Entregable 2

Trabajo de la Asignatura

ENTORNOS OPERATIVOS PARA ROBÓTICA E INFORMÁTICA INDUSTRIAL | GRUPO A1

Grado en Informática Industrial y Robótica

Escola Tècnica
Superior d'Enginyeria
Informàtica

etsinf

Espert Cornejo, Ángela
Francés Limerá, Lourdes
Grado en Informática Industrial y Robótica

ÍNDICE

ÍNDICE

INTRODUCCIÓN

RESUMEN

En este documento se encuentra la explicación de cómo se han llevado a cabo las distintas aplicaciones y configuraciones necesarias para la entrega:

- Configuración de la Máquina Virtual (*VMWare*) y *ROS2*.
- Aplicación Puente *ROS2-MQTT*.
- Configuración *Mosquitto* y *MQTT* (y *MQTTX*).
- Configuración de *OPC-UA Browser*.
- Aplicación Servidor *OPC-UA Browser*.
- Configuración de *Ignition*.
- Configuración *Ignition Designer*.

ÍNDICE

INTRODUCCIÓN

ORIGEN DE DATOS Y PUENTE ROS2-MQTT

PASO A OPC-UA

VISIÓN GENERAL 1

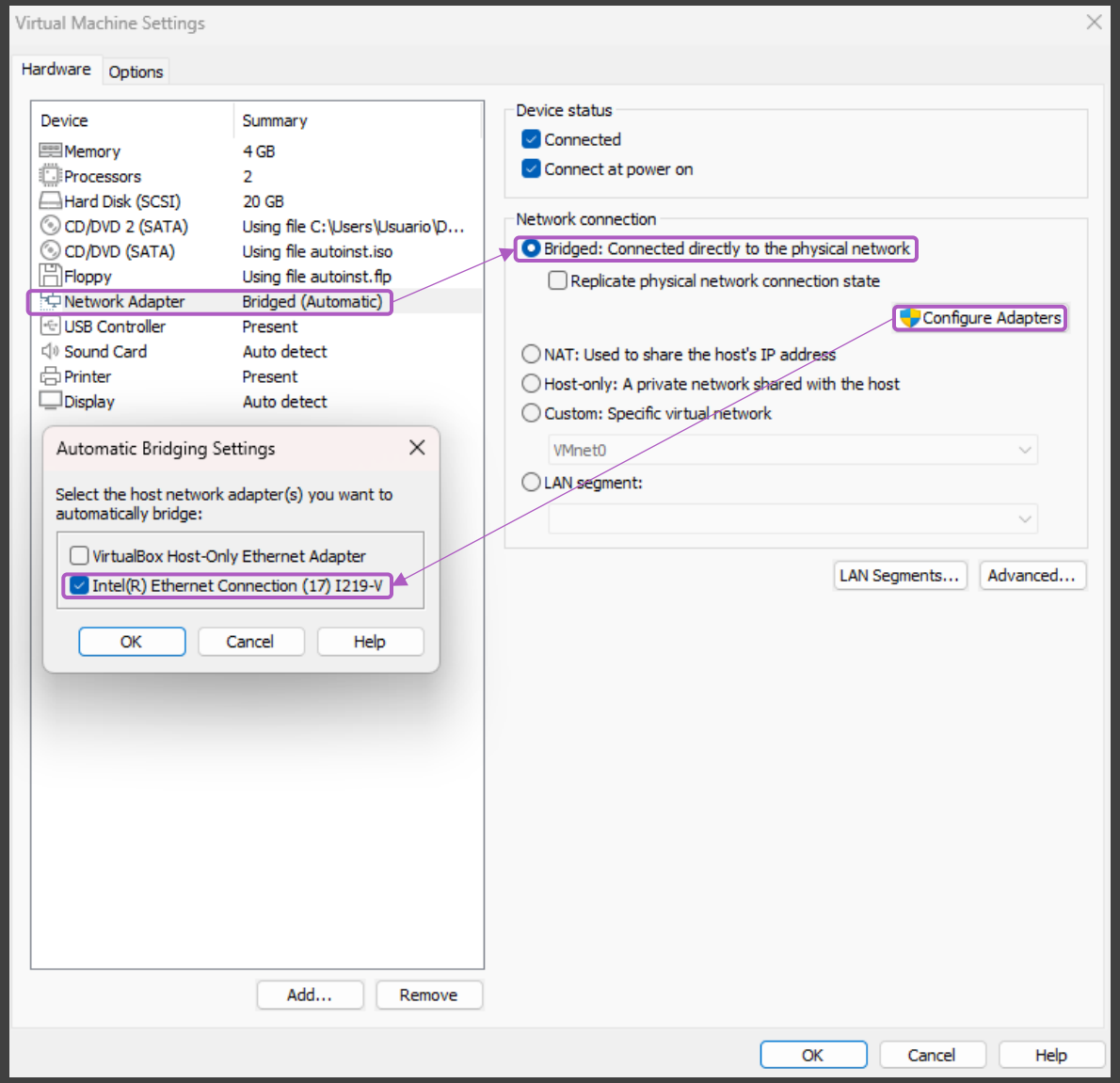
VISIÓN GENERAL 2

CIERRE

Es necesario modificar las opciones de la máquina virtual para poder realizar la comunicación *MQTT*.

Para ello, hay que acceder a "*Virtual Machine Settings*" -> "*Network Adapter*" y, en la opción "*Network Connection*" seleccionar la opción "*Bridged*".

Tras eso, es recomendable configurar los adaptadores. En este caso, se selecciona como "*host network adapter*" la segunda opción.



Para la realización de este entregable, se ha creado inicialmente una carpeta con “mkdir Entregable”.

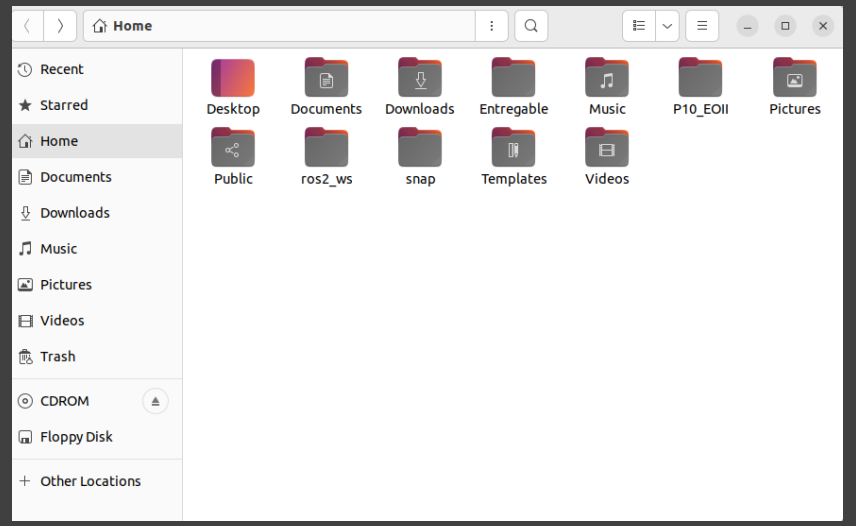
Posteriormente, dentro de esta carpeta se ha creado otra de nombre “src” empleando el mismo comando.

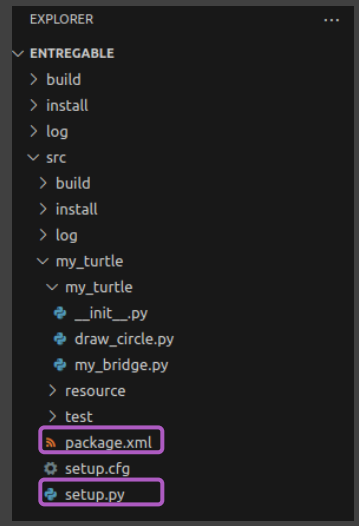
Dentro de la carpeta “Entregable”, se han empleado los comandos “colcon build” y “echo "source /home/louhlouh/Entregable/install/setup.bash" >> ~/.bashrc”

Para la creación del *package*, dentro de la carpeta “src” se emplea “ros2 pkg create my_turtle --build-type ament_python --dependencies rclpy” seguido de “colcon build”.

Dentro de este paquete se crean dos aplicaciones de *Python*. Cada uno de estos ficheros es un código ejecutable y, por tanto, un nodo. Estas serán explicadas más adelante.

Además, es necesario configurar el “*setup.py*”, incluyendo los dos nodos, y el “*package.xml*” .





En “setup.py”, se añaden las líneas “'my_bridge = my_turtle.my_bridge:main',” y “'draw_path = my_turtle.draw_path:main'”

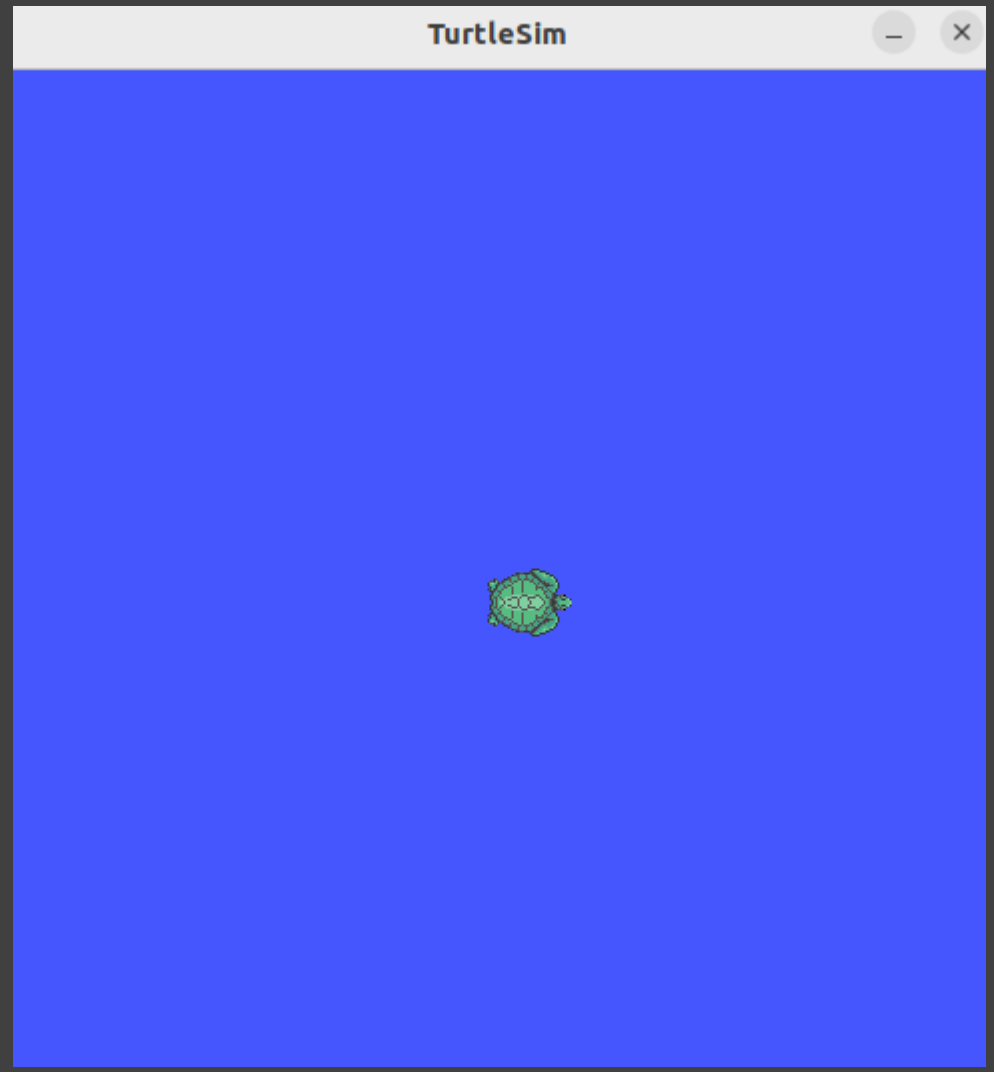
En “package.xml”, se añaden las líneas “<depend>geometry_msg</depend>” y “<depend>turtlesin</depend>”

```python
from setuptools import find_packages, setup

package_name = 'my_turtle'

setup(
    name=package_name,
    version='0.0.0',
    packages=find_packages(exclude=['test']),
    data_files=[
        ('share/ament_index/resource_index/packages',
            ['resource/' + package_name]),
        ('share/' + package_name, ['package.xml']),
    ],
    install_requires=['setuptools'],
    zip_safe=True,
    maintainer='louhlouh',
    maintainer_email='louhlouh@todo.todo',
    description='TODO: Package description',
    license='TODO: License declaration',
    tests_require=['pytest'],
    entry_points={
        'console_scripts': [
            'my_bridge = my_turtle.my_bridge:main',
            'draw_path = my_turtle.draw_path:main'
        ],
    },
)
```

```xml
<?xml version="1.0"?>
<?xml-model href="http://download.ros.org/schema/package_format3.xsd" schematypens="http://www.w3.org/2001/XMLSchema"?>
<package format="3">
  <name>my_turtle</name>
  <version>0.0.0</version>
  <description>TODO: Package description</description>
  <maintainer email="louhlouh@todo.todo">louhlouh</maintainer>
  <license>TODO: License declaration</license>

  <depend>rclpy</depend>
  <depend>geometry_msg</depend>
  <depend>turtlesin</depend>

  <test_depend>ament_copyright</test_depend>
  <test_depend>ament_flake8</test_depend>
  <test_depend>ament_pep257</test_depend>
  <test_depend>python3-pytest</test_depend>

  <export>
    <build_type>ament_python</build_type>
  </export>
</package>
```

Para la obtención de los datos es necesario ejecutar en una terminal “ros2 run turtlesim turtlesim_node”.

