Guía de usuario

Levantar Docker en Raspberry Pi:

1. Update de repositorios y upgrade del sistema.

```
sudo apt-get update && sudo apt-get upgrade -y
```

2. Instala Docker

```
curl -sSL https://get.docker.com | sh
```

3. Agrega usuario al grupo de docker, en “usuario” se pone el nombre del usuario de la raspberry.

```
sudo usermod -aG docker [usuario]
```

4. Verifica que el usuario está en el grupo

```
groups [usuario]
```

5. Reinicia la raspberry

```
sudo reboot now
```

6. Instala pip 3 (en caso de que no lo tengas)

```
sudo apt install python3-pip
```

7. Instala docker-compose

```
sudo pip3 install docker-compose
```

8. Configura docker para que se ejecute al arranque del OS

```
sudo systemctl enable docker
```

9. Crea una carpeta para el broker.
10. Crea un manifiesto (debe llamarse docker-compose.yml)

```
nano docker-compose.yml
```

11. Escribe o copia el texto de abajo en el archivo.

```
version: '3'
services:
  Mosquitto:
    container_name: Mosquitto
    image: 'eclipse-mosquitto:2.0.14'
    ports:
      - '1883:1883'
    command: mosquitto -c /mosquitto-no-auth.conf
```

```
    tty: true
```

12. En una nueva terminal navega hasta la carpeta en donde está ubicado el archivo docker-compose.yml que deseas.
13. Ejecuta el manifiesto.

```
docker-compose up -d
```

14. Para detener el contenedor.

```
docker-compose down
```

Levantar Docker en Ubuntu:

1. Descargar e instalar Docker

```
sudo apt install docker.io
```

2. Agregar el usuario al grupo de Docker, escribir en “usuario” el nombre del usuario de la máquina.

```
sudo usermod -aG docker [usuario]
```

3. Verifica que el usuario está dentro del grupo

```
groups [usuario]
```

4. Reinicia la máquina virtual

```
sudo reboot now
```

5. Instala docker-compose

```
sudo curl -L
"https://github.com/docker/compose/releases/download/1.26.0/dock
er-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-
compose
```

6. Cambia los permisos de ejecución de docker-compose

```
sudo chmod +x /usr/local/bin/docker-compose
```

7. Verifica la instalación consultando la versión

```
docker-compose --version
```

8. Crea una carpeta para el broker.

9. Crea un manifiesto (debe llamarse docker-compose.yml)

```
nano docker-compose.yml
```

10. Escribe o copia el texto de abajo en el archivo.

```
version: '3'
services:
  Mosquitto:
    container_name: Mosquitto
    image: 'eclipse-mosquitto:2.0.14'
    ports:
      - '1883:1883'
    command: mosquitto -c /mosquitto-no-auth.conf
    tty: true
```

11. En una nueva terminal navega hasta la carpeta en donde está ubicado el archivo docker-compose.yml que deseas.

12. Ejecuta el manifiesto.

```
docker-compose up -d
```

13. Para detener el contenedor.

```
docker-compose down
```

A continuación, se siguieron los siguientes pasos para poder implementar la conexión a MQTT:

1. Para implementar el protocolo de transmisión de datos MQTT en el lenguaje de programación Python se necesita instalar la librería paho-mqtt. Cabe recalcar que la versión de la librería fue la siguiente: *"paho-mqtt 1.6.1"*. Para esto se ejecuta el siguiente comando en la terminal:

```
pip install paho-mqtt
```

2. Una vez instalada la librería, se creó el nodo concentrador con el que se publica la información de los tópicos de ROS 2 en un tópico de MQTT. Para poder crear el nodo primero se creó un nuevo paquete de ROS2:

```
. ~/ros2_foxy/install/local_setup.bash
cd ~/ros2_foxy/
src
ros2 pkg create —build-type ament_python com_mqtt --dependencies
rclpy std_msgs custom_msgs geometry_msgs
```

