ROS 講習会 学習メモ

https://github.com/DYGV/ros2_practice

岡崎英佑

2022-03-02

Contents

1 環境構築

環境構築には Docker を利用する。GUI アプリケーションも使いたいので設定が煩雑となった。

1.1 動作確認済のホスト環境

- ホスト OS: Ubuntu 20.04
- Docker バージョン: 20.10.12

1.2 構築される環境

- Ubuntu 18.04
- ROS eloquent

1.3 Dockerfile

```
FROM ubuntu:18.04
RUN --mount=type=cache,target=/var/cache/apt/archives
RUN --mount=type=cache,target=/var/cache/apt/lists
# gitを入れようとするとタイムゾーンの選択が止まってしまうのでtzdataを入れる
RUN apt-get update && apt-get install tzdata sudo

# ユーザーの追加
ARG username=eisuke
RUN echo "root:pass" | chpasswd && \
    adduser --disabled-password --gecos "" "${username}" && \
    echo "${username}:${username}" | chpasswd && \
    echo "Defaults:${username} !env_reset" >> /etc/sudoers.d/${username} && \
    echo "%${username}    ALL=(ALL)   NOPASSWD:    ALL" >> /etc/sudoers.d/${username} && \
    chmod 0440 /etc/sudoers.d/${username}

# rootから作成したユーザに切り替える
USER ${username}
# ROS周りの構築に必要なパッケージ
RUN sudo apt-get install -y --no-install-recommends \
    ca-certificates \
    git \
    curl \
    gnupg2 \
    vim \
    lsb-release \
```

```
26      python3-pip
27
28  # ROS周りの構築
29  RUN curl -s https://raw.githubusercontent.com/ros/rosdistro/master/ros.
        asc | sudo apt-key add -
30  RUN sudo sh -c 'echo "deb [arch=$(dpkg --print-architecture)] http://
        packages.ros.org/ros2/ubuntu $(lsb_release -cs) main" > /etc/apt/
        sources.list.d/ros2-latest.list'
31  RUN sudo apt-get update
32  RUN sudo apt-get install -y ros-eloquent-desktop
33  RUN sudo apt-get install -y --no-install-recommends \
34      python3-rosdep \
35      python3-colcon-common-extensions \
36      ros-eloquent-v4l2-camera \
37      ros-eloquent-gazebo-ros-pkgs \
38      ros-eloquent-dynamixel-sdk \
39      ros-eloquent-turtlebot3 \
40      ros-eloquent-turtlebot3-msgs \
41      ros-eloquent-nav2-bringup \
42      ros-eloquent-navigation2 \
43      ros-eloquent-cartographer-ros \
44      ros-eloquent-cartographer
45
46  # 必須ではない
47  RUN pip3 install --upgrade pip setuptools
48  RUN pip3 install scikit-build
49  RUN pip3 install opencv-python
50
51  # ワークスペースのコピー
52  COPY . /home/${username}/
53  RUN sudo chown -R ${username}:${username} /home/${username}/
54  # .bashrcに最低限必要なパスを追加しておく
55  RUN echo '. /opt/ros/eloquent/setup.bash' >> ~/.bashrc
56  RUN echo 'export PATH="$PATH:/usr/bin"'>> ~/.bashrc
57  RUN echo 'export PATH="$PATH:/home/'${username}'/.local/bin"' >> ~/.
        bashrc
58  RUN echo 'export GAZEBO_MODEL_PATH="$GAZEBO_MODEL_PATH:/home/'${
        username}'/dev_ws/src/turtlebot3_simulations/turtlebot3_gazebo/
        models"' >> ~/.bashrc
59
60  WORKDIR /home/${username}/dev_ws/src
61  # Gazeboのworldのlaunchファイルやモデルとかが含まれている
62  RUN git clone -b eloquent-devel https://github.com/ROBOTIS-GIT/
        turtlebot3_simulations.git
63  WORKDIR face_detect
64  RUN curl -OL https://raw.githubusercontent.com/opencv/opencv/master/
        data/haarcascades/haarcascade_frontalface_default.xml
65  WORKDIR /home/${username}
66  RUN curl -OL https://gist.githubusercontent.com/DYGV/575673
        d2857dbb19c33cc6f75753fd6f/raw/
        fa56670adbbc7b6ce5a6265100ba499eb731582f/.vimrc
```

```
67
68 CMD ["/bin/bash"]
```

1.4 初回起動

Docker イメージの作成

```
1 $ DOCKER_BUILDKIT=1 docker build . -t ubuntu1804_ros2
```

コンテナから **X** サーバへ接続するための認証情報の作成

```
1 $ # dockerに対してディスレプレイの使用を許可する
2 $ xhost +local:docker
3 $ # X11のソケットファイルの環境変数
4 $ XSOCK=/tmp/.X11-unix
5 $ # Xサーバと接続するための認証情報(xauthファイル)のパス
6 $ XAUTH=/tmp/.docker.xauth
7 $ # xauthファイルを作成する
8 $ xauth nlist $DISPLAY | sed -e 's/^..../ffff/' | xauth -f $XAUTH
    nmerge -
```

コンテナの作成

```
1 $ docker create --interactive --tty \
2 --name=container_ros2 \
3 --env=DISPLAY=$DISPLAY \
4 --volume=$XSOCK:$XSOCK \
5 --volume=/dev:/dev \
6 --device-cgroup-rule='c *:* rw' \
7 ubuntu1804_ros2 /bin/bash
```

- – name: 作成するコンテナの名前
- – interactive: コンテナの標準入力にアタッチ
- – tty: 疑似ターミナルを割り当てる
- –env: 環境変数の設定
  - 接続する X サーバの指定
- –volume: ホスト側のディレクトリをとコンテナ側でマウントする
  - X サーバのソケットファイルのディレクトリをマウント
  - ビデオデバイスを使いたいので、/dev/をマウント
- –device-cgroup-rule: デバイスリストのルール追加
  - プロセスをグループ化、リソースの利用を制限・監視する Linux カーネルの機能cgroupに、マウントしたデバイスのうちキャラクタデバイスの読み書きを可能にするルールを追加する。

1.5 コンテナの起動と接続

X サーバ関係のファイルを/tmp/に置いているので、PC の再起動後などは、コンテナからXサーバへ接続するための認証情報の作成を行ってからコンテナを起動すること。

```
$ docker start container_ros2
$ docker exec -it container_ros2 /bin/bash
```

コンテナに入ると起動したアプリケーションの GUI がホスト側で表示できるようになる。

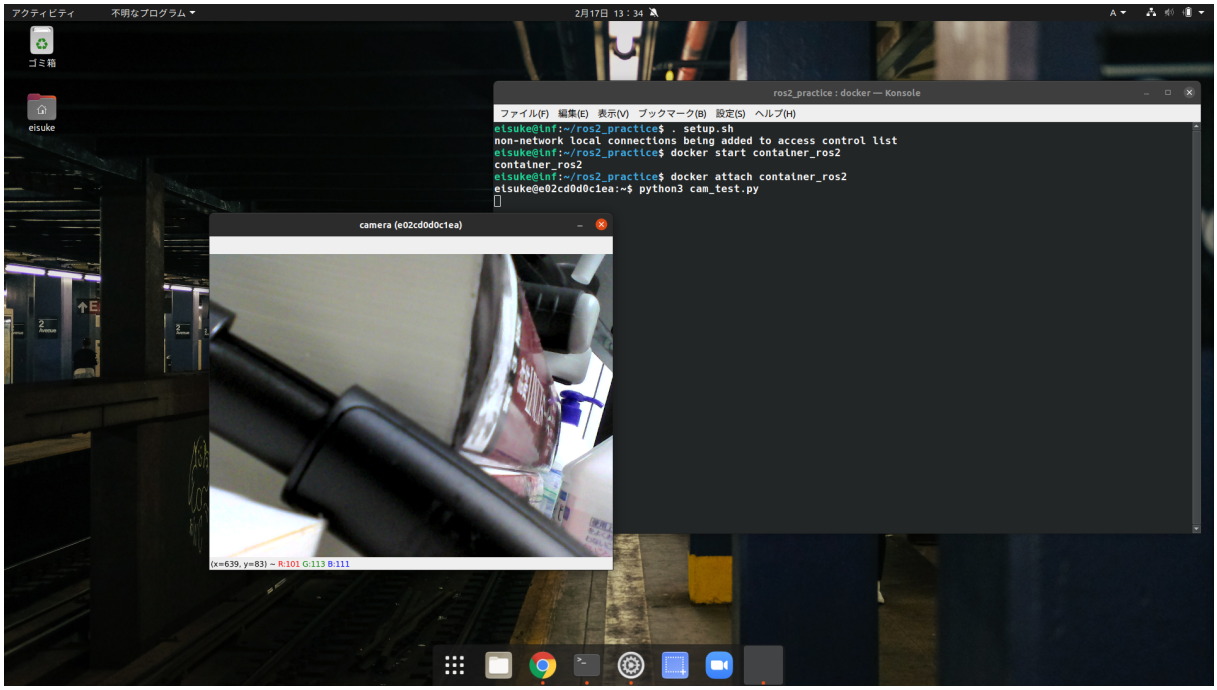


Figure 1: GUI アプリケーションを起動した様子

1.6 参考

- Docker-docs-ja
- ubuntu:20.04 docker コンテナ内でホスト側に GUI を出力（xeyes を出力）する検討メモ
- Docker Ubuntu18.04 で tzdata を install するときに timezone の選択をしないで install する
- cgroup の Device Whitelist Controller について
- cgroups(7) - Linux manual page

