EMBRY-RIDDLE
AERONAUTICAL UNIVERSITY

Notebook -- ADS-B Spoofing with Drone Simulation

Doug Chamberlain
CI 490: Cyber-Security Capstone
Dr. Sampigethaya
30 April 2019

Table of Contents:

1 Code I Used:

1.1 PyParrot: https://github.com/amymcgovern/pyparrot

1.1.1 PyParrot is a Python module for interacting with and programming Parrot drones.

1.2 ADS-B Out: https://github.com/lyusupov/ADSB-Out

1.2.1 ADS-B Out is a Python module to encode ADS-B messages. It only supports encoding for Downlink Format 17.

1.2.2 It is unclear whether it officially supports encoding functionality for surface messages. I only used airborne messages during this experiment. The primary difference between the lat/lon encoding of airborne and surface messages is the precision of the coordinates. In airborne messages, the lat/lon is represented by 34 bits (17 bits per). In surface messages, the lat/lon is represented by 38 bits (19 bits per).

1.2.3 Downlink Format 18 is reserved for messages sent from equipment that cannot be interrogated.

1.3 pyModeS: https://github.com/junzis/pyModeS

1.3.1 pyModeS is a Python module that supports decoding functionality for many variations of ADS-B messages. In practice, I only used it to decode Downlink Format 17 (DF17) messages.

2 Why I chose PyParrot over ByBop:

2.1 ByBop seemed to be extremely limited in functionality. I would have been able to use it, but it would have been more difficult. PyParrot offered more functionality and made the code easier to read, write, and understand.

3 Importing Modules:

3.1 Import Bebop object from PyParrot module

3.1.1
```
from pyparrot.Bebop import Bebop
```

3.2 Import module to allow printing in different colors and formats

3.2.1
```
from termcolor import cprint, colored
```

3.3 Import multi-processing and multi-threading modules

3.3.1
```
import multiprocessing as mp
from _thread import *
```

3.3.1.1 I was unable to get the multi-threading to work for the main purpose I needed these modules for. The primary problem that I had was when I wanted to stop the thread/process. I was unable to stop the thread for some reason. When I used the multiprocessing library, I found it pretty straightforward.

3.4 Import some core functions like networking, os, and others

3.4.1
```
import netifaces as net
import socket, logging, ctypes, sys, os
import cv2
```

3.4.1.1 netifaces was used so I could automatically gather my internal ip address for the specific adapter I wanted to use when I was establishing a connection between the client computer and this one.

3.4.1.2 socket was used to bind a socket and create threaded clients (the use for the _thread module in this context)

3.4.1.3 cv2 was not actually used, but I believe I imported it because I had previously been establishing a video stream from the drone to my computer. It may have been used for something else that I cannot remember.

3.4.1.4 logging, ctypes, sys, os for various core functions

3.5 Import math and timing modules

```
import math, time
from datetime import datetime
```

3.5.1

3.5.1.1 These are used in my collision avoidance function. The math module is used so I can make a simple function call to return the hypotenuse of the two arguments. Time and datetime were used so that I could better keep track of time during the collision avoidance iterations.

3.6 Import geography modules

```
from geographiclib.geodesic import Geodesic
import geopy.distance
```

3.6.1

3.6.1.1 Geographiclib.geodesic was going to be used to establish the bearing to/from the current drone and the threat. Because the drone considers "North" to be whatever direction the drone is facing in when it is turned on, it would have been a bit more complicated to establish which direction was North, and it also would take time each time I wanted to test the script, so I opted to just forget about it.

3.6.1.2 Geopy.distance.distance is just a simple function to get the distance between two GPS locations on Earth. It specifically differs from a simple spherical calculation because it takes into account the fact that Earth is an "oblate spheroid."

3.7 Finally, import pyModeS

```
import pyModeS as pms
```

3.7.1.1 pyModeS is a library of functions to decode ADS-B and perform various verifications of the messages. It was used for all decoding of ADS-B messages in this project. It does not encode ADS-B.