3. Ya que se creó el paquete con las dependencias necesarias, se crea en la carpeta del mismo nombre del paquete, el archivo *"lisros_pubmqtt.py"*. Dentro del nodo concentrador se crea el código con la estructura de un nodo suscriptor de ROS y se agregan las siguientes líneas de código para poder instanciar la conexión al broker y poder publicar la información obtenida de los tópicos de ROS2:

```
import paho.mqtt.client as mqttClient

def on_connect(client, userdata, flags, rc):
    """Función que establece la conexión

    """
    if rc==0:
        print("Conectado al broker")
        global Connected
        Connected = True
    else:
        print("Falla en la conexión")
    return
self.Connected = False
self.broker_address= "10.48.60.51"
self.port= 1883
self.tag1 = "/Equipo1/Temperatura"
self.tag2 = "/Equipo1/Humedad"
self.tag3 = "/Equipo1/CO2"
self.client=mqttClient.Client("identificador")
self.client.on_connect = on_connect #agregando la función
self.client.connect(self.broker_address, self.port)
self.client.loop_start() #inicia la instancia
val1=json.dumps({"Temperatura":
self.dato.temperatura,"Humedad":
     self.dato.humedad,"CO2": self.dato.bateria})
print(self.tag1,val1,'\n')
self.client.publish(self.tag1,val1,qos=0)
```

4. Con el objetivo de tener un mejor manejo de ejecución, se crea un archivo launch dentro de la carpeta launch *"lisros_pubmqtt_launch.launch.py"*:

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

def generate_launch_description():
    return LaunchDescription([
        Node(
            package='listener',
            executable='lisros_pubmqtt',
            output='screen'),
        ])
```

5. Se modifica el setup.py agregando dos líneas en las que se especifica el uso de archivos launch y la existencia del nodo concentrador:

```
(os.path.join('share',                                  package_name),
glob('launch/*.launch.py'))

entry_points={
        'console_scripts': [
            'lisros_pubmqtt = listener.lisros_pubmqtt:main',
        ],
```

```
    },
```

6. Ya que esté listo el archivo launch se compila el paquete:

```
colcon build --packages-select com_mqtt
. ~/ros2_foxy/install/local_setup.bash
```

Como una opción para poder almacenar los datos se utilizó MongoDB, esto para poder realizar un histórico de los datos que se vayan recibiendo. Al ser una base de datos no relacional tiene como característica que cada bucket o colección no cuenta con una estructura definida por lo que esto facilita que nuevos datos entren a la colección. Por otro lado, al tener diferentes visualizadores de datos estos facilitan el acceso que se tiene para poder ver el contenido de cada colección. Además, cuenta con su lenguaje de búsqueda derivado del de la base de datos relacional SQL, por lo que esto ayuda a acceder a datos específicos en cualquier lenguaje de programación: Python, Java, C#. Para poder instalar Mongo DB se siguieron los siguientes pasos:

1. Para poder descargar MongoDB en Windows o en Ubuntu se ingresa al siguiente link:

   https://www.mongodb.com/try/download/community

2. A continuación, se escoge el sistema operativo en el que se encuentra y se descarga el zip con el gestor de MongoDB (imagen 1).

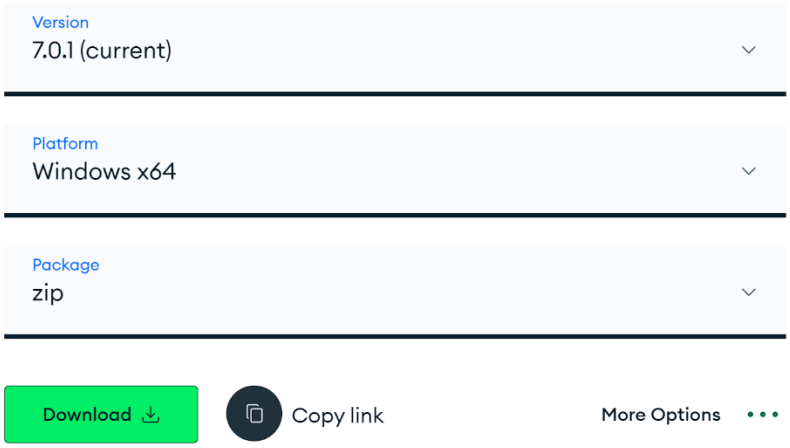

Imagen 1. Página de descarga (Download MongoDB Community Server)

3. Una vez que se ha descargado el zip, se descomprime en la carpeta que se desee. A continuación, en una terminal se ejecuta el siguiente comando:

```
mkdir -p /data/db
```

4. Dentro de la carpeta que se eligió se accede a la carpeta descomprimida y dentro de esta, se accede a la carpeta bin. Dentro de la carpeta bin se ejecuta el siguiente comando:

```
./mongod
```

Una vez que se tiene el servidor local de Mongo DB, se descarga un visualizador para la base de datos. El visualizador que se utilizó fue Mongo DB Compass, este es un visualizador bastante amigable en el que se pueden realizar búsquedas de datos específicos, así como tener un histograma de cada dato ingresado. Para poder instalar Mongo DB Compass se siguieron los siguientes pasos:

1. Se accedió a la siguiente liga para poder descargar el visualizador:

   https://www.mongodb.com/try/download/compass

2. Una vez en la página de descarga se localiza el cuadro de descarga que se presenta en la imagen 2 y se descarga el archivo ejecutable para el sistema operativo necesario (cabe recalcar que el visualizador no puede ser instalado en una máquina virtual).

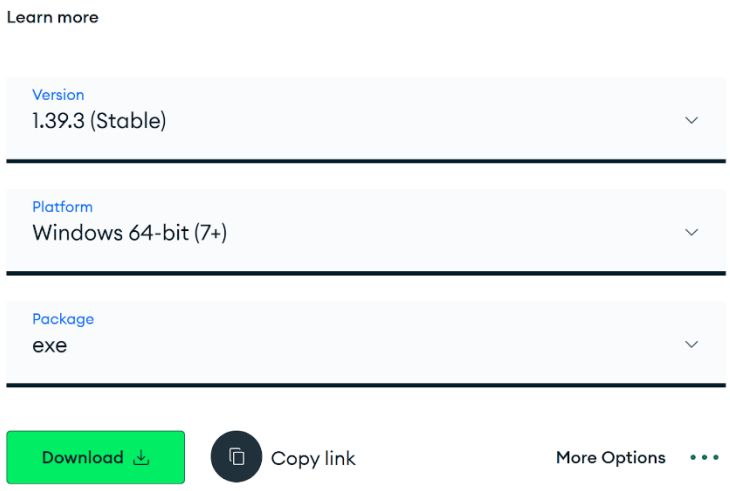

Imagen 2. Página de descarga (MongoDB Compass Download)

Una vez establecido el nodo concentrador que establece la comunicación de un punto a otro, se necesita crear el nodo que recibe la información del tópico de MQTT al que se le publica la información (“suscriptormqttmongo.py”). Para este programa se implementó la base de datos no relacional Mongo DB, con la cual se puede acceder y guardar datos a través de Python. Los pasos que se siguieron para poder crear este código se describen a continuación.

1. Para utilizar la librería de MQTT junto con Python se instala la librería como se realizó con anterioridad.
2. Para poder utilizar los recursos que proporciona Mongo DB se debe instalar la librería pymongo. Se utilizó la librería “pymongo”, la versión que se utilizó fue la siguiente: pymongo 4.5.0.

```
pip install pymongo
```

3. Se escribe el código para poder suscribirse al tópico de MQTT en el que se transmite la información (“suscriptormqttmongo.py”). A continuación se inserta el siguiente código para poder insertar un json en la colección de la base de datos que se requiera:

```
client = pymongo.MongoClient("mongodb://localhost:27017/")
db = client["Datos"]
collection = db["prueba"]
data = json.loads(payload)
resultado = collection.insert_one(data)
```

4. Con el código desarrollado, se implementó en el mismo código la librería previamente descrita “paho-mqtt”, esto para poder tener un suscriptor al tópico de MQTT en el que se están enviando los datos del rover. Dentro del código se escriben las siguientes líneas de código para poder establecer la conexión al broker deseado, suscribirse al tópico deseado, y poder transformar el dato a formato json.

```
def on_connect(client, userdata, flags, rc):
    """Función que establece la conexión
    """
    if rc==0:
        print("Conectado al broker")
        global Connected
        Connected = True
    else:
        print("Falla en la conexión")
    return

def on_message(client,userdata,message):
    """Función que recibe los mensajes del broker
    """
    payload = message.payload.decode("utf-8")
    data = json.loads(payload)
    resultado = collection.insert_one(data)
    print("ID del documento insertado:", resultado.inserted_id)
    return

Connected = False
broker_address="10.48.60.51"
port= 1883 #puerto por defecto de MQTT
tag = "/Equipo1/#"  #tag, etiqueta o tópico
client1=mqttClient.Client()
client1.on_connect=on_connect
client1.on_message=on_message
client1.connect(broker_address,port)
client1.loop_start()

while Connected != True:
    time.sleep(0.1)
    client1.subscribe(tag)
```

Con el código del suscriptor de MQTT y que almacena datos en Mongo DB, se necesita un visualizador en donde se pueda observar en tiempo real la información que se obtiene de

Mongo DB. Se utilizó la librería “Custom tkinter” para poder tener una interfaz gráfica con la que el usuario pudiera interactuar. Para desarrollar la interfaz se realizaron los siguientes pasos:

1. Para utilizar la librería de Python custom tkinter, específicamente se utilizó la versión 5.2.0, se instala la librería a través de la terminal con los siguientes comandos:

```
pip install customtkinter
```

2. Una vez que se ha instalado la librería se define la estructura general de la interfaz definiendo un aspecto modo nocturno y una ventana de 700x650:

```
customtkinter.set_appearance_mode("dark")
customtkinter.set_default_color_theme("dark-blue")
root = customtkinter.CTk()
root.geometry("700x650")
```

3. A continuación se utiliza las funciones CtkImage y CtkLabel para poder desplegar la imagen con el logo de Quantum Robotics:

```
my_image =
customtkinter.CTkImage(light_image=Image.open("Interfaz/QLogo.png"),dar
k_image=Image.open("Interfaz/QLogo.png"),size=(70, 70))

image_label = customtkinter.CTkLabel(root, image=my_image,text="")
image_label.pack(padx=1, pady=1)
```

4. Para facilidad de visualización se dividió la ventana en distintas pestañas que presentan distintas secciones de la interfaz. Utilizando la función CtkTabview se crearon 5 distintas pestañas: “Datos”, “Envio”, “Graphs”, “Maps”, “Camera”, “Datos”.

```
tabView = customtkinter.CTkTabview(master=root)
tabView.pack(padx=10, pady=10, expand=True)

tabView.add("Datos")
tabView.add("Envio")
tabView.add("Graphs")
tabView.add("Maps")
tabView.add("Camera")

tabView.set("Datos")
```

5. Para la pestaña de “Datos” se implementó la función CtkLabel y CTkProgressBar para poder visualizar en texto los datos de Temperatura, Humedad, CO2 y le nivel de batería. Esto se hizo a través del siguiente código:

```
#Titulos de datos del robot
labelco2 = customtkinter.CTkLabel(master=tabView.tab("Datos"), text="CO2:")
labelco2.grid(row=0, column=0, pady=12, padx=10)

labelhum = customtkinter.CTkLabel(master=tabView.tab("Datos"), text="Humedad:")
labelhum.grid(row=1, column=0, pady=12, padx=10)
```

```
labeltem = customtkinter.CTkLabel(master=tabView.tab("Datos"),
text="Temperatura:")
labeltem.grid(row=2, column=0, pady=12, padx=10)

labelbat = customtkinter.CTkLabel(master=tabView.tab("Datos"), text="Bateria:")
labelbat.grid(row=3, column=0, pady=12, padx=10)

#Labels con datos del robot
label1 = customtkinter.CTkLabel(master=tabView.tab("Datos"), text="")#,
text_font=("Roboto", 24))
label1.grid(row=0, column=1, pady=12, padx=10)

label2 = customtkinter.CTkLabel(master=tabView.tab("Datos"), text="")
label2.grid(row=1, column=1, pady=12, padx=10)

label3 = customtkinter.CTkLabel(master=tabView.tab("Datos"), text="")
label3.grid(row=2, column=1, pady=12, padx=10)

#Barra de nivel de bateria
bar =
customtkinter.CTkProgressBar(master=tabView.tab("Datos"),orientation="horizonta
l")
bar.grid(row=3, column=1,pady=10, padx=10)
```

6. Para poder actualizar los datos de los label's se utiliza una función que se ejecuta cada 2 segundos para poder obtener los datos de la base Mongo DB y obtener cada dato para presentar en la interfaz:

```
def subs():
    # Retrieve the most recent document from the collection
    most_recent_document = collection.find_one({}, sort=[('_id',
DESCENDING)])
    del most_recent_document['_id']
    co2 = round(most_recent_document['CO2'],1)
    hum = most_recent_document['Humedad']
    tem = round(most_recent_document['Temperatura'],2)
    batery = round(most_recent_document['Bateria'],2)
    label1.configure(text=co2)
    label2.configure(text=hum)
    label3.configure(text=tem)
    bar.set(batery)
      root.after(2000, subs)
```

7. Para la pestaña de "Envio", se definen cajas de entrada de texto y un botón para poder transmitir el mensaje vía MQTT. Se utilizaron las funciones de CtkEntry y CtkButton para desplegar los gráficos en la ventana:

```
#Entradas de datos para enviar al robot
entry1 = customtkinter.CTkEntry(master=tabView.tab("Envio"),
placeholder_text="Coordenada en x")
entry1.pack(pady=12, padx=10)
entry2 = customtkinter.CTkEntry(master=tabView.tab("Envio"),
placeholder_text="Coordenada en y")
entry2.pack(pady=12, padx=10)
#Boton para enviar datos por MQTT
```

```
button = customtkinter.CTkButton(master=tabView.tab("Envio"),
text="Send", command=msg)
button.pack(padx=12, pady=10)
```

8. A continuación, se define la función “msg” para poder enviar los mensajes en formato json vía MQTT.

```
def msg():
    x = int(entry1.get())
    y = int(entry2.get())
    val1=json.dumps({"PosicionX": x,"PosicionY": y})
    print(tag1,val1)
    client.publish(tag1,val1,qos=0)
    root.after(2000, subs)
```

9. Para poder establecer comunicación con el broker y enviar al tópico que se desea, se añade el siguiente código:

```
def on_connect(client, userdata, flags, rc):
    """Función que establece la conexión
    """
    if rc==0:
        print("Conectado al broker")
        global Connected
        Connected = True
    else:
        print("Falla en la conexión")
    return

Connected = False
broker_address= "10.48.60.51" #dirección del Broker
port= 1883 #puerto por defecto de MQTT
tag1 = "/Robot1"  #tag, etiqueta o tópico
client = mqttClient.Client() #instanciación
client.on_connect = on_connect #agregando la función
client.connect(broker_address, port)
client.loop_start() #inicia la instancia
```

10. Para la pestaña de “Graphs” se utiliza un ComboBox para desplegar diferentes opciones y se ejecute un proceso dependiendo de la opción elegida. Además se utiliza la función “FigureCanvasTkAgg” para mostrar una gráfica de la librería matplotlib. Al código se agregan las siguientes líneas:

```
#Opciones de graficas.
optionmenu = customtkinter.StringVar(value="")

combobox_graf =
      customtkinter.CTkComboBox(master=tabView.tab("Graphs"),
      values=["Humedad","Temperatura","CO2", "Vel Lineal", "Vel
      Angular"],variable=optionmenu)
      combobox_graf.pack(pady=12, padx=10)
```

```
#Grafica de datos
fig, ax = plt.subplots()
canvas = FigureCanvasTkAgg(fig,master=tabView.tab("Graphs"))
canvas.get_tk_widget().pack()
```

11. A continuación se le agrega a la función “subs()” los comandos para poder obtener los últimos 5 datos y actualizar la gráfica con los datos que se especifican. Se agregan las siguientes líneas de código:

```
    pipeline = [
    {
        '$sort': {'_id': -1}
    },
    {
        '$limit': 5
    }
    ]
    result = list(collection.aggregate(pipeline))
    data = []
    for document in result:
        del document['_id']
        opcion = combobox_graf.get()
        if opcion == "Humedad":
            data.append(document['Humedad'])
            ax.set_ylim([35, 85])
        elif opcion == "Temperatura":
            data.append(document['Temperatura'])
            ax.set_ylim([10, 40])
        elif opcion == "CO2":
            data.append(document['CO2'])
            ax.set_ylim([0, 100])
        elif opcion == "Vel Lineal":
            data.append(document['VelLin'])
            ax.set_ylim([0, 15])
        elif opcion == "Vel Angular":
            data.append(document['VelAng'])
            ax.set_ylim([-1, 1])

    data.reverse()
    ax.clear()
    ax.plot(data)
    ax.set_xlabel('X-axis')
    ax.set_ylabel('Y-axis')
    ax.set_title(combobox_graf.get())
    canvas.draw()
```

12. Para la pestaña de “Maps” se utilizó un botón el cual activa una función llamada “mapas()”:

```
#Mensaje de pestania mapas
labelmap = customtkinter.CTkLabel(master=tabView.tab("Maps"), text="Haga click
para mostrar ubicacion")
labelmap.pack(pady=12, padx=10)

#Boton para mostrar mapa
button = customtkinter.CTkButton(master=tabView.tab("Maps"), text="Show",
command=mapas)
button.pack(padx=12, pady=10)
```

13. A continuación se realiza la función “mapas()” con la cual se utiliza la librería “subprocess” esto para ejecutar el código llamado “mapa.py” con el cual despliega un mapa con la ubicación actual:

```
import subprocess
def mapas():
    subprocess.Popen(["python", "mapa.py"])
    root.after(2000, subs)
```

14. Se desarrolla el código para poder desplegar el mapa con la ubicación actual del rover. Para esto se descarga la librería “tkintermapview”, específicamente la versión “1.29”. Se ejecuta el siguiente comando para instalar la librería:

```
pip install tkintermapview
```

15. Una vez instalada la librería, se crea una ventana de custom tkinter, dentro de esta ventana se despliega un mapa con la ubicación del campus. Esto se logra utilizando la función “tkintermapview.TkinterMapView()”:

```
customtkinter.set_appearance_mode("dark")
customtkinter.set_default_color_theme("dark-blue")
root = customtkinter.CTk()
root.geometry("700x650")

map_widget = tkintermapview.TkinterMapView(root,width=800,
height=600, corner_radius=0)
map_widget.place(relx=0.5, rely=0.5,
anchor=customtkinter.CENTER)

#Set marker
marker_2 = map_widget.set_marker(19.597450907484646, -
99.22712693520945, text="CEM")

#Set coordinates
map_widget.set_position(19.597450907484646, -99.22712693520945)
map_widget.set_zoom(19)

root.mainloop()
```

16. A continuación se desarrolla una función llamada “subs()” en la cual se pueda actualizar la posición actual en el mapa. Esto se logra obteniendo la posición que se haya registrado en Mongo DB (una vez que se haya instanciado la conexión a la base de datos):

```
def subs():
    most_recent_document = collection.find_one({}, sort=[('_id', DESCENDING)])
    del most_recent_document['_id']
    coorx = most_recent_document['CoordenadaX']
    coory = most_recent_document['CoordenadaY']
    map_widget.set_position(coorx,coory) #CEM
    marker_2.set_position(coorx,coory)


    root.after(1000, subs)
```

17. Finalmente, para poder desplegar video se utiliza la librería de subprocess con la cual se manda a llamar a otro programa de Python para ejecutarse. Para esto se define el siguiente código el cual define el Label de la pestaña y el botón para mostrar el video:

```
#Mensaje de pestania camara
labelmap = customtkinter.CTkLabel(master=tabView.tab("Camera"),
text="Haga click para mostrar video")
labelmap.pack(pady=12, padx=10)

#Boton para mostrar imagen
button = customtkinter.CTkButton(master=tabView.tab("Camera"),
text="Show", command=video)
button.pack(padx=12, pady=10)
```

18. El botón manda a llamar la función “video()” la cual ejecuta el programa de Python para visualizar la cámara por mensajes de MQTT:

```
def video():
    subprocess.Popen(["python", "camaramqtt.py"])
    root.after(2000, subs)
```

19. El programa “camaramqtt.py” se conecta a MQTT al tópico de /Equipo1/Camara, para desplegar el mensaje en formato de video se importa la librería de Opencv, específicamente la versión “opencv-python 4.7.0.72”. Para utilizar la librería de opencv se corre el siguiente comando en la terminal:

```
pip install open-cv
```

20. Una vez instalada la librería de OpenCV, dentro de la función para suscribirse al mensaje de MQTT, se colocan las siguientes líneas de código para poder decodificar el mensaje enviado:

```
def on_message(client, userdata, msg):
    global frame
    img = base64.b64decode(msg.payload)
    npimg = np.frombuffer(img, dtype=np.uint8)
    frame = cv.imdecode(npimg, 1)
```

21. A continuación dentro de un ciclo while se escribe el código para transformar la imagen en formato openCV de BGR a RGB. Una vez transformado, se convierte de RGB al formato PIL Image. Por último se transforma de PIL Image a PhotoImage para desplegar la imagen en una ventana de CustomTkinter:

```
while True:
    rgb_frame = cv.cvtColor(frame, cv.COLOR_BGR2RGB)
    pil_image = Image.fromarray(rgb_frame)

    ctk_image = ImageTk.PhotoImage(image=pil_image)
    image_label.configure(image=ctk_image)
    image_label.image = ctk_image
```

Además de todo esto, en la estación de control se necesita un nodo que se encargue de recibir los datos enviados por el nodo concentrador y una vez teniendo los datos correspondientes, tenemos que desempaquetar el archivo json obtenido para convertirlo a datos de ROS2 y que estos se puedan usar para publicarlos al rover o directamente en este mismo entorno.

Por lo tanto, la construcción del nodo que recibe vía MQTT se construyó de la siguiente manera:

1. Ya que tenemos instaladas las librerías necesarias y en paquete creado, lo primero que se tiene que hacer es crear el nuevo nodo con el nombre *"lismqtt.py"* donde importamos esas librerías a utilizar.

```
import rclpy
import numpy as np
# import the ROS2 python libraries
from rclpy.node import Node
#from std_msgs.msg import String
from custom_msgs.msg import Sensores
from rclpy.qos import ReliabilityPolicy, QoSProfile
# importamos librerias MQTT
import paho.mqtt.client as mqttClient
import time
import json
```

2. Basándonos en un archivo Python para suscriptor de MQTT, comenzaremos con la creación de la función que nos instancia la conexión al broker.

```
def on_connect(client, userdata, flags, rc):
    """Función que establece la conexión
    """
    if rc==0:
        print("Conectado al broker")
        global Connected
        Connected = True
    else:
        print("Falla en la conexión")
    return
```

3. Posteriormente, se genera una función *"on_message"* que es aquella que recibirá los mensajes provenientes del broker y los guarda en un archivo JSON que es el que después vamos a desempaquetar para poder usar esos datos en nuestro entorno de ROS2.