2 演習 1

2.1 目的

- Topic通信(Publish/Subscribe通信)の基礎を理解する
- rclpyの基本的な関数・メソッドがどんなことをしているのかをふんわりと理解する
- (Python の基礎を復習する)

2.2 実装

2.2.1 パッケージの作成

```
$ cd ~/dev_ws/src/
$ ros2 pkg create --build-type ament_python ex1
$ cd ex1
```

2.2.2 ex1/talker.py

```
import random

import rclpy
from rclpy.node import Node
from std_msgs.msg import UInt8

"""
Talker: rclpyというライブラリのnode.pyに実装されているNodeクラスを継承する。
Nodeクラスの実装は
https://github.com/ros2/rclpy/blob/eloquent/rclpy/rclpy/node.py#L92
にある。
継承により、Talkerクラス内からNodeクラスに定義されたメソッドを呼ぶことができる。
"""


class Talker(Node):
    def __init__(self):
        """
        super()は親クラス、つまりNodeを表す。super.__init__()で親クラスのコンストラクタを呼び出す(インスタンス化)。
        Nodeクラスのコンストラクタを見ると、第一引数は"node_name: str"となっている。
        例えば、第一引数に"talker"と渡せば、このクラスは、talkerというノード名が割り当てられる。
```

```
22          """
23          super().__init__("talker")
24
25
26          """
27          ノードは作れたがトピックにメッセージを流すにはトピックの登録が必要である。
28          create_publisherメソッドを使う。
29          軽くcreate_publisherの実装を見た感じでは、rclpy.publisher.pyに定義されているPublisherというクラスのインスタンス化をして、
30          Nodeクラス内publishersというリストに作ったインスタンスを保持している。
31          これによって、1つのノードで複数のトピックに対してpublishできるようになっている。
32          create_subscription()も似たようにSubscriptionクラスをリストに持つようにしている。
33          Publisherクラスの実装: https://github.com/ros2/rclpy/blob/eloquent/rclpy/rclpy/publisher.py
34
35          create_publisherの
36          第1引数は、トピックに流すメッセージの型
37          第2引数は、Publishするトピック名(送信先)
38          第3引数は、キューサイズ
39          (Subscriberの動作が重い場合はメッセージを取りこぼす可能性があるので、それが困る場合は高い値にしておくとよさそう?)
40
41          戻り値はPublishクラスのインスタンスである。
42          つまり、self.publisherという変数で、Publisherクラスに定義されているメソッドを呼ぶことができる。
43          実装を見ると引数は他にもあるが、いずれもオプションである。
44          今回は0-100までのいずれかの値を"numeric"というトピックにPublishし、
45          Subscriberでそれを2倍することを考えるので、符号なしの8ビット整数で間に合う。
46          """
47          self.publisher = self.create_publisher(UInt8, "numeric", 10)
48
49          # create_timerで使う変数
50          timer_period = 1.0
51
52          """
53          タイマーを作る
54          第1引数は、何秒カウントしてコールバック関数を発火させるか
55          第2引数は、タイマーが0になったら発火させるコールバック関数
56          実装を見ると引数は他にもあるが、いずれもオプションである。
57          今回は1秒カウントごとにpublish_numericというメソッドが呼ばれる
58          """
59          self.create_timer(timer_period, self.publish_numeric)
60
61      def publish_numeric(self):
62          """
```

```
63          メッセージ型UInt8クラスのインスタンス化
64          メッセージ型の定義は/opt/ros/eloquent/share/std_msgs/msg/にあり、
65          プログラムとしての実体は、IDL(インターフェース記述言語)で記述されている。(拡張子:.idl)
66          UInt8の場合以下のような記述がされていた。
67
68          module std_msgs {
69              module msg {
70                  struct UInt8 {
71                  uint8 data;
72                  };
73              };
74          };
75
76          std_msgsのモジュールにmsgというモジュールがあり、
77          その中の構造体UInt8はdataというメンバ変数を持つ。
78          つまり、UInt8というメッセージ型で通信するには
79          PublisherがUInt8のdataに値を代入、Subscriberがdataを参照すればよい。
80          UInt8のインスタンス化
81          """
82          msg = UInt8()
83
84          # msg(UInt8)のdataというメンバに対してランダムな値を代入する。
85          msg.data = random.randint(0, 100)
86          # msgを"numeric"というトピックに対してPublishする。
87          # publishメソッドの引数は、送信したいメッセージのみ。
88          self.publisher.publish(msg)
89          # どのノードが出力しているメッセージも合わせて出力してくれるので便利
90          self.get_logger().info("Publishing: {}".format(msg))
91
92
93  def main(args=None):
94      """
95      rclpyパッケージをインポートした時(import rclpy)に使える関数は
96      https://github.com/ros2/rclpy/blob/eloquent/rclpy/rclpy/__init__.py
97      に定義されている。
98      rclpyパッケージ全体の初期化関数initの引数はいずれもオプション
99      """
100     rclpy.init(args=args)
101
102     # 上で定義したtalkerノードのインスタンス化
103     talker = Talker()
104
105     """
106     以下のspin(talker)でtalkerを実際に動かす。
107     rclpy.spinではタスクを管理するexecutor(実行機)の初期化処理とノードが立ち上がっているときブロッキングし続ける役割がありそう。
108     spin()から軽く追ってみたけど、読み解くのが大変な箇所かもしれない。
```

```
109         https://github.com/ros2/rclpy/blob/eloquent/rclpy/rclpy/__init__.py
                #L175
110         最終的にTaskというクラスにたどり着き、そこでexecutorの設定を行って
                いるような感じだった。
111
112         読めてないけど、
113         http://design.ros2.org/articles/node_lifecycle.html
114         あたりを読むとROS全体の内部状態と動作がつかめるかも。
115         """
116         rclpy.spin(talker)
117
118         """
119         ノードの破壊
120         https://github.com/ros2/rclpy/blob/eloquent/rclpy/rclpy/node.py#
                L1467
121         でdestory_publisher()を呼び、
122         destroy_publisher()では、PublisherをNodeクラスで管理しているリスト
                から削除する。
123         """
124         talker.destory_node()
125
126         # rclpyの初期化時に作成されたデータを破棄する
127         # https://github.com/ros2/rclpy/blob/eloquent/rclpy/rclpy/__init__.
                py#L90
128         rclpy.shutdown()
129
130
131     # このプログラムがトップレベル(エントリポイント)として実行されたなら
            mainを実行する
132     if __name__ == "__main__":
133         main()
```

2.2.3 ex1/listener.py

```
1   import rclpy
2   from rclpy.node import Node
3   from std_msgs.msg import UInt8
4
5
6   class Listener(Node):
7       def __init__(self):
8           # 継承したNodeクラスの初期化
9           # listenerというノード名を割り当てる
10          super().__init__("listener")
11
12          """
13          ノードは作れたがトピックからメッセージを受信するにはトピックの
                登録が必要である。
14          talkerノードの実装で"numeric"というトピックに数字を流すようにし
```

```
            たので、"numeric"から受信する。
15          データを受け取るとdoubleというメソッドが呼ばれる。
16          create_publisherのときのようにNodeクラスがsubscriptionsというリ
                ストでSubscriptionクラスを保持しているので、
17          1つのノードで複数のsubscriptionすることもできるようになってい
                る。
18
19          また、諸々を端折って実際にコールバック関数を呼んでいるのは、
                executors.pyのawait_or_execute関数内で
20          return callback(*args)っぽい。
21          executor: https://github.com/ros2/rclpy/blob/eloquent/rclpy/
                rclpy/executors.py#L111
22          create_subscription(): https://github.com/ros2/rclpy/blob/
                eloquent/rclpy/rclpy/node.py#L1120
23          """
24          self.create_subscription(UInt8, "numeric", self.double, 10)
25
26      def double(self, msg):
27          self.get_logger().info("{0}*2 = {1}".format(msg.data, msg.data
                * 2))
28
29
30  def main(args=None):
31      rclpy.init(args=args)
32      listener = Listener()
33      rclpy.spin(listener)
34      listener.destory_node()
35      rclpy.shutdown()
36
37
38  if __name__ == "__main__":
39      main()
```

2.2.4 setup.py

setup.py内の setup(...) に以下を追記する。

```
1  entry_points={
2      'console_scripts': [
3          "talker=ex1.talker:main",
4          "listener=ex1.listener:main"
5      ],
6  },
```