4 Main function:

4.1 Sharing variables between multiple processes:

```
keepGoing = mp.Value('i', 1)
```

4.1.1.1 This is specifically to allow for a way to stop the while loop in the main function.

4.2 Create Bebop object

```
# make my bebop object
bebop = Bebop()
bebop.smart_sleep(2)
```

4.2.1.1 Creates a default Bebop object.

4.2.1.2 Bebop.smart_sleep(2) tells the bebop to sleep for 2 seconds. The reason that this is used in place of time.sleep(x) is to prevent the drone from being confused and losing packets. I did not ever experiment with time.sleep(x) to see what would happen in that case.

4.3 Connect to the Bebop and Create Processes

```
# connect to the bebop
success = bebop.connect(5)
if(success):
    bebop.smart_sleep(3)
    bebop.ask_for_state_update()

    cprint(("Battery: " + str(bebop.sensors.battery)), "yellow")
```

4.3.1.1 Stores the result of the attempt to connect to the bebop in the variable "success"

4.3.1.2 If the bebop was able to connect successfully, I told it to sleep for 3 seconds and then ask for a state update from the drone because I was having problems getting the battery life from the drone right after connecting. It is initialized as 100, so the drone continued to display that it had 100% battery life every time I turned it on, so I gave it some time to work out whatever processes it goes through when a controller connects.

4.3.1.3 Prints the battery life in yellow text to make it more visible during all of the other information printed to the console when connecting to the drone.

4.3.2
```
flightProc = mp.Process(target = flight_plan, args = (keepGoing, ))
flightProc.start()
```

4.3.2.1 Upon successful connection, create a flight process as flightPlan(keepGoing). It is formatted as (keepGoing,) to convert it to a tuple.

4.3.2.2 Start the flight process.

4.3.3
```
collisionProc = mp.Process(target = collision_avoidance, args = (keepGoing, 5.0, 2.0))
collisionProc.start()
```

4.3.3.1 Create a collision avoidance process as collision_avoidance(keepGoing, 5.0, 2.0).

4.3.3.2 In the function call, 5.0 is the distance at which the drone responds to the collision. 2.0 is the interval between iterations of the while loop within the function.

4.3.3.3 Start the collision avoidance process.

4.3.3.4 I started the collision_avoidance process 2nd because I was originally passing different thread objects and such to it so that it could hopefully shut down the flight thread. I later switched to the multiprocessing approach and I think that I just never really reconsidered the

order. I think I was concerned about a race condition, but I can't recall specifically what I was concerned about.

4.3.3.5 This is the last line of the if block

4.4 Begin infinite loop in main

```
while(keepGoing.value == 1):
    continue
```

4.4.1

4.4.1.1 This just waits for the flight process or the collision avoidance process to change keepGoing to 0 to begin the exit procedures that are located after the infinite loop

4.5 Shutdown procedure in main

```
print("executing emergency landing")
bebop.emergency_land()
print("bebop emergency land sent")
bebop.smart_sleep(5)

if (not bebop.is_landed()):
    bebop.emergency_land()

bebop.disconnect()
```

4.5.1

4.5.1.1 This could have been done better. I could have included a 2nd multiprocess variable to control whether or not the shutdown should be handled as a standard shutdown or an emergency shutdown. As it stands, this will perform emergency shutdown procedures even if the bebop completes the flight process without incident. It will still print those two print statements even in a normal case.

4.5.1.2 bebop.emergency_land() is different from bebop.land() because it sends the land command to the drone on the high priority buffer. This causes it to immediately land even if it is in the middle of a maneuver. It gives the drone 5 seconds to land.

4.5.1.3 If the drone is still not landed, it will send the emergency land command again. This is a leftover from when I previously just used the normal land command. I found that it could take a long time for it to actually land, so it would basically always do an emergency land anyway. I could remove it without changing the actual effect of the code.

4.5.1.4 Disconnect from bebop and then the script is done.

5 Flight function

5.1 Definition

5.1.1 `def flight_plan(keepGoing):`

5.1.1.1 No parameters besides the keepGoing mp variable

5.2 Video configuration (commented out)

```
#     #Video Config
#     bebop.set_video_framerate("30_FPS")
#     bebop.set_video_resolutions("rec1080_stream480")
#     bebop.set_video_recording("quality")
```

5.2.1

5.2.1.1 The framerate can be a few different things. I think it can be 24 fps, 25fps, or 30 fps

5.2.1.2 The bebop has 2 options for streaming/recording. It can record in 1080p and stream in 480p, or it can record in 720p and stream in 720p. It is not clear to me why it couldn't just record and stream in 1080p. I imagine that whatever onboard storage it has is at least

reasonably fast, and streaming 1080p is not a super demanding requirement as far as I can tell.

