Tutorial for developing applications for the drone challenge

Objective: The objective of this tutorial is to help develop and verify sample applications using the programmable platform offered by CentMesh drones. We also introduce mavutil, a python module that provides additional wrappers for some functions offered by pymavlink. Code snippets that contrast the two ways of achieving the desired functionality (with and without mavutil) are also discussed.

Change History:
Version 1.0: not tracked
Version 1.1: Created by Ramachandra Kasyap Marmavula, March 1, 2014

More Context: CentMesh website and wiki: http://centmesh.csc.ncsu.edu

---

Introduction

Part I: General Tutorial

1.1 How to write programs using pymavlink
- pymavlink
- Developing an application
- Utility functions
- Commonly used functionality
  - Given a requirement, how to choose the right API?

1.2 Introduction to mavutil
- Using mavutil
  - Creation of an object of type mavlink_apm
  - Waiting for a particular type of message
  - Getting the current mode of the UAV
  - Setting the UAV in a particular mode
  - Arming the UAV
- Are there wrappers for each and every function in mavutil?
- Do we need mavlink_apm.py as well?

1.3 Sample applications
- Sample application 1
- Sample application 2

1.4 Sample applications - alternate approaches
- Sample application 1
  - Creation of a ‘connection object’

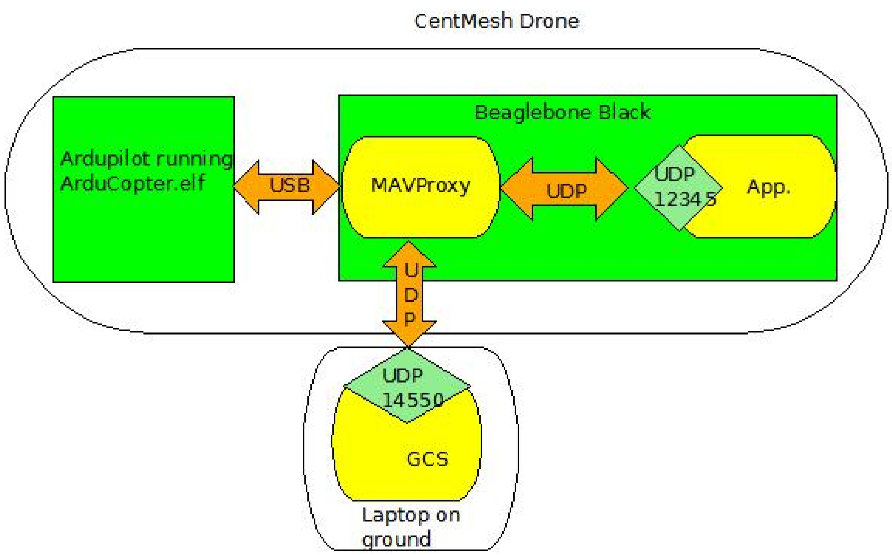
Setting the UAV mode to AUTO
ARM the UAV
Waiting for messages with GPS information
Saving GPS information
Change in saved file format
Sample APP 2
Creation of a ‘connection object’:
ARM the UAV
Change in parse_gps_info()
Setting the waypoint
Saving sensor data

Part II: Virtual Environment Usage Guide

2.1 Using SITL with Sample applications
Sample application 1
Steps for running the sample application
Sample application 2
Steps for running the sample application
Running the sensing platform:

Introduction

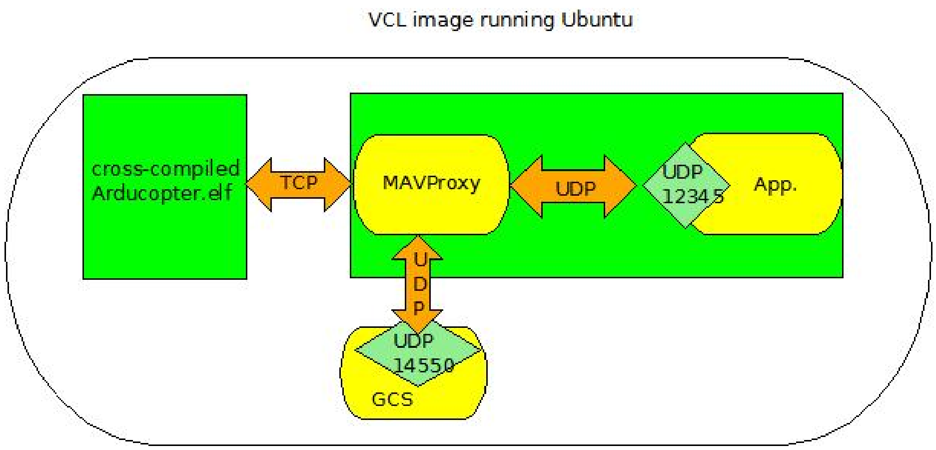
The figure below shows the components involved in any application that runs on the CM drones.



The detailed description of each of these components is available in the CentMesh wiki documentation:

[APM](#)
[Beaglebone black](#)
[GCS](#)
[MAVProxy](#)

Virtual Environment: Using a Software-in-the-Loop (SITL - documentation and further detail [here](#)) simulation, the entire setup can be run in a single Virtual Machine image. The representation (simplified) is displayed by the following figure. Such a VM images has been created by the CentMesh team, and made available as a loadable image in the NCSU Virtual Computing Laboratory (VCL).



The rest of this tutorial is organized as follows. Part I is written in general terms, and should apply equally well to the real drone platform, or the VCL VM, or a stand-alone VM built according to the instructions on the CentMesh wiki. However, for ease of reference, we repeatedly make references to the VCL image, in boxes such as the one below.

Information about sample code locations, sample commands, etc. specifically referring to the VCL image will appear in boxes like these.

Part II is entirely a guide to using the virtual drone programming environment, using SITL on VCL.

Part I: General Tutorial

1.1 How to write programs using pymavlink

pymavlink

Python bindings for MAVLink are provided by [pymavlink](#). Using the interfaces provided by pymavlink, developers can:

- Receive MAVLink messages from the UAV, and
- Send MAVLink Messages to the UAV.

We are using pymavlink to communicate with the MAV and this tutorial describes its usage. You are welcome to use any other resources (like this) that you may find useful.

Developing an application

1) Import 'mavlink_apm.py': This module can 'decode' and 'encode' (pack and unpack) MAVLink messages. It needs to be included in your python program using 'import':

import mavlink_apm

In the VCL image, the module is already present under $HOME/application_code/lib. If you need to generate the same in a VM, please follow the steps mentioned here

2) 'mavlink_apm.py' depends on other modules in the same source tree. So, to make sure that the dependencies are resolved, the 'generator' directory (generated by the python code generator that takes xml as its input) and all of its sub-directories are present in same directory where mavlink_apm.py resides and the following is to be included:

```
sys.path.insert(0, os.path.join(os.path.dirname(os.path.realpath(__file__)), '../../lib'))
```

3) Import mavutil.py using:

import mavutil

Please refer this section for details on mavutil

4) 'mavutil.py' depends on other modules in the same source tree. So, to make sure that the dependencies are resolved, the following is to be included:

```
sys.path.insert(0, os.path.join(os.path.dirname(os.path.realpath(__file__)),
'../../mavlink/mavlink/pymavlink'))
```

5) Import the following commonly used python libraries:

import re, sys, os, socket, select, time, tempfile, struct, ast

6) The application needs to create a UDP socket for reading messages from MAVProxy.
Sample code:

```
HOST = ''
```

```
 mavproxy_port = 12345
# Create a server socket for MAVProxy
mavproxy_sock = socket.socket (socket.AF_INET,socket.SOCK_DGRAM)
mavproxy_sock.setblocking(0)
mavproxy_sock.bind((HOST, mavproxy_port))
```

7) Create a MAVLink instance, which will perform ‘encode’/’decode’ operations on the socket created above

```
mav_obj = mavlink_apm.MAVLink (mavproxy_sock)
```

8) To read messages from the UAV, the decode() method is used. Sample code:

```
 # Call to receive data over UDP socket. 1024 is the buffer size
data_from_mavproxy,address_of_mavproxy = mavproxy_sock.recvfrom (1024)
decoded_message = mav_obj.decode(data_from_mavproxy)
```

9) To access individual fields, there are several useful methods. For example, get_msgId() returns the message ID sent by the UAV (the message ID is used to distinguish the different possible messages between the UAV and the sample application). Sample code:

```
print("Got a message with id %u, fields: %s, component: %d, System ID: %d" %
(decoded_message.get_msgId(), decoded_message.get_fieldnames(),
decoded_message.get_srcComponent(), decoded_message.get_srcSystem()))
# Prints the entire decode message
print (decoded_message)
```

Sample output for a heartbeat message:

```
Got a message with id 0, fields: ['type', 'autopilot', 'base_mode', 'custom_mode',
'system_status', 'mavlink_version'], component: 1, System ID: 1
 # the decoded message
 HEARTBEAT {type : 2, autopilot : 3, base_mode : 89, custom_mode : 3,
 system_status : 4, mavlink_version : 3}
```

10) To send commands to the UAV, there are variations of encode() method. For example, to set a waypoint the UAV has to travel to, the following method is used:

```
mission_item_encode()
```

Utility functions

The current mode of UAV is present in the 'hearbeat' message generated periodically by the UAV. We expect 'reading the current UAV mode' and setting the current UAV mode to be common operations. So, these operations are provided in a module called FTL_util.py (can be imported by using *import FTL_util*)
To use this, an instance needs to be created:

FTL_util_obj = FTL_util.FTL_util()

The current mode of the UAV can be obtained by passing the heartbeat_message as an argument to get_mav_mode() function. Sample code that returns the mode as a string:

mode = get_mav_mode (heartbeat_message)

The following sample code sets the UAV in AUTO mode:

FTL_util_obj = FTL_util.FTL_util()
FTL_util_obj.set_mav_mode(FTL_util_obj.auto_mode,mav_obj,
mavproxy_sock, address_of_mavproxy)

More utility functions may be added as needed.