2.2.5 launch/ex1_launch.py

launch_ros の node.pyの Node クラスをみると、立ち上げるノードの指定やパラメータの設定などができる

ようになっている。パラメータの詳細は演習 2 で説明する。

```python
from launch import LaunchDescription
from launch_ros.actions import Node


def generate_launch_description():
    return LaunchDescription(
        [
            Node(package="ex1", node_executable="talker", output="
                screen"),
            Node(package="ex1", node_executable="listener", output="
                screen"),
        ]
    )
```

talkerノードはnumericトピックに対してメッセージを送り、listenerノードはnumericトピックからメッセージを受け取っていることがわかる。

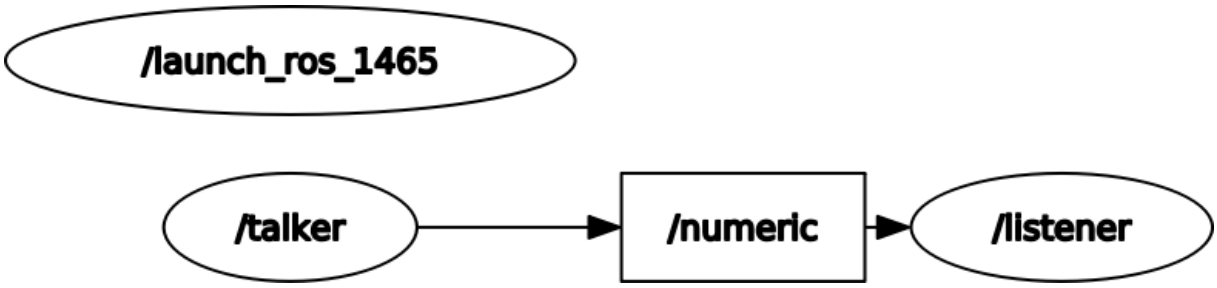


Figure 2: rqt_graph

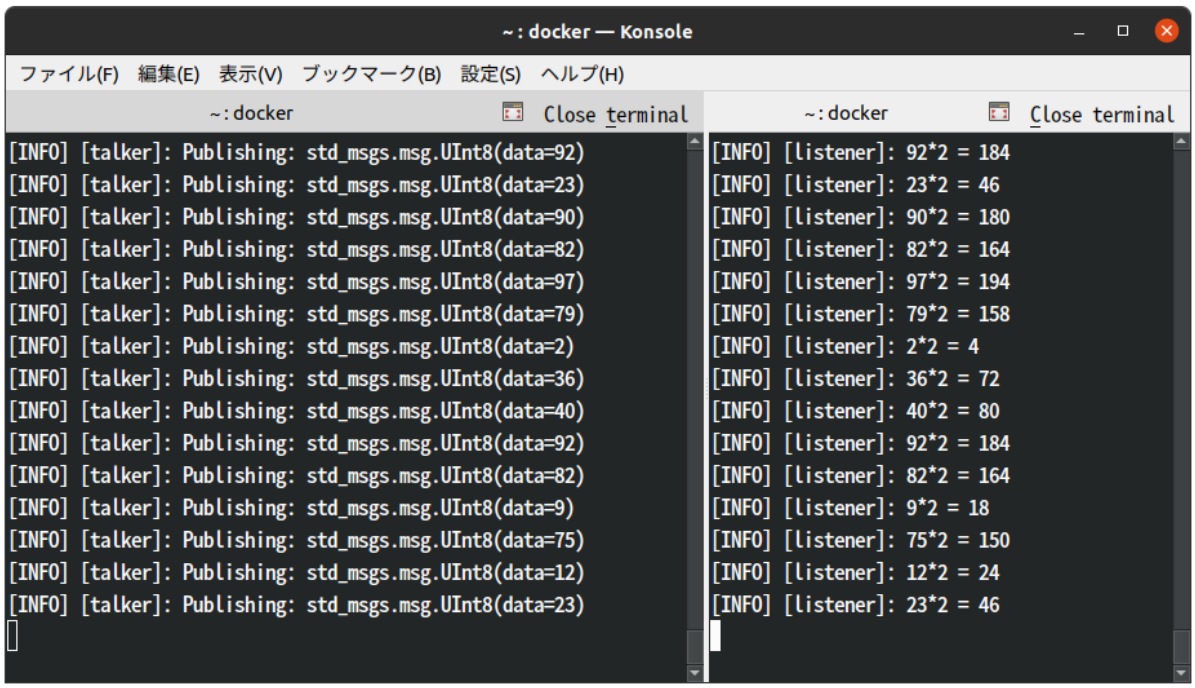


Figure 3: 実行画面

2.3 参考

- Writing a simple publisher and subscriber (Python)
- ros2/rclpy at eloquent
- ros2/launch_ros at eloquent

3 演習 2

3.1 目的

- 独自メッセージの作り方を学ぶ
- パラメータの作り方、設定の仕方を学ぶ

3.2 仕様

演習 1 のようなシンプルなシステムを作る。

- 2 つの数値x,yを格納できるメッセージ型を作りPublish/Subscribe通信をする

- Publisherは、2 つのランダムな数値x,yをトピックに流す
- Subscriberは、トピックから数値をもらい、x,yをそれぞれ N 倍し、画面に出力する
- Publisher側のパラメータ
  - rand_min: 取りうるランダムな数値の下限値
  - rand_max: 取りうるランダムな数値の上限値
  - freq: 1 秒間にPublishする回数
- Subscriber側のパラメータ
  - N: 何倍するか

3.3 独自メッセージを定義するパッケージの作成

3.3.1 パッケージを作る

```
$ cd ~/dev_ws/src/
$ ros2 pkg create --build-type ament_cmake ex2_msgs
$ cd ex2_msgs
$ mkdir msg
```

3.3.2 メッセージ型を定義する

msg/Ex2.msg

```
int32 x
int32 y
```

3.3.3 package.xml の編集

演習 1 で説明したとおり、メッセージ型は IDL という記述言語で記述されるため、これを生成するために必要な依存パッケージを追記する。

```
<buildtool_depend>rosidl_default_generators</buildtool_depend>
<exec_depend>rosidl_default_runtime</exec_depend>
<member_of_group>rosidl_interface_packages</member_of_group>
```

3.3.4 CMakeLists.txt の編集

ビルド時にメッセージ型を作るように、以下を追記する。

```
# メッセージ型の生成で必要な依存パッケージ
find_package(rosidl_default_generators REQUIRED)
# 作成したメッセージ型の定義ファイル(.msg)からIDLを生成し、実際に使えるようにする
rosidl_generate_interfaces(${PROJECT_NAME} "msg/Ex2.msg")
```

3.3.5 ビルド

```
$ colcon build
```

3.3.6 確認

```
$ ros2 interface show ex2_msgs/msg/Ex2
int32 x
int32 y
```

3.4 作成した独自メッセージで **Pub/Sub** 通信をするパッケージの作成

3.4.1 パッケージを作る

```
$ ros2 pkg create --build-type ament_python ex2
```

3.4.2 ex2/talker.py

```
import random

import rclpy
from rclpy.node import Node

from ex2_msgs.msg import Ex2


class Talker(Node):
  def __init__(self):
      super().__init__("talker")
      # Publishする数値の最小値
      self.declare_parameter("rand_min", 0)
      # Publishする数値の最大値
      self.declare_parameter("rand_max", 100)
      # 1秒間に何回Publishしたいか(周波数)
      self.declare_parameter("freq", 1)
```

```
18          # パラメータにセットされた値をインスタンス変数として保持しておく
19          self.rand_min = self.get_parameter("rand_min").value
20          self.rand_max = self.get_parameter("rand_max").value
21          # タイマーの秒数として使いたいので周期にする
22          timer_period = 1.0 / self.get_parameter("freq").value
23          # Ex2という型を流すトピックex2
24          self.publisher = self.create_publisher(Ex2, "ex2", 10)
25          self.create_timer(timer_period, self.publish_callback)
26
27      def publish_callback(self):
28          msg = Ex2()
29          msg.x = random.randint(self.rand_min, self.rand_max)
30          msg.y = random.randint(self.rand_min, self.rand_max)
31          self.publisher.publish(msg)
32          self.get_logger().info("Publishing: {}".format(msg))
33
34
35  def main(args=None):
36      rclpy.init(args=args)
37      talker = Talker()
38      rclpy.spin(talker)
39      talker.destory_node()
40      rclpy.shutdown()
41
42
43  if __name__ == "__main__":
44      main()
```

3.4.3 ex2/listener.py

```
1   import rclpy
2   from rclpy.node import Node
3
4   from ex2_msgs.msg import Ex2
5
6
7   class Listener(Node):
8       def __init__(self):
9           super().__init__("listener")
10          # 何倍するかのパラメータの宣言
11          self.declare_parameter("N", 1)
12          # パラメータにセットされた値をインスタンス変数として保持しておく
13          self.n = self.get_parameter("N").value
14          self.create_subscription(Ex2, "ex2", self.times, 10)
15
16      def times(self, msg):
17          self.get_logger().info("{0}*{1} = {2}".format(msg.x, self.n, msg.
                x * self.n))
18          self.get_logger().info("{0}*{1} = {2}".format(msg.y, self.n, msg.
```

