

Ros2 - Humbel

Quick Approach

*For those seeking straightforward information,
a practical guide*



That note compilation system is not prepared for directly software community or zero to professional bootcamp guide and books learners. Moreover, I am not a professional directly to involved topic.

That is collection of my notion works in same time coding and earning the knowledge about what I am willing to learn that period. Suppose that situation, I want to share my works like style of e-books and practicable and remainder style basic note collections.

I thought, that is much more useful for beginners and new started to learn involved the topic.

////////////////////////////////////

I examined ROS2 very superficial but effectively what important to developing materials in ROS area with Turtlesim nodes and some basic publisher, /chatter, subscriber areas and basic client-side server architecture.

Tutorial which guided to creation of that work is mentioned in references part. Moreover, readers of that work should examine whether any problems occur in their process.

That is all...
Enjoy then.

Made by [?]



S-1 (1-6)

▼ Author	Madeby [?]
----------	------------

What is node?

Fundamentally, any program that working on ROS communication system. Nodes ile her haltı ros ağı içinde yapabiliyoruz.

Heralde workspace içinde bunu yazdığımız direkt kodlar ile birlikte yapabileceğiz.

First Node

→ROS2 ile beraber run kullanıldığında direkt olarak hazır node yapıları kullanılabilir.
bu node yapıları 2 sistem arasında

- Talker
- Listener

→ bir yerden belli bir yayın yaparken biri de bu yayını almaktadır.

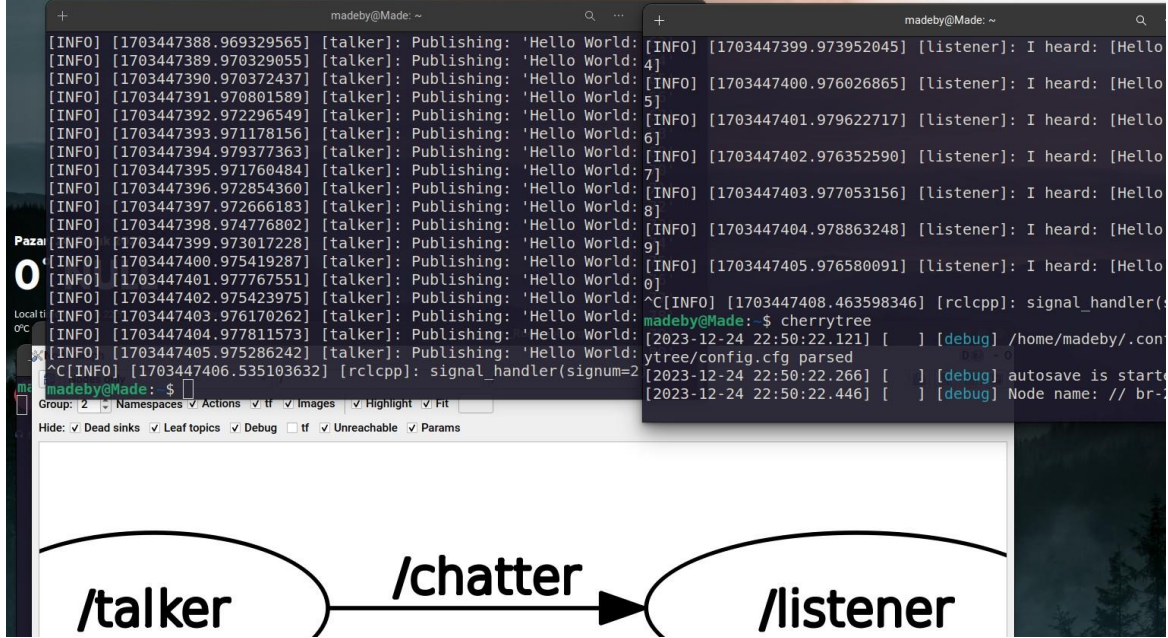
bunu direkt

→ `rqt_graph`

komutu ile hazırda hangi nodun nasıl çalıştığını görebiliriz.

listener olarak hala belli bir source'u bekliyor olacağız, öyle beklediğimiz sürece.

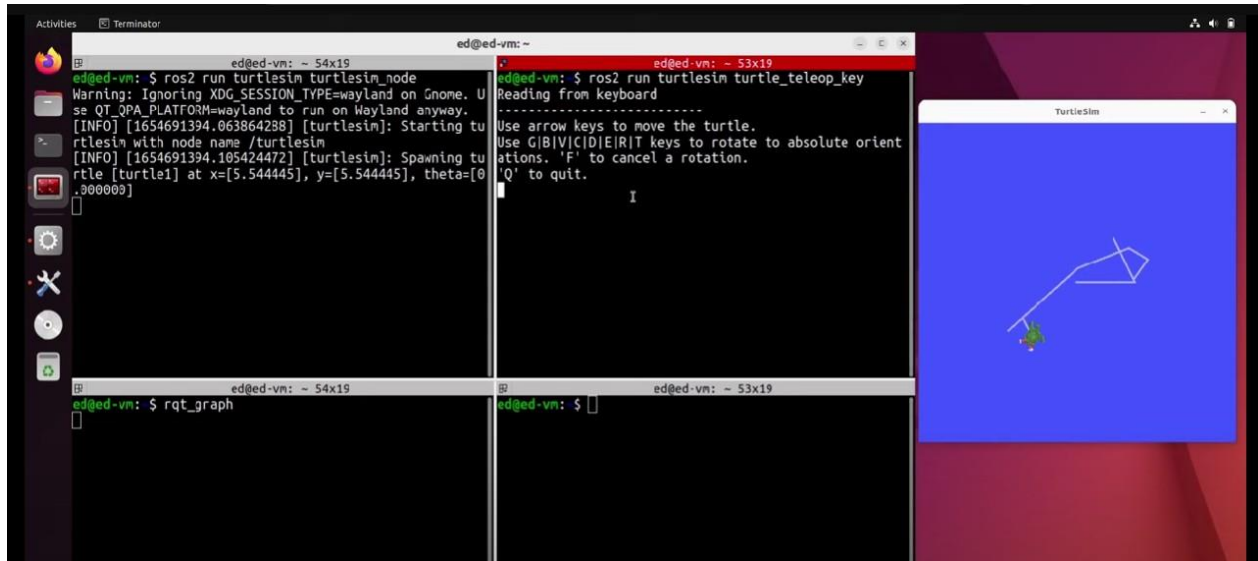
demo_node olarak seçilen dosya herhangi bir dilde olabilir.



→ rqt graph içinden direkt chart olarak node ağı görülebilir.

Turtle sim

→ farklı node türleri var bunlar içerisinde farklı işlemler gerçekleştirilebiliyor.



→ turtle simde npde ayrı çalıştırılıp ayrı şekilde bir teleop_key çalıştırılarak bu görülebiliyor, işlenebiliyor.

Creating workspace

→ ros2 node'larının çalışacağı bir workpace inşa edeceğiz.

bunun öncesinde colcon_common extensionların çalışmasında ve colcon otomatik argümanlarının tanınmasında bashrc dosyasına source ekledik .

```
source /usr/share/colcon_argcomplete/hook/colcon-argcomplete.bash
```

→ otomatik olarak terminal açıldığında colcon argümanlarını direkt olarak çalıştıracak bash dosyasını çalıştıracaktır.

```
,,,,,,,,,,,,,
```

şimdi workspace oluşturma zamanı.

colcon build ile bunu sağladık.

// oto tamamlamayı source vermemiz burada işe yaradı.

```
madeby@Made:~/ros2_ws/install$ ls
COLCON_IGNORE      local_setup.sh      local_setup.zsh     setup.sh
local_setup.bash   local_setup_util_ps1.py  setup.bash          setup.zsh
local_setup.ps1    local_setup_util_sh.py  setup.ps1
madeby@Made:~/ros2_ws/install$ source ~/ros2_ws/install/setup.bash
madeby@Made:~/ros2_ws/install$
```

// ROS için otomatik çalıştırmasını çalıştırmak için source olarak `/.bashrc` dosyasının içerisine ekledim.

source sisteminde colcon sistemi entegre edildiğine göre bu klasör işleme hazır demektir.

Creating python package

// → ros workpsace içine package ekleme zamanı,

colcon build system tool;

ament buişld system dir.

bu noktada python package yüklerken `--build-amend` ile python seçimi yapacağımızda aslında anlam budur.

→ şimdi bu söylemi yapığımıza göre

```
ros2 pkg create ..... --build-type ament-python
```

→ dependencies kısmında oluşturduğumuz proje için extensionları referans vereceğiz.

rclpy;

bu noktada kullanacağımız dependencies.

```
madeby@Made:~/ros2_ws/src$ ros2 pkg create robot_v1_controller --build-type ament
_python --dependencies rclpy
going to create a new package
package name: robot_v1_controller
destination directory: /home/madeby/ros2_ws/src
package format: 3
version: 0.0.0
description: TODO: Package description
maintainer: ['madeby <ta.aydin01@gmail.com>']
licenses: ['TODO: License declaration']
build type: ament_python
```

→ böylelikle Python için ros2 içinde node yapısı kurulmuş oldu.

paket işlemlerini buradan ilerleteceğiz.