Commonly used functionality

Each MAVLink message has a message ID that corresponds to the type of message. For example, the 'heartbeat' message that is sent by the UAV has a message ID of zero. This is given by **MAVLINK_MSG_ID_HEARTBEAT** in mavlink_apm.py. Similarly, a message with GPS information has the message ID **MAVLINK_MSG_ID_GPS_RAW_INT**. The following code checks if a message received is a heartbeat or it contains GPS information:

```
if msg_id == mavlink_apm.MAVLINK_MSG_ID_GPS_RAW_INT:
       print 'This is a message with GPS information!'
elif msg_id == mavlink_apm.MAVLINK_MSG_ID_HEARTBEAT:
       print 'This is a heartbeat message'
```

Refer page 4 of this document for description of other commonly used message IDs.

Each command sent to the UAV for performing actions (i.e. setting mode, setting a way-point etc.) needs to have a command ID. For example, to arm/disarm the UAV, there is

MAV_CMD_COMPONENT_ARM_DISARM. It can be accessed by using mavlink_apm.MAV_CMD_COMPONET_ARM_DISARM. Similarly to set a way-point, MAV_CMD_NAV_WAYPOINT is used and it can be accessed by mavlink_apm.MAV_CMD_NAV_WAYPOINT.

Given a requirement, how to choose the right API?

Refer mavlink_apm.py. Browse through the methods under ‘mavlink’ class (line 3008) to find the one that best suits your requirement. If you are still not sure, refer mavproxy.py and mavutil.py. mavproxy.py uses of the same library for sending messages. For example, entering ‘arm throttle’ in MAVProxy terminal will ARM the UAV. Checking mavproxy.py, we see that the cmd_arm() function handles this command, which in turn invokes arducopter_arm() in mavuitl.py. This is a wrapper function which finally calls command_long_send() which is defined in mavlink_apm.py and that is what we need to use in our sample applications.

1.2 Introduction to mavutil

mavutil offers many useful functions that can minimize the amount of necessary code for an application. The following sections list the changes that mavutil introduces for performing a certain functionality. The sample applications (introduced later in this document) have been implemented both with and without using mavutil. This section contrasts the code snippets involved.

Using mavutil

We review the most commonly used functions in this section.

Creation of an object of type mavlink_apm

With mavutil, we create an object of type mavfile which has a reference to an object of type mavlink_apm. To create an object of type mavfile, we need to build a string in the following format: “udp:<IP address of the machine running mavproxy><port number used by mavproxy>”. Since mavproxy always runs in the same system, the IP address is always 127.0.0.1 (localhost). Sample code:

```
device = "udp:127.0.0.1:12345"
mavproxy_conn = mavutil.mavlink_connection(device)
```

Waiting for a particular type of message

We frequently come scenarios where the application ‘blocks’ waiting for a particular message - heartbeat, GPS etc. We can perform the same using a single function recv_match():

```
data_from_mavproxy = mavproxy_conn.recv_match (type='HEARTBEAT', blocking = True)
```

The ‘type’ specifies the type of the message that we wish to wait for, ‘blocking’ specifies whether we wish to block waiting for this message.

Getting the current mode of the UAV

This involves two steps:

- Wait for the heartbeat message
- Once the heartbeat message is received, obtain the mode from it:

Sample code:

```
heartbeat_message = mavproxy_conn.recv_match (type = 'HEARTBEAT', blocking = True)
flightmode = mavutil.mode_string_v10(heartbeat_message) # Gets the mode as a string
```

Setting the UAV in a particular mode

To set a UAV in auto mode, the following call is sufficient:

```
mavproxy_conn.set_mode_auto()
```

Similarly, to set the UAV in RTL mode, the following call is sufficient:

```
mavproxy_conn.set_mode_rtl()
```

Arming the UAV

Arming the UAV can be performed using ‘arducopter_arm()’ function. The following code snippet arms the UAV if it is not already armed. Then it waits untill the UAV is armed:

```
if not mavproxy_conn.motors_armed(): # Function to check if UAV is armed
    mavproxy_conn.arducopter_arm()    # Function to ARM the UAV
    #Do not proceed until the UAV is armed
    mavproxy_conn.motors_armed_wait() # Function to wait till the UAV is armed.
```

Are there wrappers for each and every function in mavutil?

No. As mentioned earlier, mavutil provides wrappers for functions present in mavlink.py/mavlink_apm.py. However this is not an exhaustive list. For those functions which do not have wrappers, we need to work with the ‘mavlink’ object in mavfile. More details of how to do this with examples are available in the ‘Sample applications - alternate approaches’ section.

Do we need mavlink_apm.py as well?

Yes. We still need mavlink_apm.py to access some of the constants. For example, in the second sample application where the UAV is directed to traverse to specific waypoints, we need the following:

frame = **mavlink_apm**.MAV_FRAME_GLOBAL_RELATIVE_ALT
command = **mavlink_apm**.MAV_CMD_NAV_WAYPOINT

1.3 Sample applications

In this section, we discuss two sample applications. Two implementations of each sample application are discussed in this tutorial:

- Using mavutil (no FTL_util module needed)
- No mavutil involved (FTL_util module needed)

> The VCL image has source code for these sample applications. Code for the implementations using mavutil (no FTL_util module needed) is in the '/home/droneusr/application_code/sample_prog/' directory. Code for the FTL_util version (no mavutil needed) is in the '/home/droneusr/application_code/sample_prog_mavutil'/ directory.

Sample application 1

This sample application performs the following operations:

1) Sets the UAV to AUTO mode
2) Listens on UDP port 12345 for messages from MAVProxy
3) Reads the GPS data that is generated by autopilot (in a virtual environment, the SITL generates simulated GPS data) and forwarded by MAVProxy. Saves this raw data to a file every 5 seconds.

Code for the sample application is available in
/home/droneusr/application_code/sample_prog/program_1 (without using mavutil)
/home/droneusr/application_code/sample_prog_mavutil/program_1 (using mavutil)

Sample application 2

This sample application performs the following operations:

1) Listens on UDP port 12345 for messages from MAVProxy
2) Reads the file that has been generated by the first sample application, converts each reading to a WayPoint and guides the UAV to that WayPoint. This process is repeated for each line in the file every 5 seconds. Once all the entries in the file are read, the UAV is set to 'Return to Launch (RTL)' mode.

Code for the sample application is available in
/home/droneusr/application_code/sample_prog/program_2 (without using mavutil)
/home/droneusr/application_code/sample_prog_mavutil/program_2 (using mavutil)

1.4 Sample applications - alternate approaches

As mentioned above, we can achieve the same functionality with or without using mavutil. In most cases, the number of lines of code for achieving a particular functionality reduces with the usage of mavutil. However, it is important to understand both the approaches as mavutil has limitations and we need to work with mavlink_apm.py in such cases.

This section provides both the code snippets for achieving a particular functionality in the sample applications.

Sample application 1

The app parameters: ./uav_auto_mode.py <mavproxy port> <File to save GPS info>

Creation of a 'connection object'

When mavutil is not used, we create a UDP server socket to listen for messages from mavproxy. When using mavutil, we don't deal with sockets directly, as they are abstracted by the mavfile object. We obtain a reference to the connection object by using the following lines. Also, we don't need to specifically obtain the address of mavproxy.

Code snippet (no mavutil):

```
        mavproxy_port = int(sys.argv[1])
         HOST = ''
        # Create a server socket for MAVProxy
        mavproxy_sock = socket.socket (socket.AF_INET,socket.SOCK_DGRAM)
        print 'created UDP socket for MAVProxy'
        mavproxy_sock.setblocking(0)
        mavproxy_sock.bind((HOST,mavproxy_port))
        print 'Binding socket for MAVProxy connection'
        # Create the mavproxy object
        mav_obj = mavlink_apm.MAVLink (mavproxy_sock)
…
          #Get the address of mavproxy
         address_of_mavproxy = get_mavproxy_address (mav_obj, mavproxy_sock)
         return_status = FTL_util_obj.set_mav_mode(FTL_util_obj.auto_mode,mav_obj,
                                             mavproxy_sock, address_of_mavproxy)
```

Code snippet (using mavutil):

```
        HOST = ''
        # Create a mavlink connection to communicate with MAVProxy
        device = "udp:127.0.0.1:"+str(mavproxy_port)

        mavproxy_conn = mavutil.mavlink_connection(device)
```

```
        print 'created MAVLink connection to communicate with MAVProxy'
        print mavproxy_conn.mav
        data_from_mavproxy = mavproxy_conn.recv_match (type='HEARTBEAT', blocking
= True)
```

Setting the UAV mode to AUTO

Without mavutil, FTL_util is used to set the UAV mode. It is not required when mavutil is used. Please note that in the code that uses mavutil, we set the mode to AUTO and then verify if it is actually set (and reset if it is not).
Code snippet (no mavutil):

```
        return_status = FTL_util_obj.set_mav_mode(FTL_util_obj.auto_mode,mav_obj,
                                                  mavproxy_sock, address_of_mavproxy)
        if return_status < 0:
            print "Error while setting mode, please check the parameters passed..."
            sys.exit(1)
```

Code snippet (using mavutil):

```
        is_mode_set = False
        while (not is_mode_set):
          # Call to set the UAV in auto mode
          mavproxy_conn.set_mode_auto()
          # Wait for hearbeat message. This is needed to verify the mode
          # as the hearbeat message contains the mode
          heartbeat_message = mavproxy_conn.recv_match (type = 'HEARTBEAT',
blocking = True)
          flightmode = mavutil.mode_string_v10(heartbeat_message)
          if (flightmode.lower() == 'AUTO'.lower()):
              print 'mode set to AUTO'
              is_mode_set = True
```