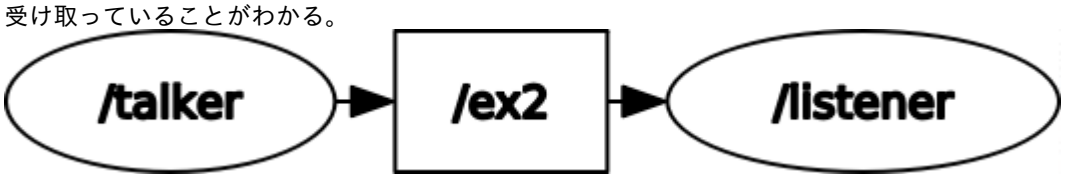
```
            y * self.n))
19
20
21  def main(args=None):
22      rclpy.init(args=args)
23      listener = Listener()
24      rclpy.spin(listener)
25      listener.destory_node()
26      rclpy.shutdown()
27
28
29  if __name__ == "__main__":
30      main()
```

3.4.4 setup.py

setup.py内の setup(...) に以下を追記する。

```
1    entry_points={
2        'console_scripts': [
3            "talker=ex2.talker:main",
4            "listener=ex2.listener:main"
5        ],
6    },
```

talker ノードは ex2 トピックに対してメッセージを送り、listener ノードは ex2 トピックからメッセージを受け取っていることがわかる。



3.5 パラメータの値の設定

talker.pyとlistener.pyで定義したパラメータは、コマンドライン引数として設定する方法と launch ファイルから設定する方法がある。どちらもパラメータを直接指定する方法と、YAML ファイルに記述されたパラメータ設定を読ませる方法の 2 通りがある。### コマンドライン引数から個別に指定する場合

ros2 run <パッケージ名> <実行名> --ros-args -p <パラメータ名>:=<値> -p <パラメータ名>:=<値> ...

```
1  $ # Listenerの何倍するかというパラメータを3とする例
2  $ ros2 run ex2 listener --ros-args -p N:=3
3  $ # ランダムな数値の下限値を100、最大値を200、1秒間に2回Publishする例
4  $ ros2 run ex2 talker --ros-args -p rand_min:=100 -p rand_max:=200 -p
       freq:=2
```

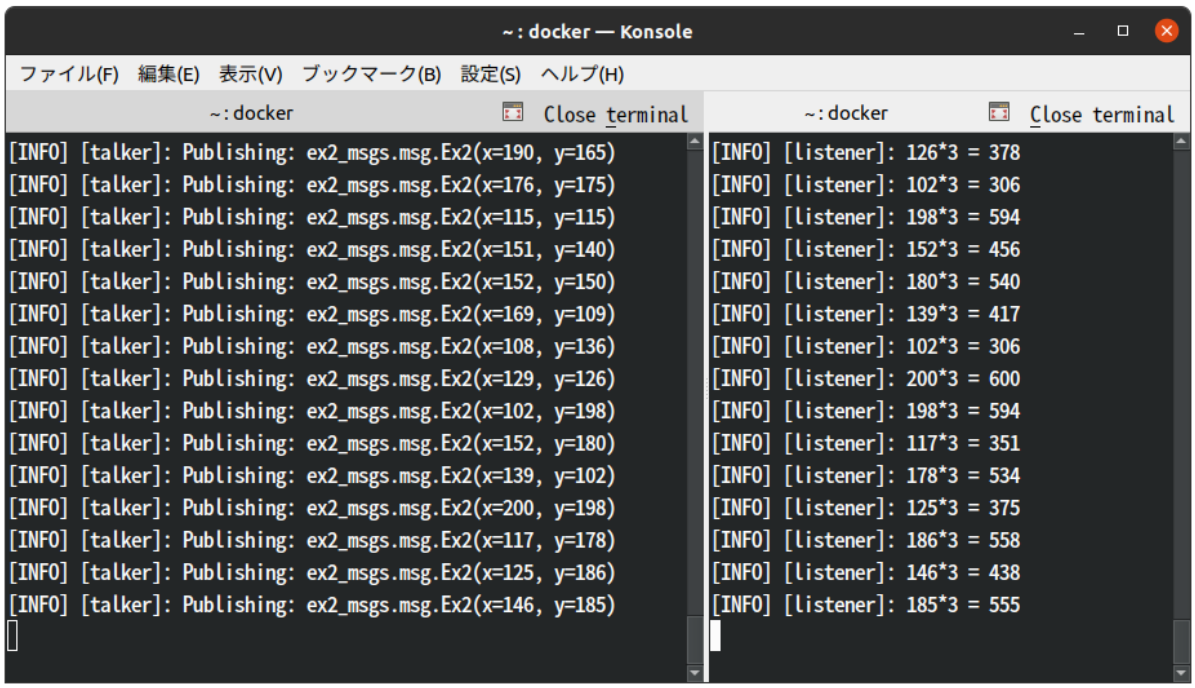


Figure 4: 実行画面

3.5.1 コマンドライン引数から **YAML** ファイルを使って指定する場合

ROS2 rcl YAML paramter parserに記載のフォーマットに従った YAML ファイルを読ませる。

`ros2 run <パッケージ名> <実行名> --ros-args --params-file <YAMLファイルのパス>`

```yaml
talker:
  ros__parameters:
    rand_min: 100
    rand_max: 200
    freq: 2

listener:
  ros__parameters:
    N: 3
```

例えば、上記のような YAML ファイル (ex2_params.yaml) なら、

```
$ ros2 run ex2 talker --ros-args --params-file ./ex2_params.yaml
$ ros2 run ex2 listener --ros-args --params-file ./ex2_params.yaml
```

とすればよい。また、パラメータを YAML ファイル形式で出力したいときは、ノードを立ち上げた状態で、`ros2 param dump /talker`とすればよい。パラメータが多く、手打ちで YAML の構造を作るのが大変なときに有用だと考えられる。

3.5.2 launch ファイルから指定する場合

launch_ros の node.py にある Node クラスのコンストラクタを見ると、parametersという引数がある。

> def __init__(
> self, *,
> node_executable: SomeSubstitutionsType,
> package: Optional[SomeSubstitutionsType] = None,
> node_name: Optional[SomeSubstitutionsType] = None,
> node_namespace: SomeSubstitutionsType = ”,
> parameters: Optional[SomeParameters] = None,
> remappings: Optional[SomeRemapRules] = None,
> arguments: Optional[Iterable[SomeSubstitutionsType]] = None,
> **kwargs
>) -> None:

parametersの説明には、

> The parameters are passed as a list, with each element either a yaml file that contains parameter rules (string or pathlib.Path to the full path of the file), or a dictionary that specifies parameter rules.

ということなのでリストで、YAML ファイルのパスを渡すか、パラメータが載った辞書形式 (key-value) で渡せばよい。

```
1  from launch_ros.actions import Node
2
3  from launch import LaunchDescription
4
5
6  def generate_launch_description():
7      return LaunchDescription(
8          [
9              Node(
10                 package="ex2",
11                 node_executable="talker",
12                 parameters=["/home/eisuke/dev_ws/src/ex2/ex2_params.
                       yaml"],
13                 # parameters=[{"rand_min": 100, "rand_max": 200, "freq
                       ": 2}],
14                 output="screen",
15             ),
16             Node(
17                 package="ex2",
18                 node_executable="listener",
```

```
19                  parameters=["/home/eisuke/dev_ws/src/ex2/ex2_params.
                        yaml"],
20                  # parameters=[{"N": 3}],
21                  output="screen",
22              ),
23          ]
24      )
```

3.6 参考

- Creating custom ROS 2 msg and srv files
- Understanding ROS 2 parameters
- ros2/rclpy at eloquent
- ros2/rcl at eloquent

4 演習 3

4.1 目的

- 複数のSubscriberを動かして動作を理解する
- 同じ実行名のノードを別ノードとして立ち上げる方法を学ぶ

4.2 仕様

演習2で作成したパッケージを使い回し、複数のSubscriberで処理をする。ランダムな数値x,yをex2トピックに流すノードが 1 つと、ex2トピックから受信した数値x,yを 2 倍、3 倍、4 倍するノード 3 つを立ち上げることを考える。この仕様を満たすには、同じ実行名のノードを別のノード名として設定する (remap: 書き換える) 方法と演習 2 で行ったパラメータの設定を行うことで実現する。