→ birçok extra file ları birbirine bağlama ve conditions ekleyerek birbirleri üzerinde bir bağlam oluşturulabilir. Bunu direkt dosya temelinde yapıyor olabilmemiz de bir artı.

```
<depend>rclpy</depend>
```

→ buraya eklediğimiz her file aslında geriye bağlı olarak buna bağlı olduğunu belirtiyor.

bu her ROS2 paketinde olan bir xml dosyasıdır. Prosedürler ve birbirlerine bağlamaları bu dosya içinde cofigure etme şansımız vardır.

→ oluşturduğumuz package ile aynı isimde bir init dosyasıda vardır.

// bu direkt dosyanın kendisi ile aynı isimde olma zorundallığı var. (init sonuçta)

şimdi oluşturulmuş workspace direkt olarak çalıştırabiliriz.

```
madeby@Made:~/ros2_ws$ colcon build
Starting >>> robot_v1_controller
Finished <<< robot_v1_controller [2.08s]

Summary: 1 package finished [3.71s]
madeby@Made:~/ros2_ws$
```

- Colcon build ile bunu sağlıyoruz.

→ bunun öncesinde setuptools adlı pip3 içerisindeki ilgili kütüphanenin sürümünü değiştirmek gerekiyor. bunu direkt

~ pip3 install setuptools== <sürümmodeli> yaparak sağlayabiliyoruz.

- bunu yapmadığımızda hata ile karşılaştık. (4. video içinde de bundan bahsediliyordu zaten.)
-

Creating ROS2 Nodes with C++ or Python

→ buradan itibaren işler iyice anlamlı hale geliyor.

→ biz node dosyalarını bahsettiğimiz `__init__` kısmı yerinde oluşturacağız. Buda zaten ana dosya ile aynı yerde bulunmaktadır. // oluşturulan dosyaya execute yetkisini vermeyi unutmamak lazımdır.

```
madeby@Made:~/ros2_ws/src/robot_v1_controller/robot_v1_controller$ ls
first_node.py  __init__.py
madeby@Made:~/ros2_ws/src/robot_v1_controller/robot_v1_controller$ chmod 7 first_node.py
madeby@Made:~/ros2_ws/src/robot_v1_controller/robot_v1_controller$ ls
first_node.py  __init__.py
madeby@Made:~/ros2_ws/src/robot_v1_controller/robot_v1_controller$
```

→ direkt her yetkiyi verdim. artık executable yetkisi sağlanıyor.

⇒ önemli olan birkaç nokta var bu konuda;

1. Node dosyalarını yazarken implementasyonda ilk import edilecek şey ROS için olan python kütüphanesidir.
 - bu dependencies kısmında da verdiğimiz `rclpy` simli kütüphane. // bunu her zaman ekleyeceğiz.
-

→ temel olarak bir nodeun nasıl yazıldığına buradaki görelden bakabiliriz.

```

my_robot_controller > my_robot_controller > my_first_node.py > main
1  #!/usr/bin/env python3
2  import rclpy
3
4  def main(args=None):
5      rclpy.init(args=args)
6
7      #
8
9      rclpy.shutdown()
10
11 if __name__ == '__main__':
12     main()

```

→ bir main fonksiyonu içinde rclpy init ile node başlatılıyor ve shutdown ile sonlandırılıyor.

yani ana görev bu iki kodun arasına gelecektir.

// if sorgusunun sebebini direkt anlayamamda direkt terminalden çalıştırmalarda kolaylık sağlayabilecek bir şey diye hatırlıyorum.

args olarak dosya açılırken bir şey vermedik, terminalden direkt bilgi veya belirli bir dosyaya referans ile de çalıştırılabilir. (ileiki uygulamalarda değinilebilir.)

,,,,,,,,,,,,,,,,,,,,,

Node un yazımında dikkat edilecek noktalardan biri class oriented olarak ilerlemesi. birden fazla node aynı script içinde işlenebilir demek oluyor.

ama kendine göre fonksiyonları bi tık daha özelleşmiş diyebiliriz.

```

1  #!/usr/bin/env python3
2
3  import rclpy
4  from rclpy.node import Node
5
6  #-> node extension olarak inceleme kütüphanesi
7
8
9
10
11 # nodelar direkt class oriented object olarak alınacaktır.
12 class node_1(Node):
13     def __init__(self):
14         super().__init__("First_Node")
15         # super ile bu class objesinin kendisine ismini verdik. // First node
16         self.get_logger().info("Goodbye ROS2!")
17
18
19 def main(args = None):
20     rclpy.init(args=args)
21     node = node_1()
22     # ilgili node burada çağrılacaktır. Çalışma alanı burasıdır.
23
24     rclpy.shutdown()
25
26 if __name__ == '__main__':
27     main()
28

```

- !!! - super() classı içinde Nodun ismini yazarken arada boşluk karakteri bırakılmaz.
 - Harici olarak yazım baya kritik, hele de rosun içindeki kendi kütüphanesinin çalışmalarında daha anlamlı bu.

⇒ Spin functionality

→ Node un yine aktif olarak çalışmasına devam etmesini sağlamanın yollarından biri spin metodunu kullanmaktır.

bu sayede shutdown ile sonlanmadan önce hala canlı bir kod gibi çalışmaya devam edebilir.

```

madeby@Made:~/ros2_ws/src/robot_v1_controller/robot_v1_controller$ ./first_node.py
[INFO] [1705746241.241842784] [First_Node]: Goodbye ROS2!

```

// spin de main içinde atadığımız class objesini argüman olarak vermeyi unutmamız lazım.

How to install the Node?

```
edged-vn: /ros2_ws/src/my_robot_controller/my_robot_controller$  
edged-vn: /ros2_ws/src$ code .  
edged-vn: /ros2_ws/src$ cd ..  
edged-vn: /ros2_ws$ ls  
build testall log ws  
edged-vn: /ros2_ws$ colcon build  
Starting >>> my_robot_controller  
Finished <<< my_robot_controller [1.40s]  
  
Summary: 1 package finished [1.68s]  
edged-vn: /ros2_ws$ source ~/.bashrc  
edged-vn: /ros2_ws$ ros2 run my_robot_controller  
--prefix test_node  
edged-vn: /ros2_ws$ ros2 run my_robot_controller test_node  
[INFO] [1654761667.296578747] [First_node]: Hello from ROS2
```

→ Node installation, ilk derslerdeki gibi ros run yaparak kendi Nodumuzun çağırma işlemidir.

bunun için setup içinde ve yazdığımız scriptteki hangi metodun içindeki kısma referans verdiğimizizi vs yazmamız gereklidir.

```
entry_points={  
    'console_scripts': [  
        "test_node_1 = robot_v1_controller.First_Node:main"  
    ],  
},
```


kendi ana fonksiyonumu kodun içinde hangisi olduğunu söylüyorum.

sonrasında colcon build yapıp, source ile ilk başta bashrc içinde tanımladığımız source kaynaklarını çağırıp, ros2 run <scriptin olduğu klasör> <node name >

```
madeby@Made:~/ros2_ws$ colcon build
Starting >>> robot_v1_controller
Finished <<< robot_v1_controller [1.02s]

Summary: 1 package finished [1.83s]
madeby@Made:~/ros2_ws$ source ~/.bashrc
madeby@Made:~/ros2_ws$ ros2 run robot_v1_controller test_node_1
[INFO] [1705747248.724536681] [First Node]: Goodbye ROS2!
```

→ dosya adını ilgili setup.py uzantılı dosya içerisinde duüzgün yazmak önemli yoksa *not founded module* hatası veriyor.

 !!! - colcon buid --symlink-install

→ bu şekilde build yaptığımızda bir daha extradan değişiklik yaptığımız zaman vs yeniden yeniden sürekli bir şey yapmaya ihtiyacımız olmadan, nodu çağırdığımızda yeni haliyle çalışacaktır.

→ timer usage

```

class MyNode(Node):
    def __init__(self):
        super().__init__("first_node")
        self.create_timer(1.0, self.timer_callback)

    def timer_callback(self):
        self.get_logger().info("Hello")

def main(args=None):
    rclpy.init(args=args)
    node = MyNode()
    rclpy.spin(node)
    rclpy.shutdown()

```

→ time rve spin kombinasyonu ile bir loop sistemi oluşturup saniyede birer kere ilgili fonksyonu (timer callback()) i çağırarak şekilde bir node yazılmış.

ve bunu spin ile sürekli çağırma yapabilirdim için, create_time rile sürekli çalışan bir sistem elde edebiliyorum.

