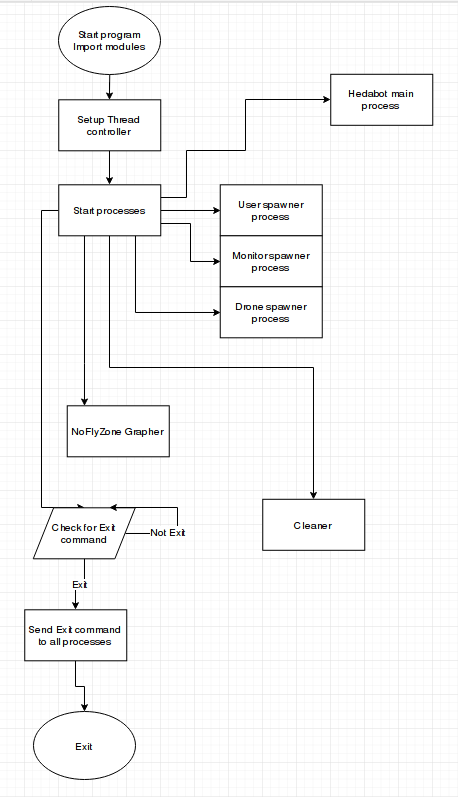
**AATC\_Server\_Starter.py**



This module is used to launch the server part of the AATC server. On Linux cannot be run in IDLE due to the use of multiprocessing causing the IDLE interface to crash, requiring the program to be killed.

**Import**

import multiprocessing,socket,AATC\_NoFlyZoneGrapher,sys,time,AATC\_GPIO,HedaBot

import AATC\_Server\_002 as AATC\_Server

These lines import modules required for the functioning of the server. Multiprocessing and socket are required to connect to the user and run the instances in separate processes to reduce interference. Multiprocessing is used due to the Global Interpreter lock in python, meaning that a single python process cannot use more than 1 CPU core at a given time. Running multiple clients on a single core would result in slow downs and a poor user experience. AATC\_NoFlyZoneGrapher is a module to convert NoFlyZones from the database into Node costs in the graph used to find a flight path. Sys is imported to allow reliable exiting when the server is to be shutdown. TIME IS CURRENTLY NOT USED?. AATC\_GPIO contains the Thread\_Controller object, used to manage a number of processes automatically through a single queue among other things. The HedaBot module contains the components for the Telegram bot aspect of the server.

from AATC\_Coordinate import \*

AATC\_Coordinate contains multiple objects and methods necessary for the path finding algorithm of the server and the Coordinate object which is used to handle coordinates as objects with normalised string and print methods. The items in AATC\_Coordinate are added to the namespace of AATC\_Server\_Starter as when loading Coordinate objects using pickle requires the objects to exist in the \_\_main\_\_ namespace of the program otherwise the object will appear as not being defined.

**ProcessSpawner**

def ProcessSpawner(Name,Communications\_Queue,Port,Type,Target):

Exit = False

Spawner\_Control\_Queue = AATC\_GPIO.Create\_Controller()

ID\_Counter = 1

DisplayName = "["+str(Name)+":"+str(Type)+"]"

while not Exit:

try:

HOST = ''

PORT = Port

s = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

s.setsockopt(socket.SOL\_SOCKET, socket.SO\_REUSEADDR, 1)

print(DisplayName,'Socket created')

s.bind((HOST, PORT))

print(DisplayName, 'Socket bind complete')

s.listen(10)

print(DisplayName, 'Socket now listening')

while not Exit:

try:

conn, addr = s.accept()

print(DisplayName, ' Connected with' , addr[0] , ':' , str(addr[1]), "Type:",Type)

Thread\_Name = Type+str(ID\_Counter)

Spawner\_Control\_Queue.put(("Controller","Create\_Process",(Thread\_Name,Target,(conn,))))

ID\_Counter +=1

except Exception as e:

print("Error creating" ,Type,"connection",str(e))

#Check for commands from Communications\_Queue

if not Communications\_Queue.empty():

data = Communications\_Queue.get()

Command,Arguments = data[0],data[1]

if Command == "Exit":

self.Exit = True

except Exception as e:

print("Error in",Type,"Process Spawner",str(e))

Spawner\_Control\_Queue.put(("Controller","Exit",(True,)))

The ProcessSpawner function is used to allow clients to connect to the server, and passes this connection to the relevant type of connection eg for a User -> UserConnection. It will continue to do this until it receives a command to exit and then attempts to join all of its child processes.

def ProcessSpawner(Name,Communications\_Queue,Port,Type,Target):

Exit = False

Spawner\_Control\_Queue = AATC\_GPIO.Create\_Controller()

