SIT310 – Task Xx  
Drone on a Mission

short line

# Overview

This task focuses on controlling a drone using the ROS2 service-client communications to travel to and extract information from mission pads.

## Task requirements

1. Ensure you have already completed all previous tasks
2. To complete this task, you will need to use your ROS2 Virtual Machine and your physical robot
3. You will also need your DJI Tello Drone. Note that in between using it, you should have the battery charging.
4. Review this week’s lectures before you start.
5. Read the introduced concepts/theories / recommended readings below.
6. Read the task instructions
7. Complete the activities in the task guide
8. Review your progress with your lab tutor.

# Task guide

This task will focus on controlling the drone using ROS2. You will use the service-client communications mechanism to control the drone and interact with mission pads.

### Planning the Service API.

### To continue our development of an application that uses services, we will quickly go over what we have already accomplished.

### For our drone application, there are a range of types of commands, as follows:

### Instructions to move (e.g. forward, backward)

### Actions to perform (e.g. flip forwar)

### State/Parameters (e.g. speed)

### Status (e.g. battery)

### Data request (e.g. video)

### We have created services for each command that include the data type needed. These are specified in the SDK. NOTE: Some commands such as *Takeoff* and *Land* do not need to be provided with any additional arguments whereas others that require moving a direction such as move\_forward requires a distance in integer format.

### Takeoff, request: none, response: none

### Land, request: none, response: none

### Move\_forward, request: int distance, response: String

### Move\_backward: request: int distance, response: String

### Flip\_forward, request: none, response: none

### Set\_speed, request: int speed, response: none

### get\_battery\_status, request: none, response: int

### get\_video request: none, response: video\_feed

### Writing the Drone Service

* 1. Create some custom services.
     + - As with the previous exercise, we will be using CMAKE to create a custom service through C and uses these in our Python packages.
       - Firstly, we should already have our drone interfaces package folder from the previous exercise, if not simply create a new package.

$ cd ~/dev\_ws/src

$ ros2 pkg create --build-type ament\_cmake drone\_interfaces

* + - * Create a service MPad

$ cd ~/dev\_ws/src/drone\_interfaces

$ cd srv

$ nano MPad.srv

* + - * Paste the following

int16 x

int16 y

int16 z

int16 x2

int16 y2

int16 z2

int16 speed

int16 rot

string pad1

string pad2

---

string result

* + - * This is a custom service that will requests an multiple integers and strings and provides the drone with the necessary parameters to preform commands with the mission pads.
      * To convert these to Python, we need to make some changes to CmakeLists.txt. Edit this file as follows

$ cd ~/dev\_ws/src/drone\_interfaces

$ nano CMakeLists.txt

* + - * Just below the section added to include the Move service but inside the parenthesis, add the following.

"srv/MPad.srv"

* + - * The dependencies should already exist but if they are missing they will need to be added to package.xml. Edit this file as follows. Just before the export block is ideal.

$ cd ~/dev\_ws/src/drone\_interfaces

$ nano package.xml

* + - * Add the following (just before the export block)

<build\_depend>rosidl\_default\_generators</build\_depend>

<exec\_depend>rosidl\_default\_runtime</exec\_depend>

<member\_of\_group>rosidl\_interface\_packages</member\_of\_group>

* + - * Build the package as follows.

$ cd ~/dev\_ws/

$ colcon build --packages-select drone\_interfaces

* + - * To test this issue the following commands in the terminal

$ cd ~/dev\_ws/

$ . install/setup.bash

$ ros2 interface show drone\_interfaces/srv/MPad

* + - * We will know that we have created our custom service, if you get the following output.

int16 x

int16 y

int16 z

int16 x2

int16 y2

int16 z2

int16 speed

int16 rot

string pad1

string pad2

---

string result

* 1. Create a project for the service
     + - We will now create a new package and create a service-client based application. Issue the following commands in a terminal:

$ cd ~/dev\_ws/src

$ ros2 pkg create --build-type ament\_python drone\_mission --dependencies rclpy drone\_interfaces

* + - * Using –dependencies actually sorts out the dependencies that we manually added in the publish-subscribe example.
  1. Create a service node in the following location.

