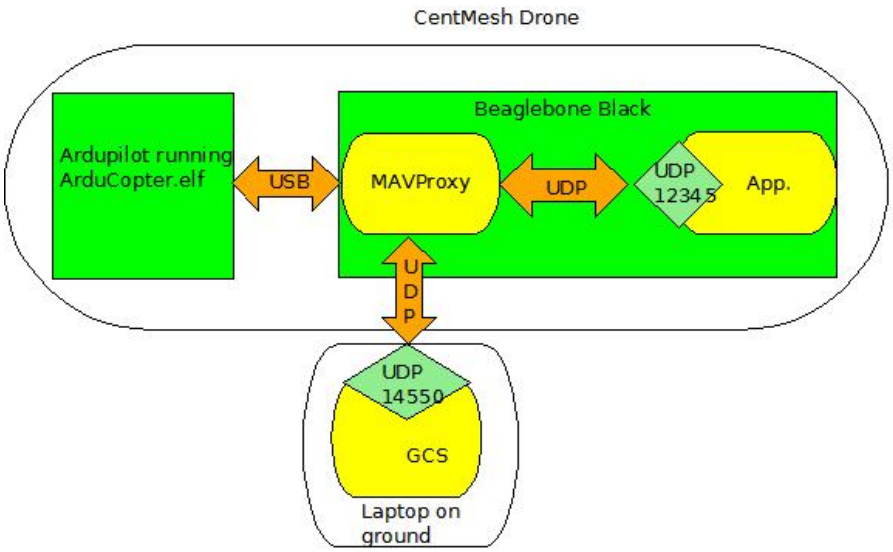
Tutorial for developing applications for the drone challenge

Objective: The objective of this tutorial is to help develop and verify sample applications using the programmable platform offered by CentMesh drones.

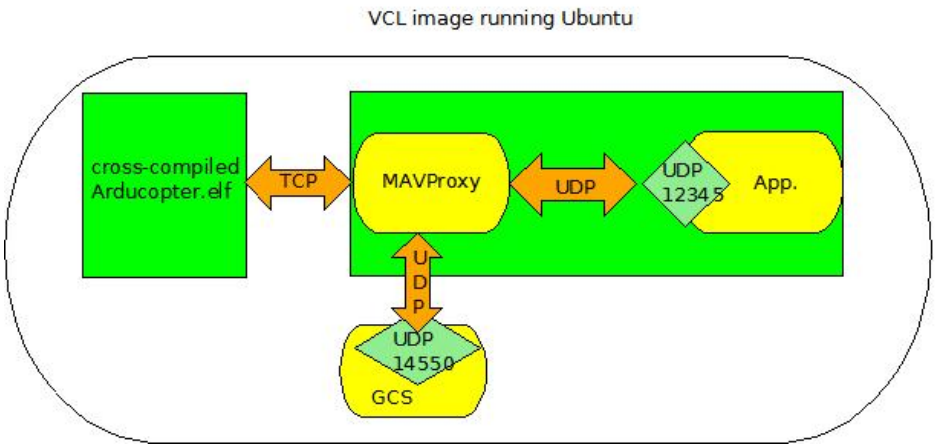
Introduction

As given in the following figure, the following components are involved in any application that runs on the CM drones.



The detailed description of each of these components is given in the documentation:
APM
Beaglebone black
GCS
MAVProxy

Using SITL (documentation here), the entire setup can be run in a single Virtual Machine or VCL image. The representation (simplified) is displayed by the following figure:



How to write programs using pymavlink

pymavlink

Python bindings for MAVLink are provided by pymavlink. Using the interfaces provided, both the cases are possible:

- MAVLink messages can be read from the UAV
- MAVLink Messages can be sent 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/sample_prog/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 the sample program resides and the following is to be included:

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

3) Import the following commonly used python libraries:

import re, sys, os, socket, select, time

4) 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))
```

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

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

6) 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)
```