ARM the UAV

Without mavutil, we have

```
          # ARM the UAV
          component_id = mavlink_apm.MAV_COMP_ID_SYSTEM_CONTROL
          # Same command for arming or disarming, arm_flag controls whether the UAV
          # armed or disarmed. arm_flag=1->arm, arm_flag=0->disarm
          command = mavlink_apm.MAV_CMD_COMPONENT_ARM_DISARM
```

```
    arm_flag = 1
    # Number of confirmations needed for this command. 0 means immediately
    confirmation = 0
    # Other parameters are ignored by this command and are to be set to zero.
    PARAM_IGNORE = 0
    msg = mav_obj.command_long_encode (1,component_id,command,confirmation,
                                       arm_flag,PARAM_IGNORE,PARAM_IGNORE,

PARAM_IGNORE,PARAM_IGNORE,PARAM_IGNORE,
                                       PARAM_IGNORE)
    try:
        mavproxy_sock.sendto(msg.get_msgbuf(),(address_of_mavproxy))
    except socket.error as v:
        print "Exception when trying to ARM the copter:"
        print os.strerror(v.errno)
    print "ARMED"
```

which can be replaced by

```
# Check if the UAV is already armed, if it is, skip this step
if not mavproxy_conn.motors_armed():
    mavproxy_conn.arducopter_arm()
    #Do not proceed until the UAV is armed
    mavproxy_conn.motors_armed_wait()
print 'ARMED!'
```

Waiting for messages with GPS information

Code snippet (no mavutil):

```
    data_from_mavproxy,address_of_mavproxy = mavproxy_sock.recvfrom
(MAX_SIZE)
    decoded_message = mav_obj.decode(data_from_mavproxy)
    msg_id = decoded_message.get_msgId()
    # Check if this information is GPS information.
    if msg_id == mavlink_apm.MAVLINK_MSG_ID_GPS_RAW_INT:
        save_gps_info(decoded_message, file_gps_info)
```

Code snippet (using mavutil):

```
    gps_info = mavproxy_conn.recv_match (type = 'GPS_RAW_INT', blocking = True)
    save_gps_info (gps_info, file_gps_info)
```

Saving GPS information

The way we save GPS information in this tutorial is slightly different than the one presented during the tutorial on Feb. 12. This change is not related to mavutil. The following is a better way of saving the GPS information such that the second sample app can read it easily without the need of complex regular expressions. This is being noted here to highlight the change. We perform two steps in the updated code:

- The payload of the message is obtained using the function get_payload(). This step is needed to get the message as a stream of bytes rather than a string.
- This stream of bytes is 'unpacked' using a format string and saved as a tuple to the file.

For more details about structure packing and unpacking, please refer this.

Code in 'save_gps_info()' function (used in the sample application implementation without mavutil):

```
new_time = datetime.now()
if (new_time - current_time).seconds >= int(INTERVAL):
    print "Time to save!!!!"
    with open(file_gps_info, "a") as file_reference:
        file_reference.write("\n" + str(decoded_message))
        file_reference.close()
    # Update the current timers
    current_time = new_time
```

Replacement code:

```
new_time = datetime.now()
if (new_time - current_time).seconds >= int(INTERVAL):
    print "Time to save!!!!"
    # Get the payload, this returns the message in the form of bytes
    # (rather than a string).
    payload = decoded_message.get_payload()
    fmt =
            mavlink_apm.mavlink_map
        [mavlink_apm.MAVLINK_MSG_ID_GPS_RAW_INT][0]
    tuple = struct.unpack (fmt,payload)

    with open(file_gps_info, "a") as file_reference:
        file_reference.write("\n" + repr(tuple))
        file_reference.close()
```

```
# Update the current timers
current_time = new_time
```

Change in saved file format

Due to the change in the way the file is saved, the format of a line in the saved file is changed from:

```
GPS_RAW_INT {time_usec : 251908000, fix_type : 3, lat : 357713121, lon : -786743912, alt : 584090, eph : 0, epv : 65535, vel : 0, cog : 0, satellites_visible : 10}
```

to

```
(251908000, 357713121, -786743912, 584090, 0, 65535, 0, 0, 3, 10)
```

Sample APP 2

Creation of a 'connection object':

When mavutil is not used, we create a UDP server socket to listen for messages from mavproxy. When using mavutil, we don't deal with sockets directly here, it is abstracted by the mavfile object. We obtain a reference to it by using the following lines. Also, we don't need to explicitly obtain the UDP port and address of mavproxy.
Code snippet (no mavutil):

```
        mavproxy_port = int(sys.argv[1])
         HOST = ''
        # Create a server socket for MAVProxy
        mavproxy_sock = socket.socket (socket.AF_INET,socket.SOCK_DGRAM)
        print 'created UDP socket for MAVProxy'
        mavproxy_sock.setblocking(0)
        mavproxy_sock.bind((HOST,mavproxy_port))
        print 'Binding socket for MAVProxy connection'
        # Create the mavproxy object
        mav_obj = mavlink_apm.MAVLink (mavproxy_sock)
…
          #Get the address of mavproxy
        address_of_mavproxy = get_mavproxy_address (mav_obj, mavproxy_sock)
        return_status = FTL_util_obj.set_mav_mode(FTL_util_obj.auto_mode,mav_obj,
                                        mavproxy_sock, address_of_mavproxy)
```

Code snippet (using mavutil)

```
HOST = ''
# Create a mavlink connection to communicate with MAVProxy
device = "udp:127.0.0.1:"+str(mavproxy_port)

mavproxy_conn = mavutil.mavlink_connection(device)
print 'created MAVLink connection to communicate with MAVProxy'
print mavproxy_conn.mav
data_from_mavproxy = mavproxy_conn.recv_match (type='HEARTBEAT', blocking
= True)
```

ARM the UAV

Without mavutil, we have

```
# ARM the UAV
component_id = mavlink_apm.MAV_COMP_ID_SYSTEM_CONTROL
# Same command for arming or disarming, arm_flag controls whether the UAV
# armed or disarmed. arm_flag=1->arm, arm_flag=0->disarm
command = mavlink_apm.MAV_CMD_COMPONENT_ARM_DISARM
arm_flag = 1
# Number of confirmations needed for this command. 0 means immediately
confirmation = 0
# Other parameters are ignored by this command and are to be set to zero.
PARAM_IGNORE = 0
msg = mav_obj.command_long_encode (1,component_id,command,confirmation,
                                   arm_flag,PARAM_IGNORE,PARAM_IGNORE,

PARAM_IGNORE,PARAM_IGNORE,PARAM_IGNORE,
                                   PARAM_IGNORE)
try:
    mavproxy_sock.sendto(msg.get_msgbuf(),(address_of_mavproxy))
except socket.error as v:
    print "Exception when trying to ARM the copter:"
    print os.strerror(v.errno)
print "ARMED"
```

which can be replaced by

```
# Check if the UAV is already armed, if it is, skip this step
if not mavproxy_conn.motors_armed():
    mavproxy_conn.arducopter_arm()
    #Do not proceed until the UAV is armed
```

```
    mavproxy_conn.motors_armed_wait()
print 'ARMED!'
```

Change in parse_gps_info()

Because of the way the file is saved by sample application 1, the use of regular expressions is avoided.

Code snippet (no mavutil):

```
latitude = longitude = altitude = 0
# Get Latitude
match_for_latitude =  re.search(r'lat : (.+?),',gps_info)
if not match_for_latitude:
    print "Error - no latitude information found, return!"
    return 0,0,0
else:
    latitude = match_for_latitude.group(1)

# Get longitude
match_for_longitude =  re.search(r'lon : (.+?),',gps_info)
if not match_for_longitude:
    print "Error - no longitude information found, return!"
    return 0,0,0
else:
    longitude = match_for_longitude.group(1)

# Get altitude
match_for_altitude =  re.search(r'alt : (.+?),',gps_info)
if not match_for_altitude:
    print "Error - no altitude information found, return!"
    return 0,0,0
else:
    altitude = match_for_altitude.group(1)

return latitude, longitude, altitude
```

Code snippet (using mavutil):

```
latitude = longitude = altitude = 0
# The first sample program has saved the GPS information as a tuple.
# We need to convert the string (read from the file) back to a tuple.
```

```
# From the format of the tuple, we know that second, third and fourth
# values correspond to latittude,longitude and altitude respectively.
tuple_from_string = ast.literal_eval(gps_info)
latitude = tuple_from_string[1]
longitude = tuple_from_string[2]
altitude = tuple_from_string[3]
return latitude, longitude, altitude
```

Setting the waypoint

There is not a significant difference between the two approaches here. Since no wrapper exists for mission_item_encode(), we use it as is. The slight difference is when mavutil is not used, we access mission_item_encode() using

mav_obj.mission_item_encode()

whereas in other case, we access it using

mavproxy_conn.mav.mission_item_encode()

The other difference (as mentioned earlier) is we do not deal with the UDP socket directly rather make use of the wrappers provided.