4.3 コマンドラインからノードを立ち上げる場合

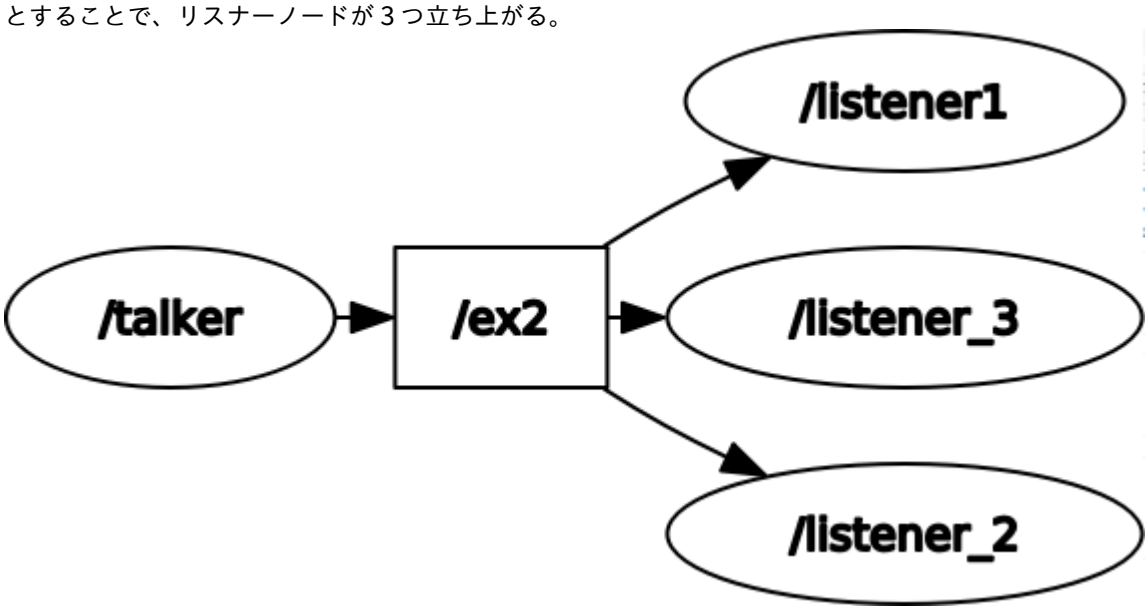
ros2 run <パッケージ名> <実行名> --ros-args -r __node:=<ノード名>でノード名を指定した上で、ノードを立ち上げることができる。例えば、talker を

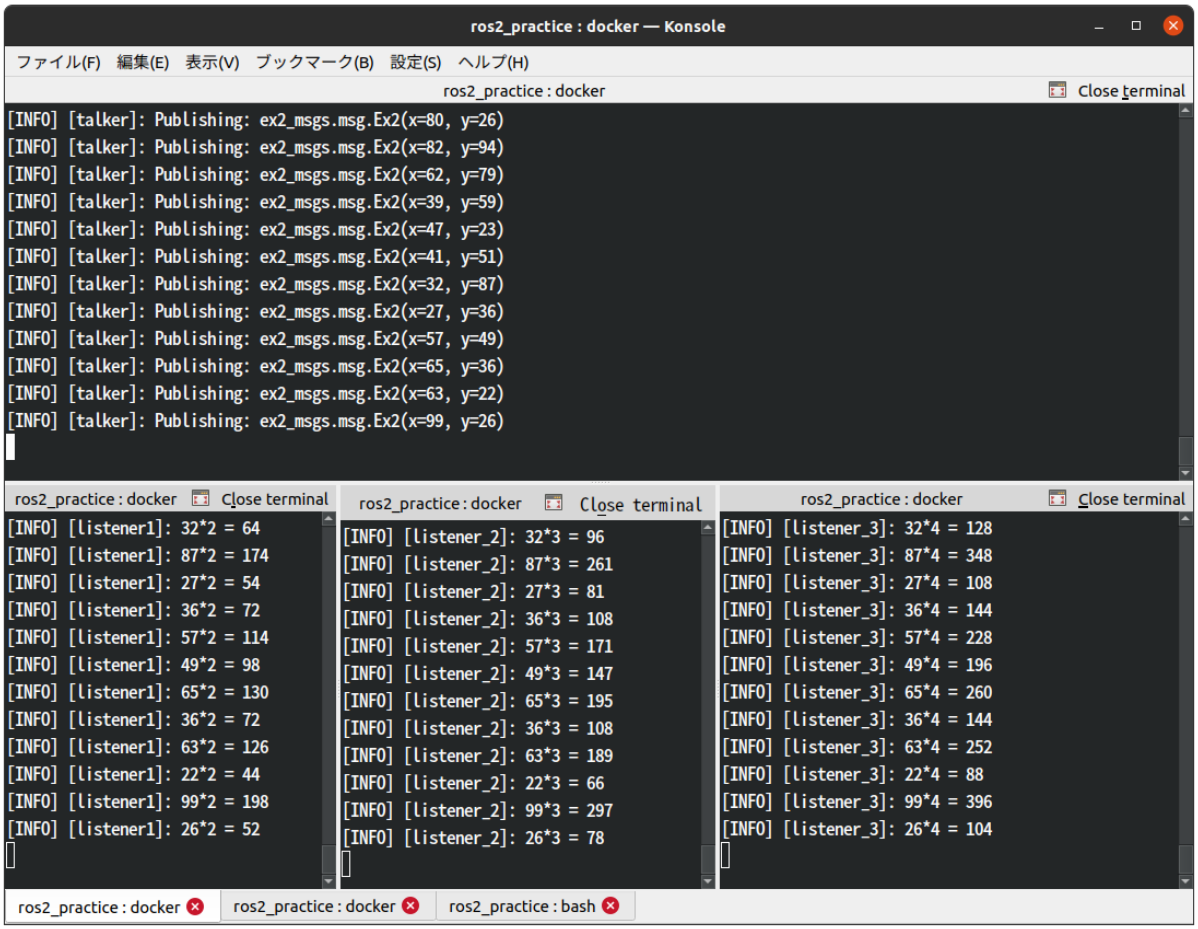
```
1 $ ros2 run ex2 talker
```

とし、listener を

```
$ ros2 run ex2 listener --ros-args -p N:=2 -r __node:=listener_1 &
$ ros2 run ex2 listener --ros-args -p N:=3 -r __node:=listener_2 &
$ ros2 run ex2 listener --ros-args -p N:=4 -r __node:=listener_3 &
```

とすることで、リスナーノードが 3 つ立ち上がる。





4.4 launch ファイルからノードを立ち上げる場合

Node のコンストラクタの説明によれば、

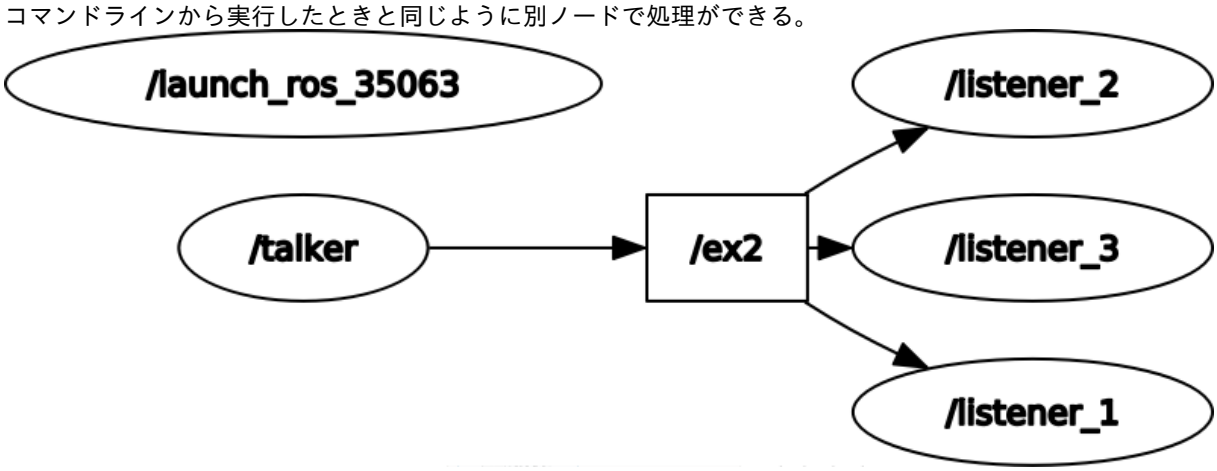
> If the node_name is not given (or is None) then no name is passed to the node on creation and instead the default name specified within the code of the node is used instead.