```
def on_message(client,userdata,message):
    """Función que recibe los mensajes del broker
    """
    payload = message.payload.decode("utf-8")
    data = json.loads(payload)
    with open("data.json", "w") as file:
        json.dump(data, file, indent=4)
    print("Archivo JSON recibido y guardado")
    print("Mensaje - {}:{}".format(message.topic, message.payload))
```

4. Posteriormente, se crea una clase en donde dentro de la función *"def __init__"* se inicializará el nodo, se definirá el tipo de dato y todo lo necesario para poder conectarnos y comunicarnos con el broker para recibir los datos solicitados. Y en una función aparte llamada *"motion"* se creará el suscriptor que se estará ejecutando cada cierto tiempo que se define por el *"timer_period"* y cuando se detecte que el usuario detiene el programa, se imprime un mensaje de notificación y detendrá la conexión con el broker.

```
class ListenerMQTT(Node):

    def __init__(self):
        # call the class constructor
        super().__init__('lismqtt')
        # create the publisher object
        # define the timer period for 0.5 seconds
        self.timer_period = 0.5
        # define the variable to save the received info
        self.dato = Sensores()
        self.timer = self.create_timer(self.timer_period, self.motion)
        self.Connected = False
        self.broker_address="10.48.60.51"
        self.port= 1883 #puerto por defecto de MQTT
        self.tag = "/Robot1"  #tag, etiqueta o tópico

        self.client1=mqttClient.Client("cliente")
        self.client1.on_connect=on_connect
        self.client1.on_message=on_message
        self.client1.connect(self.broker_address,self.port)
        self.client1.loop_start()

    def motion(self):
        self.client1.subscribe(self.tag)
        try:
            time.sleep(0.5)
        except KeyboardInterrupt:
            print("Recepción de mensajes detenida por el usuario")
            self.client1.disconnect()
            self.client1.loop_stop()
```

5. Lo último por hacer es crear la función *"main"* que ejecutará todo el código. Para facilitar la ejecución del código, se crea un launcher dentro de la carpeta launch que lleva por nombre *"lismqtt_launch.launch"*:

```
from launch import LaunchDescription
from launch_ros.actions import Node
def generate_launch_description():
    return LaunchDescription([
        Node(
            package='listener',
            executable='lisros_pubmqtt',
            output='screen'),
        ])
```

Una vez que tenemos el mensaje MQTT guardado en un archivo JSON, necesitamos desempaquetarlo para poder usar esos datos, como anteriormente se menciona.

1. Para esto, se creará un nodo única y exclusivamente encargado de leer JSON y convertirlos a datos o mensajes de ROS. Por lo tanto, primero se crea el nodo de la misma manera que todos los anteriores. Comenzamos importando las librerías a utilizar.

```
import rclpy
from rclpy.node import Node
from geometry_msgs.msg import Point
import json
```

2. Una vez hecho esto, empezamos a formular el nodo con la misma estructura de los anteriores empezando por crear la clase. En la función inicial se define el nombre del nodo, el tipo de dato a utilizar, el nombre del archivo JSON y el publicador a algún tópico. Además, creamos la función donde se desempaqueta el mensaje y se publica en ROS.

```
class jsonros(Node):
    def __init__(self):
        super().__init__('pub_json')
        self.publisher = self.create_publisher(Point, 'data_rover', 10)
        self.timer_period = 1
        self.dato = Point()
        self.timer = self.create_timer(self.timer_period, self.motion)
        self.json_file = 'data.json'

    def motion(self):
        with open(self.json_file, 'r') as json_file:
            jsonD = json.load(json_file)
        self.dato.x = float(jsonD['PosicionX']) #La funcion get toma el
valor de key
        self.dato.y = float(jsonD['PosicionY'])
        self.dato.z = 0.0

        self.publisher.publish(self.dato)
        self.get_logger().info(f'Publicando dato JSON en ROS')
```

3. Finalmente, definimos la función *"main"* que nos inicia la ejecución del código.

```
def main(args=None):
    # initialize the ROS communication
    rclpy.init(args=args)
    # declare the node constructor
    talker = jsonros()
    # pause the program execution, waits for a request to kill the node
(ctrl+c)
    rclpy.spin(talker)
    # Explicity destroy the node
    talker.destroy_node()
    # shutdown the ROS communication
    rclpy.shutdown()
```