Code snippet (no mavutil):

```
# Send the waypoint message and wait till the ACK is received for 2 seconds
list_read_sockets = [mavproxy_sock]
list_write_sockets = list_error_sockets = []
while 1:
    msg = mav_obj.mission_item_encode(target_system,
target_component,seq,frame,command,current,autocontinue,param1,param2,param3
param4,latitude,longitude,altitude - 584.0)
    try:
       mavproxy_sock.sendto(msg.get_msgbuf(),(address_of_mavproxy))
    except socket.error as v:
       print "Exception when setting WP:"
       print os.strerror(v.errno)
       time.sleep(1)
       continue
    readable, writable, error = select.select(list_read_sockets,
                                   list_write_sockets,
                                   list_error_sockets,INTERVAL)
    if readable:
```

```
            data_from_mavproxy,address_of_mavproxy = mavproxy_sock.recvfrom
(MAX_SIZE)
            decoded_message = mav_obj.decode(data_from_mavproxy)
            msg_id = decoded_message.get_msgId()

            # Check if this is a waypoint request message
            if msg_id == mavlink_apm.MAVLINK_MSG_ID_MISSION_ACK:
                print 'ACK for this way-point received received...'
                break
            else:
                #print "Expected message ID 47, received %d" % msg_id
                continue
        else:
            # Send the message again
            print "Timeout on receiving waypoint ACK message"
            continue
```

Code snippet (using mavutil):

```
    while 1:
        msg = mavproxy_conn.mav.mission_item_encode(target_system,
target_component,seq,frame,command,current,autocontinue,param1,param2,param3
param4,latitude,longitude,altitude - 584.0)
        try:
            mavproxy_conn.write(msg.get_msgbuf())
        except socket.error as v:
            print "Exception when setting WP:"
            print os.strerror(v.errno)
            time.sleep(1)
            continue
        # Wait for a waypoint ACK
        msg = mavproxy_conn.recv_match(type='MISSION_ACK', blocking = True,
timeout = 2)
        print str(msg)
        # If we have reached here beacause of a timeout, continue
        if (not msg):
            continue
        else:
            print 'Waypoint set as expected..'
            break
```

Saving sensor data

Any Sensor data that is read by applications running on the UAV can be saved to periodically to a repository. To enable this, a sensing daemon runs continually as part of the CentMesh Mobile Node platform. Applications can provide sensor data they collect to this daemon, the daemon then periodically reports this data to a distributed server running on CentMesh, and
Sensing daemon (client): This reads sensor data periodically and reports the readings to a server. The type of sensor, the frequency of reading etc. are specified in a file 'config.properties'.
Server: The server receives readings for various data from different clients and saves this to a repository. This server runs on static CentMesh nodes, and not on the CentMesh Mobile Node.

The VCL image has a basic sensing daemon that achieves this functionality, and sample cod that demonstrates its usage. The sample code works with sample application 1. In this example, the GPS readings are treated as sensor data. To recall, the sample application keeps saving the GPS readings to a file every 5 seconds. In addition to that, it also writes the current GPS reading to a file "/tmp/gps_data". So each time the UAV receives a message wit GPS information, the older reading is replaced by the current one in this file. The sensing application reads this file periodically and sends it to the server. Whenever the sensing application reads the file, it is guaranteed to get the latest GPS reading (as that is written continuously).

Part II: Virtual Environment Usage Guide

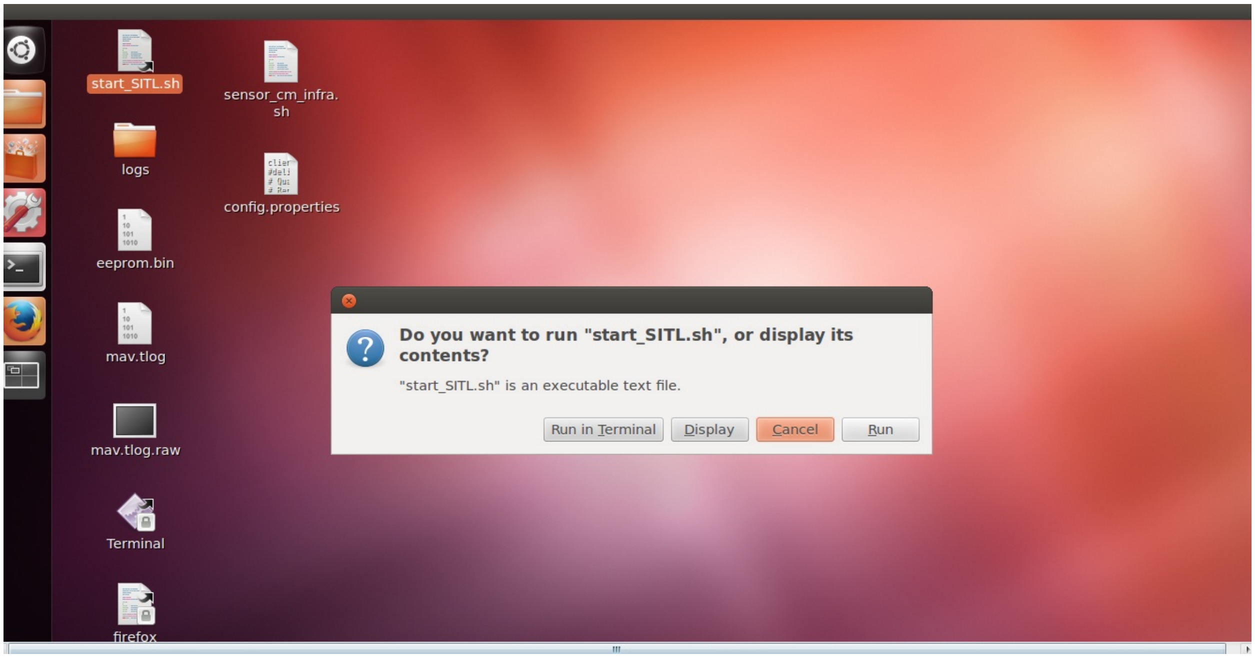
2.1 Using SITL with Sample applications

This section describes how to validate a sample application using SITL.

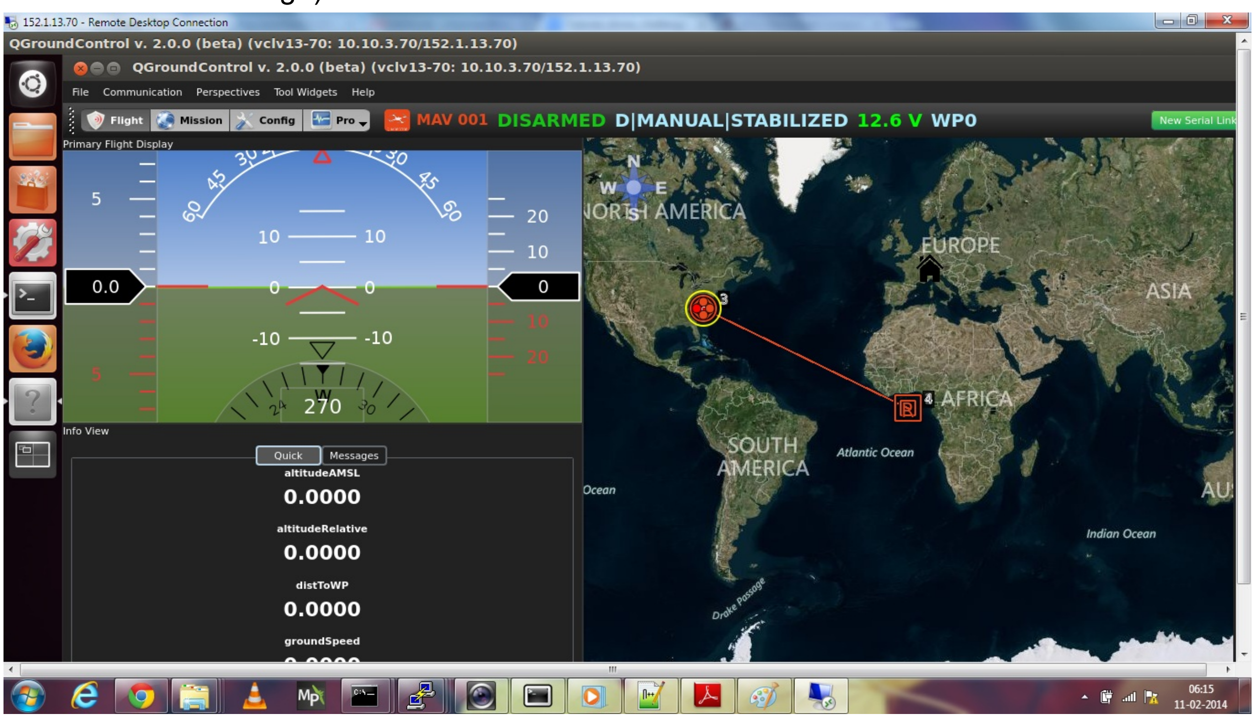
Sample application 1

Steps for running the sample application

Step 1: Make a reservation for 'APM_Copter_3DRobotics_SITL_Image' using
https://vcl.ncsu.edu/
Step 2: Use the Windows Remote Desktop Connection to this computer using the following credentials:
Username: droneusr
Password: drone123
Step 3: Running SITL: Double click start_SITL.sh. Select 'Run' when prompted:



This should open 4 tabs one of which runs the qgroundcontrol (the Ground Control Station that is bundled with this image):