// real time işlemlerinde Arduino'daki loop sisteminin aynısı gibi düşünülebilir. ama daha ayrıntılı ve müdahaleye açık.

```

[ros2run]: Interrupt
madeby@Made:~/ros2_ws$ ros2 run robot_v1_controller test_node_1
[INFO] [1705748228.163129712] [First_Node]: Goodbye ROS2!0
[INFO] [1705748228.862995683] [First_Node]: Goodbye ROS2!1
[INFO] [1705748229.551135265] [First_Node]: Goodbye ROS2!2
[INFO] [1705748230.250722605] [First_Node]: Goodbye ROS2!3
[INFO] [1705748230.951372050] [First_Node]: Goodbye ROS2!4
[INFO] [1705748231.651354961] [First_Node]: Goodbye ROS2!5
[INFO] [1705748232.350873840] [First_Node]: Goodbye ROS2!6
[INFO] [1705748233.051958128] [First_Node]: Goodbye ROS2!7
[INFO] [1705748233.752782158] [First_Node]: Goodbye ROS2!8
[INFO] [1705748234.450773287] [First_Node]: Goodbye ROS2!9
[INFO] [1705748235.151238517] [First_Node]: Goodbye ROS2!10
[INFO] [1705748235.851736775] [First_Node]: Goodbye ROS2!11
[INFO] [1705748236.553597785] [First_Node]: Goodbye ROS2!12
[INFO] [1705748237.251968513] [First_Node]: Goodbye ROS2!13
[INFO] [1705748237.953767671] [First_Node]: Goodbye ROS2!14

```

→ bu şekilde bir döngü sistemi kurulabiliyor.

```

# nodelar direkt class oriented object olarak alınacaktır.
class node_1(Node):
    def __init__(self):
        super().__init__("First_Node")
        # super ile bu class obejsinin kendisine ismini verdik. // First node

        # create time ve spin kombinasyonu ile de sürekli döngüye alabileceğimiz bir si
        self.counter_1 = 0

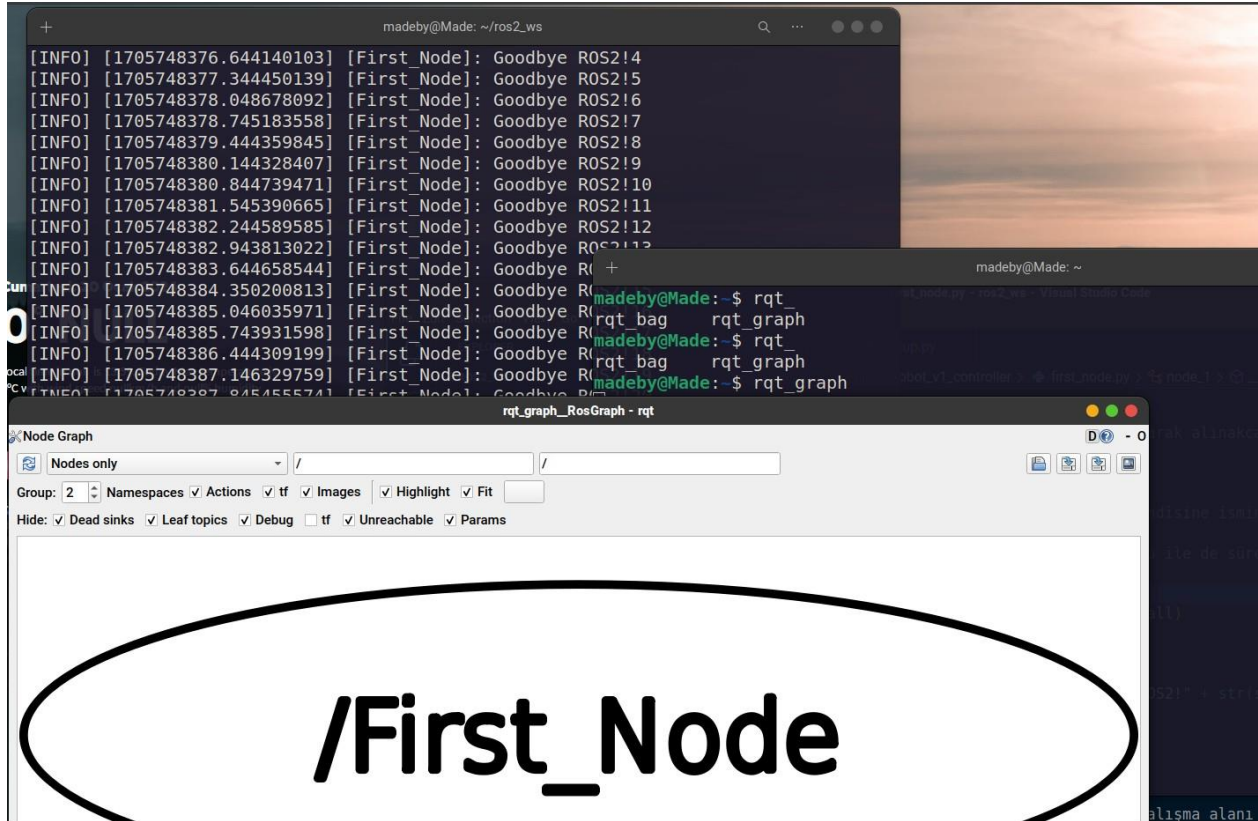
        self.create_timer(0.7, self.get_call)

    def get_call(self):
        self.get_logger().info("Goodbye ROS2!" + str(self.counter_1))
        self.counter_1 += 1

```

- bahsettiğimiz gibi spin kullanarak bu create olayını etkili şekilde gerçekleştirebiliyor ve sürekli aslında nodu aktif olarak işleyişini izleyebiliyoruz.

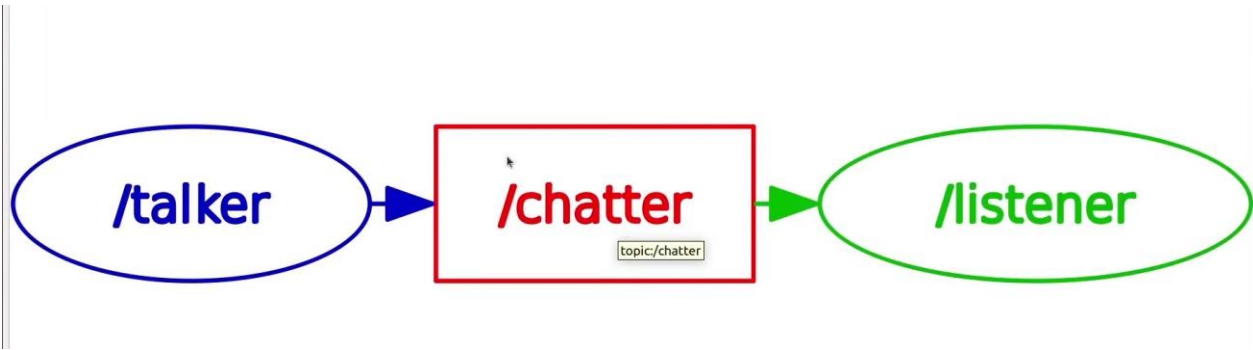
bunun takibini ilk zaman verdiğimiz node takip eddici chartda // rqt_graph // ile takibi sağlanabilir.



super() ile isim verdiğimiz nodun çağırılmasını burada görüyoruz. ve aktif çalışma sürecindeki nodeların takibi burada bu şekilde gerçekleşmektedir.

What is ROS topic?

→ how to nodes communicate each other? // bu sorunun cevabı aranacak.



→ aslında öncesidne rqt_graph içinde gördüğümüz mevzusudur.

Talker yani node u chatter içine publish eden; bir yayın yapar bu da /chatter ortamınadır.

Listener yani subscriber bu chattera katılır ve datayı almaya başlar.

mevzu bu şekilde çalışmaktadır.

```
ed@ed-vm: $ rqt_graph
ed@ed-vm: $ ros2 topic list
/chatter
/parameter_events
/rosout
ed@ed-vm: $ ros2 topic info /chatter
Type: std_msgs/msg/String
Publisher count: 1
Subscription count: 1
ed@ed-vm: $ ros2 interface show std_msgs/msg/String
# This was originally provided as an example message.
# It is deprecated as of Foxy
# It is recommended to create your own semantically meaningful message.
# However if you would like to continue using this please use the equivalent in example_msgs.

string data
ed@ed-vm: $
```

→ chatter ile ilgili genel bilgiye, hangi nodun ne tipte bir çıktı verdiğini vs interface show yardımıyla öğrenebiliyoruz.

topic info /chatter dediğimizde de direkt olarak ilgili topicin bilgilerini, tipini ve içerideki listener ve talker rollerinde kaç tane olduğunu vs görebiliyoruz.

interface show ile de type olarak verilen ifadeyi inceleyebiliyor hakkında bilgi alabiliyoruz.



ros2 topic echo /chatter

yaptığımızda string olan data tipini bir başka komutla direkt dinlemeye başlıyoruz.

Yani /chatter içine biri daha aktif olmuş gibi düşünülebilir.

bu durumda şu oluşuyor :

→ 2 subscriber in area of chatter, // görseldeki gibi

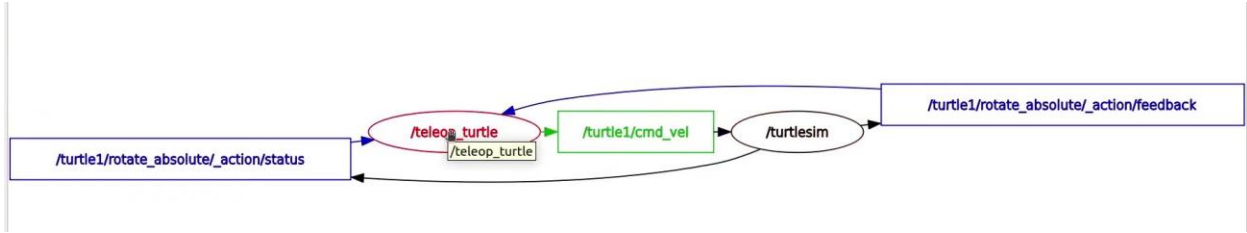
bunu bir node üzerinden gösterildi. Tek fark bunu topic olarak ifade ediliyor olması.