ということなので、node_nameを指定してやることで、ノード名が違う同じような動作をするノードを複製することができる。

```
1  from launch_ros.actions import Node
2
3  from launch import LaunchDescription
4
5
6  def generate_launch_description():
7      return LaunchDescription(
```

```
        [
            Node(
                package="ex2",
                node_executable="talker",
                parameters=[{"rand_min": 100, "rand_max": 200, "freq":
                    2}],
                output="screen",
            ),
            Node(
                package="ex2",
                node_executable="listener",
                node_name="listener_1",
                parameters=[{"N": 2}],
                output="screen",
            ),
            Node(
                package="ex2",
                node_executable="listener",
                node_name="listener_2",
                parameters=[{"N": 3}],
                output="screen",
            ),
            Node(
                package="ex2",
                node_executable="listener",
                node_name="listener_3",
                parameters=[{"N": 4}],
                output="screen",
            ),
        ]
    )
```

コマンドラインから実行したときと同じように別ノードで処理ができる。



4.5 参考

- ros2/rclpy at eloquent
- Passing ROS arguments to nodes via the command-line

5 サービス通信で人の顔検出を行う

5.1 概要

演習 2 で独自メッセージを作成したときに参考にしたCreating custom ROS 2 msg and srv filesで、トピック通信ではなく、サービス通信という通信方法ががあったため、これを試す。

5.2 目的

- トピック通信との違いを学ぶ
- 独自メッセージの作り方を学ぶ
- 独自メッセージを使った ROS パッケージの作り方を学ぶ
- launch ファイルの作り方を学ぶ
- パラメータの作り方を学ぶ

5.3 トピック通信との違い

トピック通信は、PublisherはTopicに対してメッセージを送信し、SubscriberはTopicからメッセージをもらってくる通信モデルである。一方で、サービス通信は、ClientからServerに要求 (リクエスト)、ServerからClientに対して応答 (レスポンス) し、相互にメッセージを送信することができる。いわゆるソケット通信のクライアント・サーバモデルみたいなもの。

5.4 作成する題材

サービス通信でカメラを用いた人の顔の検出を行う。処理手順は以下のとおりである。

- Client
    1. カメラから画像を入力し、画像をグレースケールに変換
    2. グレースケール画像をServerに送信
    3. Serverから座標と大きさの受信
    4. 座標と大きさからマーカー描画

5. Image 型のトピックに画像送信 (確認用)

- Server

1. グレースケール画像受信
2. 顔の検出 (今回は学習済みのカスケード分類器を使う)
3. 検出した座標と大きさをClientに送信

5.5 作成する手順

はじめに、サービス通信で用いる独自メッセージを作成する。その後、ServerとClientを作成する。

5.6 サービス通信で用いる独自メッセージの作成

5.6.1 パッケージの作成

```
$ ros2 pkg create --build-type ament_cmake face_msgs && cd face_msgs/
```

5.6.2 独自メッセージの定義

メッセージには、Clientから送信する画像 image、Serverから送信する顔が検出された左上の x,y 座標とその大きさ w,h を含める。---をデリミタとしてClientがServerに送信するメッセージ (リクエスト) を上に、ClientがServerから受信するメッセージ (レスポンス) を下に記述すれば良い。また、後述する顔検出で、検出する関数がnumpy.int32を返すので、それにビット幅を合わせておく。

```
$ mkdir srv && cd srv
$ echo -e \
"sensor_msgs/Image image\n\
---\n\
int32 x\n\
int32 y\n\
int32 w\n\
int32 h"\
> Face.srv
```

5.6.3 CMakeLists.txt への追記

ビルドで独自メッセージを作成するのにCMakeLists.txtへその情報を追加する必要がある。23 行目あたりに以下を追記する。

```
find_package(sensor_msgs REQUIRED)
find_package(rosidl_default_generators REQUIRED)
rosidl_generate_interfaces(${PROJECT_NAME}
      "srv/Face.srv"
      DEPENDENCIES sensor_msgs)
```

5.6.4 package.xml への追記

14 行目あたりに以下を追記する

```
<build_depend>rosidl_default_generators</build_depend>
<exec_depend>rosidl_default_runtime</exec_depend>
<member_of_group>rosidl_interface_packages</member_of_group>
```

5.6.5 ビルド

いつもどおりcolcon buildで OK
ビルドが通ったら、以下のようにインターフェースができたか確認する。

```
$ . install/setup.bash
$ ros2 interface list | grep face_msgs
face_msgs/srv/Face
```

5.7 サービス通信でのアプリケーションの作成

ServerとClientが実装すべき処理は作成する題材で記述したとおりである。
また、パラメータとして、Serverではカスケード分類器のファイルパス、Clientではカメラのデバイス番号と顔検出処理の周波数を指定する。
パッケージの作成

```
$ ros2 pkg create --build-type ament_python face_detect && cd
    face_detect/
```

5.7.1 カスケード分類器のダウンロード

```
$ wget https://raw.githubusercontent.com/opencv/opencv/master/data/
    haarcascades/haarcascade_frontalface_default.xml
```

5.7.2 face_detect/server.py

```python
import cv2
import rclpy
from cv_bridge import CvBridge
from rclpy.node import Node

from face_msgs.srv import Face

"""
顔検出パッケージのサーバ側
送られてきた画像を顔検出器にかけて、顔の座標・大きさを返す
"""
class Server(Node):
    def __init__(self):
        super().__init__("service")
        self.declare_parameter("cascade_path", "")
        self.srv = self.create_service(Face, "face_detect", self.
            callback)
        self.bridge = CvBridge()
        cascade_path = self.get_parameter("cascade_path").value
        self.face_cascade = cv2.CascadeClassifier(cascade_path)

    def callback(self, msg, pos):
        boxes = [0, 0, 0, 0]
        image = self.bridge.imgmsg_to_cv2(msg.image, "mono8")
        detection_boxes = self.face_cascade.detectMultiScale(image)
        if len(detection_boxes) != 0:
            boxes = detection_boxes[0]
        pos.x = int(boxes[0])
        pos.y = int(boxes[1])
        pos.w = int(boxes[2])
        pos.h = int(boxes[3])
        return pos


def main(args=None):
    rclpy.init(args=args)
    server = Server()
    rclpy.spin(server)
    server.destroy_node()
    rclpy.shutdown()


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

5.7.3 face_detect/client.py

```python
import cv2
import rclpy
from cv_bridge import CvBridge
from rclpy.callback_groups import ReentrantCallbackGroup
from rclpy.node import Node
from sensor_msgs.msg import Image

from face_msgs.srv import Face

"""
顔検出パッケージのクライアント側
要求された周波数・ビデオデバイスを用いて、
映像取得・グレースケール変換・サーバ側への送信をする。
サーバ側から顔が検出された座標・大きさが返ってきたら、
マーカーを引き、映像確認用のトピックに流す。
"""
class Client(Node):
    def __init__(self):
        super().__init__("client")
        self.declare_parameter("freq", 10)
        self.declare_parameter("video_device", -1)

        freq = self.get_parameter("freq").value
        video_device = self.get_parameter("video_device").value

        self.cam = cv2.VideoCapture(video_device)
        self.bridge = CvBridge()
        cb_group = ReentrantCallbackGroup()
        self.cli = self.create_client(Face, "face_detect",
            callback_group=cb_group)
        self.img_pub = self.create_publisher(Image, "image_display",
            10)
        self.req = Face.Request()
        timer_period = 1.0 / freq
        self.timer = self.create_timer(
            timer_period, self.pub_callback, callback_group=cb_group
        )

    def pub_callback(self):
        ret, self.frame = self.cam.read()
        src_gray = cv2.cvtColor(self.frame, cv2.COLOR_BGR2GRAY)
        self.req.image = self.bridge.cv2_to_imgmsg(src_gray, "mono8")
        future = self.cli.call_async(self.req)
        future.add_done_callback(self.draw)

    def draw(self, future):
        boxes = future.result()
        print(boxes)
        cv2.rectangle(
            self.frame,
```

```
49                 (boxes.x, boxes.y),
50                 (boxes.x + boxes.w, boxes.y + boxes.h),
51                 (255, 0, 255),
52                 2,
53             )
54             self.img_pub.publish(self.bridge.cv2_to_imgmsg(self.frame, "
                   bgr8"))
55
56
57     def main(args=None):
58         rclpy.init(args=args)
59         client = Client()
60         rclpy.spin(client)
61         client.destroy_node()
62         client.cam.release()
63         rclpy.shutdown()
64
65
66     if __name__ == "__main__":
67         main()
```

5.7.4 launch/face_detect_launch.py

```
1  from launch import LaunchDescription
2  from launch_ros.actions import Node
3
4
5  def generate_launch_description():
6      return LaunchDescription(
7          [
8              Node(
9                  package="face_detect",
10                 node_executable="client",
11                 output="screen",
12                 parameters=[{"video_device": -1, "freq": 30}],
13             ),
14             Node(
15                 package="face_detect",
16                 node_executable="server",
17                 output="screen",
18                 parameters=[
19                     {
20                         "cascade_path": "/home/eisuke/dev_ws/src/
                               face_detect/haarcascade_frontalface_default.
                               xml"
21                     }
22                 ],
23             ),
24         ]
```

```
25         )
```

5.7.5 package.xml への追記

先程作った独自メッセージを使うために**package**.xmlの 14 行目あたりに以下のように追記する。

```
1     <build_depend>face_msgs</build_depend>
2     <exec_depend>face_msgs</exec_depend>
```

5.7.6 setup.py への追記

setup.pyに以下を追記する。

```
1  import os
2  from glob import glob
```

data_files に以下を追記する。

```
1  (os.path.join("share", package_name), glob("launch/*_launch.py")),
```

さらに、entry_points に以下を追記する。