5.2.1.3 The third command allows the selection between quality and time for the video mode. I'm not sure if this is doing other than what the 2nd command is doing. Information about all of these commands can be found in the ardrone3.xml and common.xml. There are a few other xml files that have information and commands, but they really aren't important here.

5.3 Takeoff

```
print("in flight plan")
print("sleeping")
 bebop.smart_sleep(3)
bebop.safe_takeoff(10)
print("Takeoff Done")
```

5.3.1

5.3.1.1 Printing "in flight plan" is to see that the flight process has started successfully.

5.3.1.2 During testing, the drone was taking off and then landing again before I was able to see if the collision avoidance worked, but instead of just adding a sleep command after takeoff like a normal person, I somehow decided (for some dumb reason) that it made more sense to put the sleep command before the takeoff command.

5.3.1.3 Because the drone does not exit a command function until the drone reports that all maneuvers are done, it is not always obvious when the takeoff is finished. The drone will take off and be in a stable hover for a few seconds before the drone reports that it is ready for the next command. Sending commands via the high priority buffer overrides this.

5.4 After-Takeoff routine

```
bebop.move_relative(0, 0, -1, 0)  #move_relative(self, dx, dy, dz, dradians)
print("Move Done")
#bebop.smart_sleep(5)

bebop.smart_sleep(20)

print("Landing")
bebop.safe_land(10)
```

5.4.1 `keepGoing.value = 0`

5.4.1.1 This is where the scripted actions for the drone to do are to be placed. In this case, it is a very simple routine.

5.4.1.2 The first command is a relative move. In this case, the drone is instructed to move 0 meters to the front, 0 meters to the right, 1 meter up (**negative is up**), and 0 radians of yaw to the right.

5.4.1.3 After the move has ended, the drone triggers the event [RelativeMoveEnded](#1-34-0). If you do send another relative move to the drone using the high priority buffer, it will trigger this event with the offsets it did before the new command came in, and then the new command will execute and, on completion, trigger this same event with the offsets performed by the 2nd maneuver.

5.4.1.4 After this, the most entertaining part of the show is the 20 second smart_sleep.

5.4.1.5 If the drone manages to get through all 20 seconds of that sleeping without getting slammed by a spoofed drone, it will do a normal landing and then set the mp keepGoing's value to 0. This ends the infinite loop in main and then executes whatever is after that loop.

6 Collision Avoidance Function

6.1 Definition

```
def collision_avoidance(keepGoing, personalSpace = 5.0, interval = 1.0):
    """
    Begin Collision Avoidance Routine

    :param keepGoing:
    :param personalSpace:
    :param interval:
    :return:
    """
```

6.1.1

6.1.1.1 This is the collision_avoidance function. personalSpace is the radius in FEET, interval is how often the infinite loop within the function repeats.

6.2 Initialization of Variables

```
print("collision avoidance started")

currentTime = time.time()
lastTime = 0

clearance = 0.00007

collisionState = 0
safe, caution, danger = 0, 1, 2
```

6.2.1

6.2.1.1 Like in the flight function, the first print statement is just to make it obvious that the collision avoidance process started successfully.

6.2.1.2 currentTime and lastTime are used to try to get the actual interval of the while loop closer to the specified interval by subtracting the time it took to execute the commands within the while loop from the interval and then telling it to sleep by that result.

6.2.1.3 clearance is used for testing the collision avoidance without the use of another computer. It represents the initial value subtracted from the drones latitude to represent the threat latitude. 0.00007 has the 7 in the one hundred thousandth's place. That coordinates to approximately 7.77 meters (7 * 1.11 meters). The distance passed to the function is in feet

because feet are better than meters (this is not up for debate ☺). I will make it clear further down where to change this if you want. clearance is decremented with each iteration of the loop to simulate the drone moving closer.

6.2.1.4 collisionState starts at safe (0) and can be changed to warning (1) or danger (2). I never finished implementing this, but you can create another mp variable in main to save the state that the drone exited the infinite while loop in main with. This can be used to print different things to avoid what happens in my code where it tries to do an emergency landing even if the flight process ends without incident.

6.3 Infinite loop to continually check collision hazards

```
while(1):
    lastTime = currentTime

    currentLat = bebop.sensors.sensors_dict["GpsLocationChanged_latitude"]
    currentLon = bebop.sensors.sensors_dict["GpsLocationChanged_longitude"]
6.3.1   currentAlt = bebop.sensors.sensors_dict["GpsLocationChanged_altitude"]
```

6.3.1.1 At the beginning of each iteration, lastTime is set to the time that the last iteration ended at.

6.3.1.2 Important Point: currentLat, currentLon, and currentAlt are examples of how to get data from the sensors of the drone. The drone updates its sensors internally 10 times per second. Printing out the sensors dictionary will provide a long list of the data you can get from the various sensors.

6.3.1.3 currentLat is the current GPS latitude of the drone

6.3.1.4 currentLon is the current GPS longitude of the drone

6.3.1.5 currentAlt is the current GPS altitude of the drone in meters above sea level, not above the ground.

```
#drone stores 500.0 when no reliable GPS signal
if(currentLat == 500.0):
    cprint("ERROR: Unreliable GPS Signal", "red")
    flightProc.terminate()
    keepGoing.value = 0
    return

if(currentLon == 500.0):
    cprint("ERROR: Unreliable GPS Signal", "red")
    flightProc.terminate()
    keepGoing.value = 0
    return

if(currentAlt == 500.0):
    cprint("ERROR: Unreliable GPS Signal", "red")
    flightProc.terminate()
    keepGoing.value = 0
    return
```

6.3.2

6.3.2.1 These 3 if statements are used to handle an inadequate GPS signal. 500.0 is stored for the latitude, longitude, ~~and altitude~~ if the GPS cannot get a good connection.

6.3.2.2 You can modify this to just leave the collision avoidance function (thereby terminating the collision avoidance process) without terminating the flight process. However, I have it programmed such that it will treat it as a collision threat and perform the shutdown routine after keepGoing is set to 0. **The infinite loop in main will exit and the scripted shutdown procedure will execute when keepGoing is 0, NOT when either process is terminated**. I am pretty sure that if the flight process is not terminated, it is technically possible that a takeoff command in the flight process will execute just before the disconnect happens in main, leaving the drone hovering at 1 meter with nothing controlling it. If it is let in a hover, it will stay in that state until the battery drains or the drone drifts into an object (**DO NOT, under any circumstances, ask how I know what happens when the drone crashes**). If you did then I'd have to admit that I have a tendency to crash drones and I have a **strict** policy of never admitting when I make mistakes ☺.

6.3.2.3 If the GPS cannot get an adequate signal, it will print out a red error message, terminate the flight process, set keepGoing to 0, and then exit the function. I am unsure if there is a race condition created by not explicitly terminating the collision avoidance function. The process may not terminate if the main function is completed before the process fully shuts down. I think it's a race that the collision avoidance process would always lose, but I'm not positive of that which is not a good practice.

```
currentCoords = (currentLat, currentLon)

threatLat = currentLat - clearance
threatLon = currentLon
threatAlt = currentAlt

clearance = clearance - 0.0000025

threatCoords = (threatLat, currentLon)
```

6.3.3

6.3.3.1 If the GPS signal is adequate, the currentLat and currentLon are stored together in a tuple to be passed to the geopy.distance.distance function.

6.3.3.2 The latitude of the threat is set to the latitude of the drone's current position minus the value in clearance. There is no check to ensure that the resulting latitude value is valid (between -90.0 and 90.0) which would likely cause an error when the distance function is called.

6.3.3.3 The threat longitude and threat altitude are set to the drone's current longitude and altitude. This is just because I was trying to test the collision avoidance function without the use of a separate computer. These values would all be received from the attacking computer in the scenario I was trying to simulate.

6.3.3.4 clearance is decremented by a value I chose. With this value and the default interval value, it shouldn't take too long, but will take long enough to watch what is happening.

6.3.3.5 The threat coordinates are stored together as a tuple in threatCoords

```
            threatDistance = geopy.distance.distance(currentCoords, threatCoords).feet
            deltaAlt = abs(currentAlt - threatAlt)
            #Doug Chamberlain
            threatSeparation = math.hypot(threatDistance, deltaAlt)

6.3.4       print(threatSeparation)
```

6.3.4.1 threatDistance is the top-down distance between the drone's location and the threat's location in **feet**. Changing the .feet to .m will change the units to meters to match the rest of the code. It's just a matter of personal preference. I personally choose to cause unit consistency problems wherever I can.

6.3.4.2 The difference in altitude is stored in deltaAlt. It does not matter if it is negative or positive because it will be squared in the math.hypot() function.

6.3.4.3 threatSeparation stores the true 3D distance between the two targets. This means that a target can fly directly under the drone as long as the difference in altitude is greater than the required separation.

6.3.4.4 Finally is a print statement with the true separation **in whatever unit was specified after the geopy.distance.distance function call**.

```
        if(threatSeparation <= personalSpace):
            cprint("Collision Avoidance Activated", "red")
            flightProc.terminate()
            #cprint("Flight Process Terminated")
            keepGoing.value = 0
            #cprint("keepGoing set to 0")
6.3.5       return
```

6.3.5.1 If the threatSeparation is less than or equal to personalSpace, an impending collision is detected.

6.3.5.2 A red message is posted to the console stating that the collision avoidance has been activated.

6.3.5.3 The flight process is immediately terminated. This is to ensure that there is only one source that is sending commands to the drone at any given time.

6.3.5.4 keepGoing is set to 0 which will stop the infinite loop in main before the next iteration.

6.3.5.5 The collision avoidance process terminates on, or sometime after, the return. I am not completely sure.

```
currentTime = time.time()
timeUntilNextFrame = interval - (currentTime - lastTime)
bebop.smart_sleep(timeUntilNextFrame)
```

6.3.6 `currentTime = time.time()`

6.3.6.1 This is a continuation of the stuff I mentioned before. At the beginning of the iteration, lastTime was set to currentTime. currentTime is either the time when the variable was initialized, or the time just before sleeping at the end of the last iteration. I previously did not have another currentTime = time.time() at the end, but testing elsewhere had issues if I did not. I don't know why it didn't have problems before.

6.3.6.2 The purpose of this is to subtract the time at the beginning of this iteration from the time at the end to determine how long it took to run the code in the iteration. Then, that value can be subtracted from the desired interval to achieve a more accurate interval over time. If it were not coded this way, there still shouldn't be any major issues, but the actual interval would be slightly longer than the interval passed to the function because it would not account for the time the commands take to execute on each iteration.

7 ADS-B Decoding

7.1 Example ADS-B Messages

7.1.1 Original Even Message:

7.1.1.1 8D40621D58C382D690C8AC2863A7

7.1.2 Original Odd Message:

7.1.2.1 8D40621D58C386435CC412692AD6

7.2 Downlink Format [1-5] = 10001 = 17

7.3 Capability [6-8] = 101 = 5

7.4 ICAO Address [9-32] = 010000000011000100000111101 = 0x40621D

7.5 Type Code [33-37] = 01011 = 11

7.6 Aircraft Identification

7.6.1 Immediately following the Type Code **if the type code is 1 – 4**, the aircraft's airline/flight information is given.

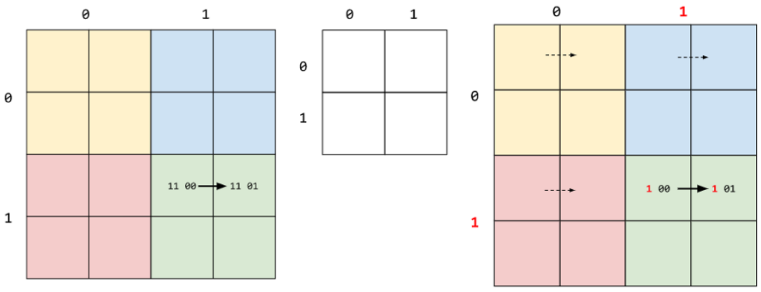
7.6.1.1 These identification messages can be decoded using the lookup table provided at https://mode-s.org/decode/adsb/identification.html

7.6.2 These example messages are type 11, so they contain the airborne position in the data segment of the message

7.7 Compact Position Reporting

7.7.1 In order to fit more data into the same message size, the position given by one message is globally ambiguous. It does not narrow the location down to an absolute location. With an even message and an odd message, the absolute location can be determined.

7.7.2 The following two diagrams illustrate how a location that requires 4 bits to positively identify can be illustrated with 3 bits in two messages. The first bit of each message is combined together to take the place of the "missing" bit. For this reason, two even message would not help. Neither would two odd messages. One even message and one odd message is required to positively identify an absolute, unambiguous location. The middle box split into quarters is intended to demonstrate that 2 bits are required to identify the location within one of the 4 main quadrants of the 16 box grid. With only one message, there are still two possible boxes that would be valid locations absent the other message.

7.7.3 

7.7.4 For positional ADS-B messages (Type Code 11 is among them), the 54th bit is 0 in odd messages and 1 in even messages.

7.8 Surveillance Status [38-39] = 00 = 0

7.9 NIC supplement-B [40] = 0 = 0

7.10 Altitude [41-52] = 110000111000 = 38000 ft.

7.11 Time [53] = 0

7.12 Even/Odd Frame Flag [54] = 0 or 1 (Even or Odd)

7.13 CPR Latitude (Even) [55-71] = 10110101101001000

7.14 CPR Longitude (Even) [72-88] = 01100100010101100

7.15 CPR Latitude (Odd) [55-71] = 10010000110101110

7.16 CPR Longitude (Odd) [71-88] = 01100010000010010

7.17 CRC Parity Code

7.17.1 The 24 parity bits in an ADS-B message are the result of a Cyclic Redundancy Check (CRC) in order to verify that the message received is the message sent, and that the message received is properly encoded.

7.18 CRC Parity Code (Even) [89-112] = 001010000011000111010011

7.19 CRC Parity Code (Odd) [89-112] = 011010010010101011010110

7.20 Decoding with pyModeS

7.20.1 To make the console output easier to read, I used termcolor to print colored messages to the console. The colors made it easy to discern the different messages from each other. I defined a function to do the ADS-B printing for me because I'm lazy and got sick of typing it out by hand.

```
def adsbPrint(adsb, textColor = "white", attrs=[]):
    cprint("ADS-B Hex:    " + str(adsb), textColor)
    cprint("ADS-B Binary:    " + str(bin(int(adsb, 16))[2:]), textColor)
    cprint("")
    cprint("ICAO:    " + str(pms.adsb.icao(adsb)) + "          " +
        ...."Typecode:    " + str(pms.adsb.typecode(adsb)) + "          " +
        ...."BDS Reg.:    " + str(pms.bds.infer(adsb)) + "          " +
        ...."Altitude:    " + str(pms.decoder.bds.bds05.altitude(adsb)) + "ft.", textColor)
    cprint("")
```

7.20.1.1

7.20.1.2 As an example, here are 4 calls to this function:

```
cprint("Original Even Message: ", "magenta", attrs=["underline"])
adsbPrint(adsb_A, "magenta")
cprint("Encoded Even Message: ", "cyan", attrs=["underline"])
adsbPrint(even, "cyan")
cprint("Original Odd Message: ", "magenta", attrs=["underline"])
adsbPrint(adsb_B, "magenta")
cprint("Encoded Odd Message: ", "yellow", attrs=["underline"])
adsbPrint(odd, "yellow")
```

7.20.1.2.1

7.20.1.3 For the Even ADS-B Message, the following is the output:

```
Original Even Message:
ADS-B Hex:    8D40621D58C382D690C8AC2863A7
ADS-B Binary:    10001101010000000110001000011101010110001100001110000010110101101001000011001000101011000010100001100011101001111
```

7.20.1.3.1

```
ICAO:   40621D          Typecode:   11          BDS Reg.:   BDS05          Altitude:   38000ft.
```

8 Encoding DF17, Type 11 ADS-B Messages

8.1 Problems with unmodified github code

8.1.1 Using the code straight from github produced errors because it was written for (I think) python 2.7. Python 2.7 did not require parenthesis in the print statement (print xyz) vs (print(xyz)). The code I've included has been fixed, to the best of my ability, for python 3.7.