// ki anlamlıdır; bir talkerın ve bir listenerin olduğu bir ortamda konuşulan bir şey olması gerekir.

bir konu olmalıdır yani...

buna dayanarak sistem oluşur.

⇒ Turtle sim örneği,



Şemadan şunu anlıyoruz, Node lar kendi aralarında haberleşebilir ve bağlantıya geçebilirler. ve birden fazla ortamda topic olabilir.

Biri başka bir topğin dinleyecisi iken aynı zamanda başka bir topğin talkeri olabilir.

Bu şekilde close loop mekanizmasında çalışabilir.

ve aynı zamanda ortam değişkenleri ve nasıl iletildiğine dair ayrıntılı info *interface show* komutu ile öğrenilebilir.

// Pratik yapmamız gereken nokta topic info komutunun çokça kullanıp nodun nasıl çalıştığını iyice anlamak diyebiliriz.

```

madeby@Made: ~$ ros2 topic list
/parameter_events
/rosout
/turtle1/cmd_vel
/turtle1/color_sensor
/turtle1/pose
madeby@Made: ~$ ros2 topic info /turtle1/cmd_vel
Type: geometry_msgs/msg/Twist
Publisher count: 1
Subscription count: 1
madeby@Made: ~$ ros2 interface show geometry_msgs/msg/Twist
# This expresses velocity in free space broken into its linear and angular
Vector3 linear
float64 x
float64 y
float64 z
Vector3 angular
float64 x
float64 y
float64 z
madeby@Made: ~$

```

→ topic list yapıp aktif topikleri bulup sonrasında da istediğimiz topic hakkında detaylı bilgiyi yakalayabiliriz.

Topic prensibini ve aralarındaki hiyerarşiyi anlatan baya iyi bir örnek bu.

direkt yorum olarak da gerçekleştirilen uygulamanın detaylı bilgisini yakalama şansımız oluyor görüldüğü üzere.

→ bundan sonrasında aralarında haberleşen Nodeların yazımları üzerine değinilecek.

Yeni bir partta ele alalım bunu.



S-2 (7-11)

▼ Author	Madeby [?]
----------	------------

Creating talker Node

→ bir draw circle Node u üzerinden ilerliyor.

ve önceki Node yazımıyla aynı, farklılık publisher eklentisi noktasında başlayacak ama, XML dosyasının içine kullanılan kütüphaneleri dependencies olarak eklemek önemlidir.

bu yüzden kullandığımız her kütüphaneyi dependencies kısmına eklemekte yarar vardır.

```
<depend>roscpp</depend>  
<depend>geometry_msgs</depend>  
<depend>turtlesim</depend>
```

Olay aslında tamamen yayınlanan topiğin tipine ve ismine dayanıyor.



→ çünkü görselden anlaşılacağı üzere kendi publisherımız direkt olarak topiğin kendisine müdahale edecek ve aynı tipte bir ifade döndürerek turtle sim içerisinde harekete manipülede bulunacaktır.

''''''''

```
draw_circle_node(Node):
    buraya dikkat, burada bir node çalıştırıyoruz, bu noktada inheritance verirken bunu Node ana sınıfından vermemiz öneliydi.
    def __init__(self):
        super.__init__("drawing a circle")
        self.cmd_vel_pub = self.create_publisher(Twist, "/turtle1/cmd_vel", 10)
        # burada sistem komutu ve bir publisher yayını yapılması için gerekli parametreler ve istekler var.
        # publisher isteklerinde önce type, sonrasında ilgili topiğin adı sonrasında bir alan numarası // 10 ideal dendi.

        self.timer_ = self.create_timer(0.5, self.circle_act)
        # timer sistemini ayarladık, bir değişkene atanmasında problem yok, sonuçta create timerdan referans alıyoruz.
        self.get_logger().info("Drawing Node tool has been started")
        --> buradaki tüm çalışmaların hepsi init içinde gerçekleştirildi. self değerinde direkt olarak ilgili node objesinin kendisine atar

    def circle_act(self):
        msg = Twist
        msg.linear.x = 2.0
        msg.angular.z = 1.0
        self.cmd_vel_pub.publish(msg)
        # self.cmd_vel_pub değişkeni create publisher dan üretildi, tip ve topic ismi belli yani.
        # bunu publish ederken içeriğindeki mesajda circle actin içinde verdik ki turtle'ne ne yapacağını söylesin.
        # bunun detaylarını, mesaj içeriğinde ne olmasının gerektiğini interface show kısmından öğrendik ve msg içeriğinde belirttik.
        # msg değişkenini oluştururken ne tipte olduğunu öğrenmiştik, direkt bu değişkene ne olduğunu fonksiyonun ilk satırında söyledik.
```

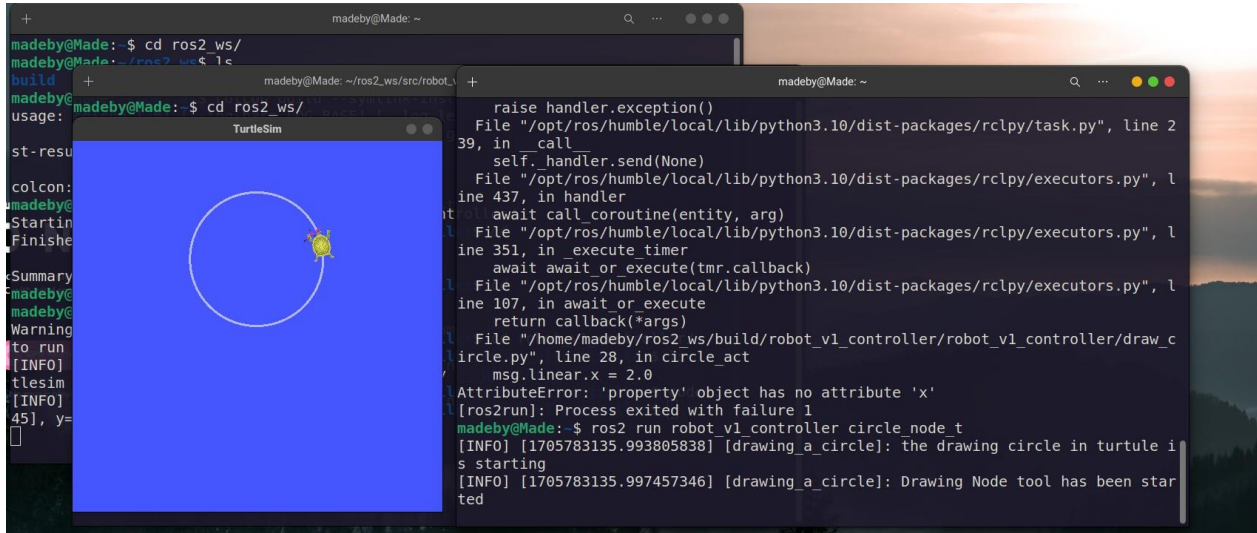
- Bu node hakkında detaylı bir işleyiş yaptım kağıt üzerinde ama yine de nasıl işlediğini nasıl adım adım ilerlediğini gözlemlemek önemlidir. Özellikle
 - spin ve createtime ile olan birlikte kullanım

- Publish mekanizmasının argümanları ve nasıl yazılacağı.

→ Arada sırada geçerken buralara bakmakta kati yarar vardır.

→ Önceisdeki gibi benzer şekilde node çalıştırıldığı zaman dirket listener durumdaki turtle emir gidecektir. ve çalışmaya başlayacaktır.

Bu da sonucu...



Writing a subscriber

→ talker yazdığımıza göre sırada bir listener var. ama öncesinde basit bir node şablonu;

```
#!/usr/bin/env python3
import rclpy
from rclpy.node import Node

class PoseSubscriberNode(Node):
    def __init__(self):
        super().__init__("pose_subscriber")

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

    rclpy.shutdown()
```

hatırlarsak bir iletişimde bazı şeyler ihtiyaçtır.

- type (ros2 topic info <topicname> ile öğrenilebilir)
- topiğin adı ((ros2 topic info ile aktif topicler öğrenilebilir))
- ve bunun içeriğindeki değişkenlerin türleri (ros2 interface show <topicname> ile öğrenilebilir.)