$ cd ~/dev\_ws/src

$ nano drone\_mission/drone\_mission/services.py

* + - * Paste in the following code. The code is also available alongside this document. This is explained in the lecture.

import socket

import threading

import time

import rclpy

from rclpy.node import Node

from drone\_interfaces.srv import Move, MPad

from std\_srvs.srv import Empty

class DroneServer(Node):

    drone\_response = "no\_response"

    def \_\_init\_\_(self):

        super().\_\_init\_\_('drone\_server')

        self.local\_ip = ''

        self.local\_port = 8889

        self.socket = socket.socket(socket.AF\_INET, socket.SOCK\_DGRAM)  # socket for sending cmd

        self.socket.bind((self.local\_ip, self.local\_port))

        # thread for receiving cmd ack

        self.receive\_thread = threading.Thread(target=self.\_receive\_thread)

        self.receive\_thread.daemon = True

        self.receive\_thread.start()

        self.socket2 = socket.socket(socket.AF\_INET, socket.SOCK\_DGRAM)  # socket for mission pads

        self.socket2.bind(('', 8890))

        # thread for receiving mission pad ack

        self.receive\_thread2 = threading.Thread(target=self.\_receive\_thread\_state)

        self.receive\_thread2.daemon = True

        self.receive\_thread2.start()

        self.tello\_ip = '192.168.10.1'

        self.tello\_port = 8889

        self.tello\_address = (self.tello\_ip, self.tello\_port)

        self.MAX\_TIME\_OUT = 15.0

        self.srv = self.create\_service(Move, 'move\_forward', self.move\_forward\_callback)

        self.srv = self.create\_service(Move, 'move\_backward', self.move\_backward\_callback)

        self.srv = self.create\_service(Move, 'move\_left', self.move\_left\_callback)

        self.srv = self.create\_service(Move, 'move\_right', self.move\_right\_callback)

        self.srv = self.create\_service(Move, 'move\_up', self.move\_up\_callback)

        self.srv = self.create\_service(Move, 'move\_down', self.move\_down\_callback)

        self.srv = self.create\_service(Empty, 'flip\_forward', self.flip\_forward\_callback)

        self.srv = self.create\_service(Empty, 'flip\_backward', self.flip\_backward\_callback)

        self.srv = self.create\_service(Empty, 'takeoff', self.takeoff\_callback)

        self.srv = self.create\_service(Empty, 'land', self.land\_callback)

        self.srv = self.create\_service(Empty, 'battery', self.battery\_callback)

        self.srv = self.create\_service(Empty, 'mon', self.mon\_callback)

        self.srv = self.create\_service(Empty, 'mdirection', self.mdirection\_callback)

        self.srv = self.create\_service(MPad, 'mid\_go', self.mid\_go\_callback)

        self.srv = self.create\_service(MPad, 'mid\_jump', self.mid\_jump\_callback)

        self.srv = self.create\_service(MPad, 'mid\_curve', self.mid\_curve\_callback)

    def send\_command(self, msg):

        command = msg #the actual command string

        self.get\_logger().info('I heard: "%s"' % msg)

        self.socket.sendto(command.encode('utf-8'), self.tello\_address)

        print('sending command: %s to %s' % (command, self.tello\_ip))

        start = time.time()

        now = time.time()

        diff = now - start

        if diff > self.MAX\_TIME\_OUT:

            print('Max timeout exceeded... command %s' % command)

            return

        print('Done!!! sent command: %s to %s' % (command, self.tello\_ip))

    def move\_forward\_callback(self, request, response):

        global drone\_response

        self.get\_logger().info('Incoming request: Move Forward: %dcm' % (request.distance))

        command = "forward %d" % request.distance

        print(command)

        self.send\_command("command")

        time.sleep(1)

        self.send\_command(command)

        time.sleep(1) #wait for the response

        response.result = drone\_response

        return response

    def move\_backward\_callback(self, request, response):

        global drone\_response

        self.get\_logger().info('Incoming request: Move Backward: %dcm' % (request.distance))

        command = "back %d" % request.distance

        print(command)

        self.send\_command("command")

        time.sleep(1)

        self.send\_command(command)

        time.sleep(1) #wait for the response

        response.result = drone\_response

        return response

    def move\_left\_callback(self, request, response):

        global drone\_response

        self.get\_logger().info('Incoming request: Move Left: %dcm' % (request.distance))

        command = "left %d" % request.distance

        print(command)

        self.send\_command("command")

        time.sleep(1)

        self.send\_command(command)

        time.sleep(1) #wait for the response

        response.result = drone\_response

        return response

    def move\_right\_callback(self, request, response):

        global drone\_response

        self.get\_logger().info('Incoming request: Move Right: %dcm' % (request.distance))

        command = "right %d" % request.distance

        print(command)

        self.send\_command("command")

        time.sleep(1)

        self.send\_command(command)

        time.sleep(1) #wait for the response

        response.result = drone\_response

        return response

    def move\_up\_callback(self, request, response):

        global drone\_response

        self.get\_logger().info('Incoming request: Move Up: %dcm' % (request.distance))

        command = "up %d" % request.distance

        print(command)

        self.send\_command("command")

        time.sleep(1)

        self.send\_command(command)

        time.sleep(1) #wait for the response

        response.result = drone\_response

        return response

    def move\_down\_callback(self, request, response):

        global drone\_response

        self.get\_logger().info('Incoming request: Move Down: %dcm' % (request.distance))

        command = "down %d" % request.distance

        print(command)

        self.send\_command("command")

        time.sleep(1)

        self.send\_command(command)

        time.sleep(1) #wait for the response

        response.result = drone\_response

        return response

    def flip\_forward\_callback(self, request, response):

        self.get\_logger().info('Incoming request: flip forward')

        command = "flip f"

        print(command)

        self.send\_command(command)

        return response

    def flip\_backward\_callback(self, request, response):

        self.get\_logger().info('Incoming request: flip backward')

        command = "flip b"

        print(command)

        self.send\_command(command)

        return response

    def takeoff\_callback(self, request, response):

        self.get\_logger().info('Incoming request: Takeoff')

        command = "takeoff"

        print(command)

        self.send\_command("command")

        time.sleep(2)

        self.send\_command(command)

        return response

    def land\_callback(self, request, response):

        self.get\_logger().info('Incoming request: Land')

        command = "land"

        print(command)

        self.send\_command("command")

        time.sleep(2)

        self.send\_command(command)

        return response

    def battery\_callback(self, request, response):

        global drone\_response

        self.get\_logger().info('Incoming request: Battery')

        command = "battery?"

        print(command)

        self.send\_command("command")

        time.sleep(1)

        self.send\_command(command)

        time.sleep(1)

        print("Battery is : %s" % self.response.decode('utf-8'))

        return response

    def mon\_callback(self, request, response):

        self.get\_logger().info('Activate Mission Pads')

        command = "mon"

        print(command)

        self.send\_command("command")

        time.sleep(2)

        self.send\_command(command)

        return response

    def mdirection\_callback(self, request, response):

        self.get\_logger().info('Detect Mission Pads below')

        command = "mdirection 2"

        print(command)

        self.send\_command("command")

        time.sleep(2)

        self.send\_command(command)

        return response

    def mid\_go\_callback(self, request, response):

        global drone\_response

        self.get\_logger().info('Incoming request: Move on Current Mission Pad: %s' % (request.pad1))

        command = "go %d %d %d %d %s" % (request.x, request.y, request.z, request.speed, request.pad1)

        print(command)

        self.send\_command("command")

        time.sleep(1)

        self.send\_command(command)

        time.sleep(1) #wait for the response

        #response.result = drone\_response

        return response

    def mid\_jump\_callback(self, request, response):

        global drone\_response

        self.get\_logger().info('Incoming request: Move to Mission Pad: %s' % (request.pad2))

        command = "jump %d %d %d %d %d %s %s" % (request.x, request.y, request.z, request.speed, request.rot, request.pad1, request.pad2)

        print(command)

        self.send\_command("command")

        time.sleep(1)

        self.send\_command(command)

        time.sleep(1) #wait for the response

        #response.result = drone\_response

        return response

    def mid\_curve\_callback(self, request, response):

        global drone\_response

        self.get\_logger().info('Incoming request: Curve to Mission Pad: %s' % (request.pad1))

        command = "curve %d %d %d %d %d %d %d %s" % (request.x, request.y, request.z, request.x2, request.y2, request.z2, request.speed, request.pad1)

        print(command)

        self.send\_command("command")

        time.sleep(1)

        self.send\_command(command)

        time.sleep(1) #wait for the response

        #response.result = drone\_response

        return response

    def \_receive\_thread(self):

        global drone\_response

        #Listen to responses from the Tello.