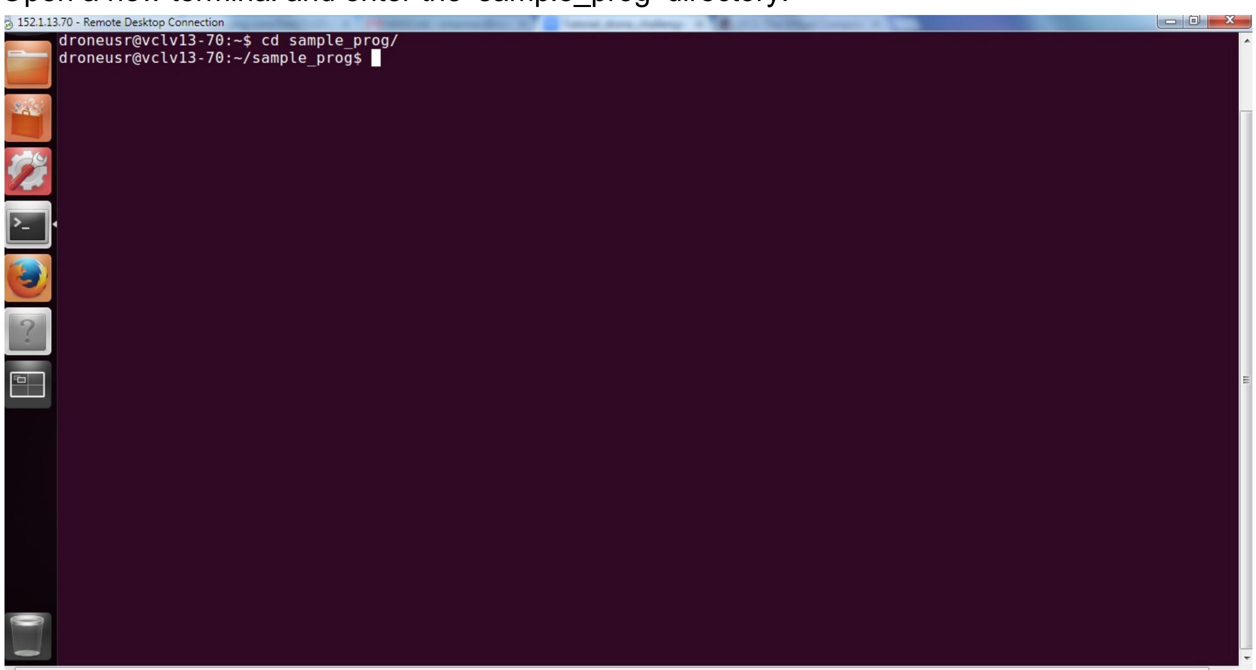


Zoom in using the side-bar on the right so that you have get a decent view of the UAV:

NOTE: This has to be performed manually the first-time you run SITL after reserving an image.For subsequent runs, you can click on ‘Last Pos’ and you will be taken to the last position that you were looking at

Step 4: Running sample app 1:

Open a new terminal and enter the ‘sample_prog’ directory:

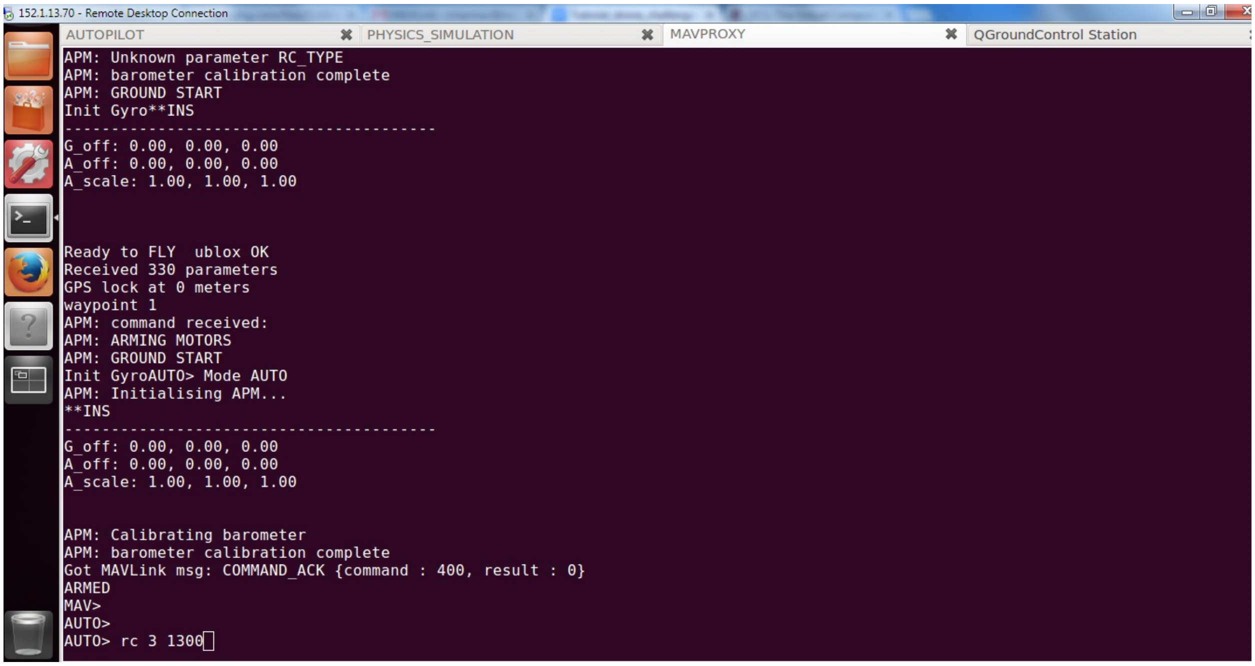


Enter program_1 directory and start the app using the following syntax:

./uav_auto_mode.py <MAVProxy port to connect to> <name of the file>

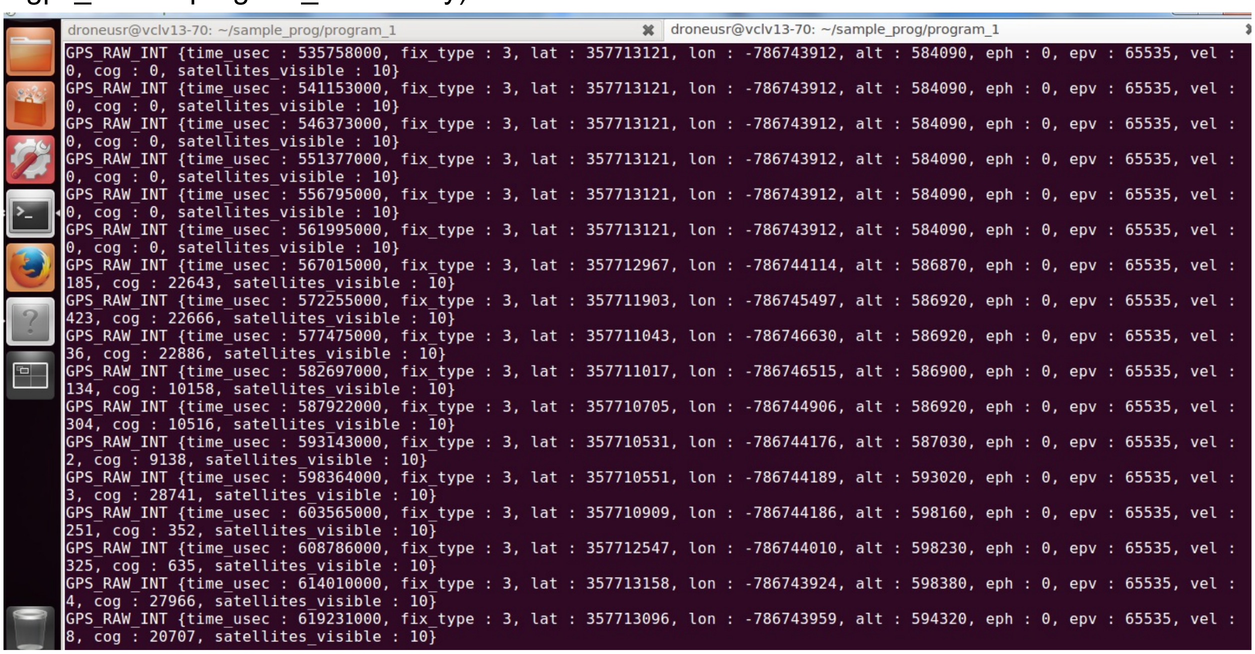
In our example, the MAVProxy port is always 12345. We specify a file named gps_data to which the readings are saved periodically (every 5 seconds).

Step 5: Provide throttle input by entering the following command on MAVProxy terminal: AUTO>rc 3 1300



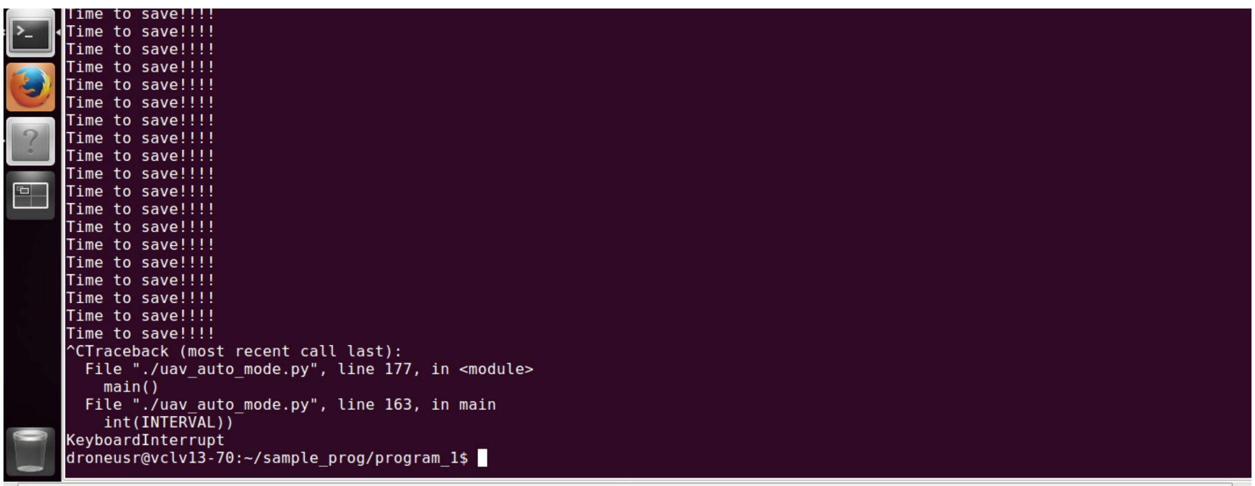
```
AUTOPILOT    PHYSICS_SIMULATION    MAVPROXY    QGroundControl Station
APM: Unknown parameter RC_TYPE
APM: barometer calibration complete
APM: GROUND START
Init Gyro**INS
----------------------------------------
G_off: 0.00, 0.00, 0.00
A_off: 0.00, 0.00, 0.00
A_scale: 1.00, 1.00, 1.00


Ready to FLY  ublox OK
Received 330 parameters
GPS lock at 0 meters
waypoint 1
APM: command received:
APM: ARMING MOTORS
APM: GROUND START
Init GyroAUTO> Mode AUTO
APM: Initialising APM...
**INS
----------------------------------------
G_off: 0.00, 0.00, 0.00
A_off: 0.00, 0.00, 0.00
A_scale: 1.00, 1.00, 1.00


APM: Calibrating barometer
APM: barometer calibration complete
Got MAVLink msg: COMMAND_ACK {command : 400, result : 0}
ARMED
MAV>
AUTO>
AUTO> rc 3 1300
```