ID\_Counter = 1

DisplayName = "["+str(Name)+":"+str(Type)+"]"

The process takes 5 arguments, none of which are optional. The first argument is the Name associated with that process by the relevant Thread\_Controller and can be any type which has a str function. The Name is also used to generate the DisplayName of the process, to clearly identify the process in the text output. The second argument is a queue, either of type queue.Queue or multiprocessing.Queue depending on if the function is being run in a thread or a process, both are almost functionally identical. This queue is used to send commands to the ProcessSpawner such as to Exit or reset. The port argument is of type integer and is the port on which the ProcessSpawner will listen for connections on. Per default this would be 8000, 8001, 8002 for User,Monitor and Drone respectively. The 4th argument is Type, of any type used to identify in the text output what Type of Process is currently running. The last argument is the target function of the Process Spawner. When a connection is formed the connection is passed into this function and run. Must be a function which takes the variables Thread\_Name, Control\_Queue and a Connection.

When the function is run Exit is set to False. Then a Thread\_Controller is created using AATC\_GPIO.Create\_Controller which returns the control queue for that Thread\_Controller, this queue is then stored in Spawner\_Control\_Queue. ID\_Counter is set to 1, this variable generates unique IDs for each process created , and is increased by 1 each time a process is created. This is used to ensure each connection in the Thread\_Controller remains available in order to end commands to the processes at a later time. Finally DisplayName is a string variable which contains a neater way to print the Name and type of the process i.e. when the process prints the connection of a new user.

while not Exit:

try:

HOST = ''

PORT = Port

s = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

s.setsockopt(socket.SOL\_SOCKET, socket.SO\_REUSEADDR, 1)

print(DisplayName,'Socket created')

s.bind((HOST, PORT))

The while not Exit loop will continue to run until Exit is set to a True value, such as when the server receives an Exit command. The section inside the loop has been broken up to improve readability. The try statement will in conjunction with the except statement will catch any exceptions created when attempting to form connections or bind sockets.

The lines containing HOST and PORT define the HOST and PORT on which the socket will accept connections. HOST is set to “”, the empty string to signal the socket to accept any connection on the localhost. Port is an integer for the port on which connections will be received on.

s = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

s.setsockopt(socket.SOL\_SOCKET, socket.SO\_REUSEADDR, 1)

These lines create a socket with a reusable address, meaning that sockets can connect to that address again even if the previous socket was not correctly closed.

print(DisplayName,'Socket created')

s.bind((HOST, PORT))

These lines display that the ProcessSpawner has correctly formed a socket, using DisplayName to identify the socket. The socket is then bound to the HOST,PORT combination as defined above. If the address is already bound to then this call will raise an error which will be caught in the try/except statement.

print(DisplayName, 'Socket bind complete')

s.listen(10)

The ProcessSpawner then displays that the Socket is correctly bound if so, and then listens for up to 10 concurrent incoming connection requests.

while not Exit:

try:

conn, addr = s.accept()

print(DisplayName, ' Connected with' , addr[0] , ':' , str(addr[1]), "Type:",Type)

Thread\_Name = Type+str(ID\_Counter)

Spawner\_Control\_Queue.put(("Controller","Create\_Process",(Thread\_Name,Target,(conn,))))

ID\_Counter +=1

except Exception as e:

print("Error creating" ,Type,"connection",str(e))

This loop will run until Exit is set to True. A new set of Try/except statements catches errors which connecting in this loop, reducing the number of times the socket has to be recreated on an error and be rebound to a port. A connection and address are returned into the respective variables from the s.accept call when a connection is formed. The connection object is a new socket object and the address, a string variable, is the address the connection is formed from. The function then prints that a new connection from that address has been formed and of which type it is. The Thread\_Name is formed from a combination of the Type and string of the current value of the ID\_Counter. This variable is a string.

The function then places a tuple into the Spawner\_Control\_Queue, instructing the Thread\_Controller to create a new process with the arguments in the third position in the tuple. This includes the previously created Thread\_Name, the target function and the arguments for this function in the form of a tuple.

ID Counter is then increased by 1.

Exceptions will be caught and displayed in the except statement.

#Check for commands from Communications\_Queue

if not Communications\_Queue.empty():

data = Communications\_Queue.get()

Command,Arguments = data[0],data[1]

if Command == "Exit":

Exit = True

This section checks the command Queue for any commands, if the queue is not empty it fetches the first command and executes it e.g. if the Command is “Exit” the variable Exit is set to True, preventing any further repeats of the connection loop.

except Exception as e:

print("Error in",Type,"Process Spawner",str(e))

Spawner\_Control\_Queue.put(("Controller","Exit",(True,)))

The final section catches any errors between the first try statement and the except statement.

The last line is executed on the exit of the ProcessSpawner, it sends a command to the Thread\_Controller to exit all processes and to wait for each to join.

**MakeConnection Statements**

def MakeUserConnection(Thread\_Name,Thread\_Queue,conn):

try:

UConn = AATC\_Server.UserConnection(Thread\_Name,Thread\_Queue,conn)

UConn.Connection\_Loop()

except Exception as e:

print("Serious error in UserConnection",e)

def MakeMonitorConnection(Thread\_Name,Thread\_Queue,conn):

try:

MConn = AATC\_Server.MonitorConnection(Thread\_Name,Thread\_Queue,conn)

MConn.Connection\_Loop()

except Exception as e:

print("Serious error in MonitorConnection",e)

def MakeDroneConnection(Thread\_Name,Thread\_Queue,conn):

try:

DConn = AATC\_Server.DroneConnection(Thread\_Name,Thread\_Queue,conn)

DConn.Connection\_Loop()

except Exception as e:

print("Serious error in DroneConnection",e)

Each function is similar, each is executed when a new connection is formed and creates the ?\_Connection object and executes the main loop for each where ? is the type of object the function is to deal with.

The functions take 3 variables. A Thread\_Name of any type, a Thread\_Queue of type queue (either type of queue) and a socket connection. In a try/except statement the function creates an instance of the relevant object , passing the three variables to this object and executes the Connection\_Loop function of the object. The except statement catches any variables which are not caught in the Connection\_Loop, displaying them as serious errors due to the expectance that the Connection\_Loop should attempt to catch any exceptions it generates.

**StartProcesses**

def StartProcesses(Control\_Queue):

Control\_Queue.put(("Controller","Create\_Process",("USpawner",ProcessSpawner,(8000,"User",MakeUserConnection))))

Control\_Queue.put(("Controller","Create\_Process",("MSpawner",ProcessSpawner,(8001,"Monitor",MakeMonitorConnection))))

Control\_Queue.put(("Controller","Create\_Process",("DSpawner",ProcessSpawner,(8002,"Drone",MakeDroneConnection))))

Control\_Queue.put(("Controller","Create\_Process",("Grapher",AATC\_NoFlyZoneGrapher.NoFlyZoneGrapher)))

Control\_Queue.put(("Controller","Create\_Process",("Cleaner",AATC\_Server.Cleaner)))

Control\_Queue.put(("Controller","Create\_Process",("Hedabot",HedaBot.TelebotLaunch,(HedaBot.telepot.Bot(HedaBot.BOT\_TOKEN),))))

print("[StartProcesses] All processes started")

This function launches all processes at the startup of the program using the Thread\_Controller. It takes one Queue as an argument, the queue used to send commands to the main Thread\_Controller of the program.

The first three command calls are very similar. Each commands the Thread\_Controller to create a new process using the arguments in the last tuple. This tuple is laid out in the form (Thread\_Name, Type, Target) of the types (any string capable object, any string capable object, function).

The 4th and 5th command calls create the NoFlyZoneGrapher and the Cleaner processes. The Cleaner removes any redundant data from the database i.e. old or completed flights.

The last call launches the Telegram bot, using the function TelebotLaunch from the HedaBot module. The only argument is a Telepot bot created using the HedaBot BOT\_TOKEN constant.

Finally the function displays that all processes have been launched.

**\_\_main\_\_ function**

if \_\_name\_\_ == "\_\_main\_\_":

print("Server is starting")

Control\_Queue = AATC\_GPIO.Create\_Controller()

StartProcesses(Control\_Queue)

Main\_Command = ""

while Main\_Command != "EXIT":

Main\_Command = input("Enter main command >>").upper()

print("Killing all Server processes....")

print("This may take time, sleeping processes will be killed when resuming from sleep")

Control\_Queue.put(("Controller","Exit",(True,)))

print("Main process is now exiting...")

sys.exit()

This section of code only runs if the program is the main program being launched, not if it was imported by another module. The program prints a startup message and creates a Thread\_Controller which returns the command Queue required to control it. Then the StartProcesses function is called, passing the queue as an argument, launching the processes required for functioning.

A loop is then setup to continually request an input which is converted into upper case from the user until “EXIT” is entered. This could be used to add main functions to the server , however currently no such functionality is needed.

Once that loop has completed an exit message is displayed and an exit command is sent to the Thread\_Controller, requesting all processes to be exited and waiting for each to join. Then the program exits using sys.exit.

**Server Process Layout**