```
1  "client=face_detect.client:main", "server=face_detect.server:main",
```

5.7.7 ビルド

```
1  $ colcon build
2  $ . install/setup.bash
```

もし、ビルド時にsetup.py関係で警告がうるさいときはsetup.pyに

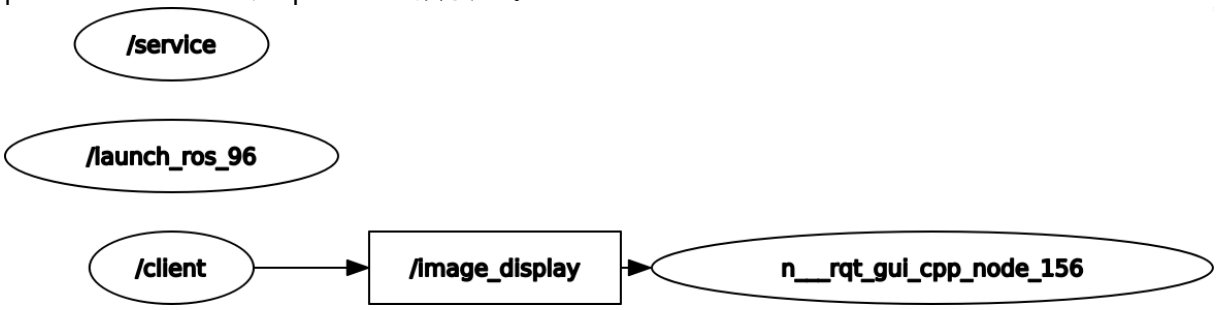
```
1  import warnings
2  warnings.filterwarnings("ignore")
```

とすればよい。

5.7.8 起動

```
1  $ ros2 launch face_detect face_detect_launch.py
```

検出してマーカーが引かれているかの確認は、clientが確認用のトピックimage_displayに画像を publish しているので、rqt でそれを確認する。



動作は以下のようになった
* 塗りつぶし処理後

https://user-images.githubusercontent.com/8480644/154873004-270db3d3-d0a7-4fa1-81e6-0e058b4f280c.mp4

5.8 参考

- Understanding ROS 2 services
- Writing a simple service and client (Python)
- Creating custom ROS 2 msg and srv files
- Haar Cascades を使った顔検出

6 Gazebo で TurtleBot3 の走行と障害物回避

6.1 概要

Gazeboは、ROS と連携してロボットを動かせるシミュレータである。今回は、Gazeboを使って、ロボットを動かす・障害物の前で止まるタスク・障害物を回避するタスクの実装を目指す。

6.2 目的

- Gazeboの基本的な操作を学ぶ
- TurtleBot3を操作する方法を学ぶ
- センサの値からTurtleBot3の操作を決定する簡単なプログラムを作れるようにする

6.3 とりあえず動かしてみる

初回の起動に時間がかかるかもしれないが、とにかく待つ。

```
1 $ gazebo & # Gazeboのバックグラウンド起動
```

すでに World があって、それを読み込むときは

```
1 $ gazebo hoge.world &
```

のようにする。

6.3.1 World を作成する

ロボットと適当な壁と障害物を配置する。TurtleBot3のモデルはWaffleにした。World ファイルはこのようになった。ファイルは適当な場所に保存する。後述する障害物検知で使う物体unit_cylinderも配置した。初めはcoke canでも良いと思ったが、センサの位置によっては物体にレーザーが照射されないので、物体の高さには注意が必要であることが後からわかった (今回は 2D の距離センサを使うので水平方向にしか飛ばない？)。

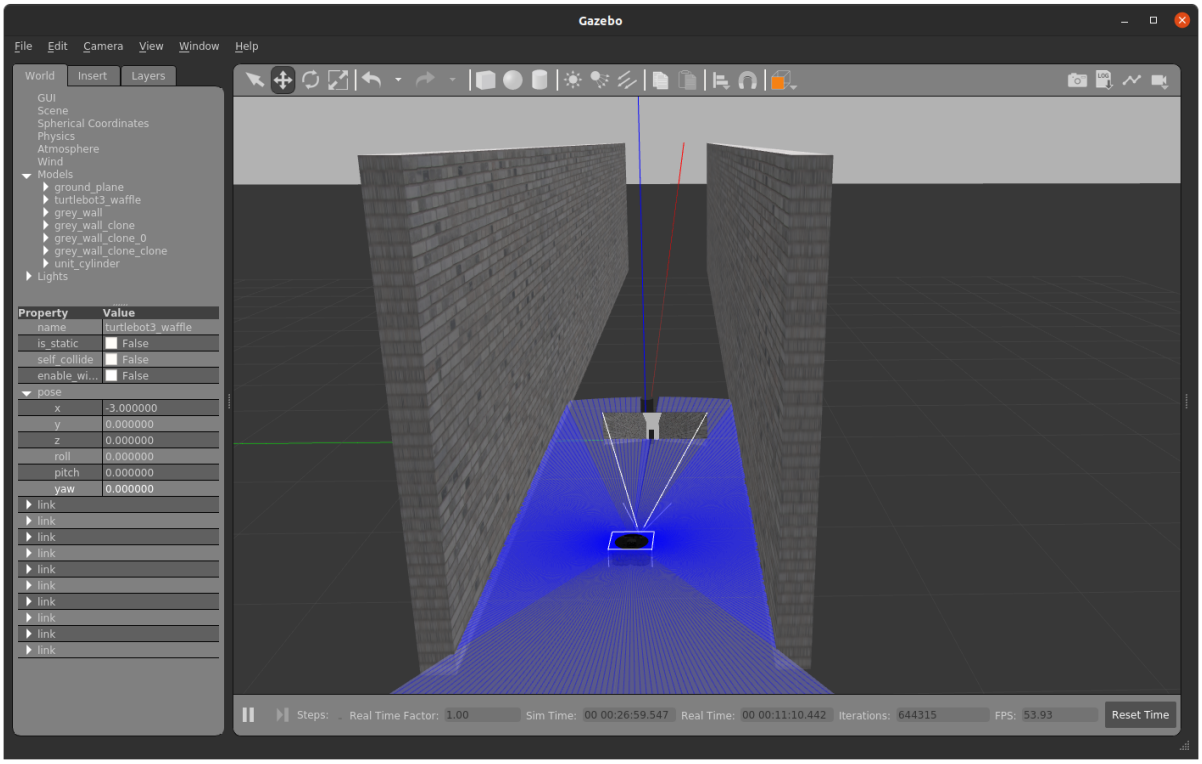


Figure 5: gazebo_world

TurtleBot3の配置が完了した時点で、以下のようにトピックが作られていることがわかる。

```
$ ros2 topic list
/camera/camera_info
/camera/image_raw
/clock
/cmd_vel
/imu
/joint_states
/odom
/parameter_events
/rosout
/scan
/tf
```

6.3.2 キーボード操作で動かしてみる

Dockerfileで立ち上げたコンテナ環境を使っている場合は必要なパッケージは既に入っているはずである。ない場合は、

```
$ sudo apt install ros-eloquent-teleop-twist-keyboard
```

で、パッケージをインストールすると、/opt/ros/eloquent/lib/にturtlebot3_teleopというパッケージが入る。

```
$ ros2 run turtlebot3_teleop teleop_keyboard
```

でTurtleBot3をキーボード操作することができる。teleop の操作は以下のとおりである。

操作	キー
前進	w
後退	x
右折	d
左折	a
停止	s

teleopの起動と rqt でカメラ画像をsubscribeすると以下のようなrqt_graphとなる。

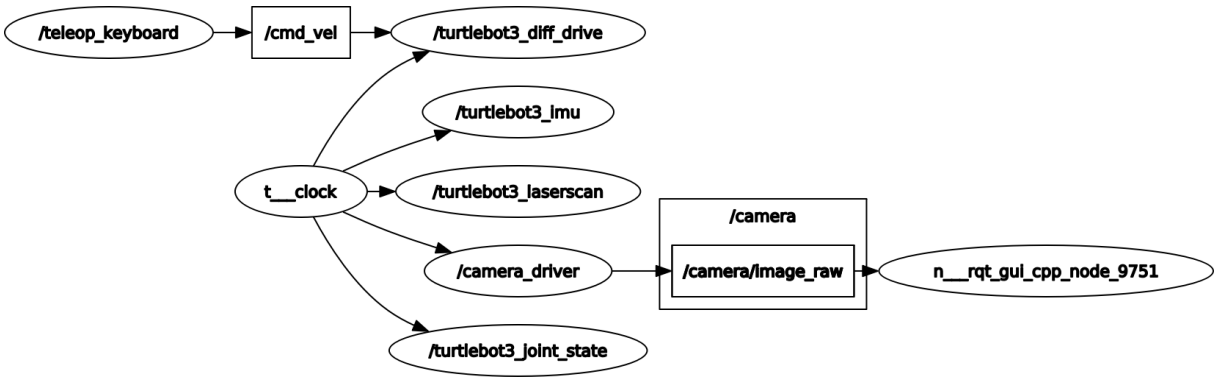


Figure 6: rqt_graph

6.4 障害物への衝突回避 (1)

TurtleBot3が運転中に前方にある障害物を検知し、ある距離まで近づいた時点で停止するタスクを考える。今回は、TurtleBot3に付属している2D 距離センサを用いて障害物を検知する。(「検知」とは、距離がinfでない数値とする。)

6.4.1 利用するトピック

このタスクを満たすために、距離の取得とTurtleBot3の移動をしたい。この距離センサから距離を取得するにはsensor_msgs/msg/LaserScanというメッセージ型を/scanトピックからsubscribeすれば良い。また、移動するためには、/cmd_velというトピックに対してTwistというメッセージ型でpublishすれば良い。