8.1.2 There are some other errors that affect the positional accuracy of the encoded GPS messages. I have not been able to identify why the lat/lon are never exactly what they should be.

8.1.3 Depending on what time is passed to the decoding and encoding functions, the resulting encoded messages is invalid. I did not fully work through this, but I did develop a workaround. With a bit more experimentation, I think I could figure out the problem, but I ran out of time.

8.2 Workaround for invalid messages

8.2.1 To get around these problems, I just called the encoding and decoding functions twice and saved the valid messages for each. As seen in the following image:

```
ca = 5
tc = 11
ss = 0
nicsb = 0
time = 0
icao = int("40621D", 16)
surface = False

trash = []

alt = pms.decoder.bds.bds05.altitude(adsb_A)

lat, lon = pms.decoder.bds.bds05.airborne_position(adsb_A, adsb_B, 1, 0)
#print(lat, lon)
df17_even, trash = out.df17_pos_rep_encode(ca, icao, tc, ss, nicsb, alt, time, lat, lon, surface)

lat, lon = pms.decoder.bds.bds05.airborne_position(adsb_A, adsb_B, 0, 0)
#print(lat, lon)
trash, df17_odd = out.df17_pos_rep_encode(ca, icao, tc, ss, nicsb, alt, time, lat, lon, surface)

even = (''.join(format(x, '02x').upper() for x in df17_even))
odd = (''.join(format(x, '02x').upper() for x in df17_odd))
```

8.2.2

9 Connection between computers

9.1 Ultimate Outcome

9.1.1 I was able to get the client/server code working and transmitting fine on my home computer and laptop. However, I had trouble when trying to replicate the setup between my laptop and a groupmate's laptop.

9.1.2 I had some issues with the encoding and decoding. For whatever reason, it seemed that the encoding happened twice. When I decoded the message, I was still left with a b'8D40621D58C382D690C8AC2863A7'. The encoded message I received was b"b'8D40621D58C382D690C8AC2863A7'" which seems to indicate that it was encoded twice. However, when trying to decode the message a second time, I was getting errors. My suspicion is that the errors were because the message was no longer encoded and that the string was simply "b'8D40621D58C382D690C8AC2863A7'". Python has libraries functions built in that can strip those characters if I didn't find a solution anyway, but I ran out of time to try to implement something for that.

9.2 Server Code:

9.2.1

```
host = str(net.ifaddresses('en0')[net.AF_INET][0]['addr'])
port = 5555

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

try:
    s.bind((host, port))
except socket.error as e:
    print(str(e))

s.listen(5)

print("Waiting for a connection")

conn, addr = s.accept()
print('connected to: ' + addr[0] + ':' + str(addr[1]))

servProc = mp.Process(threaded_client, (conn, ))
servProc.start()
```

9.2.1.1 I found this code on youtube or something. I modified it some to automate setting the internal IP to my own system. On the client, it's virtually the same, but the host is the server's IP. The way that the server side works is that it starts a separate threaded_client process (it was originally threaded, but I was traumatized by my probably 6 hour long attempt to get simple

threading to work, so I used the multiprocessing module instead). This means that multiple clients can connect to the same server.

```
def threaded_client(conn):
    conn.send(str.encode("Welcome, Type your info\n"))

    while True:
        data = conn.recv(2048)
        reply = "Server Output: " + data.decode("utf-8")
        if not data:
            break
        conn.sendall(str.encode(reply))
```

9.2.2 `conn.close()`

9.2.2.1 Implementation for the threaded_client.

10 Results; Stuff that never got completed; etc.

10.1 ADS-B messages were never fully integrated with the drone script. It would not be too difficult to do that, but I ran out of time.

10.2 I never fully worked out why the ADS-B messages were not being encoded/decoded properly. There seemed to be some small errors in the GPS location even when the encoded messages did match the original messages bit for bit. It seems very strange.

10.3 I never was able to get two computers transmitting unencrypted (or encrypted) ADS-B messages over a wireless connection.

10.4 It was fun!