Step 6: Observe the file ‘gps_data’ getting populated with GPS readings every 5 seconds (run tail -f gps_data in program_1 directory):



```
droneusr@vclv13-70: ~/sample_prog/program_1    droneusr@vclv13-70: ~/sample_prog/program_1
GPS_RAW_INT {time_usec : 535758000, fix_type : 3, lat : 357713121, lon : -786743912, alt : 584090, eph : 0, epv : 65535, vel :
0, cog : 0, satellites_visible : 10}
GPS_RAW_INT {time_usec : 541153000, fix_type : 3, lat : 357713121, lon : -786743912, alt : 584090, eph : 0, epv : 65535, vel :
0, cog : 0, satellites_visible : 10}
GPS_RAW_INT {time_usec : 546373000, fix_type : 3, lat : 357713121, lon : -786743912, alt : 584090, eph : 0, epv : 65535, vel :
0, cog : 0, satellites_visible : 10}
GPS_RAW_INT {time_usec : 551377000, fix_type : 3, lat : 357713121, lon : -786743912, alt : 584090, eph : 0, epv : 65535, vel :
0, cog : 0, satellites_visible : 10}
GPS_RAW_INT {time_usec : 556795000, fix_type : 3, lat : 357713121, lon : -786743912, alt : 584090, eph : 0, epv : 65535, vel :
0, cog : 0, satellites_visible : 10}
GPS_RAW_INT {time_usec : 561995000, fix_type : 3, lat : 357713121, lon : -786743912, alt : 584090, eph : 0, epv : 65535, vel :
0, cog : 0, satellites_visible : 10}
GPS_RAW_INT {time_usec : 567015000, fix_type : 3, lat : 357712967, lon : -786744114, alt : 586870, eph : 0, epv : 65535, vel :
185, cog : 22643, satellites_visible : 10}
GPS_RAW_INT {time_usec : 572255000, fix_type : 3, lat : 357711903, lon : -786745497, alt : 586920, eph : 0, epv : 65535, vel :
423, cog : 22666, satellites_visible : 10}
GPS_RAW_INT {time_usec : 577475000, fix_type : 3, lat : 357711043, lon : -786746630, alt : 586920, eph : 0, epv : 65535, vel :
36, cog : 22886, satellites_visible : 10}
GPS_RAW_INT {time_usec : 582697000, fix_type : 3, lat : 357711017, lon : -786746515, alt : 586900, eph : 0, epv : 65535, vel :
134, cog : 10158, satellites_visible : 10}
GPS_RAW_INT {time_usec : 587922000, fix_type : 3, lat : 357710705, lon : -786744906, alt : 586920, eph : 0, epv : 65535, vel :
304, cog : 10516, satellites_visible : 10}
GPS_RAW_INT {time_usec : 593143000, fix_type : 3, lat : 357710531, lon : -786744176, alt : 587030, eph : 0, epv : 65535, vel :
2, cog : 9138, satellites_visible : 10}
GPS_RAW_INT {time_usec : 598364000, fix_type : 3, lat : 357710551, lon : -786744189, alt : 593020, eph : 0, epv : 65535, vel :
3, cog : 28741, satellites_visible : 10}
GPS_RAW_INT {time_usec : 603565000, fix_type : 3, lat : 357710909, lon : -786744186, alt : 598160, eph : 0, epv : 65535, vel :
251, cog : 352, satellites_visible : 10}
GPS_RAW_INT {time_usec : 608786000, fix_type : 3, lat : 357712547, lon : -786744010, alt : 598230, eph : 0, epv : 65535, vel :
325, cog : 635, satellites_visible : 10}
GPS_RAW_INT {time_usec : 614010000, fix_type : 3, lat : 357713158, lon : -786743924, alt : 598380, eph : 0, epv : 65535, vel :
4, cog : 27966, satellites_visible : 10}
GPS_RAW_INT {time_usec : 619231000, fix_type : 3, lat : 357713096, lon : -786743959, alt : 594320, eph : 0, epv : 65535, vel :
8, cog : 20707, satellites_visible : 10}
```

Step 7: Stop the sample application (by giving a Ctrl-C) after the UAV has completed going through all the waypoints:

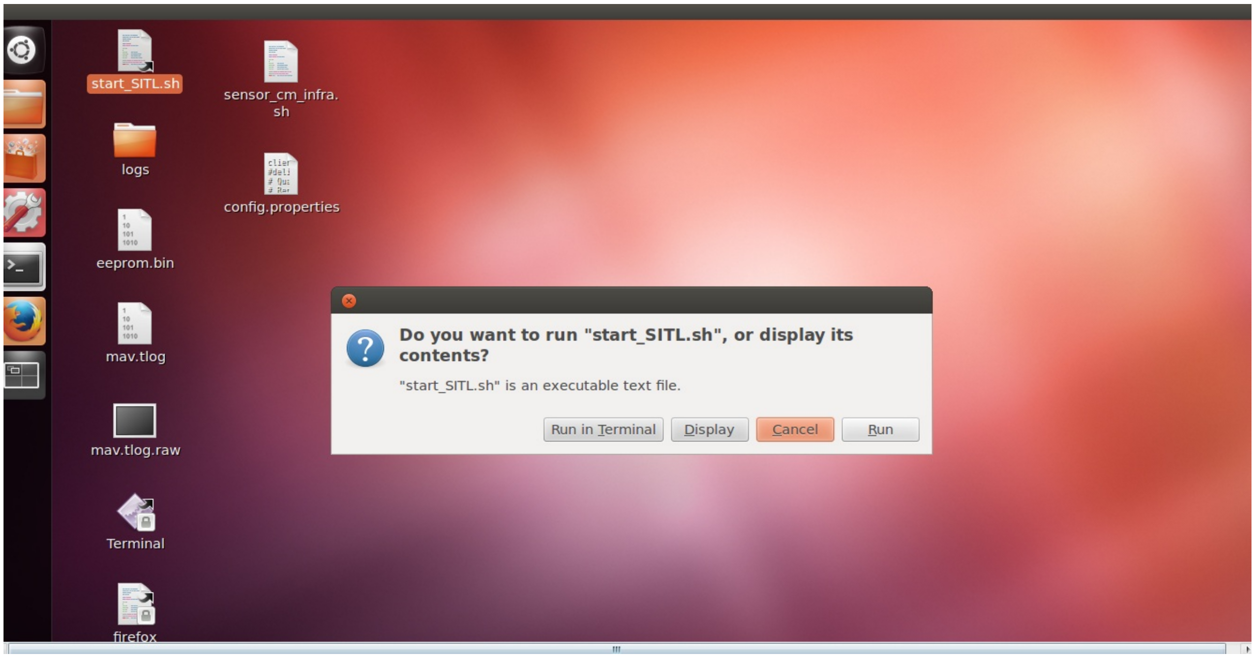


Following video demonstrates the working of this sample application (this uses Mission planner as the ground control station).