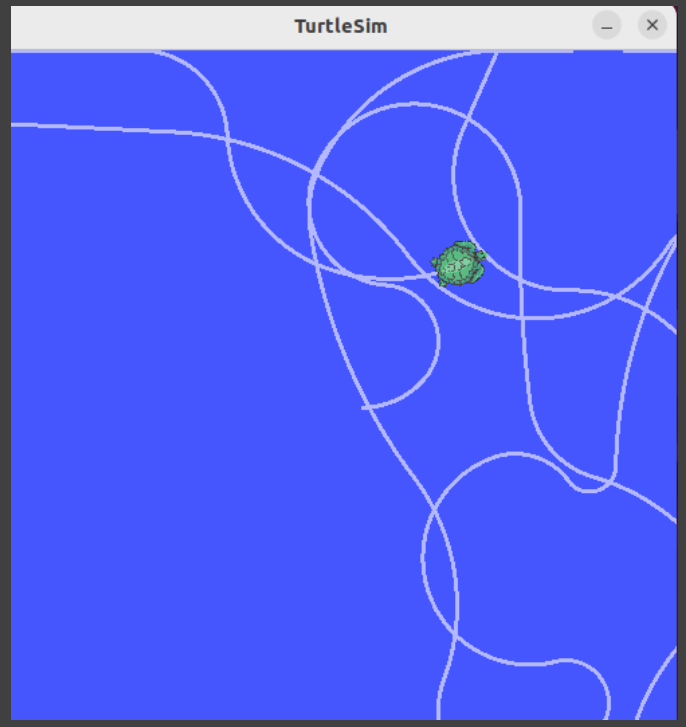
Este comando pone en ejecución el simulador de *turtlebot* “*TurtleSim*” del cual obtendremos los distintos datos solicitados.




```
louhlouh@louhlouh-virtual-machine:~$ ros2 run turtlesim turtlesim_node
```

Para no recoger datos estáticos, se ha optado por hacer uso de un código básico de movimiento de la tortuga (“draw_path.py”).

Al ejecutarlo, la tortuga modifica su velocidad lineal y angular, realizando movimientos aleatorios.




```python
import rclpy
from rclpy.node import Node
from geometry_msgs.msg import Twist
import random

class NodoDibujaCamino(Node):
    def __init__(self):
        super().__init__("drawing_path")
        self.cmd_vel_pub = self.create_publisher(Twist, "/turtle1/cmd_vel", 10)
        self.get_logger().info("Inicialización del movimiento de la tortuga.")
        self.create_timer(1.0, self.enviar_velocidad)

    def enviar_velocidad(self):
        msg = Twist()
        msg.linear.x = random.uniform(0.0, 5.0)
        msg.angular.z = random.uniform(-3.0, 3.0)
        self.cmd_vel_pub.publish(msg)
        self.get_logger().info("Velocidad modificada.")

def main(args=None):
    rclpy.init(args=args)
    nodo_pub = NodoDibujaCamino()
    rclpy.spin(nodo_pub)
    nodo_pub.destroy_node()
    rclpy.shutdown()

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

```
louhlouh@louhlouh-virtual-machine:~/Entregable$ ros2 run my_turtle draw_path
```

Para realizar la comunicación de información entre *ROS2* y *MQTT* es necesario hacer uso de una aplicación (my_bridge.py) que realice las siguientes funciones:

- Suscribirse al *topic /turtle1/pose* para obtener la posición (x, y) y orientación (theta) de la tortuga.
- Suscribirse al *topic /turtle1/cmd_vel* para obtener la velocidad lineal y angular de la tortuga.
- Publicar los datos obtenidos en un mensaje *JSON* por *MQTT*:
  - Mensaje *JSON* vector de 2 componentes y 1 dato (posición, orientación).
  - Mensaje *JSON* de 2 datos (velocidad lineal, velocidad angular).

```python
import rclpy
from rclpy.node import Node

from turtlesim.msg import Pose
from geometry_msgs.msg import Twist

import paho.mqtt.client as mqtt
import json

import time

BROKER_ADRESS = "192.168.0.13" # Varía según la IP del host
PORT = 1883

POSE_TOPIC = "turtle/pose"
VEL_TOPIC = "turtle/velocity"
```

Este código primero importa las librerías necesarias para su funcionamiento.

A continuación, establece 4 variables globales.

- *BROKER_ADRESS*: Esta variable varía dependiendo de la IP del host. Esta información puede obtenerse desde la *cmd* del host mediante el comando “ipconfig”.
- *PORT*: Puerto empleado para las conexiones.
- *POSE_TOPIC*: *topic* en el que se publicarán los datos referentes a la posición y orientación.
- *VEL_TOPIC*: *topic* en el que se publicarán los datos referentes a las velocidades.

```python
class TurtleMQTTPublisher(Node):

    def __init__(self):
        super().__init__('turtle_mqtt_publisher')

        # Subscripciones a los tópicos
        # Posición y Orientación
        self.pose_subs = self.create_subscription(
            Pose,
            '/turtle1/pose',
            self.pose_callback,
            10)
        self.pose_subs  # prevent unused variable warning
        # Velocidad lineal y angular
        self.cmd_vel_subs = self.create_subscription(
            Twist,
            '/turtle1/cmd_vel',
            self.cmd_vel_callback,
            10)
        self.cmd_vel_subs # prevent unused variable warning

        # Configuración del cliente MQTT
        self.mqtt_client = mqtt.Client("TurtleMQTTPublisher")
        self.mqtt_client.connect(BROKER_ADRESS, PORT)

        # Tiempos de publicación e intérvalos
        self.last_pose_time = 0.0
        self.last_vel_time = 0.0
        self.pose_interval = 1.0
        self.vel_interval = 0.5

    def pose_callback(self, msg):…

    def cmd_vel_callback(self, msg):…
```

Este código declara una clase "*TurtleMQTTPublisher*" que hereda de la clase *Node* de *ROS2*.

1. El método "*__init__*" inicializa la clase.
2. Mediante "*self.pose_subs*" se crea una subscripción al tópico "*/turtle1/pose*", el cual publica un mensaje tipo "*Pose*". Cada vez que se reciba un mensaje en este tópico, se llamará al método "*self.pose_callback*".
3. Mediante "*self.cmd_vel_subs*" se crea una subscripción al tópico "*/turtle1/cmd_vel*", el cual publica un mensaje tipo "*Twist*". Cada vez que se reciba un mensaje en este tópico, se llamará al método "*self.cmd_vel_callback*".
4. Se crea un cliente *MQTT* y se conecta al bróker en la dirección y puertos anteriormente establecidos.
5. Se establecen unos intervalos que permiten controlar la frecuencia de obtención de los datos (estos permiten que ambos datos se obtengan con frecuencias similares y evita problemas de pérdidas de datos *a posteriori*).

```python
class TurtleMQTTPublisher(Node):

    def __init__(self):…

    def pose_callback(self, msg):
        current_time = time.time()
        if current_time - self.last_pose_time >= self.pose_interval:
            self.last_pose_time = current_time

            # Mensaje JSON para posición y orientación
            pose_data = {
                "position": [msg.x, msg.y],
                "orientation": msg.theta
            }
            pose_json = json.dumps(pose_data)

            # Publicación por MQTT del mensaje
            self.mqtt_client.publish(POSE_TOPIC, pose_json)
            self.get_logger().info(f"Turtle pose: {pose_json}")

    def cmd_vel_callback(self, msg):
        current_time = time.time()
        if current_time - self.last_vel_time >= self.vel_interval:
            self.last_vel_time = current_time

            # Mensaje JSON para velocidad
            velocity_data = {
                "linear_velocity": msg.linear.x,
                "angular_velocity": msg.angular.z
            }
            velocity_json = json.dumps(velocity_data)

            # Publicación por MQTT del mensaje
            self.mqtt_client.publish(VEL_TOPIC, velocity_json)
            self.get_logger().info(f"Turtle velocity: {velocity_json}")
```

6. Se define el método “*pose_callback*”. Este primero comprueba si ha pasado el intervalo de tiempo requerido y extrae los datos de posición y orientación del mensaje recibido, organizándolos en un diccionario llamado “*pose_data*”. A continuación, convierte el diccionario en un *string* en formato *JSON* y lo almacena en “*pose_json*”. Por último, publica el mensaje *JSON* al *topic MQTT* especificado por *POSE_TOPIC*.

7. Se define el método “*cmd_vel_callback*”. Este primero comprueba si ha pasado el intervalo de tiempo requerido y extrae las velocidades lineal y angular del mensaje recibido y los organiza en un diccionario llamado “*velocity_data*”. A continuación, convierte el diccionario en un *string* en formato *JSON* y lo almacena en “*velocity_json*”. Por último, publica el mensaje *JSON* al *topic MQTT* especificado por *VEL_TOPIC*.

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

    turtle_mqtt_publisher = TurtleMQTTPublisher()

    # Manejo de interrupción mediante "ctrl+C" en terminal
    try:
        rclpy.spin(turtle_mqtt_publisher)
    except KeyboardInterrupt:
        pass

    # Destruir el nodo y detener MQTT
    turtle_mqtt_publisher.destroy_node()
    turtle_mqtt_publisher.mqtt_client.disconnect()  # Cerrar conexión MQTT
    rclpy.shutdown()

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

Esta parte del código define la función principal del programa.

1. Se inicializa el sistema de *ROS2* mediante "rclpy.init(args=args)".
2. Se crea una instancia del nodo *TurtleMQTTPublisher*.
3. Se ejecuta el nodo en un bucle infinito hasta que se detiene el nodo mediante una interrupción por teclado.
4. Se destruye el nodo, detiene la conexión *MQTT* y apaga el sistema de *ROS2*.

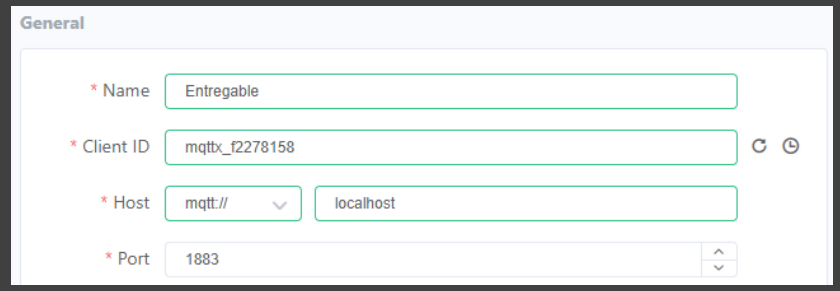
Ejemplo de impresión por terminal del código:

```
[INFO] [1735489033.524698350] [turtle_mqtt_publisher]: Turtle velocity: {"linear_velocity": 4.74617943972584, "angular_velocity": 1.4415620282933066}
[INFO] [1735489033.668561393] [turtle_mqtt_publisher]: Turtle pose: {"position": [8.348675727844238, 6.5869779586791 99], "orientation": 1.939117670059204}
[INFO] [1735489034.486584905] [turtle_mqtt_publisher]: Turtle velocity: {"linear_velocity": 1.7904409583552576, "angular_velocity": -2.8044862239483903}
[INFO] [1735489034.679653924] [turtle_mqtt_publisher]: Turtle pose: {"position": [4.679611682891846, 8.738859176635742], "orientation": 2.624359130859375}
```

Para iniciar el bróker *Mosquitto* en *Windows* es necesario ejecutar "mosquitto.exe" en la terminal de *Windows*.