→ bunları elde ettikten sonra subscriber yazılabilir.

publisherdan farklı olarak subscription bir fonksiyon// callback beklemektedir.

buna dayanarak turtlesimdeki pose değerlerini döndüren bir subscriber deneyelim. //

v-8


```
# ilgili topic direkt turtlesim içinde var. biz burada topic info yaparak son kısımdaki yere bakıyoruz ve alıyoruz.
# // ros2 interface show message type > bize msg içeriği ile ilgili bilgi verecektir. Bu bilgilere dayanarak mesaj içeriğini ineleye

# nodelar direkt class oriented object olarak alınacaktır.
class pose_subscription(Node):
    def __init__(self):
        super().__init__("Turtlesim_pose_estimation")
        self.pose_sub = self.create_subscription(Pose, "/turtle1/pose", self.callback, 10)
        # init içinde tanımlandığı için direkt self tanımını yapmayı unutmamak lazım // fonksyonlar içinde dahil.

    def callback(self, msg: Pose):
        # msg tanımındaki pose anlamında dikkat. direkt olarak message ögesinin bir pose olduğunu ifade etmiş olduk.
        # -> bir önceki örenkte de direkt önce msg = twist olarka ifade etmiştik, aynısını yine yapabilirdik.
        self.get_logger().info("X- " + str(msg.x) + " " + str(msg.y) + "Y- " + str(msg.y))
        # direkt message ögesinin kendisini yazdırabiliyoruz.

def main(args = None):
    rclpy.init(args=args)
    node_2 = pose_subscription()

    rclpy.spin(node_2)

    rclpy.shutdown()

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

→ subscribe yaparken callback olarak kullandığımız fonksiyon bu anlamda önemlidir. subscription içeiisinde nasıl yer aldığına bakarsak direkt self.function olarak çekilip mesajın içeriğinden ilgili bilgi okunmuş oldu.

spin içerisinde bu node 'u bulundurmadaydık eğer çalışmamış olacaktı.

bu anlamda bu bilgi alınımını sürekli olarak çalıştırıyor olmak önemli bir durumdur.

yorum satırlarında yazarken önemli gördüklerimi bulundurdum tekrar ederken bakarım.

Creating closed-loop system with publisher

3 kritik videonun ilkidir.

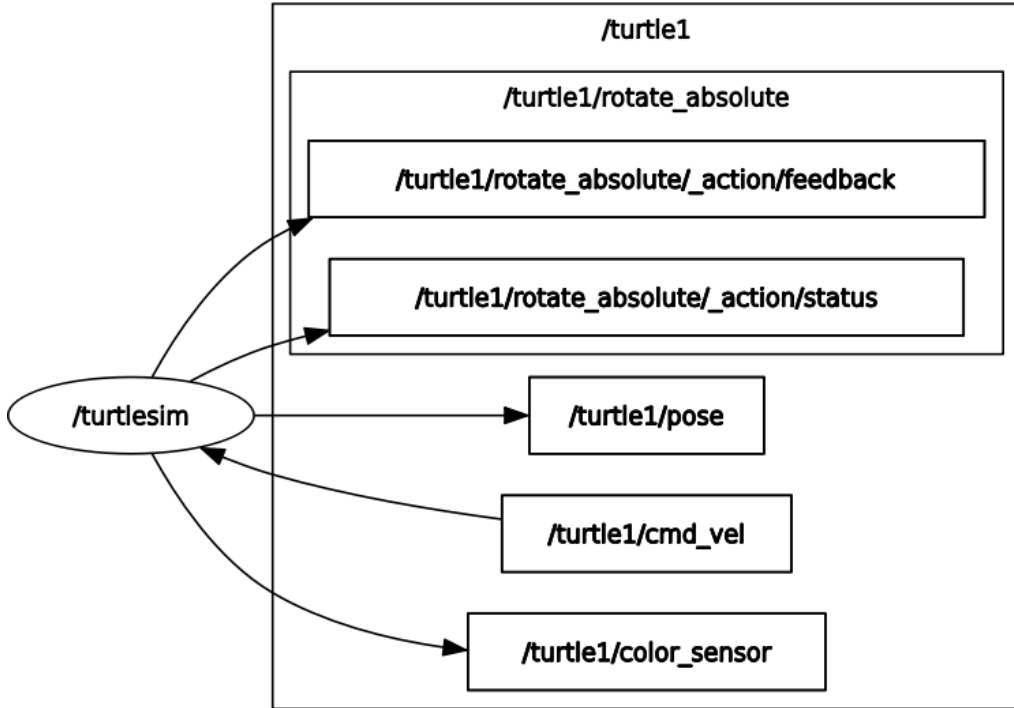
- tek bir node içerisinde hem publisher hem de subscriber bulunabilir ve ayarlanabilir.

→ rqt_graphdan baktığımızda bir nodeun istemcilerine (topiclerine) → sırasında ve hangisinin girdi, hangisinin çıktı olduğuna bağlı olarak bir sistem kurabiliriz.

ki bu zamana kadar yaptığımız ayrı ayrı bunları incelemektir, şimdi bunları birleştirmeye geldik.

Node ortamını hazırlayalım.

rqt_graph ile nodun topiclerinin nasıl sistemi olduğu bakalım.



→ görüldüğü üzere cmd_vel publisher, diğer topicler subscriber modunda.

buna dayanarak pose ve cmd_vel e bağlı olan bir kapalı loop kuracağız.

→ ana nodu inşa ettikten sonra;

```
#!/usr/bin/env python3

import rclpy

from rclpy.node import Node

class turtle_node(Node):
    def __init__(self):
        super().__init__("turtle_controller")
        self.get_logger().info("The turtle_controller Node has been started")
        # Nodun ne olduğu ile ilgili detayların hepsi kendi init sınıfının içinde tanımlanmalıdır.

def main(args=None):
    rclpy.init(args=args)
    node_1 = turtle_node()
    rclpy.spin(node_1)

    rclpy.shutdown()

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

→ ana algoritma mantığı şöyleki subscription yaptıktan sonra elimizdeki pose değerine göre bir çalışma gerçekleştirmeliyiz. obje belirli bir noktaya kadar istenilen bir profiledeki hareketi gerçekleştirecek, farklı bir pose değerine ulaşıldığında da başka bir mekanizma çalıştırılacaktır.

→ yani subscriptions sonrasında bir publisher çalışacak ve hareketi tanımlayacaktır.

```

super().__init__(turtle_controller)
self.cmd_vel_publisher_ = self.create_publisher(
    Twist, "/turtle1/cmd_vel", 10)
self.pose_subscriber_ = self.create_subscription(
    Pose, "/turtle1/pose", self.pose_callback, 10)
self.get_logger().info("Turtle controller has been started.")

def pose_callback(self, pose: Pose):
    cmd = Twist()
    cmd.linear.x = 5.0
    cmd.angular.z = 0.0

```

→ callback ile subscriptionun sağlandığı süre boyunca bir metod çalıştırılabilir. ve biz bunun içinde komutumuzu tanımlayıp bunu publish edersek eğer // bu kısımda yok ama aşağıda bir publish var callback içinde.// biz hareketi sağlamaya başlayabileceğimiz anlamına geliyor.

ve hareketi pose koordinatlarına göre bir koşul sistemi oluşturabileceğimiz anlamına geliyor.

```

from rclpy.node import Node
from turtlesim.msg import Pose
from geometry_msgs.msg import Twist

class turtle_node(Node):
    def __init__(self):
        super().__init__("turtle_node1")
        self.get_logger().info("The turtle_controller Node has been started")
        # Nodun ne olduğu ile ilgili detayların hepsi kendi init sınıfının içinde tanımlanmalıdır.

        #-----
        # hem subscription hemde publisher eklenmelidir. önceki kodlarla aynı.
        self.pose_sub = self.create_subscription(Pose, "/turtle1/pose", self.callback, 10)
        self.vel_pub = self.create_publisher(Twist, "/turtle1/cmd_vel", 10)

    def callback(self, position: Pose):
        # burada bir mesaj oluşturacağız ve pose subscriptionu sağlanana kadar kapalı bir loop gibi çalışmaya devam edecektir.
        msg = Twist()
        msg.angular.z = 0.0
        msg.linear.x = 2.0
        self.vel_pub.publish(msg)

def main(args=None):
    rclpy.init(args=args)
    node_1 = turtle_node()
    rclpy.spin(node_1)

    rclpy.shutdown()

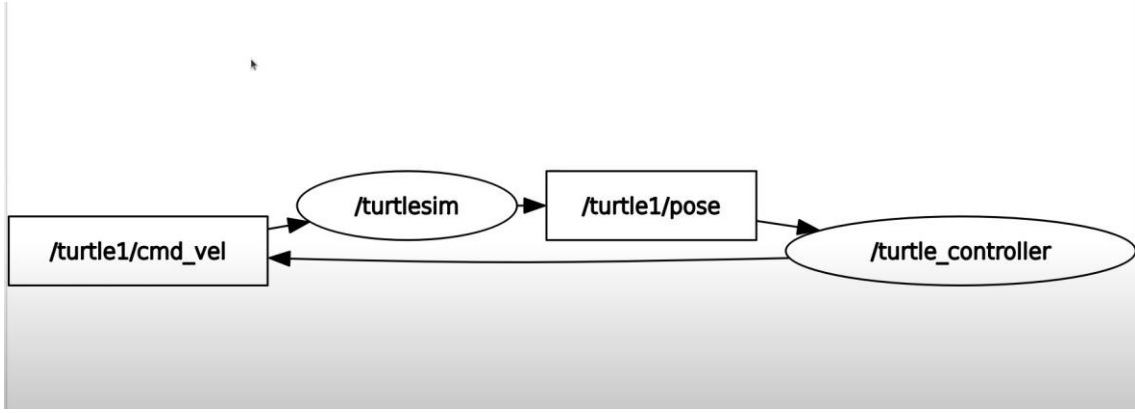
if __name__ == '__main__':
    main()

```

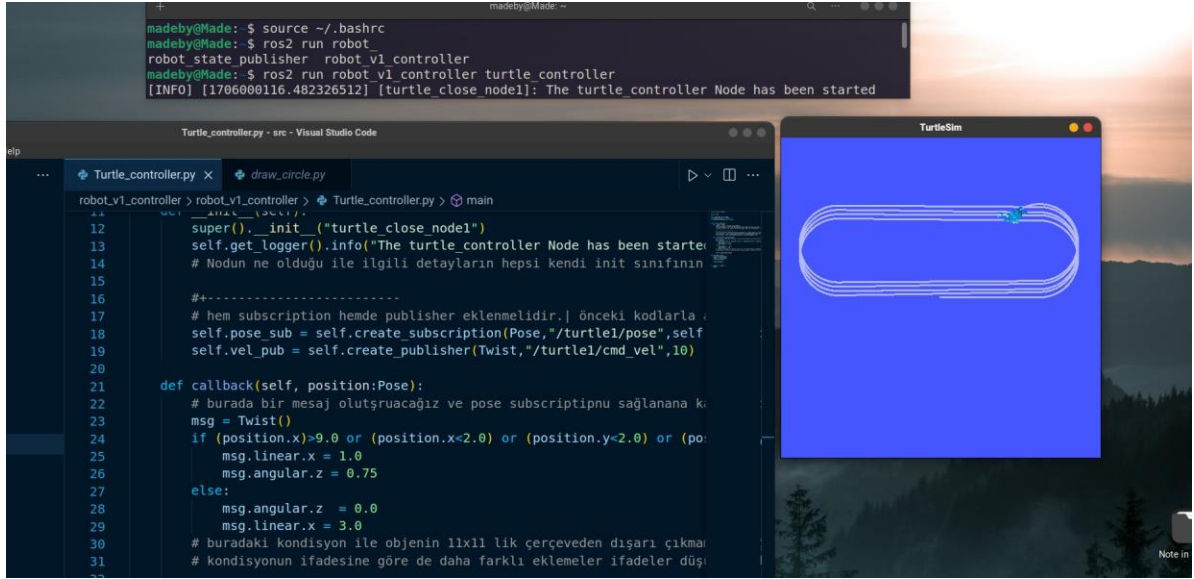
→ özellikle callback içindeki hareket eylemini gerçekleştiren mesaj içeriğinin hazırlanması bu anlamda etkili. ve direkt callback fonksiyonu içinde yazılıyor olması sürekli çağırılmasını ve hareket etmesini sağlıyor.

bunu şimdi pose a bağlı bir sistem oluşturursak, istediğimiz şekilde bir bağlam kurulabilir olacaktır.

(callback için bir position değişkeni oluşturduk pose dan geldiğine dair), bunu kullanacağız.



- Şimdi hareket algoritmasını kuralım.
 - 11 e 11 lik kare turtlesim ekranı, bunun içinde kenarlara yaklaştığı zaman dönüşünü optimize edecek ve dönecek şekilde bir angular, direct hız değerlerini değiştirerek yöneteceğiz.



pozisyonu kullanarak hangi aralığa geldiğinde nasıl davranacağını, ifade etmiş olduk.

// senaryo mantığını buna yedirerek, pose değerlerinin belirli aralıklarında statusu değiştirebilir ve hareketi yeniden planlayabiliriz.

Burada ML ile hangi senaryoda nasıl çalışacağını ve bunu deneyeceğini planlayabilir ve bir iş akışı oluşturabiliriz.

Takviyeli öğrenmede step step hangi sürecin en uygun olduğunu sisteme öğretmek kapalı bir çalışma prensibi geliştirilebilir.

status üzerinden hareket kontrolü de buradan izlenebilir.

What is The ROS Services ?

→ ROS2 server yapıları aslında localde çalışan bir mini server gibi request ve response ikilemesine çalışır. Nodes yapılarına gayet benzer ama bi tık daha farklı işlemektedir.

- ROS2 run demo_nodes_cpp addd_two_int_server // çağırıldıktan sonrasında server çalışacak ve beklemede kalacaktır. → request gelene kadar.
→ requestin de kendine göre bir formatı vardır ve request için gereksinimleri anlamamız lazım.

```
ed@ed-vm:~$ ros2 node list
/add_two_ints_server
ed@ed-vm:~$ ros2 service
call          list
find          type
--include-hidden-services
ed@ed-vm:~$ ros2 service type /add_two_ints
example_interfaces/srv/AddTwoInts
ed@ed-vm:~$ ros2 interface show example_interfaces/srv/AddTwoInts
int64 a
int64 b
---
int64 sum
ed@ed-vm:~$
```

- önce aktif node listesini gördükten ve bu aktif olan node'un bir server olduğunun tespitinden sonra ros2 services type // ile tipi öğreniyoruz .
- Sonrasında interface show ile de request ve responsun type bilgilerini elde ediyoruz.
 - Sonrasındaki işlem, istekte bulunma.

Ros2 service call <server adı> <server type'ı> "{ 'a' : 2, 'b': 6}"

→ bu sayede içeride a ve b olarak belirtilen sayıların toplamını ekrana verecektir.

Yanı sıra da çalışan node içerisinde server tarafında yapılan hareketlerin tespitini sağlamış olacağız.

- 2 taraflı olarak hem server tarafında hem de client tarafında biz sonuçları görebilme şansımız vardır. Bu anlamlıdır, birden fazla client tarafından sunucuya bağlanabilir, emir verilebilir veyahut alınabilir.

gerçekten kullanışlıdır.

```

madeby@Made: ~$ ros2 run demo_nodes_cpp add_two_ints_
add_two_ints_client add_two_ints_server
madeby@Made: ~$ ros2 run demo_nodes_cpp add_two_ints_
add_two_ints_client add_two_ints_server
madeby@Made: ~$ ros2 run demo_nodes_cpp add_two_ints_serv
er
[INFO] [1786029553.649869222] [add_two_ints_server]: Inc
oming request
a: 0 b: 0
[INFO] [1786029618.468456422] [add_two_ints_server]: Inc
oming request
a: 4 b: 8
local_
°C with wind speed nullm/h and null% humidity
madeby@Made: ~$
madeby@Made: ~$ ros2 service type /add_two_ints
example_interfaces/srv/AddTwoInts
madeby@Made: ~$ ros2 interface show /add_two_ints example_interfaces/srv/AddTwoInts
usage: ros2 [-h] [--use-python-default-buffering]
Call 'ros2 <command> -h' for more
detailed usage. ...
ros2: error: unrecognized arguments: example_interfaces/srv/AddTwoInts
madeby@Made: ~$ ros2 interface show example_interfaces/srv/AddTwoInts
int64 a
int64 b
....
int64 sum
madeby@Made: ~$ ros2 service call /add_two_ints "{a: 4, 'b': 8}"
The passed service type is invalid
madeby@Made: ~$ ros2 service call /add_two_ints example_interfaces/srv/AddTwoInts
"{a: 4, 'b': 8}"
requester: making request: example_interfaces.srv.AddTwoInts_Request(a=0, b=0)
response:
example_interfaces.srv.AddTwoInts_Response(sum=0)
{'a': 4, 'b': 8}: command not found
madeby@Made: ~$ ros2 service call /add_two_ints example_interfaces/srv/AddTwoInts "{a: 4, 'b': 8}"
requester: making request: example_interfaces.srv.AddTwoInts_Request(a=4, b=8)
response:
example_interfaces.srv.AddTwoInts_Response(sum=12)
madeby@Made: ~$

```

- öncelikle node list aktif olan nodelara baktık. // tabii ki bu sırada server açıldı.
 - sonrasında service list yaparak ilgili node un service olduğunu doğruladık ve sonrası da service type yaparak ilgili serverın tipini elde ettik.

Ros2 service call <server adı> <server type> "{a: 2, 'b': 6}"

- ile de a ve b değerinin içini doldurduktan sonra da pushladık.
ve serverın kendi side dında bir sonuç elde ettik.

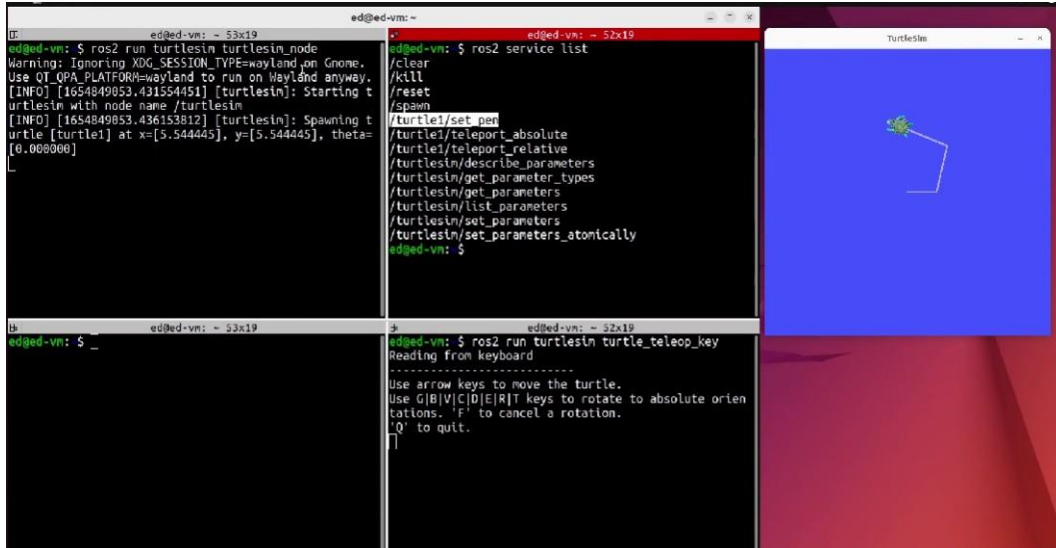
bu competition tip serverlara güzel bir örnektir.

birde bir competition yapmayan direkt sistem içerisinde ayarlar üzerinde değişiklik yapan serverlar vardır.

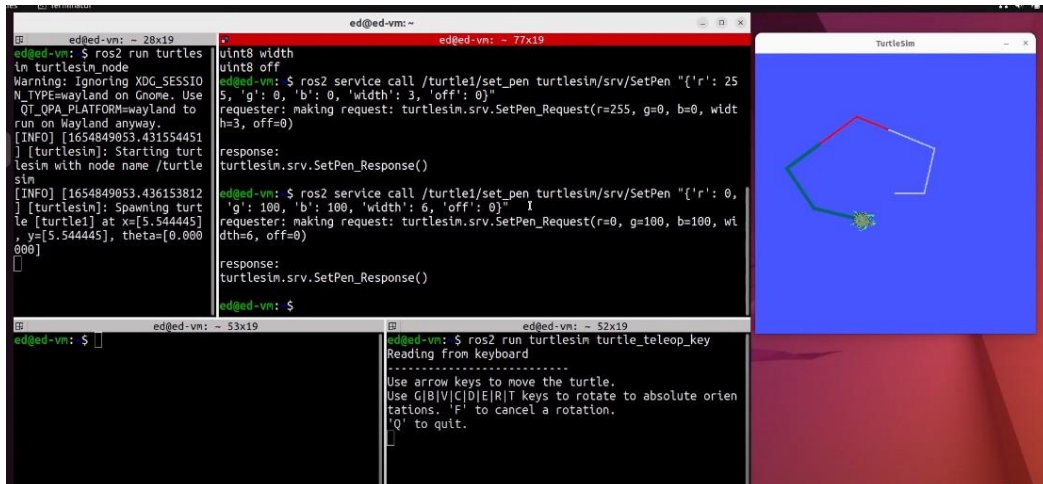
→ robotik obejlerde birden çok ayar bulunabilir, buna müdahale yine serverlar üzerinden sağlanabilir. Multi-object sistemler ile bunun yönetimi daha esnek şekilde

sağlanma şansı vardır.

→ bunu turtlesim üzerinden bir örnekle yapabiliriz.

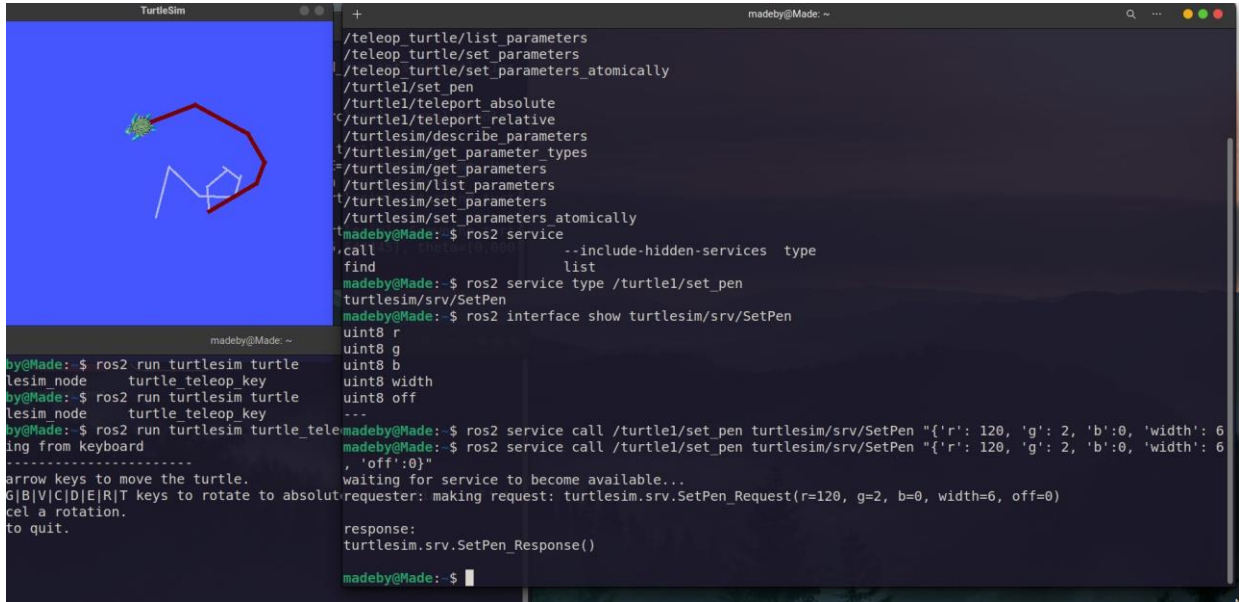


```
ed@ed-vm: ~$ ros2 run turtlesim turtlesim_node
Warning: Ignoring XDG_SESSION_TYPE=wayland on Gnome.
Use QT_OPA_PLATFORM=wayland to run on Wayland anyway.
[INFO] [1654849053.431554451] [turtlesim]: Starting t
urtlesim with node name /turtlesim
[INFO] [1654849053.436153812] [turtlesim]: Spawning t
urtle [turtle1] at x=[5.544445], y=[5.544445], theta=
[0.000000]
ed@ed-vm: ~$ ros2 service list
/clear
/kill
/reset
/spawn
/turtle1/set_pen
/turtle1/teleport_absolute
/turtle1/teleport_relative
/turtlesim/describe_parameters
/turtlesim/get_parameter_types
/turtlesim/get_parameters
/turtlesim/list_parameters
/turtlesim/set_parameters
/turtlesim/set_parameters_atomically
ed@ed-vm: ~$
```



```
ed@ed-vm: ~$ ros2 run turtlesim turtlesim_node
Warning: Ignoring XDG_SESSION_TYPE=wayland on Gnome.
Use QT_OPA_PLATFORM=wayland to run on Wayland anyway.
[INFO] [1654849053.431554451] [turtlesim]: Starting t
urtlesim with node name /turtle
sim
[INFO] [1654849053.436153812] [turtlesim]: Spawning t
urtle [turtle1] at x=[5.544445], y=[5.544445], theta=
[0.000000]
ed@ed-vm: ~$ ros2 service call /turtle1/set_pen turtlesim/srv/SetPen "{r': 25
5, 'g': 0, 'b': 0, 'width': 3, 'off': 0}"
requester: making request: turtlesim.srv.SetPen_Request(r=255, g=0, b=0, wlt
h=3, off=0)
response:
turtlesim.srv.SetPen_Response()
ed@ed-vm: ~$ ros2 service call /turtle1/set_pen turtlesim/srv/SetPen "{r': 0,
'g': 100, 'b': 100, 'width': 6, 'off': 0}"
requester: making request: turtlesim.srv.SetPen_Request(r=0, g=100, b=100, wlt
h=6, off=0)
response:
turtlesim.srv.SetPen_Response()
ed@ed-vm: ~$
```

- Belirli bir data işlemeye yönelik işlemleri Node ile ama daha çok cevap bekleme, spesifik ayarları değiştirme gibi seçeneksel ve iletişimsel işlemleri services ile sağlamakayız.



```
madeby@Made: ~  
$ ros2 service call /turtle1/set_pen turtlesim/srv/SetPen  
...  
uint8 r  
uint8 g  
uint8 b  
uint8 width  
uint8 off  
...  
madeby@Made: ~  
$ ros2 service call /turtle1/set_pen turtlesim/srv/SetPen "{r: 120, 'g': 2, 'b': 0, 'width': 6, 'off': 0}"  
waiting for service to become available...  
requester: making request: turtlesim.srv.SetPen_Request(r=120, g=2, b=0, width=6, off=0)  
response:  
turtlesim.srv.SetPen_Response()  
madeby@Made: ~
```

→ parametrelere göre turtlesim içerisindeki ayarları service tarafında böyle değiştirebiliyoruz.

interface tarafında gerekli bilgilendirmede response için belirli bir ibare yoktur. Ama çalıştığını request atıldıktan sonrasında görebiliyoruz. (adamlar yazmış)

Genel olarak böyle... Client service yazımı ile devam....

Writing a ROS2 service client

→ adım adım bir client side service nasıl yazılır inceleyelim.

```
def call_set_pen_service(self, r, g, b, width, off):
    client = self.create_client(SetPen, "/turtle1/set_pen")
    while not client.wait_for_service(1.0):
        self.get_logger().warn("Waiting for service...")

    request = SetPen.Request()
    request.r = r
    request.g = g
    request.b = b
    request.width = width
    request.off = off

    future = client.call_async(request)
    future.add_done_callback(partial(self.callback_set_pen))

    def callback_set_pen(self, future):
        try:
            response = future.result()
        except Exception as e:
            self.get_logger().error("Service call failed: %r" % (e,))
```

- Öncelikle subscriber da ve publisher da olduğu gibi oluşturulan client bir service olduğu belirtilmeli.
- sonrasında bir while not kondisyonu eklendi, service in başlatılmasını beklemkele ile ilgilidir.
- → sıra requeste geldi. Uğraştığımız service ile ilgili detaylı bilgileri aldıktan sonrasında request parametrelerini ayarlıyoruz.
- future adlı bir değişkeni response noktasında kullanacağız ve call fonksiyonu için kullanacağız client.call_async(request) /- eklenmesinin sebebi budur.
- add_dpne_callback ile de responsun nasıl döneceğini inceliyoruz. hata durumu ve direkt olarak çalışma durmunda ne olacağını inceliyoruz.

- response u herhangi bir şeye eşitlemedik. çünkü zaten kullandığımız bu service içinde direkt response üzerinden bir şey döndürülmüyor. interface show da bunu görmüştük.,
-

→ callback metodlarını iyi anlamakta yarar, takılıyorum o kısımlarda.

- görülen format basic bir service in nasıl yazıldığını ifade eder. Bu çok değişmeyecektir. genel şablon bu şekilde olmaktadır.
-

→ bir client service node sistemi içerisinde entegre edilebilir, gösterceğimiz gibi ...

- Ama bir service yapısını başlatmak ve sürekli çalışır olarak aktif tutmak performans açısından gerçekten bir kayıptır. sadece gerekli olduğu noktada çalıştırıp, sonrasında kapatmak çok daha mantıklıdır. Bunu aklımızda bulundurarak ayarların node çalıştırdığı sürece konfigürasyonunun yapılmasını ve bunu aktif olarak yönetilmesini sağlamak daha anlamlı bir süreçtir.

```

def service_call_set_pen(self,r,g,b,width,off):
    client = self.create_client(SetPen,"/turtle1/set_pen")

    while not client.wait_for_service(1.0):
        self.get_logger().warn("Waiting for the services....")
        # -> while not içinde service bağlantısı sağlanmadığı sürece bi log bilgisi ekrana yansiyacaktır.
        # bu şekilde service bağlantısı kesilmiş mi kesilmemiş mi kontrol edilebilir, çok mantıklı bir sorgudur.

    # şimdi service için request ayarlamasına geldi.

    request = SetPen.Request()
    request.r = r
    request.g = g
    request.b = b
    request.width = width
    request.off = off

    future = client.call_async(request)
    # -> future aslında ileride olacak bir işlemin şimdiden hazırlanması.
    # yani ileride yapmak istediğimiz çağrılacak requestin değişkenleri ile beraber tutulması için ve çağırılması için hazırladık
    future.add_done_callback(partial(self.future_callback_))

def future_callback_(self,future):
    # service yanıtladığında bu fonksyon devreye girecektir. /-/ işte burada responsu oluşturacağız.
    # mantıkende callback çağrısına cevap vermeyen bir service den başka zaman response beklemekte saçma olacaktır.
    try:
        response = future.result()
        # interface showda öğrenmiştik herhangi bir response yok bu set pende o yüzden herhangi bir dönüt olarka kullanmadık.

    except Exception as e:
        self.get_logger().error("service call failed :: %r" %e,)
        # direkt try ve except içerisidne bunu eridik ki belirli bir hata durumunda direkt exceptionu belirtsin.

```

→ genel bir service client tarafı nasıl olmalıdırın güzel bir örneğidir bu kod parçası.

```

class turtle_node(Node):
    def __init__(self):
        super().__init__("turtle_node")
        self.get_logger().info("The turtle_controller Node has been started")
        self.pose_sub = self.create_subscription(Pose,"/turtle1/pose",self.callback, 10)
        self.vel_pub = self.create_publisher(Twist,"/turtle1/cmd_vel",10)
        self.previous_x_ = 0

    def callback(self, position:Pose):
        msg = Twist()
        if (position.x>9.0 or (position.x<2.0) or (position.y<2.0) or (position.y>9.0):
            msg.linear.x = 1.0
            msg.angular.z = 0.75
            #self.get_logger().info("status :: TURN")
        else:
            msg.angular.z = 0.0
            msg.linear.x = 3.0
            #self.get_logger().info("status :: FORWARD")

        self.vel_pub.publish(msg)

        # service callback ile service işlemlerini entegrasyonu direkt node callback fonksyonu içinde sağlanabilir.
        if position.x > 5.5 and self.previous_x_ <=5.5:
            self.get_logger().info("The turtle on the right:: RED")
            self.service_call_set_pen(254,0,0,5,0)
            self.previous_x_ = position.x
            """
            # kondisyonun previous eklentisi sadece geçişler arasında değişiklik olacağı zaman çalışacak şekilde ayarlandı.
            -> bu sayede sürekli olarak service call yapılmaması olacak ve performans açısından kolaylık sağlanacaktır.
            """
        elif position.x <= 5.5 and self.previous_x_ > 5.5:
            self.get_logger().info("The turtle on the right:: GREEN")
            self.service_call_set_pen(0,254,0,5,0)
            self.previous_x_ = position.x
        else:
            pass

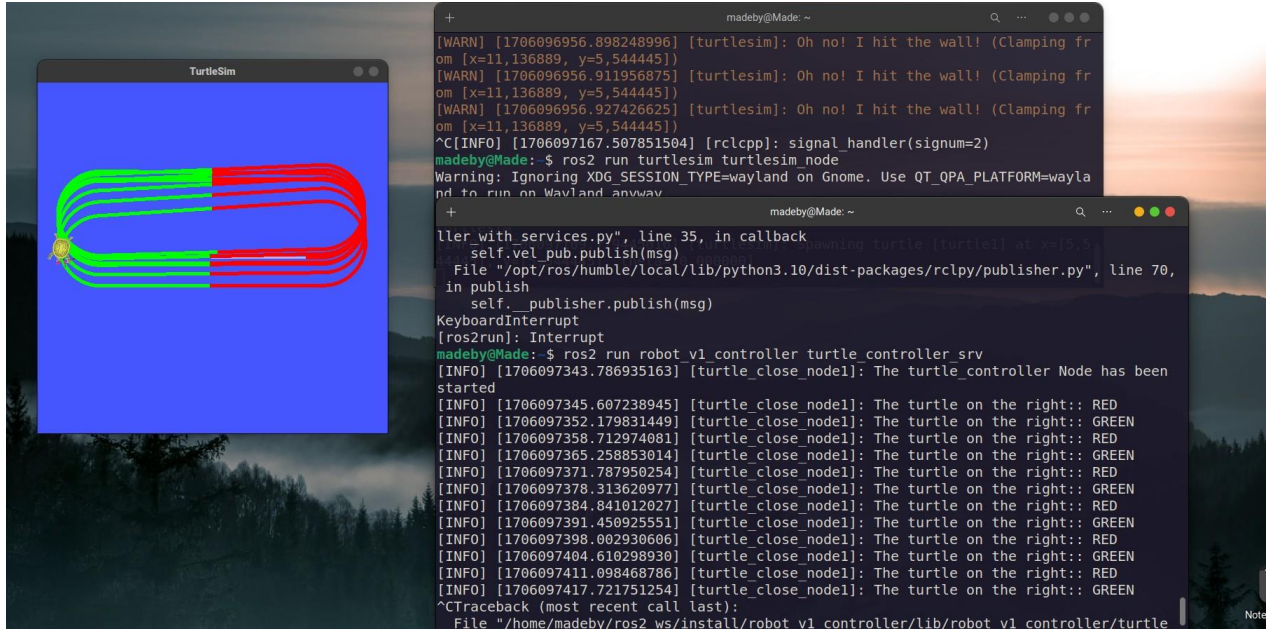
```

→ burada da genel bir node entegrasyonu içerisine nasıl implement edilebilir onun görüntüsünü içeriyor. direkt hazır callback içerisine subscriberın değişkenleri ve hareketi üzerinden işlem yapmış olduk.

ve bunun içerisinde service fonksiyonunu çağırarak robot ayarlarını (çizginin rengi kalınlığı vs gibi) yapmış olabiliyoruz. ve servisi her zaman değil çok az durumda çağırarak performanstan kayıp olmasının önüne geçmiş oluyoruz.

// Service mimarisi ilk başta anlaması zor ama baya güzel bir mimari buradaki.

Tekrar ettikçe zamanla oturacaktır diye düşünüyorum. gider ayak, SLAM içerisinde olan çalışmalar ile birlikte pratik edeceğiz.



yeşil olan kısımlar için conditiona LEFT yazılmalıydı, ana kodda düzeltildi.

<REFERENCES>