        while True:

            try:

                self.response, ip = self.socket.recvfrom(1024)

                print('from %s: %s' % (ip, self.response))

                drone\_response = str(self.response) #convert from byte string to string

            except (socket.error, exc):

                print("Caught exception socket.error : %s" % exc)

    def \_receive\_thread\_state(self):

        #Listen to responses from the Tello.

        while True:

            try:

                self.response, ip = self.socket2.recvfrom(1024)

                print('from %s: %s' % (ip, self.response))

            except (socket.error, exc):

                print("Caught exception socket.error : %s" % exc)

def main(args=None):

    rclpy.init(args=args)

    node = DroneServer()

    try:

        rclpy.spin(node)

    except KeyboardInterrupt:

        pass

    # Destroy the node explicitly

    # (optional - Done automatically when node is garbage collected)

    node.destroy\_node()

    rclpy.shutdown()

if \_\_name\_\_ == '\_\_main\_\_':

    main()

* + - * Edit the drone\_mission/setup.py file with nano as follows.

$ cd ~/dev\_ws/src

$ nano drone\_mission/setup.py

* + - * Add the following lines to within the console\_scripts section. Then exit nano with control-x, making sure you save the file.

'service = drone\_mission.services:main',

* + - * Build the package as follows.

$ cd ~/dev\_ws/

$ colcon build --packages-select drone\_mission

* + - * Run the service as follows.

$ cd ~/dev\_ws

$ . install/setup.bash

$ ros2 run drone\_mission service

* + - * The service node will sit waiting for API calls.
      * To see what the service is advertising, in a new terminal, type the following:

$ cd ~/dev\_ws

$ . install/setup.bash

$ ros2 service list -t

* 1. Test the Service with tools
     + - To test if the topic is working, make sure the drone is on, and connect to its WiFi network. Run the service in a terminal as follows.

$ cd ~/dev\_ws

$ . install/setup.bash

$ ros2 run drone\_mission service

* + - * Now open a new terminal and run the following:

$ cd ~/dev\_ws

$ . install/setup.bash

$ ros2 service call /takeoff std\_srvs/srv/Empty

$ ros2 service call /move\_forward drone\_interfaces/srv/Move "{distance: 100}"

$ ros2 service call /mon std\_srvs/srv/Empty

$ ros2 service call /mdirection std\_srvs/srv/Empty

$ ros2 service call /land std\_srvs/srv/Empty

* + - * You will see the service client program issuing a ROS2 service call, which the service Node will receive and issue commands to the Drone. Note, from when the mon service is called, the terminal will print out continual locational data as well as mission pad information.
      * This are straightforward, but what we really want to achieve is using this data for movement and location awareness with the mid\_go and mid\_jump commands.
      * Place your drone on a mission pad and run the mon and mdirection services followed by takeoff. Once the drone is in the air paste in the following command.

ros2 service call /mid\_go drone\_interfaces/srv/MPad "{x: 20, y:40, z:10, speed:30, pad1:m1"

waiting for service to become available...

requester: making request: drone\_interfaces.srv.MPad\_Request(x=20, y=40, z=10, x2=0, y2=0, z2=0, speed=30, rot=0, pad1='m1', pad2='')

response:

drone\_interfaces.srv.MPad\_Response(result='')

* + - * Note that we have a fully-fledged and working ROS2 service Node. Any Node can call this.
      * Notice that requester has empty values for the unused parameters from out Mpad service? This is because we have created a service that can be used across multiple commands with differing parameters.
  1. Test the Service with a client
     + - To complete this process, we can create a client node. Create it in the following location.

$ cd ~/dev\_ws/src

$ nano drone\_mission/drone\_mission/services\_client.py

* + - * Paste in the following code. This is explained in the lecture.

from drone\_interfaces.srv import Move, MPad

from std\_srvs.srv import Empty

import rclpy

import time

def Initilize(node):

    ##Battery##

    cli = node.create\_client(Empty, 'battery')

    req = Empty.Request()

    while not cli.wait\_for\_service(timeout\_sec=1.0):

        node.get\_logger().info('service not available, waiting again...')

    future = cli.call\_async(req)

    rclpy.spin\_until\_future\_complete(node, future)

    try:

        result = future.result()

    except Exception as e:

        node.get\_logger().info('Service call failed %r' % (e,))

    else:

        node.get\_logger().info("Battery status is shown")

    time.sleep(5)

    ##Mon##

    cli = node.create\_client(Empty, 'mon')

    req = Empty.Request()

    while not cli.wait\_for\_service(timeout\_sec=1.0):

        node.get\_logger().info('service not available, waiting again...')

    future = cli.call\_async(req)

    rclpy.spin\_until\_future\_complete(node, future)

    try:

        result = future.result()

    except Exception as e:

        node.get\_logger().info('Service call failed %r' % (e,))

    else:

        node.get\_logger().info("Mission pad On")

    time.sleep(2)

    ##mdirection##

    cli = node.create\_client(Empty, 'mdirection')

    req = Empty.Request()

    while not cli.wait\_for\_service(timeout\_sec=1.0):

        node.get\_logger().info('service not available, waiting again...')

    future = cli.call\_async(req)

    rclpy.spin\_until\_future\_complete(node, future)

    try:

        result = future.result()

    except Exception as e:

        node.get\_logger().info('Service call failed %r' % (e,))

    else:

        node.get\_logger().info("Mission pad detection set")

    time.sleep(5)

def Mission\_Pad\_Jump(node, x, y, z, speed, rot, mpad\_s, mpad\_f):

    cli = node.create\_client(MPad, 'mid\_jump')

    req = MPad.Request()

    req.x = x

    req.y = y

    req.z = z

    req.speed = speed

    req.rot = rot

    req.pad1 = mpad\_s

    req.pad2 = mpad\_f

    while not cli.wait\_for\_service(timeout\_sec=1.0):

        node.get\_logger().info('service not available, waiting again...')

    future = cli.call\_async(req)

    rclpy.spin\_until\_future\_complete(node, future)

    try:

        result = future.result()

    except Exception as e:

        node.get\_logger().info('Service call failed %r' % (e,))

    else:

        node.get\_logger().info("Move to new Mission pad successful")

    time.sleep(10)

def Mission\_Pad\_Go(node, x, y, z, speed, mpad):

    cli = node.create\_client(MPad, 'mid\_go')

    req = MPad.Request()

    req.x = x

    req.y = y

    req.z = z

    req.speed = speed

    req.pad1 = mpad

    while not cli.wait\_for\_service(timeout\_sec=1.0):

        node.get\_logger().info('service not available, waiting again...')

    future = cli.call\_async(req)

    rclpy.spin\_until\_future\_complete(node, future)

    try:

        result = future.result()

    except Exception as e:

        node.get\_logger().info('Service call failed %r' % (e,))

    else:

        node.get\_logger().info("Move on current Mission pad successful")

    time.sleep(2)

def Mission\_Pad\_Curve(node, x1, y1, z1, x2, y2, z2, speed, mpad):

    cli = node.create\_client(MPad, 'mid\_curve')

    req = MPad.Request()

    req.x = x1

    req.y = y1

    req.z = z1

    req.x2 = x2

    req.y2 = y2

    req.z2= z2

    req.speed = speed

    req.pad1 = mpad

    while not cli.wait\_for\_service(timeout\_sec=1.0):

        node.get\_logger().info('service not available, waiting again...')

    future = cli.call\_async(req)

    rclpy.spin\_until\_future\_complete(node, future)

    try:

        result = future.result()

    except Exception as e:

        node.get\_logger().info('Service call failed %r' % (e,))

    else:

        node.get\_logger().info("Curve to new Mission pad successful")

    time.sleep(2)

def main(args=None):

    rclpy.init(args=args)

    node = rclpy.create\_node('drone\_service\_client')

    ##Take off##

    cli = node.create\_client(Empty, 'takeoff')

    req = Empty.Request()

    while not cli.wait\_for\_service(timeout\_sec=1.0):

        node.get\_logger().info('service not available, waiting again...')

    future = cli.call\_async(req)

    rclpy.spin\_until\_future\_complete(node, future)

    try:

        result = future.result()

    except Exception as e:

        node.get\_logger().info('Service call failed %r' % (e,))

    else:

        node.get\_logger().info("takeoff successful")

    time.sleep(5)

    Initilize(node)

    Mission\_Pad\_Jump(node, 80, 3, 50, 25, 180, 'm1', 'm2')

    Mission\_Pad\_Go(node, 0, 10, 60, 80, 'm2')

    Mission\_Pad\_Go(node, 0, -10, 60, 80, 'm2')

    Mission\_Pad\_Go(node, 0, 10, 60, 80, 'm2')

    Mission\_Pad\_Go(node, 0, -10, 60, 80, 'm2')

    Mission\_Pad\_Jump(node, -80, 3, 120, 40, 180, 'm2', 'm1')

    ##flip##

    cli = node.create\_client(Empty, 'flip\_backward')

    req = Empty.Request()

    while not cli.wait\_for\_service(timeout\_sec=1.0):

        node.get\_logger().info('service not available, waiting again...')

    future = cli.call\_async(req)

    rclpy.spin\_until\_future\_complete(node, future)

    try:

        result = future.result()

    except Exception as e:

        node.get\_logger().info('Service call failed %r' % (e,))

    else:

        node.get\_logger().info("Flip backward successful")

    time.sleep(5)

    ##land##

    cli = node.create\_client(Empty, 'land')

    req = Empty.Request()

    while not cli.wait\_for\_service(timeout\_sec=1.0):

        node.get\_logger().info('service not available, waiting again...')

    future = cli.call\_async(req)

    rclpy.spin\_until\_future\_complete(node, future)

    try:

        result = future.result()

    except Exception as e:

        node.get\_logger().info('Service call failed %r' % (e,))

    else:

        node.get\_logger().info("land successful")

    node.destroy\_node()

    rclpy.shutdown()

if \_\_name\_\_ == '\_\_main\_\_':

    main()

* + - * You may notices a few in the client application we have migrated to a function model for some of the service calls. This is due to the necessity to call the same service multiple times with separate argument parameters each time.
      * Edit the drone\_mission/setup.py file with nano as follows.

$ cd ~/dev\_ws/src

$ nano drone\_service/setup.py

* + - * Add the following lines to within the console\_scripts section. Then exit nano with control-x, making sure you save the file.

'client = drone\_mission.services\_client:main',

* + - * Build the package as follows.

$ cd ~/dev\_ws/

$ colcon build --packages-select drone\_mission

* + - * To test if the topic is working, make sure the drone is on, and connect to its WiFi network. Run the client in a new terminal as follows.

$ cd ~/dev\_ws

$ . install/setup.bash

$ ros2 run drone\_mission client

* + - * You will see the service client program issuing ROS2 service calls, which the service Node will receive and issue commands to the Drone.
      * Note that we have a fully-fledged and working ROS2 service Node. Any Node can call this.

### Extending the drone service with battery status

### The next extensions for this tasks will be integrating the curve command

### This needs to be done in a very similar way to the way we call the jump and go commands.

ASSESSMENT 3

### This week’s lab assessment is based on the output of this lab as follows. You should start this now. Depending on what you achieve, you should upload a paragraph of explanation of what you have done, your code in a zip and a short video to the quiz “Lab Assessment 3” on the unit site. Take a video (mobile phone is fine) and zip it together with your solution. Put your zip on your onedrive, click share, and allow anyone with the link to read it. Finally, insert this link in the text box after your explanation of what you have achieved.

### It is recommended that you spend 5-6 hours to get as far as possible. Leave the quiz questions for more advanced functionality blank. Don’t worry, there is plenty of time to get great grades in the unit!

* + - * P – Explain in your own words how the service/client method of controlling the drone work. Demonstrate both working in your submission.
      * C - There are a few things we have not included in the service, e.g. curve to mission pad. To complete the ROS2 Drone service, go through the Mission Pad SDK and add each missing command to the service.
      * D - Add ability for drone to preform specific actions when at each mission pad, incorporating at least 3 mission pads.
      * HD - Add support for video in your service and use the video to determine if the drone is on a mission pad and move towards a pad if not. This might require some effort. Write this as a tutorial in the format of the task sheets and this will be integrated into the unit next year.

Mission Pad Guide: <https://dl-cdn.ryzerobotics.com/downloads/Tello/Tello%20Mission%20Pad%20User%20Guide.pdf>

Drone SDK: <https://dl-cdn.ryzerobotics.com/downloads/Tello/Tello%20SDK%202.0%20User%20Guide.pdf>

short dash