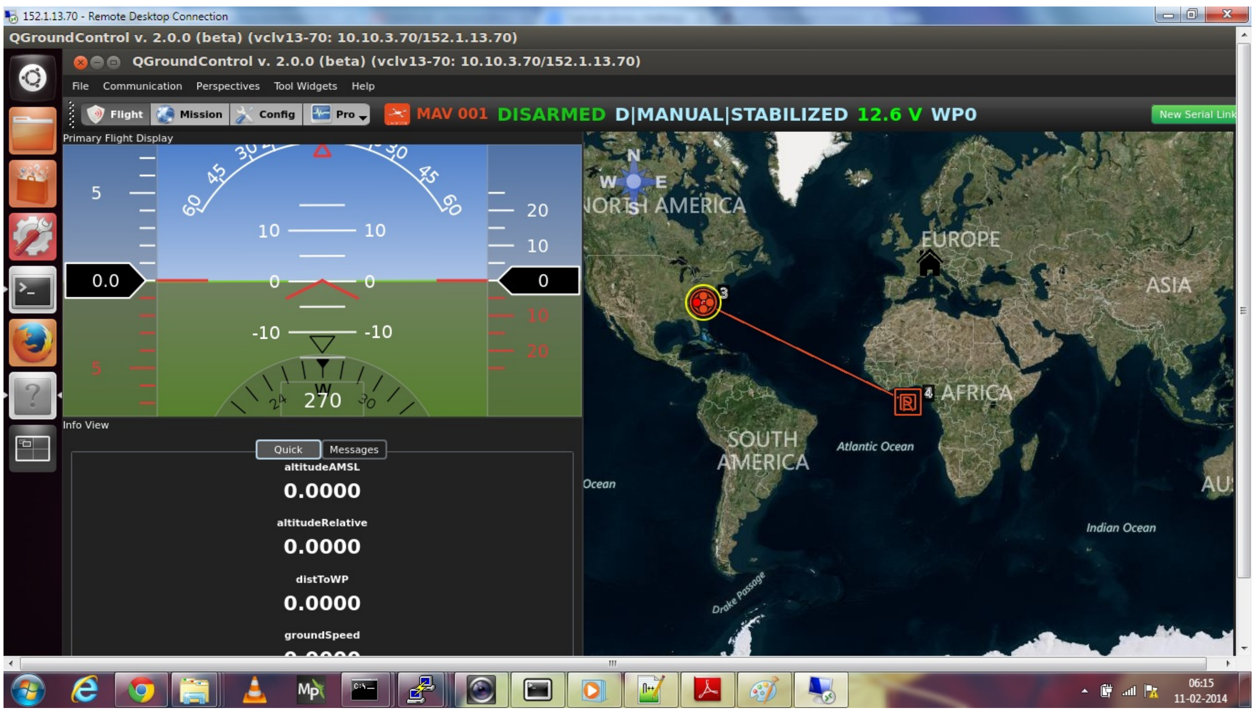
Sample application 2

Steps for running the sample application

Step 1: Make a reservation for ‘APM_Copter_3DRobotics_SITL_Image’ using https://vcl.ncsu.edu/
Step 2: Use the Windows Remote Desktop Connection to this computer using the following credentials:
Username: droneusr
Password: drone123
Step 3: Running SITL: Click start_SITL.sh. Select ‘Run’ when prompted:



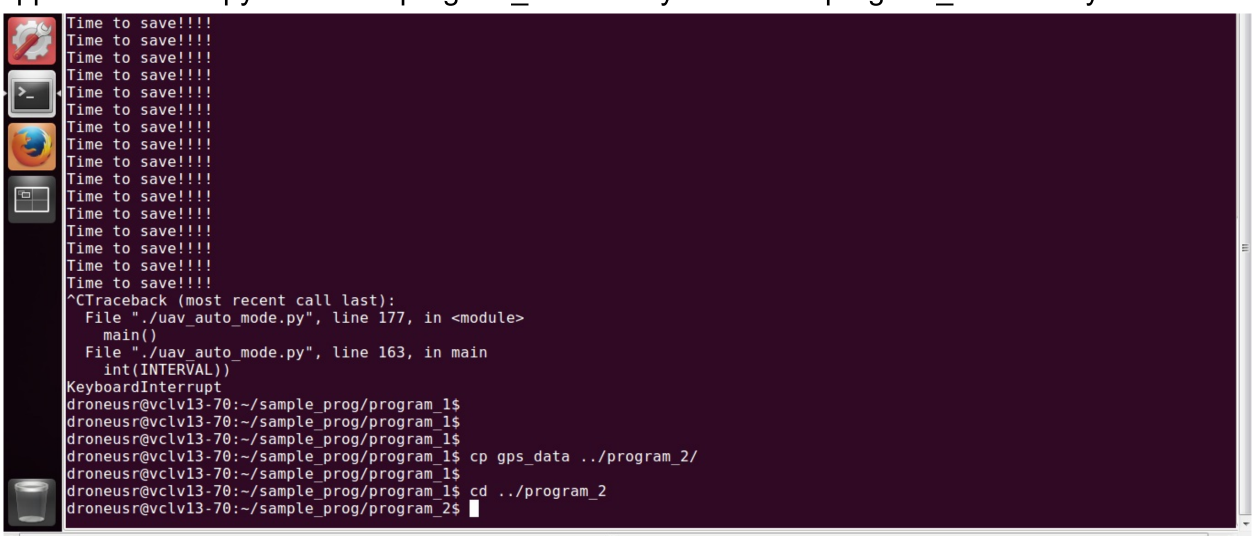
This should open 4 tabs one of which runs the qgroundcontrol (the Ground Control Station that is bundled with this image):



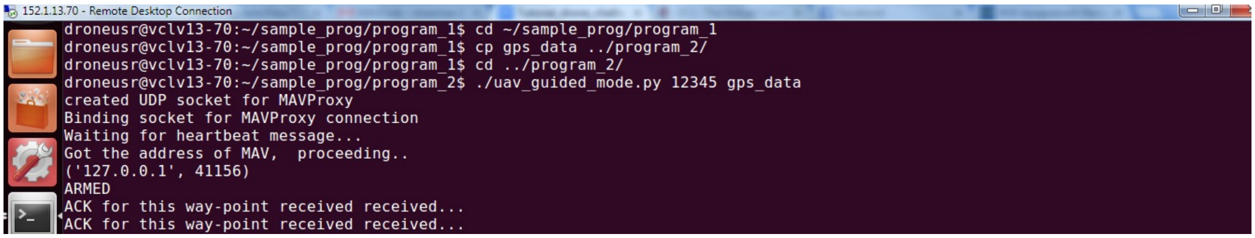
Zoom in using the side-bar on the right so that you have get a decent view of the copter:

NOTE: This has to be performed manually the first-time you run SITL after reserving an image.For subsequent runs, you can click on 'Last Pos' and you will be taken to the last position that you were looking at

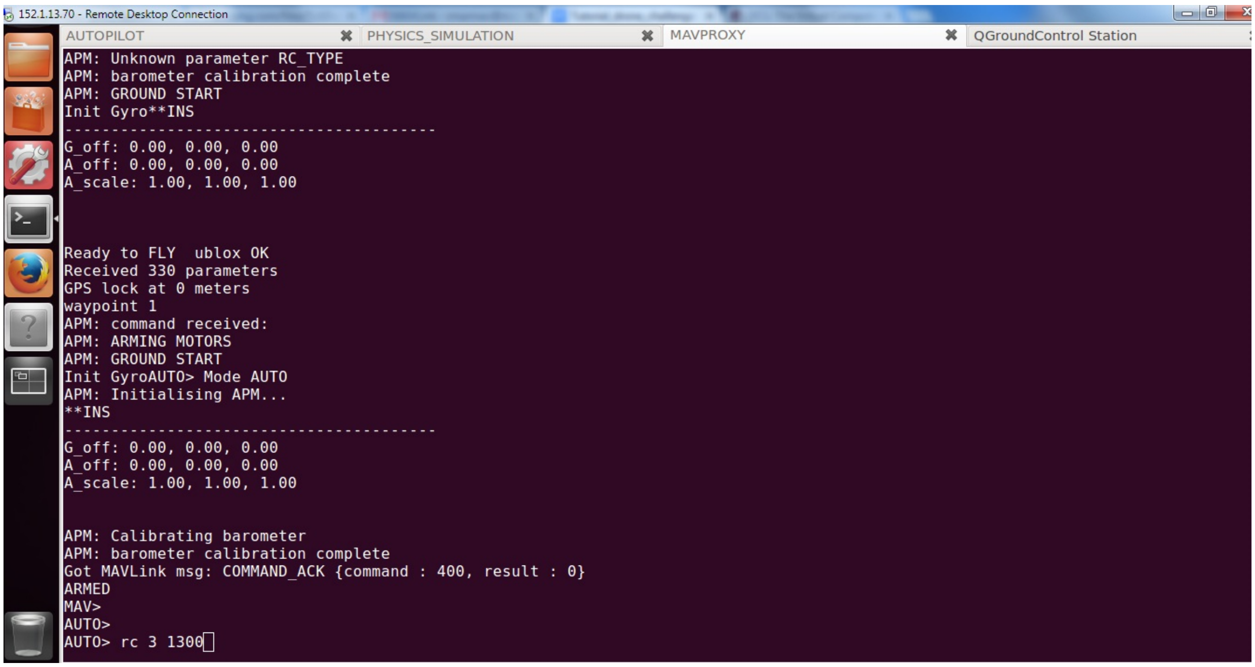
Step 4: Running sample application 2: We make use of the file that was generated by sample application 1. Copy the file to 'program_2' directory and enter 'program_2' directory:



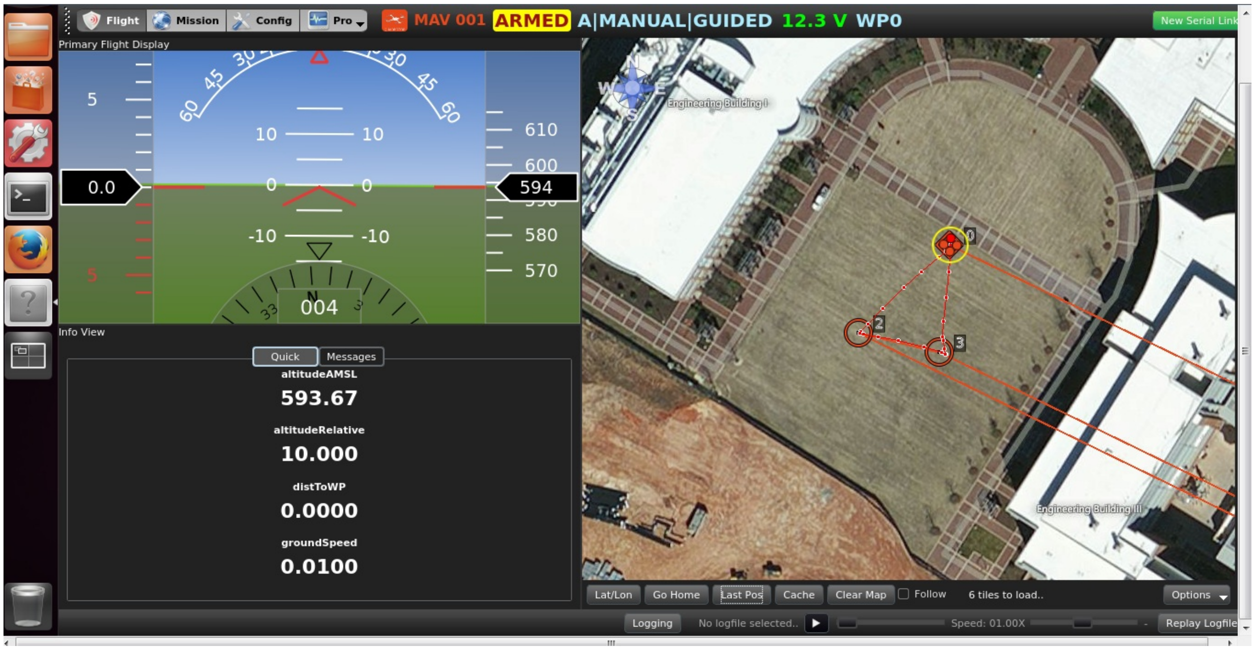
Step 6: Start the sample application:



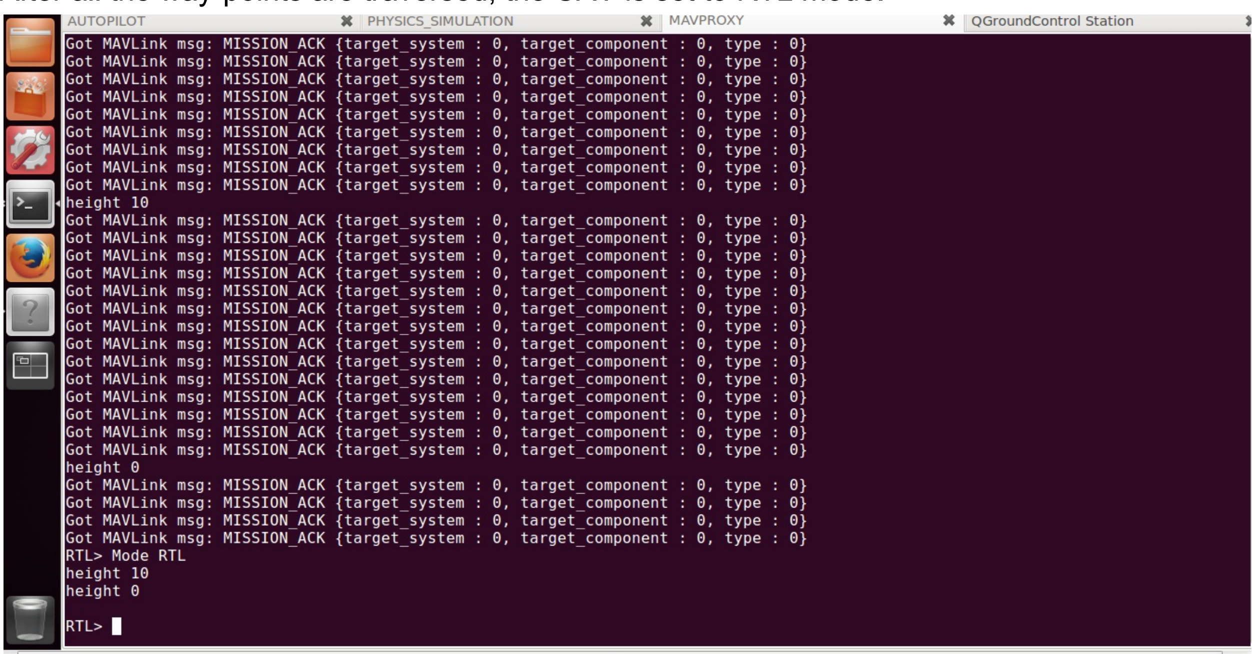
Step 7: Provide throttle by entering the following command on MAVProxy terminal:
AUTO>rc 3 1300



Step 8: The UAV should follow a similar path as the UAV that flew in AUTO mode (sample application 1).



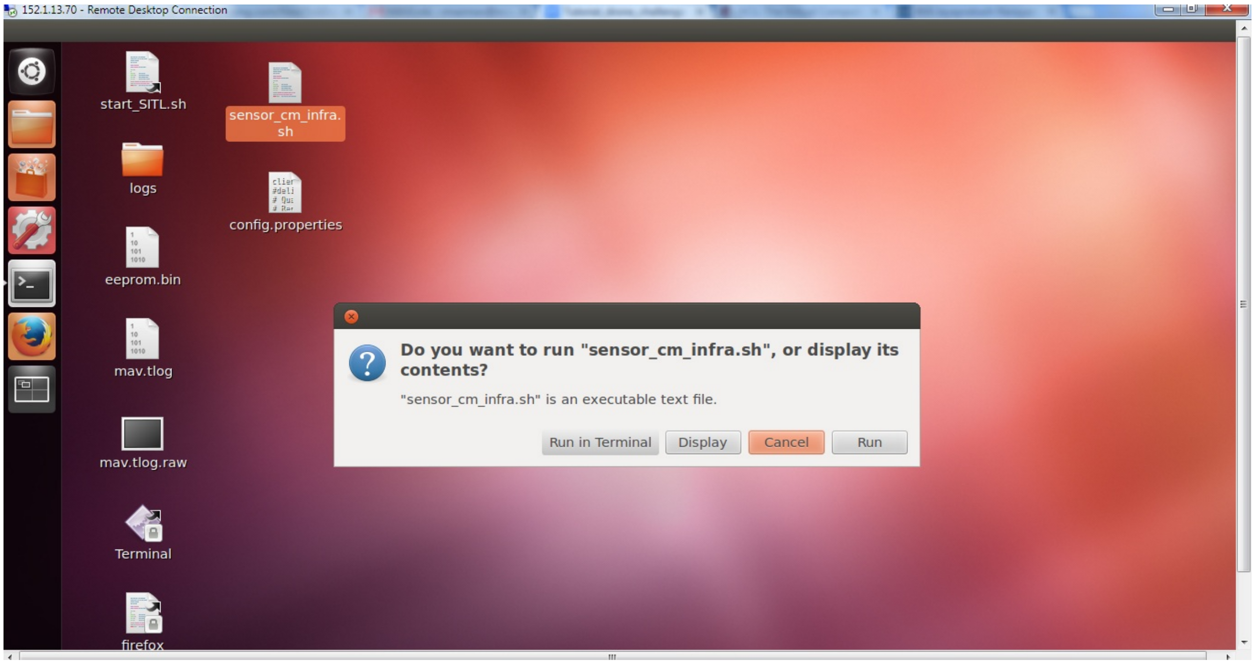
After all the way-points are traversed, the UAV is set to RTL mode.



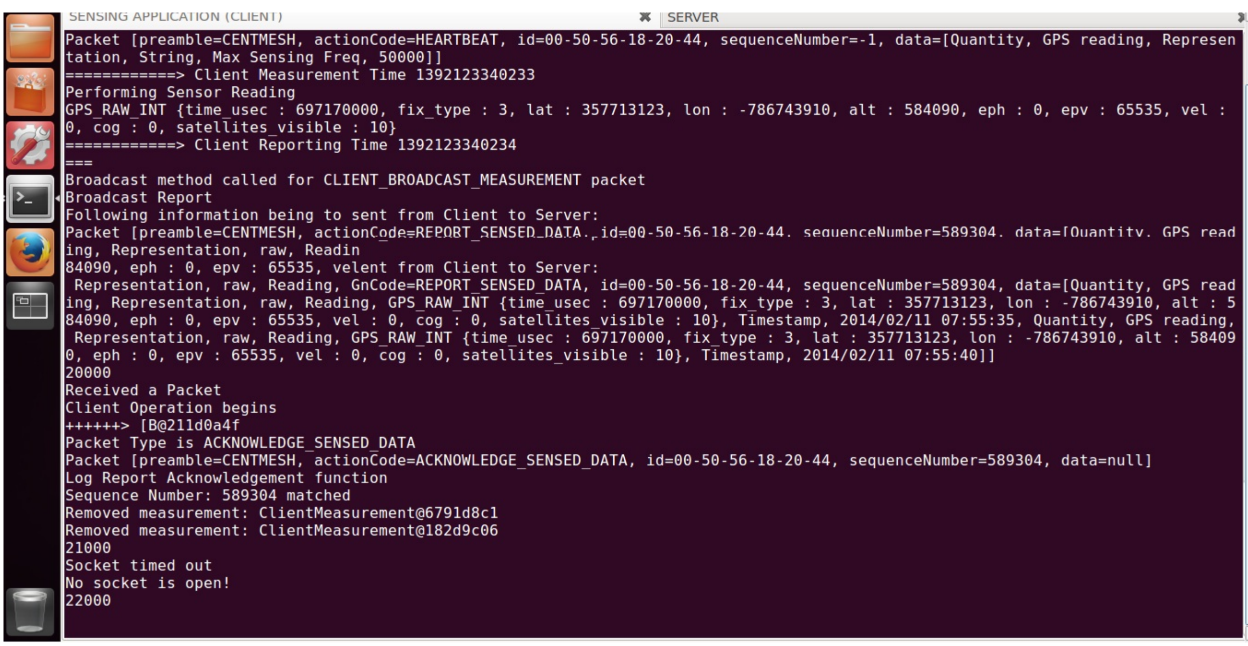
Following video demonstrates the working of this sample application.

Running the sensing platform:

Run the file sensor_cm_infra.sh in /home/droneusr/Desktop/:



It spawns a new terminal with two tabs - one tab has the client running and the other has server running:



From the tab in which the client process is running, it can be seen that this is accessing the gps reading in /tmp/gps_data.

NOTE:

- If you run this without the sample application, the same data will be read by the client again and again (as there is no process refreshing the file).

- If you run this without running the sample application at least once, each time the client tries to read the file, there will be an exception (as the file does not exist).

Following snapshot shows the tab in which server process runs. Whenever some data is received from the client, it is printed by the server: