

---

## ROS 講習会 学習メモ

[https://github.com/DYGV/ros2\\_practice](https://github.com/DYGV/ros2_practice)

岡崎英佑

2022-03-02

## Contents

<b>1</b>	<b>環境構築</b>	<b>5</b>
1.1	動作確認済のホスト環境	5
1.2	構築される環境	5
1.3	Dockerfile	5
1.4	初回起動	7
1.5	コンテナの起動と接続	8
1.6	参考	8
<b>2</b>	<b>演習 1</b>	<b>9</b>
2.1	目的	9
2.2	実装	9
2.2.1	パッケージの作成	9
2.2.2	ex1/talker.py	9
2.2.3	ex1/listener.py	12
2.2.4	setup.py	13
2.2.5	launch/ex1_launch.py	13
2.3	参考	15
<b>3</b>	<b>演習 2</b>	<b>15</b>
3.1	目的	15
3.2	仕様	15
3.3	独自メッセージを定義するパッケージの作成	16
3.3.1	パッケージを作る	16
3.3.2	メッセージ型を定義する	16
3.3.3	package.xml の編集	16
3.3.4	CMakeLists.txt の編集	16
3.3.5	ビルド	17
3.3.6	確認	17
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	17
3.4.1	パッケージを作る	17
3.4.2	ex2/talker.py	17
3.4.3	ex2/listener.py	18
3.4.4	setup.py	19
3.5	パラメータの値の設定	19
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	20
3.5.2	launch ファイルから指定する場合	21

3.6	参考	22
<b>4</b>	<b>演習 3</b>	<b>22</b>
4.1	目的	22
4.2	仕様	22
4.3	コマンドラインからノードを立ち上げる場合	22
4.4	launch ファイルからノードを立ち上げる場合	24
4.5	参考	26
<b>5</b>	<b>サービス通信で人の顔検出を行う</b>	<b>26</b>
5.1	概要	26
5.2	目的	26
5.3	トピック通信との違い	26
5.4	作成する題材	26
5.5	作成する手順	27
5.6	サービス通信で用いる独自メッセージの作成	27
5.6.1	パッケージの作成	27
5.6.2	独自メッセージの定義	27
5.6.3	CMakeLists.txt への追記	28
5.6.4	package.xml への追記	28
5.6.5	ビルド	28
5.7	サービス通信でのアプリケーションの作成	28
5.7.1	カスケード分類器のダウンロード	28
5.7.2	face_detect/server.py	29
5.7.3	face_detect/client.py	30
5.7.4	launch/face_detect_launch.py	31
5.7.5	package.xml への追記	32
5.7.6	setup.py への追記	32
5.7.7	ビルド	32
5.7.8	起動	32
5.8	参考	33
<b>6</b>	<b>Gazebo で TurtleBot3 の走行と障害物回避</b>	<b>33</b>
6.1	概要	33
6.2	目的	33
6.3	とりあえず動かしてみる	34
6.3.1	World を作成する	34
6.3.2	キーボード操作で動かしてみる	35

6.4	障害物への衝突回避 (1)	36
6.4.1	利用するトピック	36
6.4.2	LaserScan について	36
6.4.3	Twist について	37
6.4.4	実装	38
6.5	障害物への衝突回避 (2)	39
6.6	参考	41
<b>7</b>	<b>まとめ</b>	<b>41</b>

## 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

```
1 FROM ubuntu:18.04
2 RUN --mount=type=cache,target=/var/cache/apt/archives
3 RUN --mount=type=cache,target=/var/cache/apt/lists
4 # 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
5 RUN apt-get update && apt-get install tzdata sudo
6
7 # ユーザーの追加
8 ARG username=eisuke
9 RUN echo "root:pass" | chpasswd && \
10     adduser --disabled-password --gecos "" "${username}" && \
11     echo "${username}:${username}" | chpasswd && \
12     echo "Defaults:${username} !env_reset" >> /etc/sudoers.d/${username}
13     && \
14     echo "%${username}    ALL=(ALL)    NOPASSWD:    ALL" >> /etc/sudoers
15     .d/${username} && \
16     chmod 0440 /etc/sudoers.d/${username}
17
18 # rootから作成したユーザに切り替える
19 USER ${username}
20
21 # ROS周りの構築に必要なパッケージ
22 RUN sudo apt-get install -y --no-install-recommends \
23     ca-certificates \
24     git \
25     curl \
26     gnupg2 \
27     vim \
28     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/575673d2857dbb19c33cc6f75753fd6f/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、/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。

```
1 $ docker start container_ros2
2 $ 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。

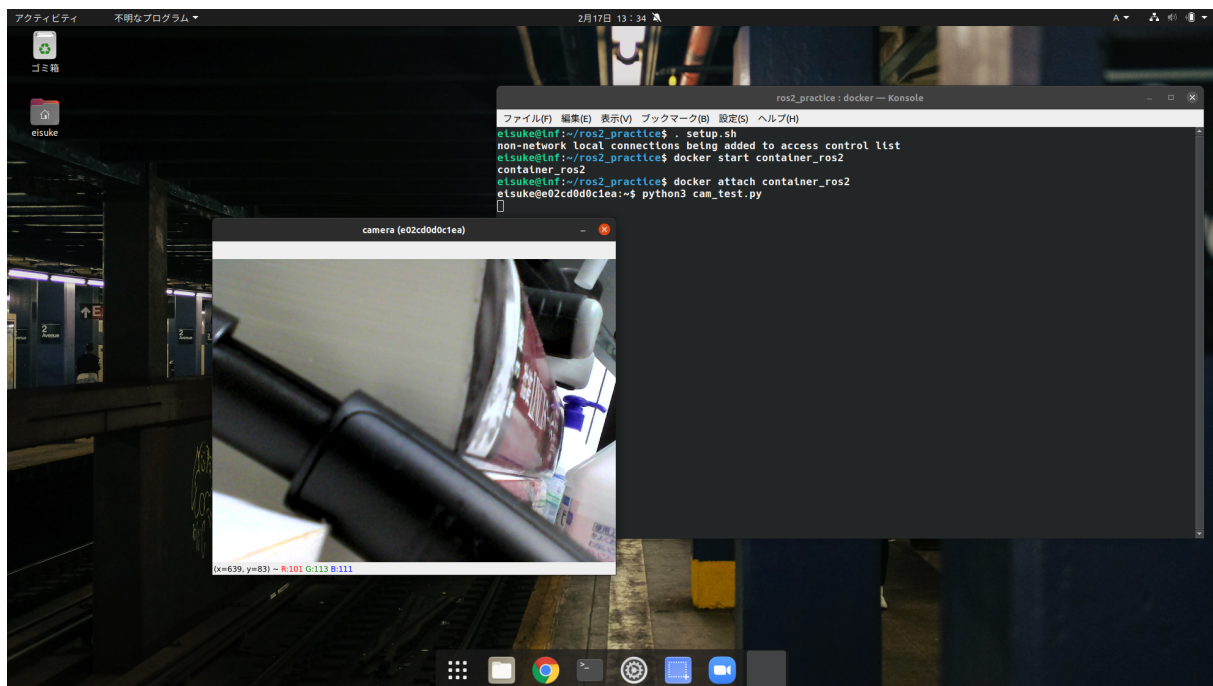


Figure 1: GUI アプリケーションを起動した様子

## 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

```
1 $ cd ~/dev_ws/src/
2 $ ros2 pkg create --build-type ament_python ex1
3 $ cd ex1
```

#### 2.2.2 ex1/talker.py

```
1 import random
2
3 import rclpy
4 from rclpy.node import Node
5 from std_msgs.msg import UInt8
6
7 """
8 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。
9 Nodeクラスの実装は
10 https://github.com/ros2/rclpy/blob/eloquent/rclpy/rclpy/node.py#L92
11 にある。
12 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。
13 """
14
15
16 class Talker(Node):
17     def __init__(self):
18         """
19         super()は親クラス、つまりNodeを表す。super.__init__()で親クラス
20         のコンストラクタを呼び出す(インスタンス化)。
21         Nodeクラスのコンストラクタを見ると、第一引数は"node_name: str"
22         となっている。
23         例えば、第一引数に"talker"と渡せば、このクラスは、talkerという
24         ノード名が割り当てられる。
25         """
```

```
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かよさそう?)
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
108         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。
109         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destroy_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.destroy_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のときにNodeクラスがsubscriptionsというリス
        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。

```
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(package="ex1", node_executable="talker", output="
9                 screen"),
10            Node(package="ex1", node_executable="listener", output="
11                screen"),
12        ]
13    )
```

`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。

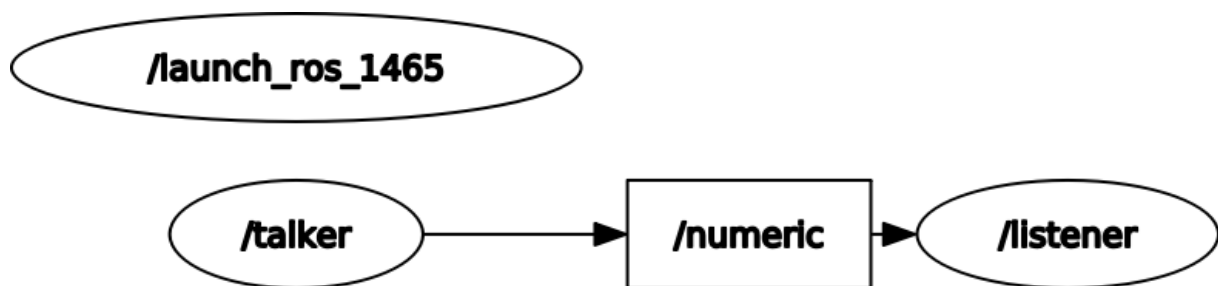
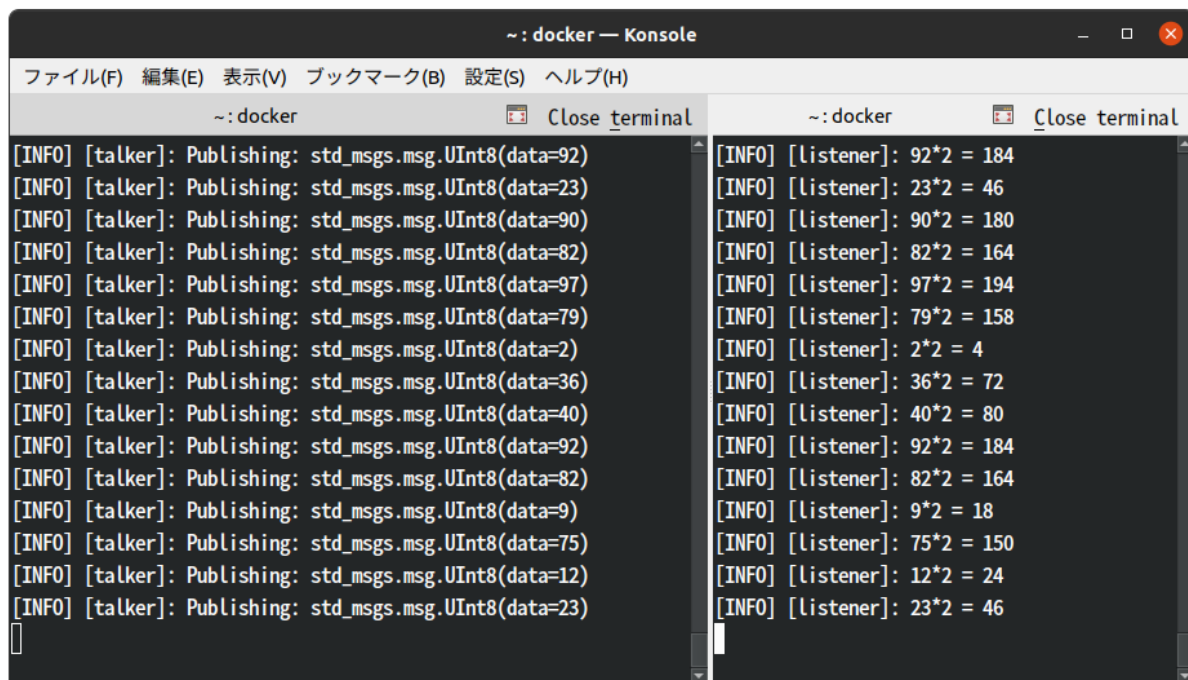


Figure 2: rqt\_graph



```
~: docker — Konsole
ファイル(F) 編集(E) 表示(V) ブックマーク(B) 設定(S) ヘルプ(H)

~: docker      Close terminal      ~: docker      Close terminal

[INFO] [talker]: Publishing: std_msgs.msg.UInt8(data=92)
[INFO] [talker]: Publishing: std_msgs.msg.UInt8(data=23)
[INFO] [talker]: Publishing: std_msgs.msg.UInt8(data=90)
[INFO] [talker]: Publishing: std_msgs.msg.UInt8(data=82)
[INFO] [talker]: Publishing: std_msgs.msg.UInt8(data=97)
[INFO] [talker]: Publishing: std_msgs.msg.UInt8(data=79)
[INFO] [talker]: Publishing: std_msgs.msg.UInt8(data=2)
[INFO] [talker]: Publishing: std_msgs.msg.UInt8(data=36)
[INFO] [talker]: Publishing: std_msgs.msg.UInt8(data=40)
[INFO] [talker]: Publishing: std_msgs.msg.UInt8(data=92)
[INFO] [talker]: Publishing: std_msgs.msg.UInt8(data=82)
[INFO] [talker]: Publishing: std_msgs.msg.UInt8(data=9)
[INFO] [talker]: Publishing: std_msgs.msg.UInt8(data=75)
[INFO] [talker]: Publishing: std_msgs.msg.UInt8(data=12)
[INFO] [talker]: Publishing: std_msgs.msg.UInt8(data=23)

[INFO] [listener]: 92*2 = 184
[INFO] [listener]: 23*2 = 46
[INFO] [listener]: 90*2 = 180
[INFO] [listener]: 82*2 = 164
[INFO] [listener]: 97*2 = 194
[INFO] [listener]: 79*2 = 158
[INFO] [listener]: 2*2 = 4
[INFO] [listener]: 36*2 = 72
[INFO] [listener]: 40*2 = 80
[INFO] [listener]: 92*2 = 184
[INFO] [listener]: 82*2 = 164
[INFO] [listener]: 9*2 = 18
[INFO] [listener]: 75*2 = 150
[INFO] [listener]: 12*2 = 24
[INFO] [listener]: 23*2 = 46
```

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值
  - `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

```
1 $ cd ~/dev_ws/src/  
2 $ ros2 pkg create --build-type ament_cmake ex2_msgs  
3 $ cd ex2_msgs  
4 $ mkdir msg
```

#### 3.3.2 メッセージ型を定義する

msg/Ex2.msg

```
1 int32 x  
2 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。

```
1 <buildtool_depend>rosidl_default_generators</buildtool_depend>  
2 <exec_depend>rosidl_default_runtime</exec_depend>  
3 <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。



```
1 # メッセージ型の生成に必要な依存パッケージ
2 find_package(rosidl_default_generators REQUIRED)
3 # 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
4 rosidl_generate_interfaces(${PROJECT_NAME} "msg/Ex2.msg")
```

### 3.3.5 ビルド

```
1 $ colcon build
```

### 3.3.6 確認

```
1 $ ros2 interface show ex2_msgs/msg/Ex2
2 int32 x
3 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

```
1 $ ros2 pkg create --build-type ament_python ex2
```

### 3.4.2 ex2/talker.py

```
1 import random
2
3 import rclpy
4 from rclpy.node import Node
5
6 from ex2_msgs.msg import Ex2
7
8
9 class Talker(Node):
10     def __init__(self):
11         super().__init__("talker")
12         # Publishする数値の最小値
13         self.declare_parameter("rand_min", 0)
14         # Publishする数値の最大値
15         self.declare_parameter("rand_max", 100)
16         # 1秒間に何回Publishしたいか(周波数)
17         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.destroy_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.
18             x * self.n))
19         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.destroy_node()
26     rclpy.shutdown()
27
28
29 if __name__ == "__main__":
30     main()

```

### 3.4.4 setup.py

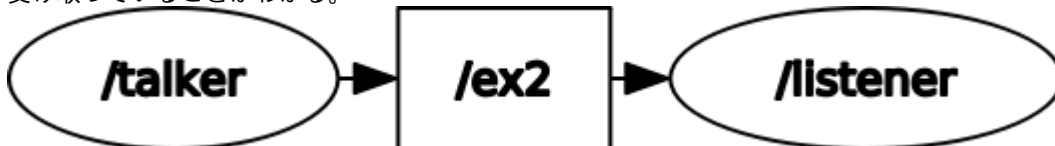
setup.py内の setup(...) に以下を追記する。

```

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

```

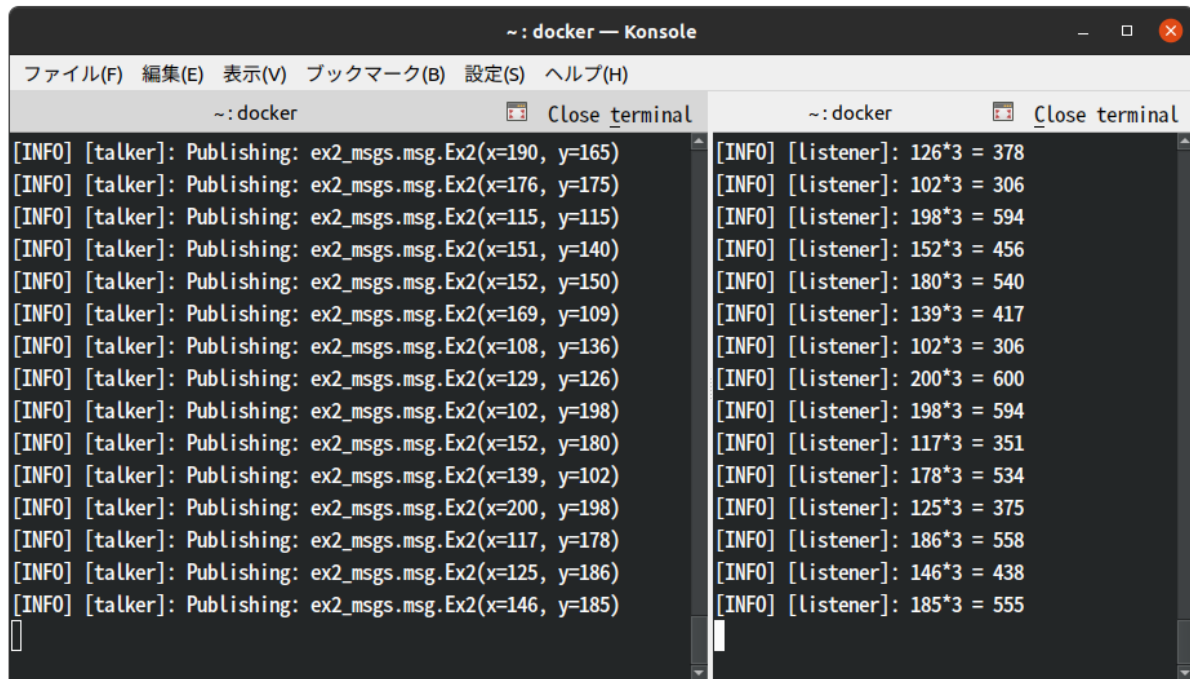


Figure 4: 実行画面

### 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>`

```
1 talker:
2   ros__parameters:
3     rand_min: 100
4     rand_max: 200
5     freq: 2
6
7 listener:
8   ros__parameters:
9     N: 3
```

例えば、上記のような YAML ファイル (ex2\_params.yaml) なら、

```
1 $ ros2 run ex2 talker --ros-args --params-file ./ex2_params.yaml
2 $ 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.
13                           yaml"],
14                # parameters=[{"rand_min": 100, "rand_max": 200, "freq
15                           ": 2}],
16                output="screen",
17            ),
18            Node(
19                package="ex2",
20                node_executable="listener",
```

```
19         parameters=["/home/eisuke/dev_ws/src/ex2/ex2_params.  
20             yaml"],  
21         # parameters=[{"N": 3}],  
22         output="screen",  
23     ),  
24 ]  
25 )
```

### 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

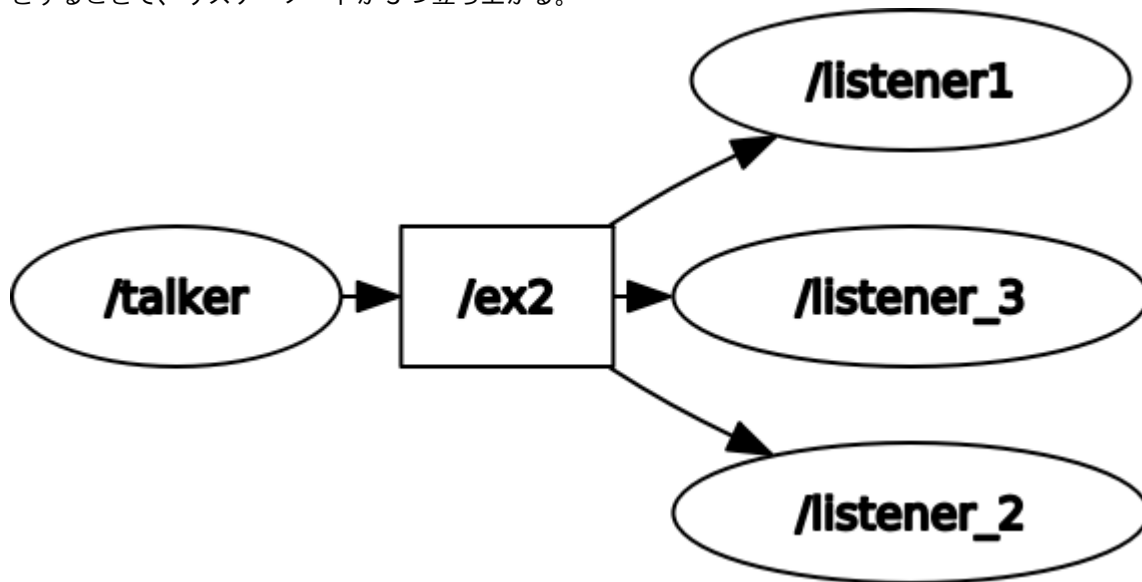
`ros2 run <パッケージ名> <実行名> --ros-args -r __node:=<ノード名>`でノード名を指定した上で、ノードを立ち上げることができる。例えば、talker を

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

とし、listener を

```
1 $ ros2 run ex2 listener --ros-args -p N:=2 -r __node:=listener_1 &  
2 $ ros2 run ex2 listener --ros-args -p N:=3 -r __node:=listener_2 &  
3 $ 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。



```

ros2_practice : docker — Konsole
ファイル(F) 編集(E) 表示(V) ブックマーク(B) 設定(S) ヘルプ(H)
ros2_practice : docker
[INFO] [talker]: Publishing: ex2_msgs.msg.Ex2(x=80, y=26)
[INFO] [talker]: Publishing: ex2_msgs.msg.Ex2(x=82, y=94)
[INFO] [talker]: Publishing: ex2_msgs.msg.Ex2(x=62, y=79)
[INFO] [talker]: Publishing: ex2_msgs.msg.Ex2(x=39, y=59)
[INFO] [talker]: Publishing: ex2_msgs.msg.Ex2(x=47, y=23)
[INFO] [talker]: Publishing: ex2_msgs.msg.Ex2(x=41, y=51)
[INFO] [talker]: Publishing: ex2_msgs.msg.Ex2(x=32, y=87)
[INFO] [talker]: Publishing: ex2_msgs.msg.Ex2(x=27, y=36)
[INFO] [talker]: Publishing: ex2_msgs.msg.Ex2(x=57, y=49)
[INFO] [talker]: Publishing: ex2_msgs.msg.Ex2(x=65, y=36)
[INFO] [talker]: Publishing: ex2_msgs.msg.Ex2(x=63, y=22)
[INFO] [talker]: Publishing: ex2_msgs.msg.Ex2(x=99, y=26)

ros2_practice : docker
[INFO] [listener1]: 32*2 = 64
[INFO] [listener1]: 87*2 = 174
[INFO] [listener1]: 27*2 = 54
[INFO] [listener1]: 36*2 = 72
[INFO] [listener1]: 57*2 = 114
[INFO] [listener1]: 49*2 = 98
[INFO] [listener1]: 65*2 = 130
[INFO] [listener1]: 36*2 = 72
[INFO] [listener1]: 63*2 = 126
[INFO] [listener1]: 22*2 = 44
[INFO] [listener1]: 99*2 = 198
[INFO] [listener1]: 26*2 = 52

ros2_practice : docker
[INFO] [listener_2]: 32*3 = 96
[INFO] [listener_2]: 87*3 = 261
[INFO] [listener_2]: 27*3 = 81
[INFO] [listener_2]: 36*3 = 108
[INFO] [listener_2]: 57*3 = 171
[INFO] [listener_2]: 49*3 = 147
[INFO] [listener_2]: 65*3 = 195
[INFO] [listener_2]: 36*3 = 108
[INFO] [listener_2]: 63*3 = 189
[INFO] [listener_2]: 22*3 = 66
[INFO] [listener_2]: 99*3 = 297
[INFO] [listener_2]: 26*3 = 78

ros2_practice : docker
[INFO] [listener_3]: 32*4 = 128
[INFO] [listener_3]: 87*4 = 348
[INFO] [listener_3]: 27*4 = 108
[INFO] [listener_3]: 36*4 = 144
[INFO] [listener_3]: 57*4 = 228
[INFO] [listener_3]: 49*4 = 196
[INFO] [listener_3]: 65*4 = 260
[INFO] [listener_3]: 36*4 = 144
[INFO] [listener_3]: 63*4 = 252
[INFO] [listener_3]: 22*4 = 88
[INFO] [listener_3]: 99*4 = 396
[INFO] [listener_3]: 26*4 = 104

```

#### 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。

```

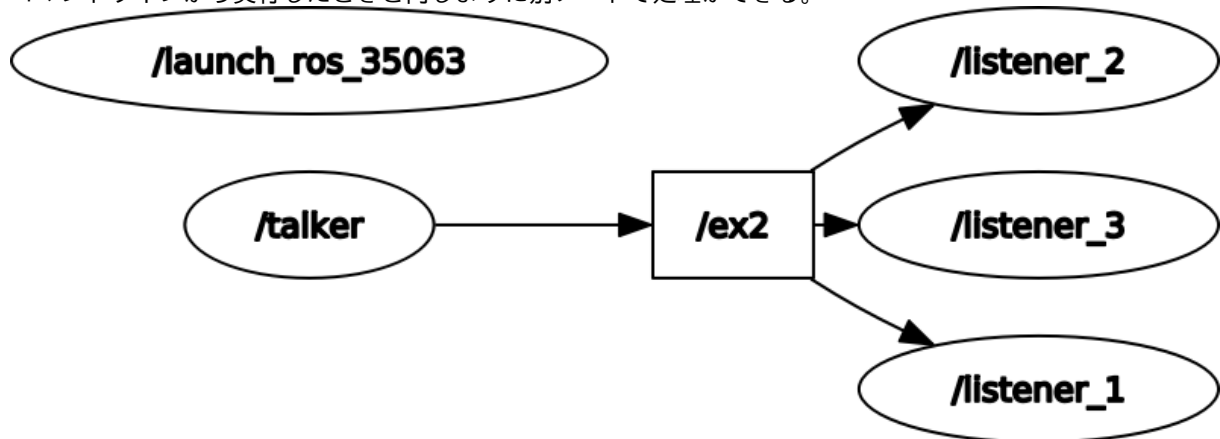
1 from launch_ros.actions import Node
2
3 from launch import LaunchDescription
4
5
6 def generate_launch_description():
7     return LaunchDescription(

```



```
8      [
9          Node(
10             package="ex2",
11             node_executable="talker",
12             parameters=[{"rand_min": 100, "rand_max": 200, "freq":
13                 2}],
14             output="screen",
15         ),
16         Node(
17             package="ex2",
18             node_executable="listener",
19             node_name="listener_1",
20             parameters=[{"N": 2}],
21             output="screen",
22         ),
23         Node(
24             package="ex2",
25             node_executable="listener",
26             node_name="listener_2",
27             parameters=[{"N": 3}],
28             output="screen",
29         ),
30         Node(
31             package="ex2",
32             node_executable="listener",
33             node_name="listener_3",
34             parameters=[{"N": 4}],
35             output="screen",
36         )
37     ]
38 )
```

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/rcldpy 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。

考え方の違いを言うなら、トピック通信は、トピックを介しているので、**Publisher**から見て購読者(購読しているノード)は誰でもいいし、**Subscriber**から見て出版者(出版しているノード)は誰でもいいという考え。しかし、サービス通信では、**Client**はデータを処理してほしい相手**Server**が決まっていて、**Server**はレスポンスする相手**Client**が決まっているという考え方ができる。

つまり、トピック通信は、1つの**Publisher**に対し複数の**Subscriber**がいるので一対多の通信で、サービス通信は、要求から応答までを1つのトランザクションと見なせば、サービス通信は一対一の通信といえる？

## 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

```
1 $ 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、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。

```
1 $ mkdir srv && cd srv
2 $ echo -e \
3 "sensor_msgs/Image image\n\
4 ---\n\
5 int32 x\n\
6 int32 y\n\
7 int32 w\n\
8 int32 h"\
9 > 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。

```
1 find_package(sensor_msgs REQUIRED)
2 find_package(rosidl_default_generators REQUIRED)
3 rosidl_generate_interfaces(${PROJECT_NAME}
4     "srv/Face.srv"
5     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

```
1 <build_depend>rosidl_default_generators</build_depend>
2 <exec_depend>rosidl_default_runtime</exec_depend>
3 <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。

```
1 $ . install/setup.bash
2 $ ros2 interface list | grep face_msgs
3 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

```
1 $ 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

```
1 $ wget https://raw.githubusercontent.com/opencv/opencv/master/data/haarcascades/haarcascade_frontalface_default.xml
```

### 5.7.2 face\_detect/server.py

```
1 import cv2
2 import rclpy
3 from cv_bridge import CvBridge
4 from rclpy.node import Node
5
6 from face_msgs.srv import Face
7
8 """
9 顔検出パッケージのサーバ側
10 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
11 """
12 class Server(Node):
13     def __init__(self):
14         super().__init__("service")
15         self.declare_parameter("cascade_path", "")
16         self.srv = self.create_service(Face, "face_detect", self.callback)
17         self.bridge = CvBridge()
18         cascade_path = self.get_parameter("cascade_path").value
19         self.face_cascade = cv2.CascadeClassifier(cascade_path)
20
21     def callback(self, msg, pos):
22         boxes = [0, 0, 0, 0]
23         image = self.bridge.imgmsg_to_cv2(msg.image, "mono8")
24         detection_boxes = self.face_cascade.detectMultiScale(image)
25         if len(detection_boxes) != 0:
26             boxes = detection_boxes[0]
27             pos.x = int(boxes[0])
28             pos.y = int(boxes[1])
29             pos.w = int(boxes[2])
30             pos.h = int(boxes[3])
31         return pos
32
33
34 def main(args=None):
35     rclpy.init(args=args)
36     server = Server()
37     rclpy.spin(server)
38     server.destroy_node()
39     rclpy.shutdown()
40
41
42 if __name__ == "__main__":
```

```
43     main()
```

### 5.7.3 face\_detect/client.py

```
1  import cv2
2  import rclpy
3  from cv_bridge import CvBridge
4  from rclpy.callback_groups import ReentrantCallbackGroup
5  from rclpy.node import Node
6  from sensor_msgs.msg import Image
7
8  from face_msgs.srv import Face
9
10 """
11 顔検出パッケージのクライアント側
12 要求された周波数・ビデオデバイスを用いて、
13 映像取得・グレースケール変換・サーバ側への送信をする。
14 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、
15 マーカーを引き、映像確認用のトピックに流す。
16 """
17 class Client(Node):
18     def __init__(self):
19         super().__init__("client")
20         self.declare_parameter("freq", 10)
21         self.declare_parameter("video_device", -1)
22
23         freq = self.get_parameter("freq").value
24         video_device = self.get_parameter("video_device").value
25
26         self.cam = cv2.VideoCapture(video_device)
27         self.bridge = CvBridge()
28         cb_group = ReentrantCallbackGroup()
29         self.cli = self.create_client(Face, "face_detect",
30                                     callback_group=cb_group)
31         self.img_pub = self.create_publisher(Image, "image_display",
32                                             10)
33         self.req = Face.Request()
34         timer_period = 1.0 / freq
35         self.timer = self.create_timer(
36             timer_period, self.pub_callback, callback_group=cb_group
37         )
38
39     def pub_callback(self):
40         ret, self.frame = self.cam.read()
41         src_gray = cv2.cvtColor(self.frame, cv2.COLOR_BGR2GRAY)
42         self.req.image = self.bridge.cv2_to_imgmsg(src_gray, "mono8")
43         future = self.cli.call_async(self.req)
44         future.add_done_callback(self.draw)
```

```
44     def draw(self, future):
45         boxes = future.result()
46         print(boxes)
47         cv2.rectangle(
48             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, "
55             bgr8"))
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/
21                            face_detect/haarcascade_frontalface_default.
```

```

21                                     xml"
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")

```

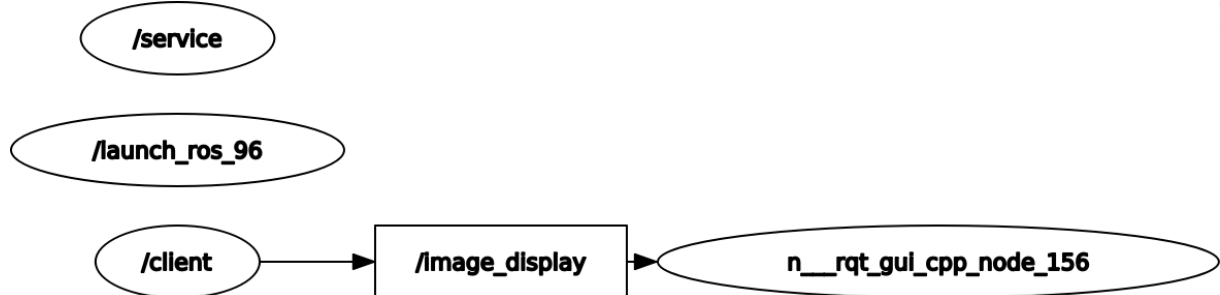
とすればよい。

### 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?)。

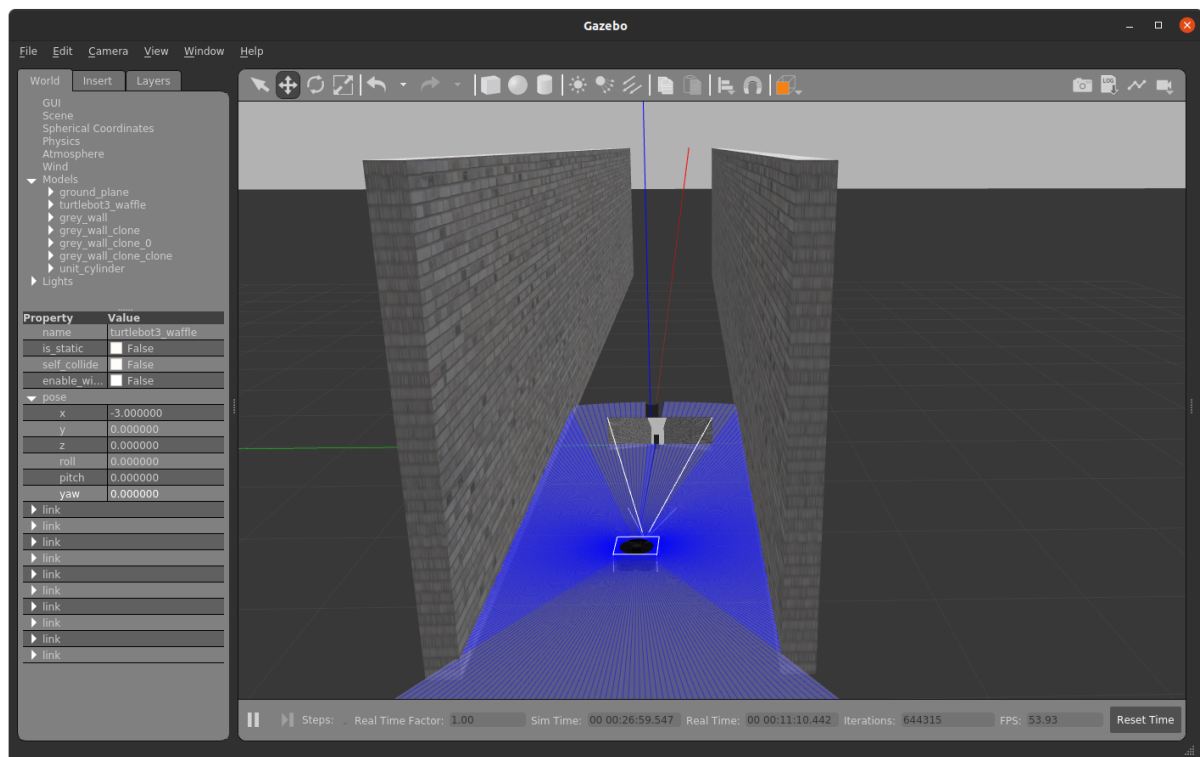


Figure 5: gazebo\_world

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。

```
1 $ ros2 topic list
2 /camera/camera_info
3 /camera/image_raw
4 /clock
5 /cmd_vel
6 /imu
7 /joint_states
8 /odom
9 /parameter_events
10 /rosout
11 /scan
12 /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、

```
1 $ 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。

```
1 $ 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。

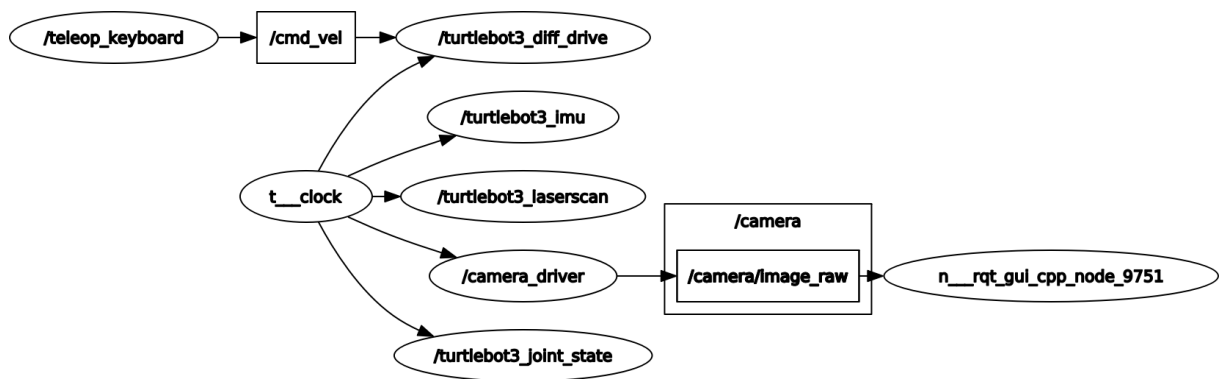


Figure 6: rqt\_graph

## 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。

```

1 $ ros2 topic info /scan
2 Type: sensor_msgs/msg/LaserScan
3 Publisher count: 1
4 Subscriber count: 0
5 $ ros2 topic info /cmd_vel
6 Type: geometry_msgs/msg/Twist
7 Publisher count: 0
8 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。

```

1 $ ros2 interface show sensor_msgs/msg/LaserScan
2 (コメントなど省略済み)
3 std_msgs/Header header
  
```

```
4 float32 angle_min
5 float32 angle_max
6 float32 angle_increment
7 float32 time_increment
8 float32 scan_time
9 float32 range_min
10 float32 range_max
11 float32[] ranges
12 float32[] intensities
```

今回使うメンバは

```
1 float32 range_min
2 float32 range_max
3 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。

```
1 $ ros2 interface show geometry_msgs/msg/Twist
2 # This expresses velocity in free space broken into its linear and
   angular parts.
3
4 Vector3 linear
5 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、

```
1 twist = Twist()
2 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。

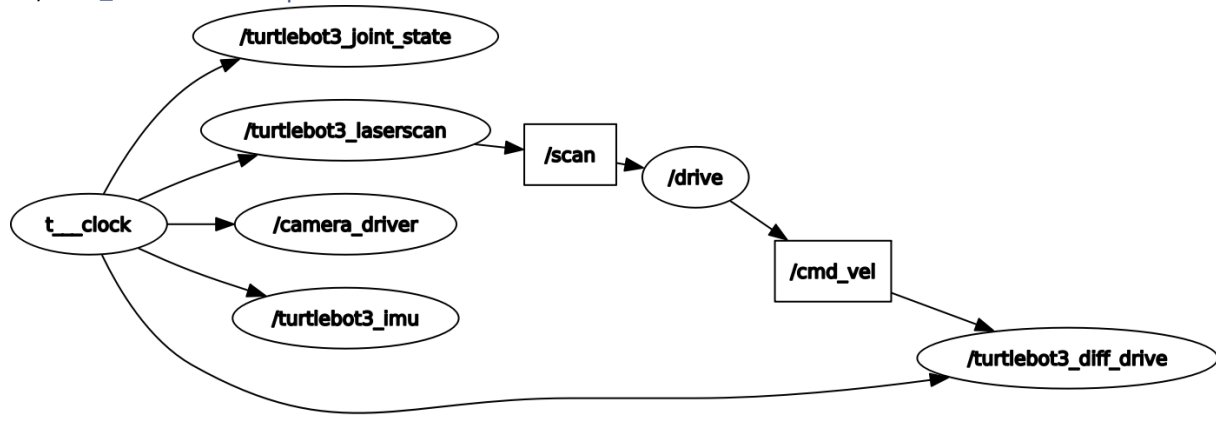
```
1 import numpy
2 import rclpy
3 from geometry_msgs.msg import Twist
4 from rclpy.node import Node
5 from rclpy.qos import qos_profile_sensor_data
6 from sensor_msgs.msg import LaserScan
7
8
9 class Drive(Node):
10     def __init__(self):
11         super().__init__("drive")
12         self.create_subscription(
13             LaserScan, "scan", self.scan_callback,
14             qos_profile_sensor_data
15         )
16         self.pub_twist = self.create_publisher(Twist, "cmd_vel", 10)
17         self.twist = Twist()
18         self.stop_twist = Twist()
19
20     def __del__(self):
21         self.pub_twist.publish(self.stop_twist)
22
23     def scan_callback(self, msg):
24         ranges = numpy.array(msg.ranges)
25         ranges = numpy.where(ranges < msg.range_min, msg.range_max,
26                             ranges)
27         ahead_object_distance = min(min(ranges[0:15]), min(ranges
28                     [345:360]))
29         if ahead_object_distance < 0.5:
30             self.twist.linear.x = 0.0
31             print("stop")
32         else:
33             self.twist.linear.x = 0.3
34             print(
35                 "{:.3f}[m] to the nearest forward obstacle".format(
36                     ahead_object_distance
37                 )
38             )
39         self.pub_twist.publish(self.twist)
40
41     def main(args=None):
42         rclpy.init(args=args)
43         drive = Drive()
```

```

42     rclpy.spin(drive)
43     drive.destroy_node()
44     rclpy.shutdown()
45
46
47 if __name__ == "__main__":
48     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は以下のようなになる。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。

```

1  import time
2
3  import numpy
4  import rclpy
5  from geometry_msgs.msg import Twist
6  from rclpy.node import Node
7  from rclpy.qos import qos_profile_sensor_data
8  from sensor_msgs.msg import LaserScan
9
10
11 class Drive(Node):
12     def __init__(self):
13         super().__init__("drive")

```

```
14     self.create_subscription(
15         LaserScan, "scan", self.scan_callback,
16         qos_profile_sensor_data
17     )
18     self.pub_twist = self.create_publisher(Twist, "cmd_vel", 10)
19     self.twist = Twist()
20     self.stop_twist = Twist()
21
22     def __del__(self):
23         self.pub_twist.publish(self.stop_twist)
24
25     def rotate(self, center_index, argmin):
26         self.twist.linear.x = 0.1
27         # 距離の最小値のインデックスが
28         # 前方中央のインデックスより左にあるなら
29         # 右へ回転する
30         if argmin <= center_index:
31             self.twist.angular.z = -0.2
32         else:
33             self.twist.angular.z = 0.2
34         print("rotate")
35         self.pub_twist.publish(self.twist)
36
37     def scan_callback(self, msg):
38         ranges = numpy.array(msg.ranges)
39         ranges = numpy.where(ranges < msg.range_min, msg.range_max,
40                             ranges)
41         ahead_object_distance_concat = numpy.concatenate(
42             (ranges[345:360], ranges[0:15])
43         )
44         ahead_object_distance = min(ahead_object_distance_concat)
45
46         if ahead_object_distance < 0.5:
47             argmin = numpy.argmin(ahead_object_distance_concat)
48             center_index = len(ahead_object_distance_concat) // 2
49             self.rotate(center_index, argmin)
50             time.sleep(0.5)
51         else:
52             self.twist.linear.x = 0.2
53             self.twist.angular.z = 0.0
54             self.pub_twist.publish(self.twist)
55         print("{:.3f}[m] to the nearest forward obstacle".format(
56             ahead_object_distance))
57
58     def main(args=None):
59         rclpy.init(args=args)
60         drive = Drive()
61         rclpy.spin(drive)
62         drive.destroy_node()
63         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](#)

## 7 まとめ

ROS の使い方について学び基礎を習得した。深く理解するには、`rcl`パッケージの実装を見たり、ROS ノードの状態遷移を知る必要を感じた。

リンク	説明
<a href="#">Managed nodes</a>	ROS ノードのライフサイクル
<a href="#">Executors ROS 2 Documentation</a>	タスク管理
<a href="#">class Executor</a>	タスク管理の実装
<a href="#">class Node</a>	ノードの実装
<a href="#">class Publisher</a>	Publisher の実装
<a href="#">class Subscription</a>	Subscription の実装