```
C:\Users\Usuario>cd "C:\Program Files\mosquitto"

C:\Program Files\mosquitto>mosquitto.exe

C:\Program Files\mosquitto>
```

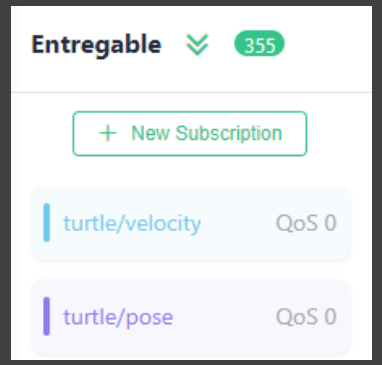
Mediante *MQTTX* se puede realizar una monitorización inicial de los datos enviados por el publicador para comprobar que todos los datos se envían y reciben correctamente.



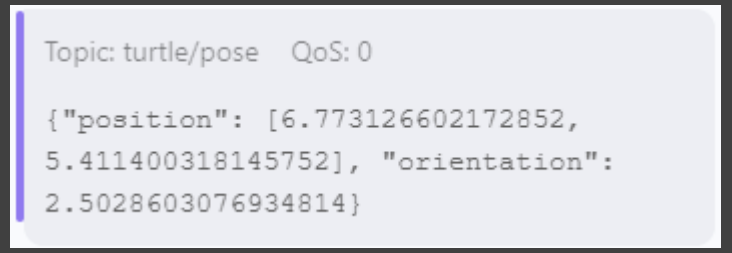
Primero, se configura una nueva conexión de nombre "Entregable".

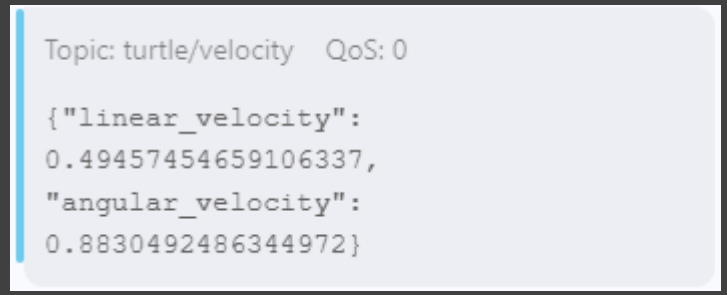
A continuación, se crean las subscripciones a los dos tópicos:

- turtle/velocity: recibe las velocidades en formato {"linear_velocity": a, "angular_velocity": b}
- turtle/pose: recibe la posición y orientación en formato {"position": [x, y], "orientation": z}



Ejemplo de mensajes recibidos:





ÍNDICE
INTRODUCCIÓN
ORIGEN DE DATOS Y PUENTE ROS2-MQTT

PASO A OPC-UA

VISIÓN GENERAL 1
VISIÓN GENERAL 2
CIERRE

Es necesario configurar una conexión en *OPC-UA Browser*.

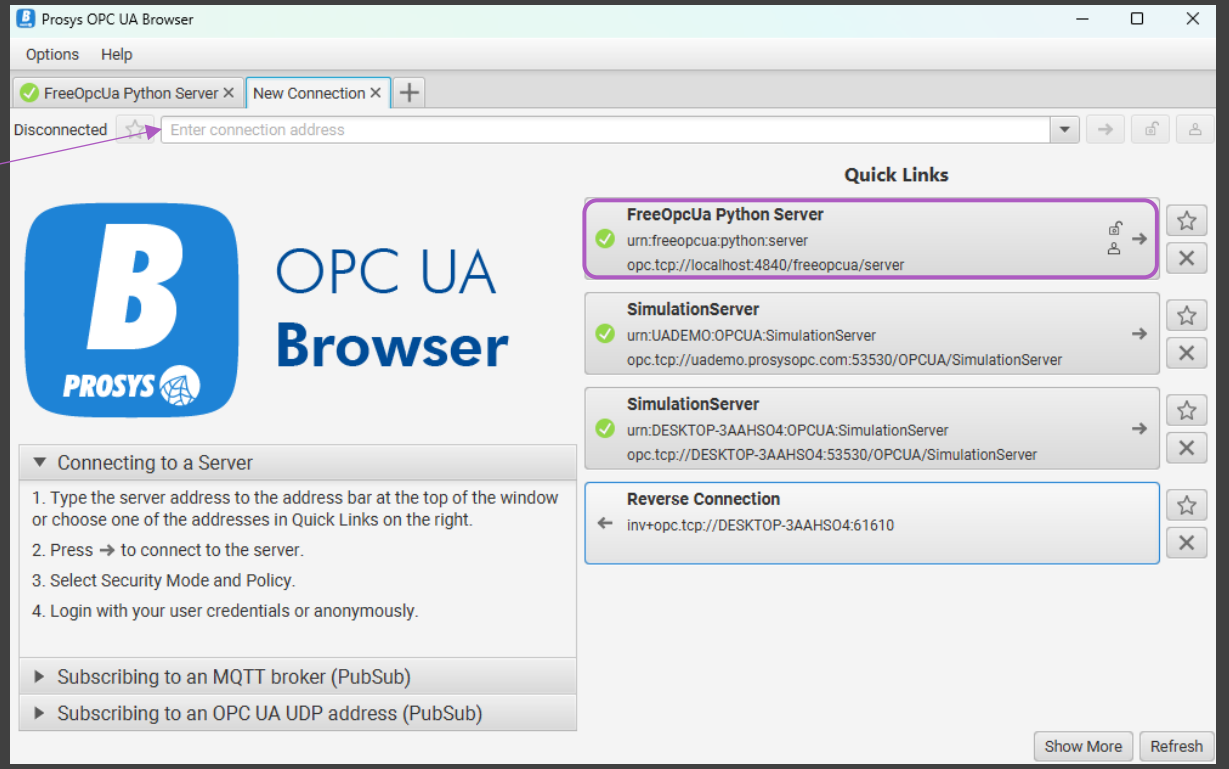
La dirección de la conexión en este caso es:
"opc.tcp://localhost:4840/freeopcua/server/"

*Para poder acceder a este servidor, es necesario estar ejecutando el código "servidor_OPC-UA_Browser.py", dedicado al manejo de este servidor.

En este se configura el *endpoint* con el mismo nombre especificado como dirección de la conexión.

```
ENDPOINT= "opc.tcp://localhost:4840/freeopcua/server/"
```

```
# Configuración del endpoint del servidor OPC-UA
server.set_endpoint(ENDPOINT)
```



Para realizar la comunicación de información entre el bróker *Mosquitto* y el servidor *OPC-UA* es necesaria la creación de una aplicación que haga de servidor *OPC-UA* que realice las siguientes funciones:

- Suscribirse al *topic turtle/pose* para obtener la posición (x, y) y orientación (theta) de la tortuga.
- Suscribirse al *topic turtle/velocity* para obtener la velocidad lineal y angular de la tortuga.
- Actualizar los datos obtenidos en los nodos correspondientes *OPC-UA*:
  - Pose: Vector posición, orientación y estado (contador y últimos valores).
  - Velocity: Velocidad lineal, velocidad angular y estado (contador y últimos valores).

```python
import threading
import time
from asyncua.sync import Server
import json
from datetime import datetime
import paho.mqtt.client as mqtt

BROKER_ADDRESS = "localhost"
PORT = 1883
ENDPOINT= "opc.tcp://localhost:4840/freeopcua/server/"

POSE_TOPIC = "turtle/pose"
VEL_TOPIC = "turtle/velocity"

pose_counter = 0
velocity_counter = 0
```

Este código primero importa las librerías necesarias para su funcionamiento.

A continuación, establece 7 variables globales.

- *BROKER_ADRESS*: Se establece como "*localhost*".
- *PORT*: Puerto empleado para las conexiones.
- *ENDPOINT*: Dirección de la conexión con el Servidor *OPC-UA*.
- *POSE_TOPIC*: *topic* del que se recibirán los datos referentes a la posición y orientación.
- *VEL_TOPIC*: *topic* del que se recibirán los datos referentes a las velocidades.
- *pose_counter*: Contador que indica la cantidad de datos de tipo *pose* recibidos.
- *velocity_counter*: Contador que indica la cantidad de datos de tipo *velocity* recibidos.

```python
class PoseSubscriber(threading.Thread):
    def __init__(self):
        super().__init__()
        self.topic = POSE_TOPIC
        self.client = mqtt.Client(f"pose_subscriber")
        self.client.on_message = self.on_message

    def run(self):
        try:
            self.client.connect(BROKER_ADDRESS, PORT)
            self.client.subscribe(self.topic)
            print(f"Pose Subscriber subscribed to topic: {self.topic}")
            self.client.loop_forever()
        except Exception as e:
            print(f"Error connecting to MQTT broker: {e}")

    def on_message(self, client, userdata, msg):
        # Lectura del mensaje recibido
        message = json.loads(msg.payload.decode())

        position = message["position"] #[x, y]
        orientation = message["orientation"]

        print(f"Received from {self.topic}: {message}")

        # Obtención de los últimos valores
        last_position = turtle_position.get_value()
        last_orientation = turtle_orientation.get_value()

        # Actualización en los nodos de OPC-UA
        turtle_position.write_value(position)
        turtle_orientation.write_value(orientation)
        turtle_last_position.write_value(last_position)
        turtle_last_orientation.write_value(last_orientation)

        print(f"Object 'Pose' updated with 'Position': {position}, Orientation: {orientation}]")

        # Actualización del contador
        global pose_counter
        pose_counter += 1

        turtle_pose_count.write_value(pose_counter)
```

Se declara una clase "*PoseSubscriber*" que hereda de *threading.Thread*, lo que permite que se ejecute en un hilo separado del programa principal.

1. Se define el constructor "*__init__*" para inicializar el tópico al que se subscribe, la conexión *MQTT* y el método de manejo de mensajes entrantes.
2. Se define el método "*run*", en el cual se realiza la conexión al bróker *MQTT* y la subscripción al tópico "*POSE_TOPIC*". Posteriormente, se llama a "*loop_forever*", que escucha de forma continua los mensajes del *topic*.
3. Se define el método "*on_message*", el cual se activa cuando se recibe un mensaje del tópico y realiza las siguientes acciones:
   - I. Extrae "*position*" y "*orientation*" del mensaje.
   - II. Obtiene los valores anteriores leyendo el valor actual de los nodos *OPC-UA*.
   - III. Actualiza los valores en *OPC-UA* con los obtenidos de "*position*" y "*orientation*" y los valores anteriores.
   - IV. Incrementa el contador global "*pose_counter*" y escribe el valor en el nodo *OPC-UA* correspondiente.

```python
class VelocitySubscriber(threading.Thread)
    def __init__(self):
        super().__init__()
        self.topic = VEL_TOPIC
        self.client = mqtt.Client(f"velocity_subscriber")
        self.client.on_message = self.on_message

    def run(self):
        try:
            # Conectar al broker MQTT y suscribirse al tema correspondiente
            self.client.connect(BROKER_ADDRESS, PORT)
            self.client.subscribe(self.topic)
            print(f"Velocity Subscriber subscribed to topic: {self.topic}")
            self.client.loop_forever()
        except Exception as e:
            print(f"Error connecting to MQTT broker: {e}")

    def on_message(self, client, userdata, msg):
        # Lectura del mensaje recibido
        message = json.loads(msg.payload.decode())

        linear_vel = message["linear_velocity"]
        angular_vel = message["angular_velocity"]

        print(f"Received from {self.topic}: {message}")

        # Obtención de los últimos valores
        last_linear = turtle_linear_vel.get_value()
        last_angular = turtle_angular_vel.get_value()

        # Actualización en los nodos de OPC-UA
        turtle_linear_vel.write_value(linear_vel)
        turtle_angular_vel.write_value(angular_vel)
        turtle_last_linear.write_value(last_linear)
        turtle_last_angular.write_value(last_angular)

        print(f"Object 'Velocity' updated with 'Linear': {linear_vel}, 'Angular': {angular_vel}")

        # Actualización del contador
        global velocity_counter
        velocity_counter += 1

        turtle_velocity_count.write_value(velocity_counter)
```

Se declara una clase "*VelocitySubscriber*" que hereda de *threading.Thread*, lo que permite que se ejecute en un hilo separado del programa principal.

1. Se define el constructor "*__init__*" para inicializar el tópico al que se subscribe, la conexión *MQTT* y el método de manejo de mensajes entrantes.
2. Se define el método "*run*", en el cual se realiza la conexión al bróker *MQTT* y la subscripción al tópico "*VEL_TOPIC*". Posteriormente, se llama a "*loop_forever*", que escucha de forma continua los mensajes del *topic*.
3. Se define el método "*on_message*", el cual se activa cuando se recibe un mensaje del tópico y realiza las siguientes acciones:
   - I. Extrae "*linear_velocity*" y "*angular_velocity*" del mensaje.
   - II. Obtiene los valores anteriores leyendo el valor actual de los nodos *OPC-UA*.
   - III. Actualiza los valores en *OPC-UA* con los obtenidos de del *topic* y los valores anteriores.
   - IV. Incrementa el contador global "*velocity_counter*" y escribe el valor en el nodo *OPC-UA* correspondiente.

```python
if __name__ == "__main__":

    # Configuración del servidor
    server = Server()
    # Configuración del endpoint del servidor OPC-UA
    server.set_endpoint(ENDPOINT)

    # Configuración del namespace
    uri = "http://localhost/mynamespace"
    idx = server.register_namespace(uri)
    print(f"Espacio de nombres registrado: {uri} con índice: {idx}")

    # Creación de los nodos objeto y variables
    # Posición y orientación
    myobj1 = server.nodes.objects.add_object(idx, "Pose")
    turtle_position = myobj1.add_variable(idx, "Position", [0.0, 0.0])
    turtle_orientation = myobj1.add_variable(idx, "Orientation", 0.0)
    turtle_position.set_writable()
    turtle_orientation.set_writable()
    # Control de estado
    turtle_pose_state = myobj1.add_object(idx, "State")
    turtle_pose_count = turtle_pose_state.add_variable(idx, "Pose Counter", 0)
    turtle_last_position = turtle_pose_state.add_variable(idx, "Last Position", [0.0, 0.0])
    turtle_last_orientation = turtle_pose_state.add_variable(idx, "Last Orientation", 0.0)
    turtle_pose_count.set_writable()
    turtle_last_position.set_writable()
    turtle_last_orientation.set_writable()
    # Velocidad
    myobj2 = server.nodes.objects.add_object(idx, "Velocity")
    turtle_linear_vel = myobj2.add_variable(idx, "Linear", 0.0)
    turtle_angular_vel = myobj2.add_variable(idx, "Angular", 0.0)
    turtle_linear_vel.set_writable()
    turtle_angular_vel.set_writable()
    # Control de estado
    turtle_velocity_state = myobj2.add_object(idx, "State")
    turtle_velocity_count = turtle_velocity_state.add_variable(idx, "Velocity Counter", 0)
    turtle_last_linear = turtle_velocity_state.add_variable(idx, "Last Linear", 0.0)
    turtle_last_angular = turtle_velocity_state.add_variable(idx, "Last Angular", 0.0)
    turtle_velocity_count.set_writable()
    turtle_last_linear.set_writable()
    turtle_velocity_count.set_writable()

    server.start()
    print("Servidor OPC-UA iniciado y en ejecución.")

    pose_subscriber = PoseSubscriber()
    vel_subscriber = VelocitySubscriber()

    pose_subscriber.start()
    vel_subscriber.start()
```

Esta parte del código define la función principal del programa.

1. Se configura el servidor *OPC-UA* inicializando una instancia "*Server()*" y configurando el *endpoint*.
2. Se registra un *namespace* para agrupar los nodos *OPC-UA*.
3. Se crean los nodos y las variables, estableciéndolas como escribibles.

Pose	Position	
	Orientation	
	State	Pose Counter
		Last Position
		Last Orientation
Velocity	Linear	
	Angular	
	State	Velocity Counter
		Last Linear
		Last Angular

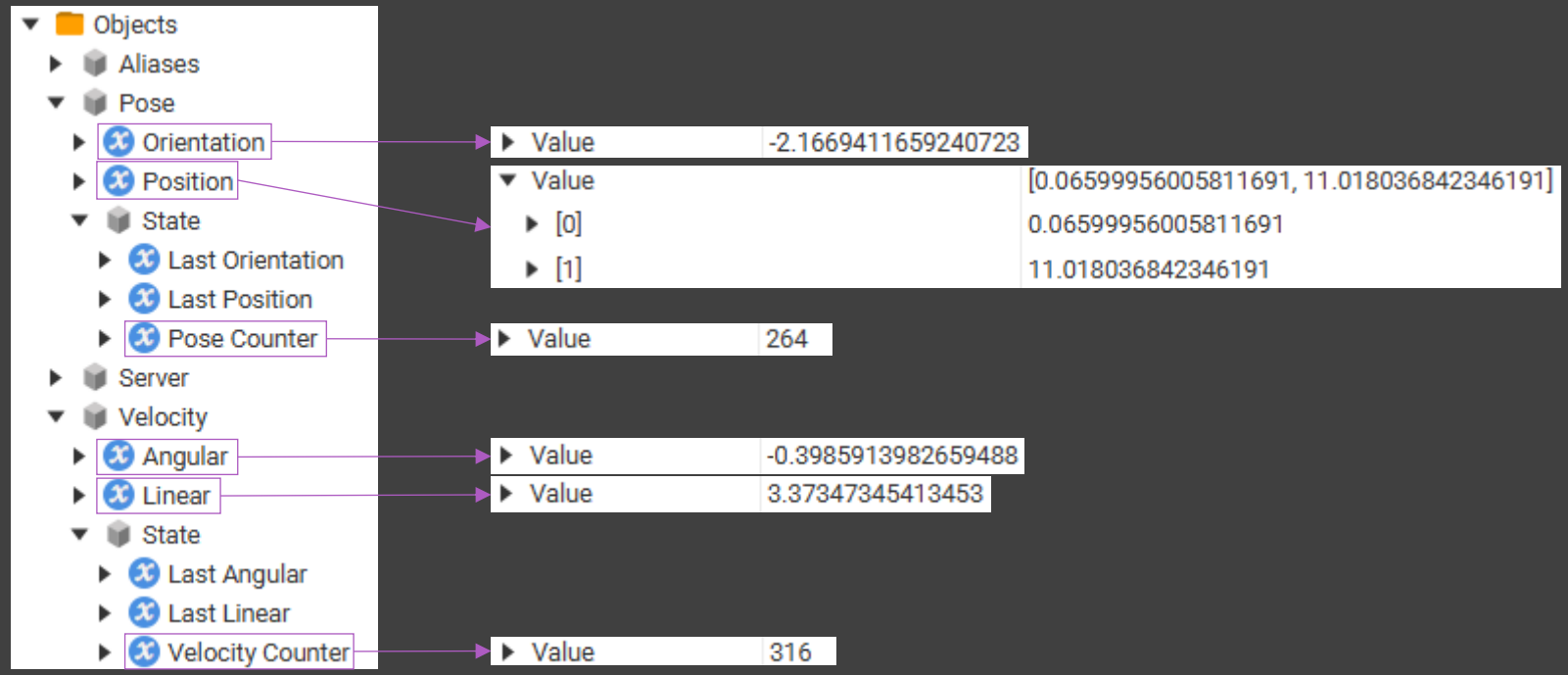
Estos se organizan en dos objetos principales (*Pose* y *Velocity*). Cada uno cuenta con dos variables y un subnodo "*State*". Este subnodo tiene tres variables.

4. Se inicializa el servidor.
5. Se inicializan los subscriptores *MQTT*, cada uno en un hilo independiente.

Datos obtenidos en terminal:

```
Received from turtle/pose: {'position': [0.31067246198654175, 11.088889122009277], 'orientation': 1.0173602104187012}
Object 'Pose' updated with 'Position': [0.31067246198654175, 11.088889122009277], Orientation: 1.0173602104187012]
Received from turtle/velocity: {'linear_velocity': 4.782541039273772, 'angular_velocity': 0.41661387486259205}
Object 'Velocity' updated with 'Linear': 4.782541039273772, 'Angular': 0.41661387486259205
Received from turtle/pose: {'position': [4.683532238006592, 10.50181770324707], 'orientation': -0.134185329079628}
Object 'Pose' updated with 'Position': [4.683532238006592, 10.50181770324707], Orientation: -0.134185329079628]
Received from turtle/velocity: {'linear_velocity': 2.296332517225874, 'angular_velocity': -0.38827364451164836}
Object 'Velocity' updated with 'Linear': 2.296332517225874, 'Angular': -0.38827364451164836
```

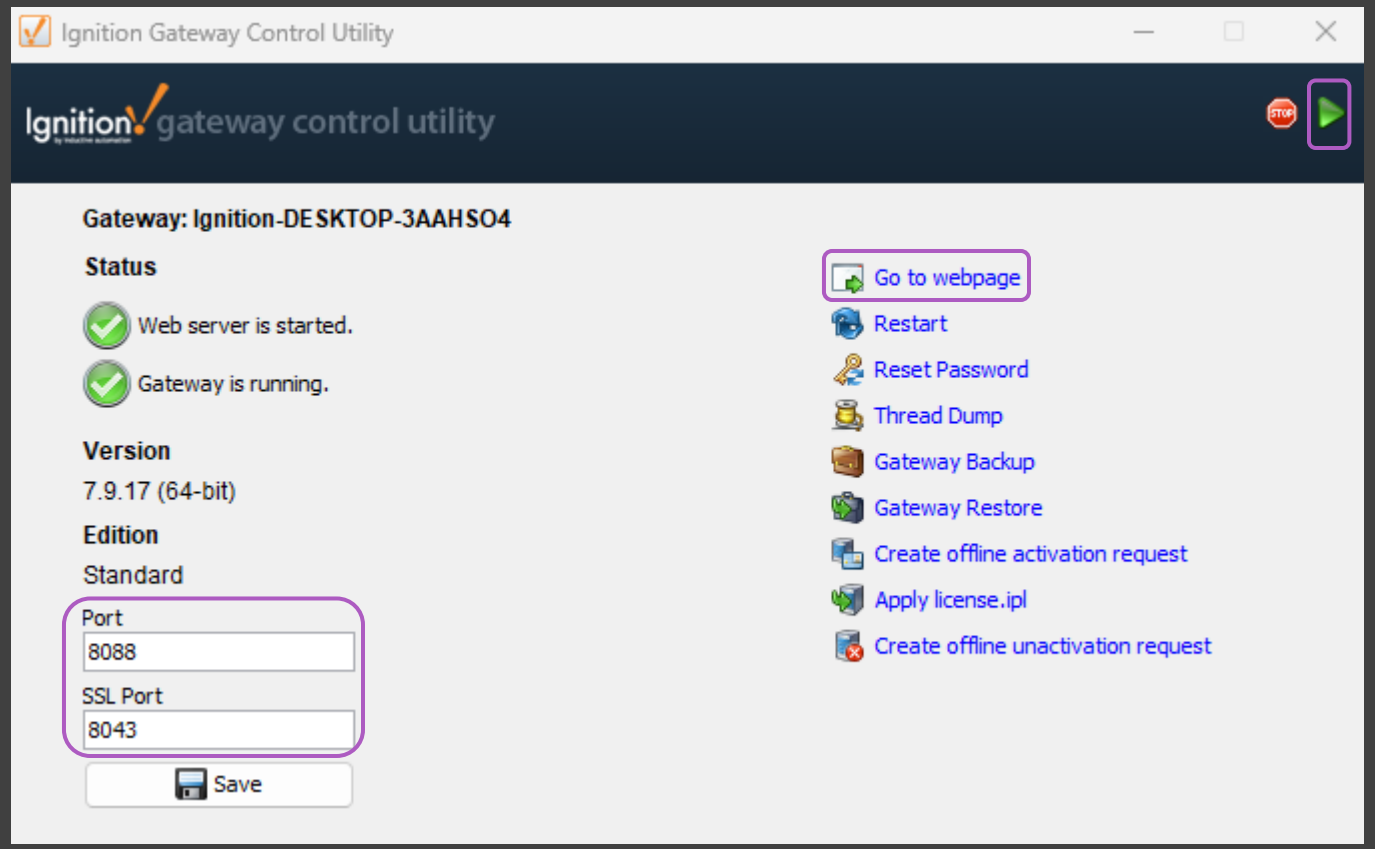
Datos obtenidos en el servidor:



ÍNDICE
INTRODUCCIÓN
ORIGEN DE DATOS Y PUENTE ROS2-MQTT
PASO A OPC-UA

VISIÓN GENERAL 1

VISIÓN GENERAL 2
CIERRE



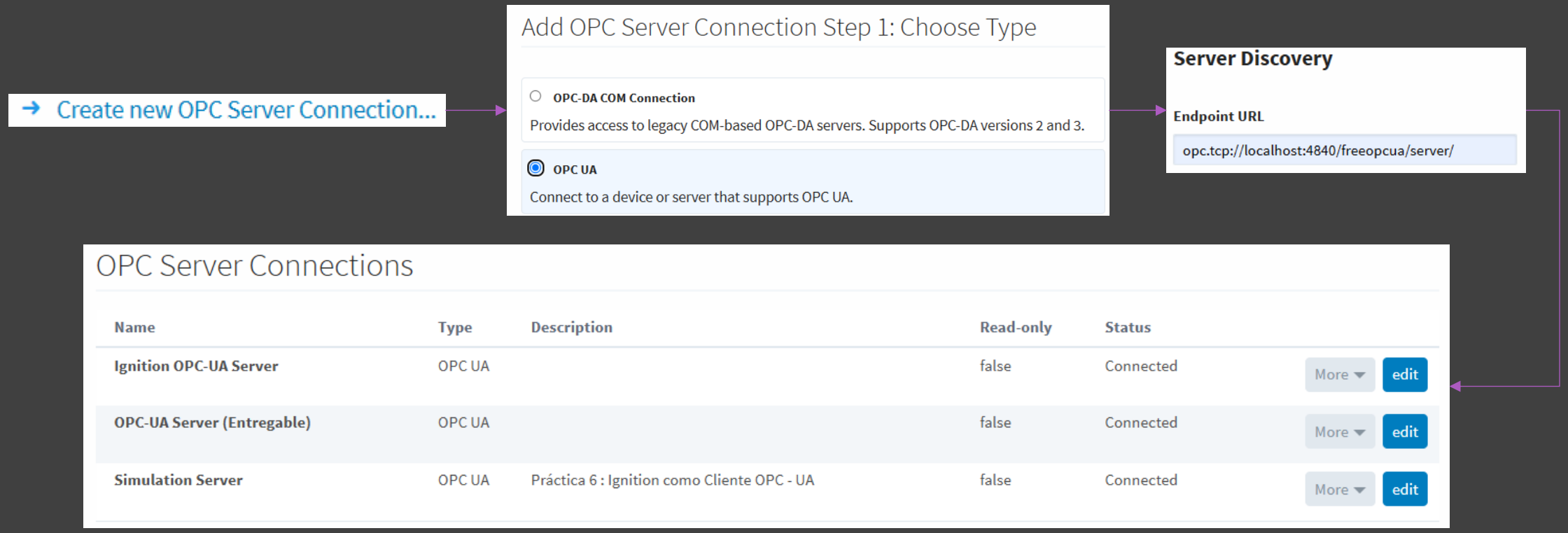
Para poder utilizar *Ignition*, es necesario inicializar el "*Gateway Control Utility*".

Primero, hay que establecer el *Port* y *SSL Port*.

Posteriormente, se presiona el botón de inicio y, una vez iniciado el servidor, se procede a ir a la página web de *Ignition*.

Una vez iniciada la sesión en *Ignition* e iniciada la versión de prueba, se procede a crear la conexión desde el apartado de "Configuración – *OPC Connections* – *Servers*".

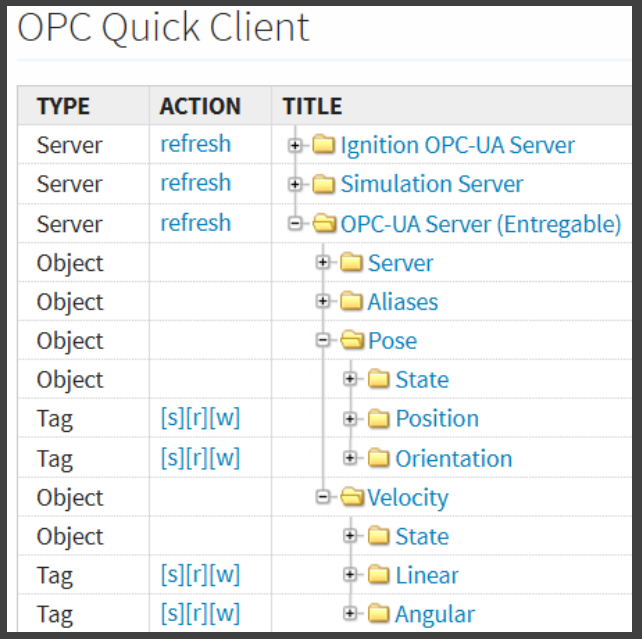
Para ello, es necesario establecer como "*Endpoint URL*" la dirección mencionada con anterioridad que se emplea tanto en el código de la aplicación *OPC-UA* como en el *OPC-UA Browser*.

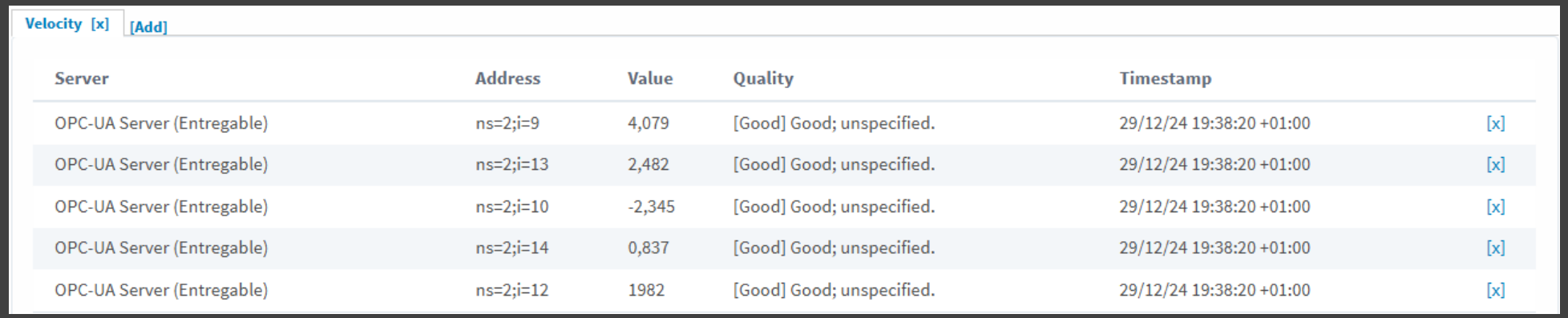


Desde "Configuración – *OPC Connections – Quick Client*" se puede acceder a los datos que se reciben en "OPC-UA Server (Entregable)". Para ello, es necesario subscribirse al tópico que se desea leer.

"*Velocity [x]*" es un ejemplo de subscripción. Esta está subscrita a:

1. *Velocity – Linear*
2. *Velocity – State – Last Linear*
3. *Velocity – Angular*
4. *Velocity – State – Angular*
5. *Velocity – Velocity Counter*





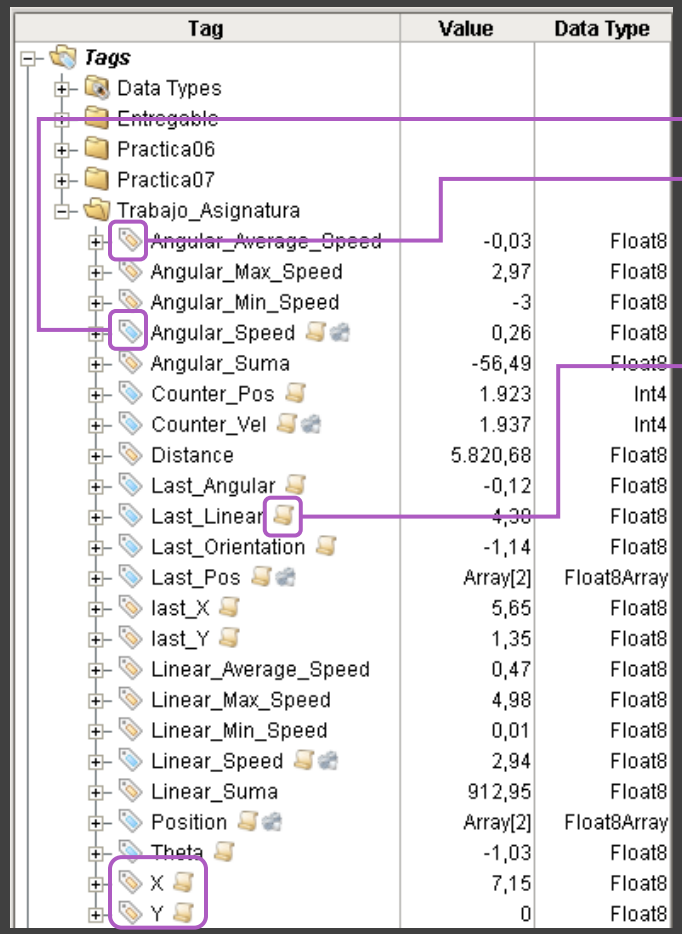
Server	Address	Value	Quality	Timestamp	
OPC-UA Server (Entregable)	ns=2;i=9	4,079	[Good] Good; unspecified.	29/12/24 19:38:20 +01:00	[x]
OPC-UA Server (Entregable)	ns=2;i=13	2,482	[Good] Good; unspecified.	29/12/24 19:38:20 +01:00	[x]
OPC-UA Server (Entregable)	ns=2;i=10	-2,345	[Good] Good; unspecified.	29/12/24 19:38:20 +01:00	[x]
OPC-UA Server (Entregable)	ns=2;i=14	0,837	[Good] Good; unspecified.	29/12/24 19:38:20 +01:00	[x]
OPC-UA Server (Entregable)	ns=2;i=12	1982	[Good] Good; unspecified.	29/12/24 19:38:20 +01:00	[x]

ÍNDICE
INTRODUCCIÓN
ORIGEN DE DATOS Y PUENTE ROS2-MQTT
PASO A OPC-UA
VISIÓN GENERAL 1

VISIÓN GENERAL 2

CIERRE

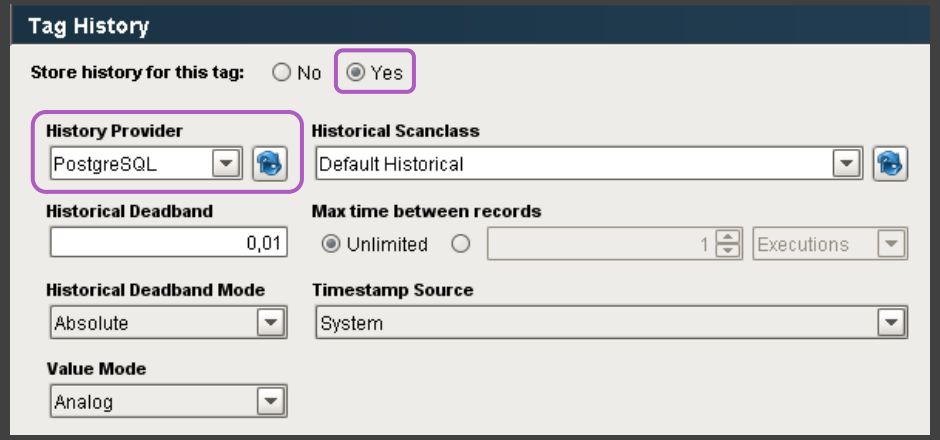
Para representar gráficamente en *Ignition Designer* los datos obtenidos en "OPC-UA Server (Entregable)" se crean varios *tags*. Estos, permitirán suscribirse al dato *OPC-UA* que se desea leer y así asociarlo a su representación correspondiente.



Dependiendo de los datos que estos reciben, se emplea un *tag* de un tipo u otro :

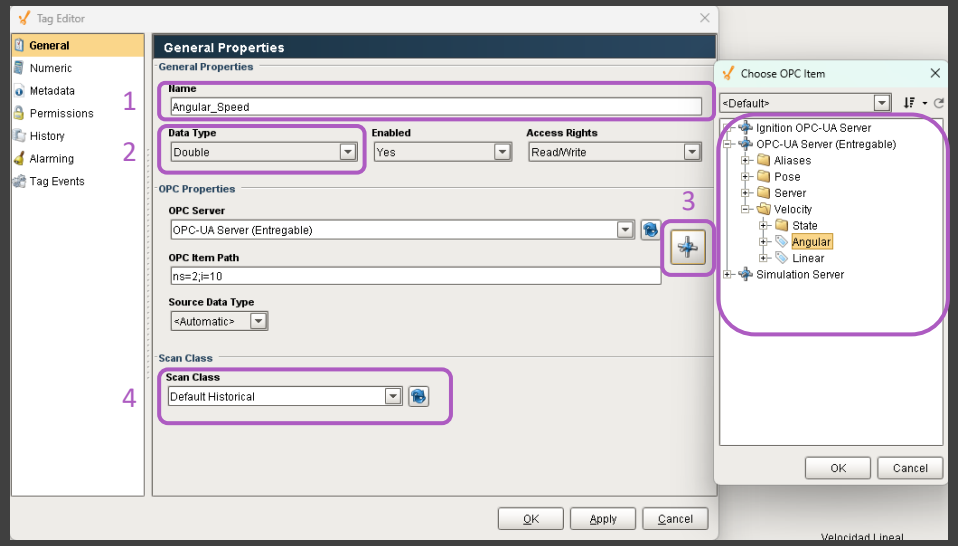
- ***OPC Tags*** : Etiquetas que tratan los datos obtenidos del servidor *OPC-UA* directamente.
- ***Memory Tags*** : Etiquetas derivadas de las *OPC Tags*; es decir, sus valores se obtienen a partir de cálculos y tratamientos de los datos obtenidos de las *OPC Tags*.

Asimismo, para analizar la variación de los valores de un tag a lo largo del tiempo, se define dicho tag como *Historical*. Para ello, tras realizar la creación de dicho tag, habrá que activar "*Store history for this tag*", seleccionando la base de datos donde se obtiene el historial, en este caso PostgreSQL.

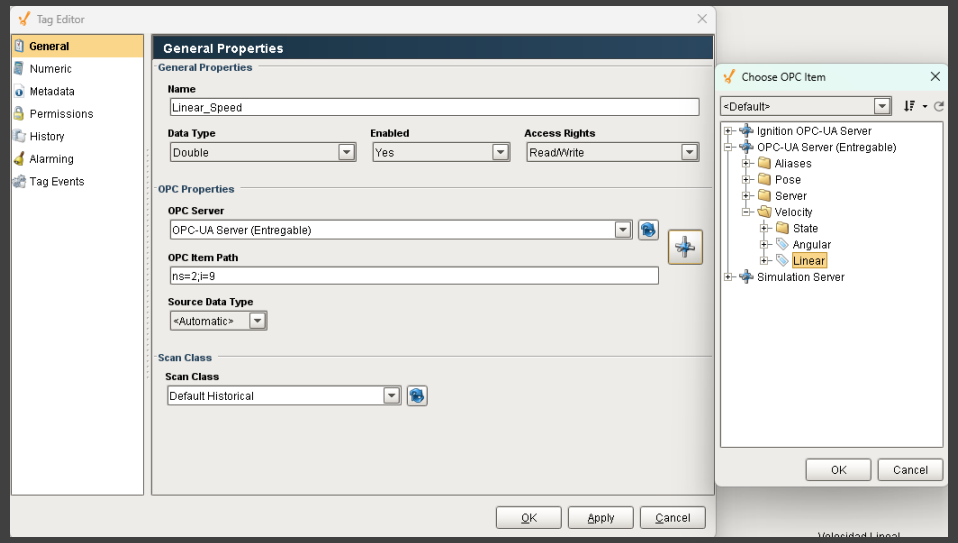


Un ejemplo de tags que requieren activar esta opción serían los tags "*X*" e "*Y*". Los cuales se han asociado a un *Easy Chart* para mostrar la variación de la posición de la tortuga.

Angular_Speed



Linear_Speed



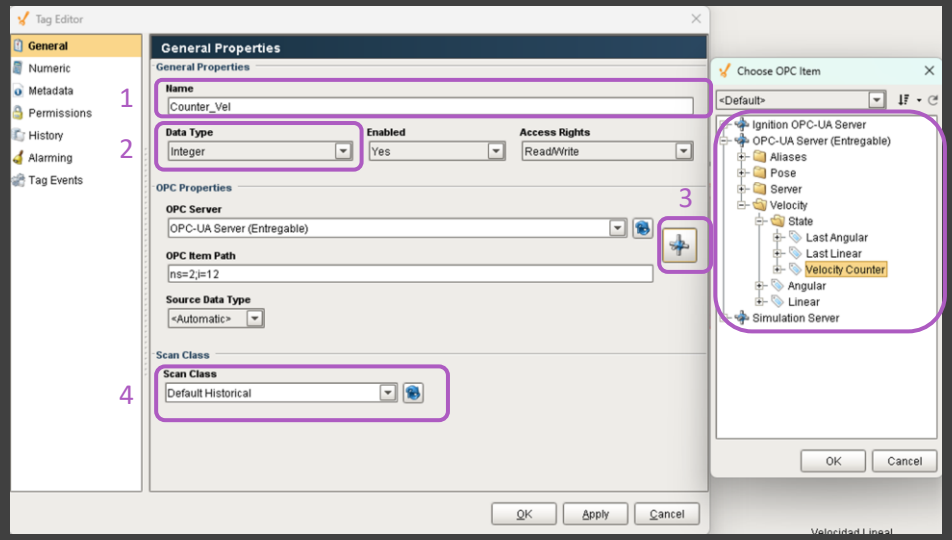
Velocity-Tags

Los *tags* "*Angular_Speed*" y "*Linear_Speed*" recogen la información relacionada con la velocidad angular y lineal de la tortuga.

Para la creación de estos *tags* se han realizado los siguientes pasos :

1. Definir el nombre del *tag*.
2. Especificar el tipo de dato que va a recibir dicho *tag* (en este caso *Double*).
3. Vincular el *topic* que recibe el servidor de *Ignition* con el *tag*.
4. Establecer el *tag* como *Default Historical*.

Counter_Vel

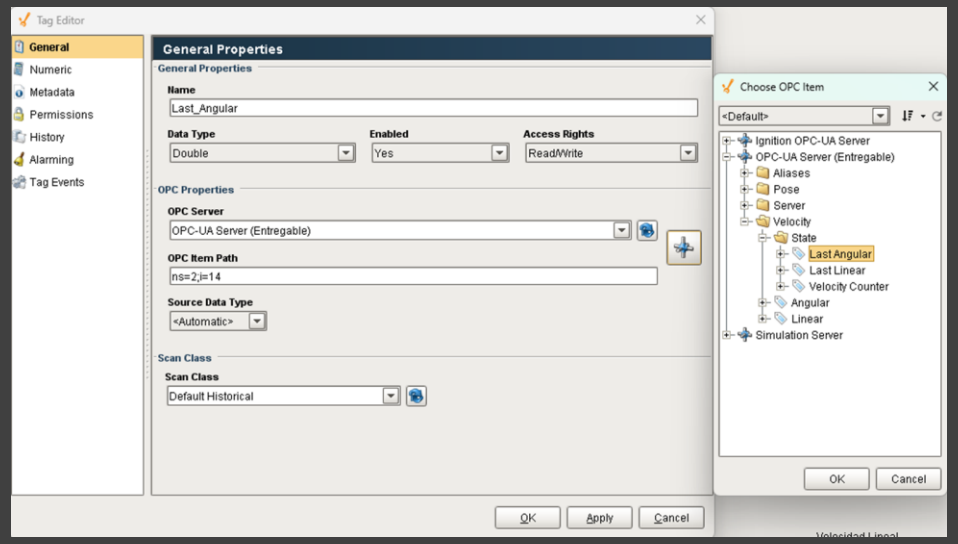


Velocity-State-Tags

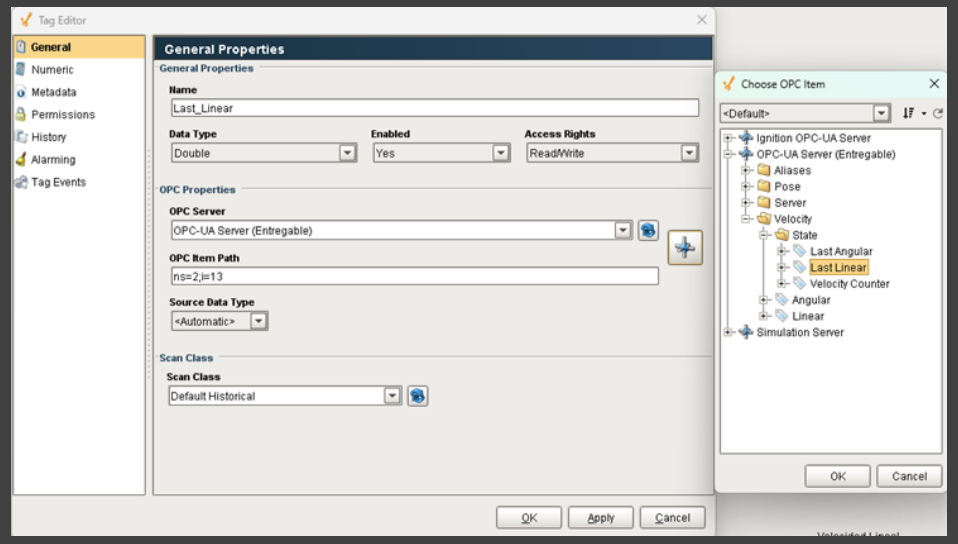
Los *tags* "*Last_Angular*", "*Last_Linear*" y "*Counter_Vel*" recogen la información relacionada con los últimos datos recibidos respecto a la velocidad angular y lineal de la tortuga, y la cantidad de datos recibidos en total en cuanto a la velocidad.

Para la creación de estos *tags* se han realizado los mismos pasos comentados anteriormente con un ligero matiz respecto al *tag* "*Counter_Vel*"; el cual, en vez de ser *Double* como el resto, es de tipo *Integer*, ya que dicho *tag* funciona como contador.

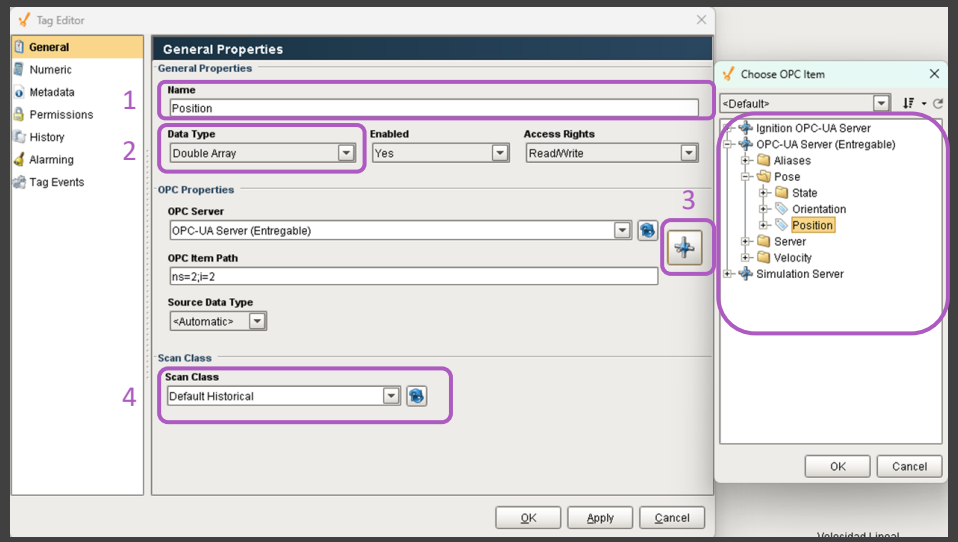
Last_Angular



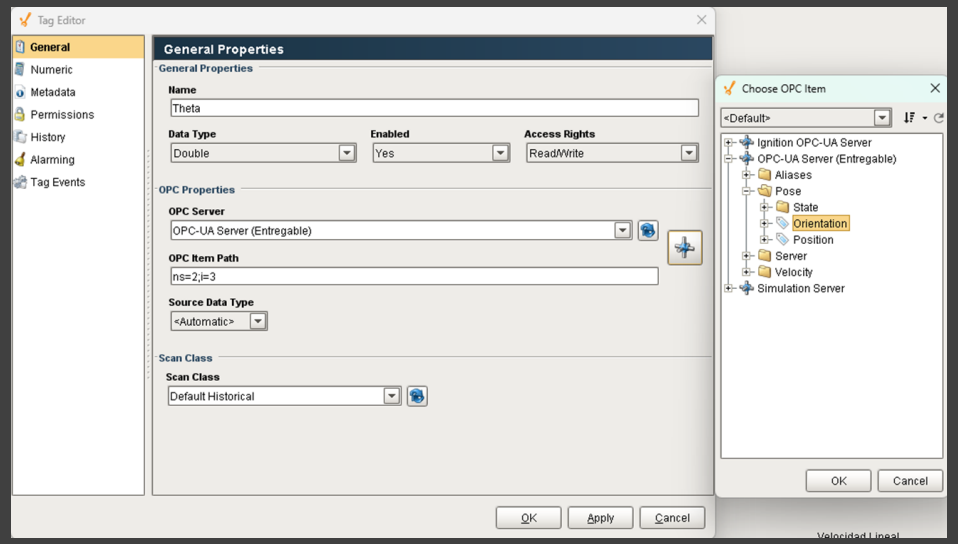
Last_Linear



Position



Theta

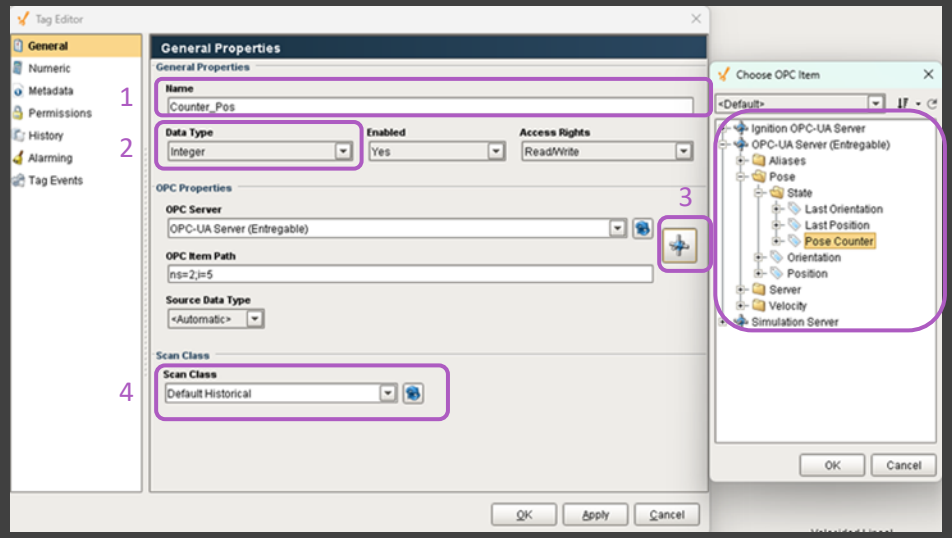


Position-Tags

Los *tags* "*Position*" y "*Theta*" recogen la información relacionada con la posición y la orientación de la tortuga.

Para la creación de estos *tags* se han realizado los mismos pasos comentados anteriormente con un ligero matiz respecto al *tag* "*Position*"; el cual, en vez de ser *Double* como el *tag* "*Theta*" o los anteriores, es de tipo *Double Array*, ya que dicho *tag* recibe la posición en formato [x, y], siendo "*X*" e "*Y*" datos de tipo *Double*.

Counter_Pos

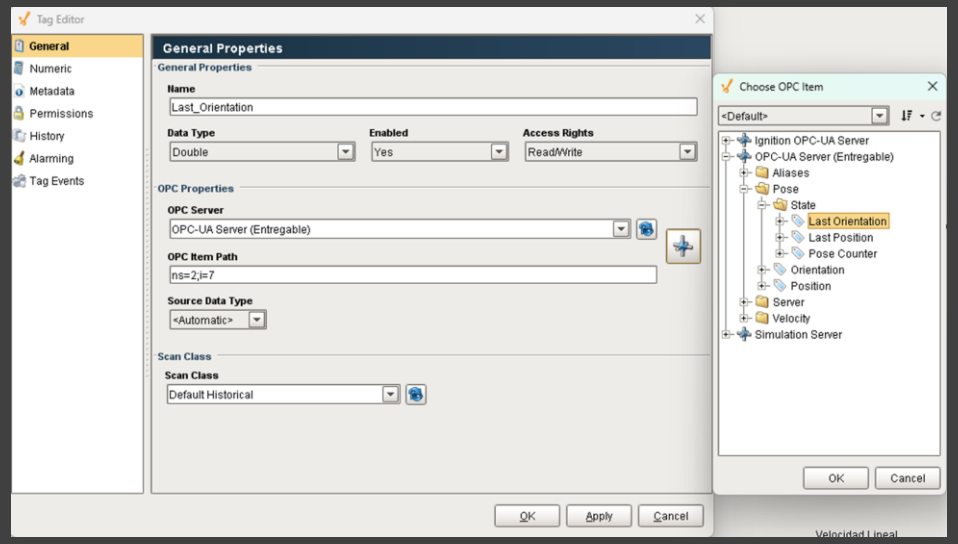


Position-State-Tags

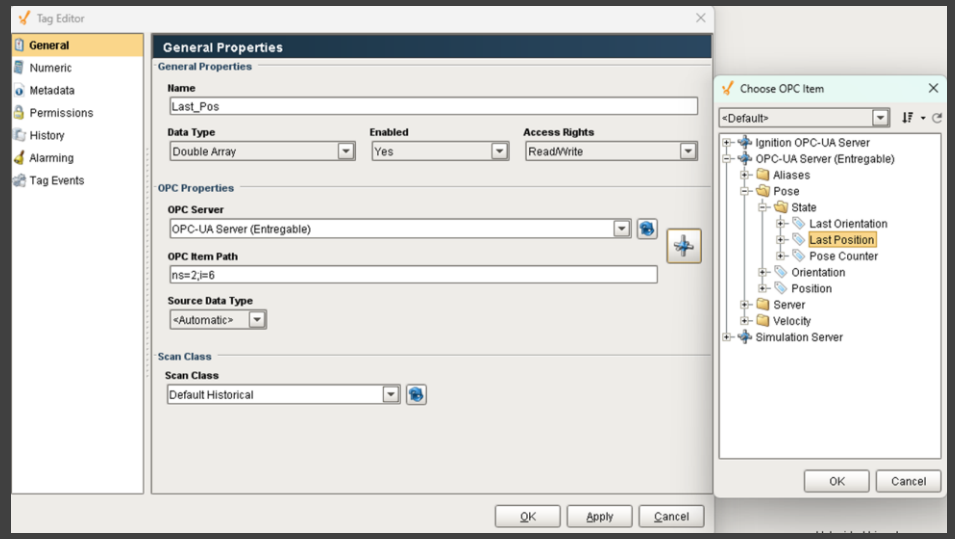
Los *tags* "*Last_Orientation*", "*Last_Pos*" y "*Counter_Pos*" recogen la información relacionada con los últimos datos recibidos respecto a la posición y la orientación de la tortuga, y la cantidad de datos recibidos en total en cuanto a la posición.

Para la creación de estos *tags* se han realizado los mismos pasos comentados anteriormente con un ligero matiz respecto al *tag* "*Counter_Pos*"; el cual, en vez de ser *Double* como el resto, es de tipo *Integer*, ya que dicho *tag* funciona como contador.

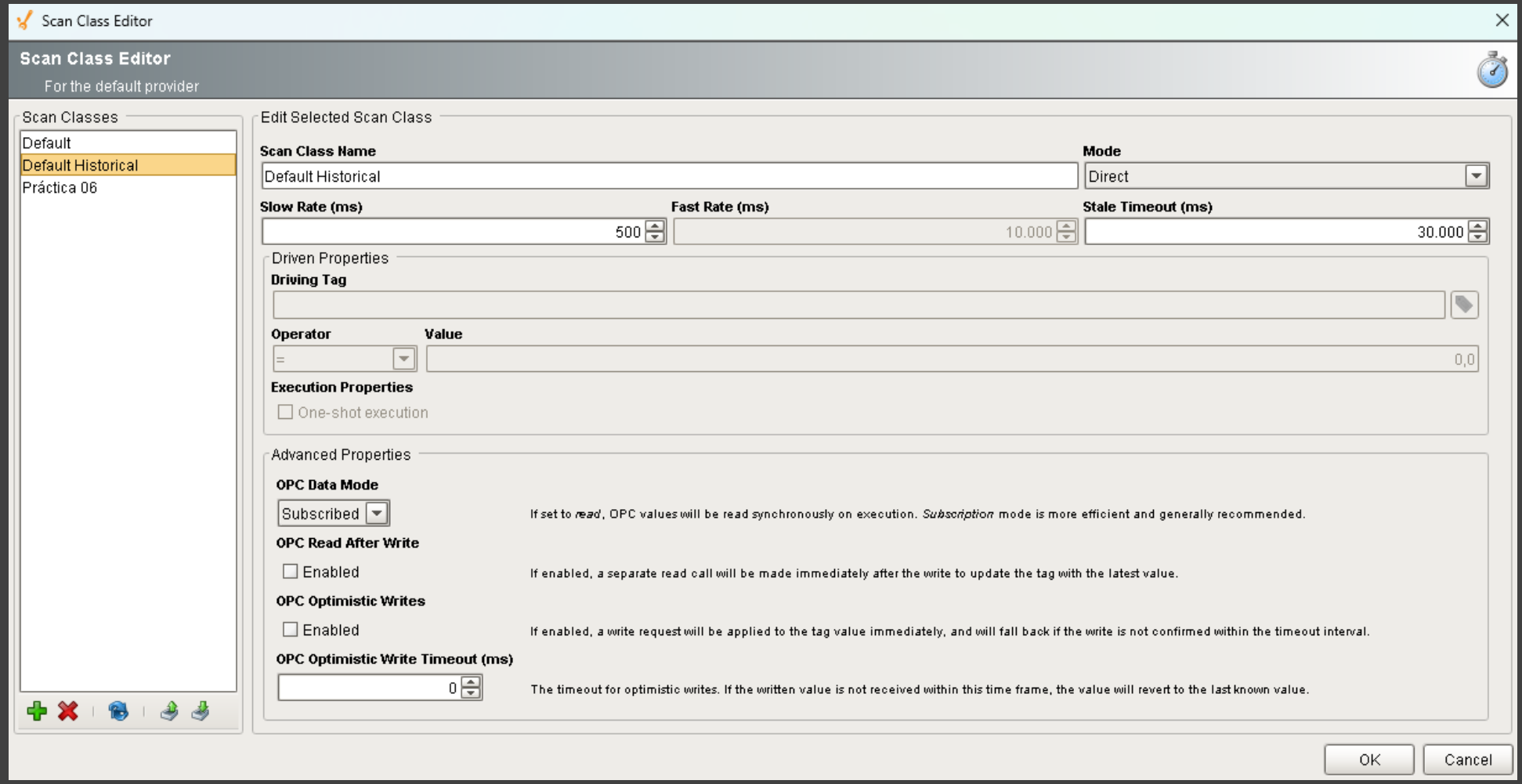
Last_Orientation



Last_Pos



Antes de nada, se establece un *Slow Rate* de 500 ms en el *Scan Class* que se ha designado a las *OPC Tags*.



Dentro de *Counter_Vel -> Tag Events -> Value Events -> Value Changed* se localiza el siguiente código.

Este permite reiniciar los datos almacenados en las *Memory Tags* cada vez que se reinicia el servidor.

El código comprueba el valor anterior y actual del contador y, en caso de que el valor previo sea superior, realiza el reinicio de variables.

```
# Si el valor del contador se reinicia, se reinician todas las memory tags
if previousValue.value > currentValue.value:
    system.tag.write("Trabajo_Asignatura/Distance", 0.0)
    system.tag.write("Trabajo_Asignatura/Angular_Suma", 0)
    system.tag.write("Trabajo_Asignatura/Angular_Average_Speed", 0)
    system.tag.write("Trabajo_Asignatura/Angular_Max_Speed", -10)
    system.tag.write("Trabajo_Asignatura/Angular_Min_Speed", 10)
    system.tag.write("Trabajo_Asignatura/Linear_Suma", 0)
    system.tag.write("Trabajo_Asignatura/Linear_Average_Speed", 0)
    system.tag.write("Trabajo_Asignatura/Linear_Max_Speed", -10)
    system.tag.write("Trabajo_Asignatura/Linear_Min_Speed", 10)
```

Este código se localiza en *Angular_Speed -> Tag Events -> Value Events -> Value Changed*

Se han definido cuatro cálculos, uno para cada *Memory Tag*. En estos cálculos se han realizado los siguientes pasos :

1. Se define una variable auxiliar que guarde el valor actual leído por el *tag*.
2. Se define una variable auxiliar que guarde la dirección del *path* del *Memory Tag* y otra que almacene su valor actual *"x_value"*.
3. Se realiza el cálculo correspondiente
4. Se escribe dicho cálculo en el *Memory Tag*.

Variaciones en cada cálculo (3):

- "*Angular_Suma*" : Se suma el valor del *Memory Tag* con el valor actual.
- "*Angular_Average_Speed*" : Se realiza la división entre el valor de "*Angular_Suma*" y "*Counter_Vel*" obteniendo así la velocidad media.
- "*Angular_Max_Speed*" : Se compara continuamente el valor anterior con el actual y se selecciona el mayor.
- "*Angular_Min_Speed*" : Se compara continuamente el valor anterior con el actual y se selecciona el menor.

```
# Añadimos a Angular_Suma el nuevo valor obtenido
angular_value = currentValue.value                                  1
suma_path = system.tag.read("Trabajo_Asignatura/Angular_Suma")      2
suma_value = suma_path.value
suma_vel = suma_value + angular_value                               3
system.tag.write("Trabajo_Asignatura/Angular_Suma", suma_vel)       4

# Calculamos la velocidad media "Angular_Average_Speed"
# Angular_Suma / Counter_Vel (suma de velocidades entre número de velocidades obtenidas)
suma_value = suma_path.value
counter_path = system.tag.read("Trabajo_Asignatura/Counter_Vel")
counter_value = counter_path.value
average_vel = suma_value / counter_value                            3
system.tag.write("Trabajo_Asignatura/Angular_Average_Speed", average_vel)

# Calculamos la velocidad máxima "Angular_Max_Speed"
max_path = system.tag.read("Trabajo_Asignatura/Angular_Max_Speed")
max_value = max_path.value
if angular_value > max_value :                                      3
        max_vel = angular_value
        system.tag.write("Trabajo_Asignatura/Angular_Max_Speed", max_vel)

# Calculamos la velocidad mínima "Angular_Min_Speed"
min_path = system.tag.read("Trabajo_Asignatura/Angular_Min_Speed")
min_value = min_path.value
if angular_value < min_value :                                      3
    min_vel = angular_value
    system.tag.write("Trabajo_Asignatura/Angular_Min_Speed", min_vel)
```

Este código se localiza en *Linear_Speed -> Tag Events -> Value Events -> Value Changed*

Se realizan los mismos cálculos anteriores con un ligero matiz, en vez de trabajar con el *OPC Tag* de velocidad angular, se trabaja con la velocidad lineal.

```
# Añadimos a Linear_Suma el nuevo valor obtenido
linear_value = currentValue.value                                    1
suma_path = system.tag.read("Trabajo_Asignatura/Linear_Suma")
suma_value = suma_path.value                                         2
suma_vel = suma_value + linear_value                                 3
system.tag.write("Trabajo_Asignatura/Linear_Suma", suma_vel)         4

# Calculamos la velocidad media "Linear_Average_Speed"
# Angular_Suma / Counter_Vel (suma de velocidades entre número de velocidades obtenidas)
suma_value = suma_path.value
counter_path = system.tag.read("Trabajo_Asignatura/Counter_Vel")
counter_value = counter_path.value
average_vel = suma_value / counter_value                             3
system.tag.write("Trabajo_Asignatura/Linear_Average_Speed", average_vel)

# Calculamos la velocidad máxima "Linear_Max_Speed"
max_path = system.tag.read("Trabajo_Asignatura/Linear_Max_Speed")
max_value = max_path.value
if linear_value > max_value :                                        3
        max_vel = linear_value
        system.tag.write("Trabajo_Asignatura/Linear_Max_Speed", max_vel)

# Calculamos la velocidad mínima "Linear_Min_Speed"
min_path = system.tag.read("Trabajo_Asignatura/Linear_Min_Speed")
min_value = min_path.value
if linear_value < min_value :                                        3
    min_vel = linear_value
    system.tag.write("Trabajo_Asignatura/Linear_Min_Speed", min_vel)
```

Este código se localiza en *Position -> Tag Events -> Value Events -> Value Changed*

Como "Position" es un *tag Double Array*, se deben extraer sus componentes "X" e "Y" para representarlas gráficamente de forma individual. Para ello se han realizado los siguientes pasos :

1. Se lee el valor de la posición
2. Separamos sus componentes en dos variables "x" e "y"
3. Escribimos su valor en los *tags* "X", "Y".

Posteriormente, se ha realizado el cálculo de la distancia recorrida por la tortuga. Para ello, se han seguido los siguientes pasos:

1. Se guardan el *path* y el valor del *tag* "Distance" en dos variables auxiliares.
2. Se guardan el *path* y el valor del *tag* "Last_Position" en dos variables auxiliares. Respecto al valor, se realizan los mismos cálculos anteriores para dividir el vector en dos componentes unitarias.
3. Se calcula la distancia empleando la siguiente fórmula :
4. Se escribe el valor en el *tag* "Distance".