```
$ ros2 topic info /scan
Type: sensor_msgs/msg/LaserScan
Publisher count: 1
Subscriber count: 0
$ ros2 topic info /cmd_vel
Type: geometry_msgs/msg/Twist
Publisher count: 0
Subscriber count: 1
```

6.4.2 LaserScan について

メッセージ型の詳細は、ros2 **interface** showで取得できる。以下は LaserScan のメンバである。

```
$ ros2 interface show sensor_msgs/msg/LaserScan
(コメントなど省略済み)
std_msgs/Header header
```

```
float32 angle_min
float32 angle_max
float32 angle_increment
float32 time_increment
float32 scan_time
float32 range_min
float32 range_max
float32[] ranges
float32[] intensities
```

今回使うメンバは

```
float32 range_min
float32 range_max
float32[] ranges
```

の 3 つで、いずれも単位はメートルである。
range_minとrange_maxは検出できた距離の最小 (最大) 値である。rangesは 360 度スキャンした結果 (距離) が収められている。配列のインデックスが角度に対応している。ただし、スキャンできる範囲に物体が検出されなかった角度は距離がinfとなる。

6.4.3 Twist について

以下はTwistのメンバである。

```
$ ros2 interface show geometry_msgs/msg/Twist
# This expresses velocity in free space broken into its linear and
    angular parts.

Vector3  linear
Vector3  angular
```

Vector3 は C++ などの動的配列のベクターではなく、X軸周りの回転(Roll), Y軸周りの回転(Pitch), Z軸周りの回転(Yaw)の値を格納する型である。linearは並進方向、angularは回転方向で、値の単位はメートル毎秒である。例えばTurtleBot3のタイヤを x 軸の正の方向に動かしたい (前進したい) なら、

```
twist = Twist()
twist.linear.x = 0.1
```

のように使用する。このtwistを/cmd_velに対してpublishすることでロボットを動かすことができる。

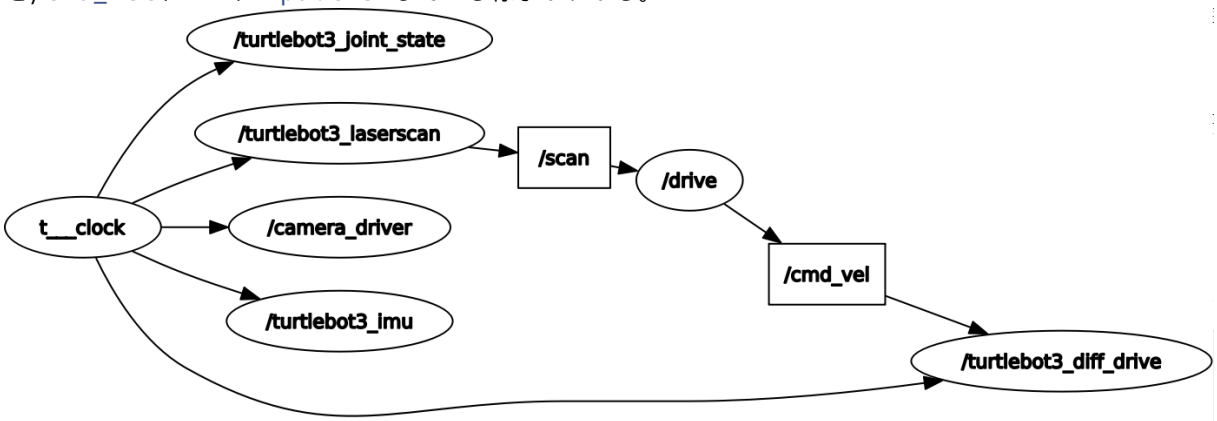
6.4.4 実装

いつもどおりパッケージを作成し、実装する。実装は以下の通りで、setup.pyへの記述を忘れないようにする。ロボットの動作は前進のみで、前方 30 度のスキャン結果のいずれかに 0.5 [m] 以下が含まれていたら停止する。

```python
import numpy
import rclpy
from geometry_msgs.msg import Twist
from rclpy.node import Node
from rclpy.qos import qos_profile_sensor_data
from sensor_msgs.msg import LaserScan


class Drive(Node):
    def __init__(self):
        super().__init__("drive")
        self.create_subscription(
            LaserScan, "scan", self.scan_callback,
                qos_profile_sensor_data
        )
        self.pub_twist = self.create_publisher(Twist, "cmd_vel", 10)
        self.twist = Twist()
        self.stop_twist = Twist()

    def __del__(self):
        self.pub_twist.publish(self.stop_twist)

    def scan_callback(self, msg):
        ranges = numpy.array(msg.ranges)
        ranges = numpy.where(ranges < msg.range_min, msg.range_max,
            ranges)
        ahead_object_distance = min(min(ranges[0:15]), min(ranges
            [345:360]))
        if ahead_object_distance < 0.5:
            self.twist.linear.x = 0.0
            print("stop")
        else:
            self.twist.linear.x = 0.3
            print(
                "{:.3f}[m] to the nearest forward obstacle".format(
                    ahead_object_distance
                )
            )
        self.pub_twist.publish(self.twist)


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

```
    rclpy.spin(drive)
    drive.destroy_node()
    rclpy.shutdown()


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

Gazeboを立ち上げたまま、実装したパッケージを立ち上げるとrqt_graphは以下のようになる。先程実装した/driveノード/scanトピックからスキャン結果をsubscribeし、機体の速度情報 (Twist) を/cmd_velトピックへpublishしている様子がわかる。



以下のように動作する。

https://user-images.githubusercontent.com/8480644/154829002-fd8a0812-61fe-4cae-a21b-7565e5335a62.mp4

6.5 障害物への衝突回避 (2)

次は、障害物を回避する、というタスク。

```
import time

import numpy
import rclpy
from geometry_msgs.msg import Twist
from rclpy.node import Node
from rclpy.qos import qos_profile_sensor_data
from sensor_msgs.msg import LaserScan


class Drive(Node):
    def __init__(self):
        super().__init__("drive")
```

```
14          self.create_subscription(
15              LaserScan, "scan", self.scan_callback,
                    qos_profile_sensor_data
16          )
17          self.pub_twist = self.create_publisher(Twist, "cmd_vel", 10)
18          self.twist = Twist()
19          self.stop_twist = Twist()
20
21      def __del__(self):
22          self.pub_twist.publish(self.stop_twist)
23
24      def rotate(self, center_index, argmin):
25          self.twist.linear.x = 0.1
26          # 距離の最小値のインデックスが
27          # 前方中央のインデックスより左にあるなら
28          # 右へ回転する
29          if argmin <= center_index:
30              self.twist.angular.z = -0.2
31          else:
32              self.twist.angular.z = 0.2
33          print("rotate")
34          self.pub_twist.publish(self.twist)
35
36      def scan_callback(self, msg):
37          ranges = numpy.array(msg.ranges)
38          ranges = numpy.where(ranges < msg.range_min, msg.range_max,
                    ranges)
39          ahead_object_distance_concat = numpy.concatenate(
40              (ranges[345:360], ranges[0:15])
41          )
42          ahead_object_distance = min(ahead_object_distance_concat)
43
44          if ahead_object_distance < 0.5:
45              argmin = numpy.argmin(ahead_object_distance_concat)
46              center_index = len(ahead_object_distance_concat) // 2
47              self.rotate(center_index, argmin)
48              time.sleep(0.5)
49          else:
50              self.twist.linear.x = 0.2
51              self.twist.angular.z = 0.0
52              self.pub_twist.publish(self.twist)
53          print("{:.3f}[m] to the nearest forward obstacle".format(
                    ahead_object_distance))
54
55
56  def main(args=None):
57      rclpy.init(args=args)
58      drive = Drive()
59      rclpy.spin(drive)
60      drive.destroy_node()
61      rclpy.shutdown()
```

```
62
63
64 if __name__ == "__main__":
65     main()
```

回避できたことが確かめられた。しかし、回避する方向を迷っているので、rotate() の処理が甘いのかもしれない？

https://user-images.githubusercontent.com/8480644/154853387-17f842be-e8e1-496f-b3e0-a3800d37830f.mp4

6.6 参考

- Gazebo
- TurtleBot3 LDS-01
- ROS の勉強　第 11 弾：センシング-LaserScan