```
import math

# Leemos la posición y la dividimos en dos tags distintos
position = currentValue.value                                          1
x = position[0]
y = position[1]                                                        2
system.tag.write("Trabajo_Asignatura/X", x)
system.tag.write("Trabajo_Asignatura/Y", y)                            3

# Calculamos la distancia recorrida
distance_path = system.tag.read("Trabajo_Asignatura/Distance")
distance_value = distance_path.value                                   1
last_position_path = system.tag.read("Trabajo_Asignatura/Last_Pos")
last_position_value = last_position_path.value
last_x = last_position_value[0]
last_y = last_position_value[1]                                        2
distance_total = distance_value + (math.sqrt((x + last_x)**2 + (y + last_y)**2))   3
system.tag.write("Trabajo_Asignatura/Distance", distance_total)        4
```

$$D_{TOTAL} = distance\_value + \sqrt{(x + last_x)^2 + (y + last_y)^2}$$

Este código se localiza en *Last_Pos -> Tag Events -> Value Events -> Value Changed*

Se divide, como se ha realizado anteriormente, el vector posición y se almacena el valor actual de sus componentes en dos *tags* “last_X” y “last_Y”.

```
# Leemos la posición y la dividimos en dos tags distintos
position = currentValue.value
x = position[0]
y = position[1]
system.tag.write("Trabajo_Asignatura/last_X", x)
system.tag.write("Trabajo_Asignatura/last_Y", y)
```

Para la realización de la representación gráfica mediante *Ignition Designer* se han empleado varios objetos :

Label
Permite mostrar texto en una pantalla o ventana.

Numeric Label
Permite mostrar datos numéricos en tiempo real.

Meter
Control visual que se utiliza para mostrar un valor dentro de un rango predefinido.

Easy Chart
Permite representar datos en forma de gráficos en tiempo real (empleando el modo "*Realtime*") para mostrar la evolución de los valores a lo largo del tiempo.

Moving Analog Indicator
Componente visual que se utiliza para mostrar un valor numérico en un formato gráfico similar a un medidor analógico, donde el valor se indica mediante el movimiento de un triángulo a lo largo de una escala.

POSICIÓN

Posición y Orientación Actual

X: 1.97

Y: 10.58

Theta: -1.71

Cantidad de Mediciones

1,799

Distancia Total Recorrida

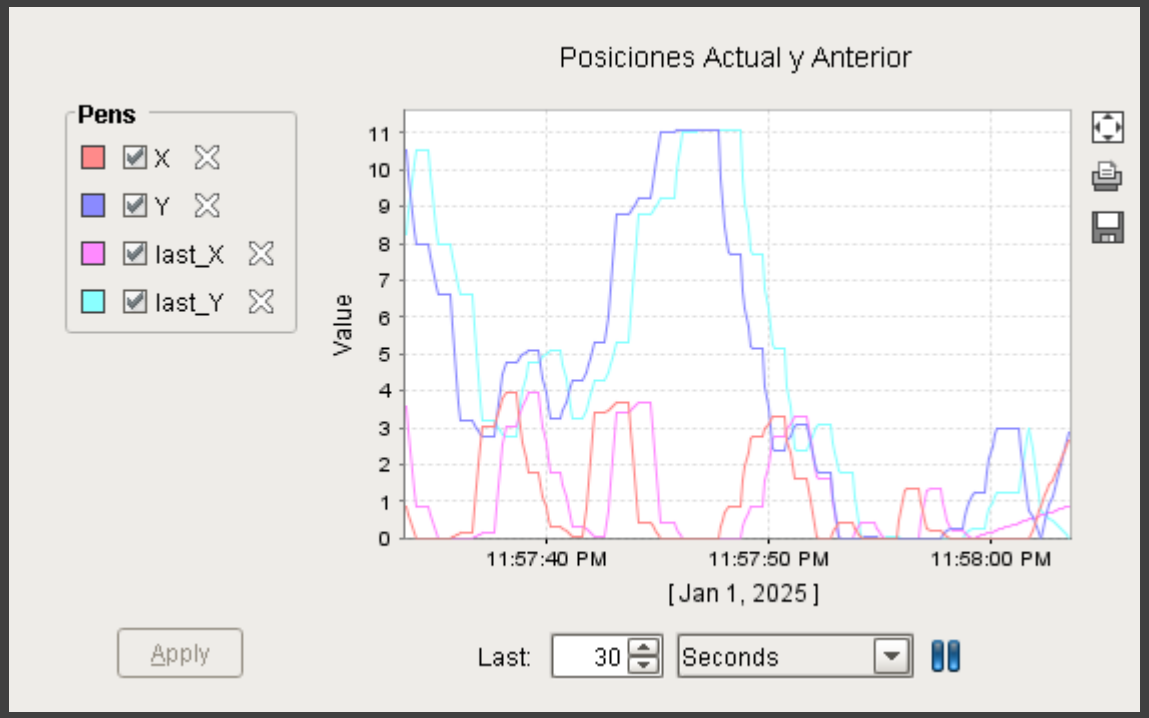
3,601.39

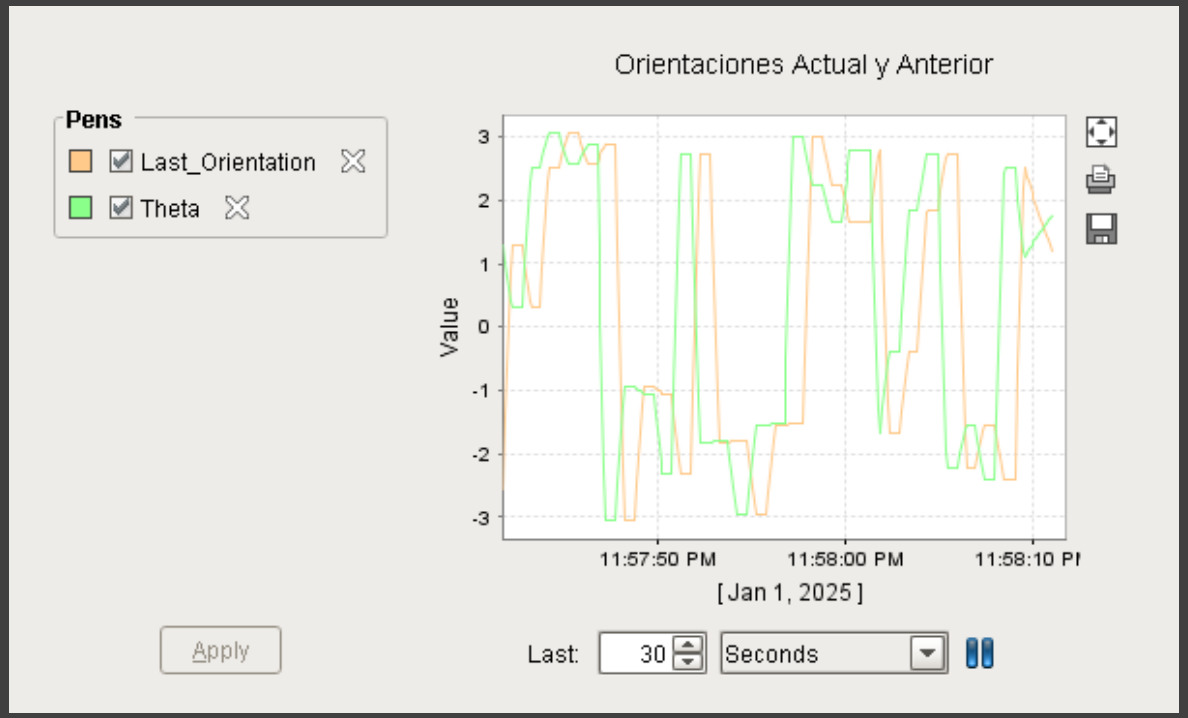
Ultima Posición y Orientación

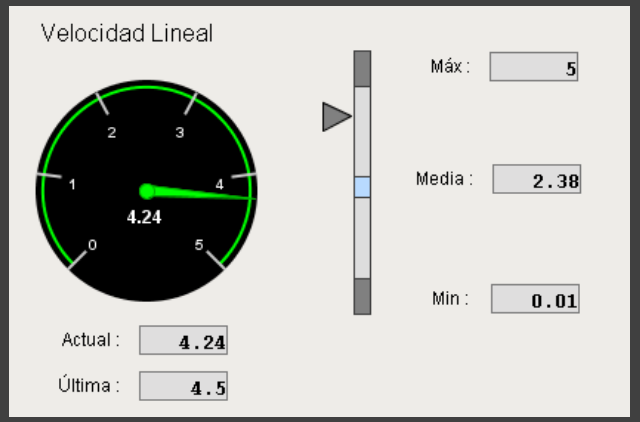
X: 2.67

Y: 11.09

Theta: 2.67







Para realizar la representación de la Velocidad Lineal se han empleado un *Meter* y un *Moving Analog Indicator*.

Common	
Name	Meter
Mouseover Text	
Data	
Value	4,816052024
Overall Low Bound	0,0
Overall High Bound	5,0
Appearance	
Units	
Tick Size	1,0

Common	
Name	Moving Analog Indicator
Data	
Range High	5,998428004
Range Low	-0,993158982
Process Value	1.488731414
High Alarm	4.998428004
Desired High	2.699516478
Desired Low	2.099516478
Low Alarm	0.006841018

Meter

Overall (Low/High) Bound : Rango [0, 5]

Tick Size : Movimiento de 1 en 1

Value : "*Linear_Speed*"

Moving Analog Indicator

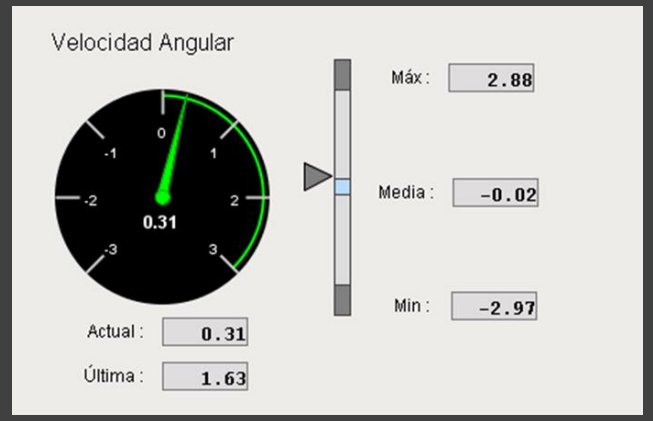
Range (Low/High) : Rango ["Linear_Min_Speed" - 1, "Linear_Max_Speed" + 1]

Process Value : "Linear_Speed"

High Alarm : "Linear_Max_Speed"

Low Alarm : "Linear_Min_Speed"

Desired (Low/High) : "Linear_Average_Speed" + [-0.3, +0.3]



Common	
Name	Meter 1
Mouseover Text	
Data	
Value	-1,96769668
Overall Low Bound	-3,0
Overall High Bound	3,0
Appearance	
Units	
Tick Size	1,0

Para realizar la representación de la Velocidad Angular también se han empleado un *Meter* y un *Moving Analog Indicator*.

Common	
Name	Moving Analog Indicator
Data	
Range High	3,875110743
Range Low	-3,969161203
Process Value	-0.123510567
High Alarm	2.875110743
Desired High	0.27331679
Desired Low	-0.32668321
Low Alarm	-2.969161203

Meter

Overall (Low/High) Bound : Rango [-3, 3]

Tick Size : Movimiento de 1 en 1

Value : "*Angular_Speed*"

Moving Analog Indicator

Range (Low/High) : Rango ["Angular_Min_Speed" - 1, "Angular_Max_Speed" + 1]

Process Value : "*Angular_Speed*"

High Alarm : "*Angular_Max_Speed*"

Low Alarm : "*Angular_Min_Speed*"

Desired (Low/High) : "Angular_Average_Speed" + [-0.3, +0.3]

ÍNDICE
INTRODUCCIÓN
ORIGEN DE DATOS Y PUENTE ROS2-MQTT
PASO A OPC-UA
VISIÓN GENERAL 1
VISIÓN GENERAL 2

CIERRE

BIBLIOGRAFÍA

- Para la realización de este trabajo se han empleado de referencia las distintas prácticas de la asignatura realizadas durante el curso.

APLICACIONES

- VMWare
- ROS2
- Mosquitto
- MQTTX
- OPC-UA Browser
- Ignition
- Ignition Designer