7) To access individual fields, there are other 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}
```

8) To send commands to the UAV, there are variations of encode() method. For example, to set a waypoint the UAV has to traverse 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 an ID that signifies 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.

Command sent to the UAV for performing actions (i.e. setting mode, setting a way-point etc.) 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 makes use 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.

Using SITL with Sample applications

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

Sample application 1

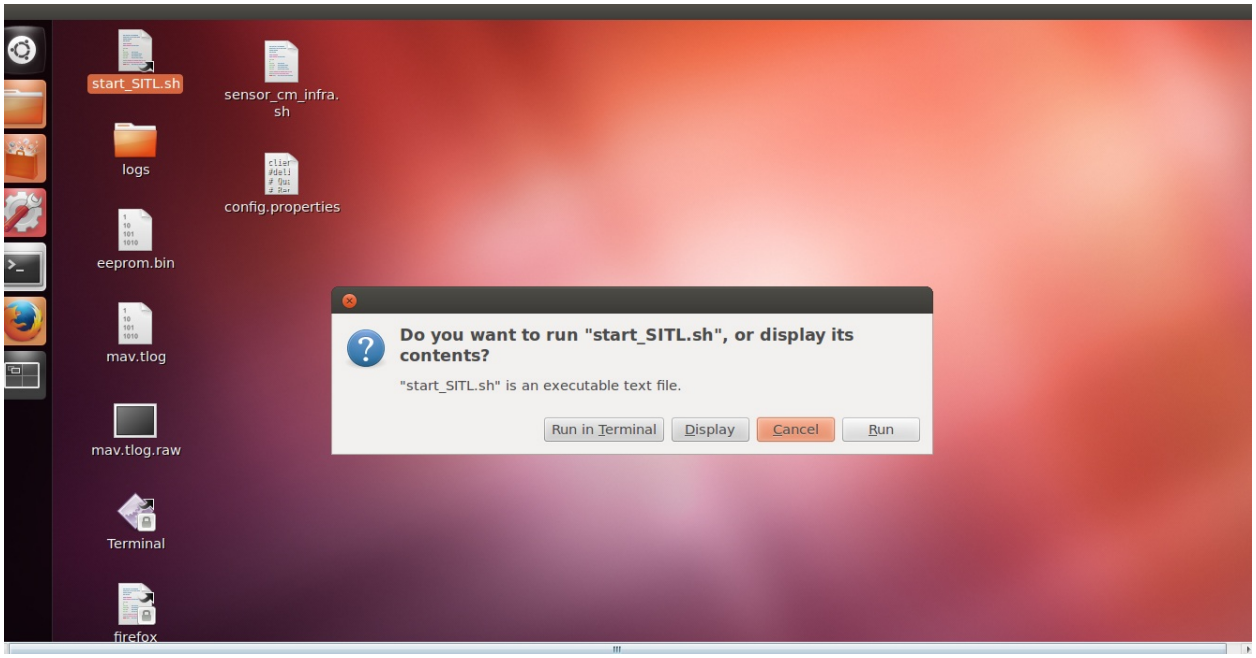
This sample application performs the following operations:

1) Sets the copter to AUTO mode
2) Listens on UDP port 12345 for messages from MAVProxy
2) Reads the GPS data that is generated by SITL and forwarded by MAVProxy. Saves this raw data to a file every 5 seconds.

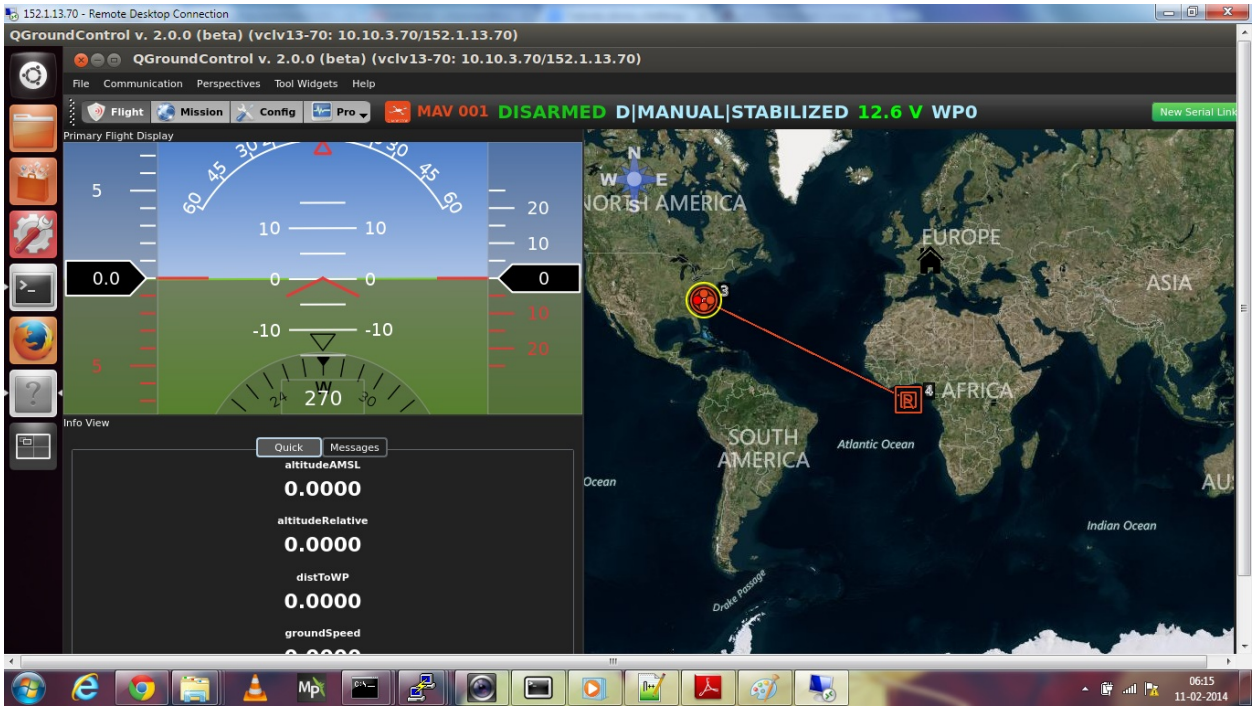
Code for the sample application is available in /home/droneusr/sample_prog/program_1

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: 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):

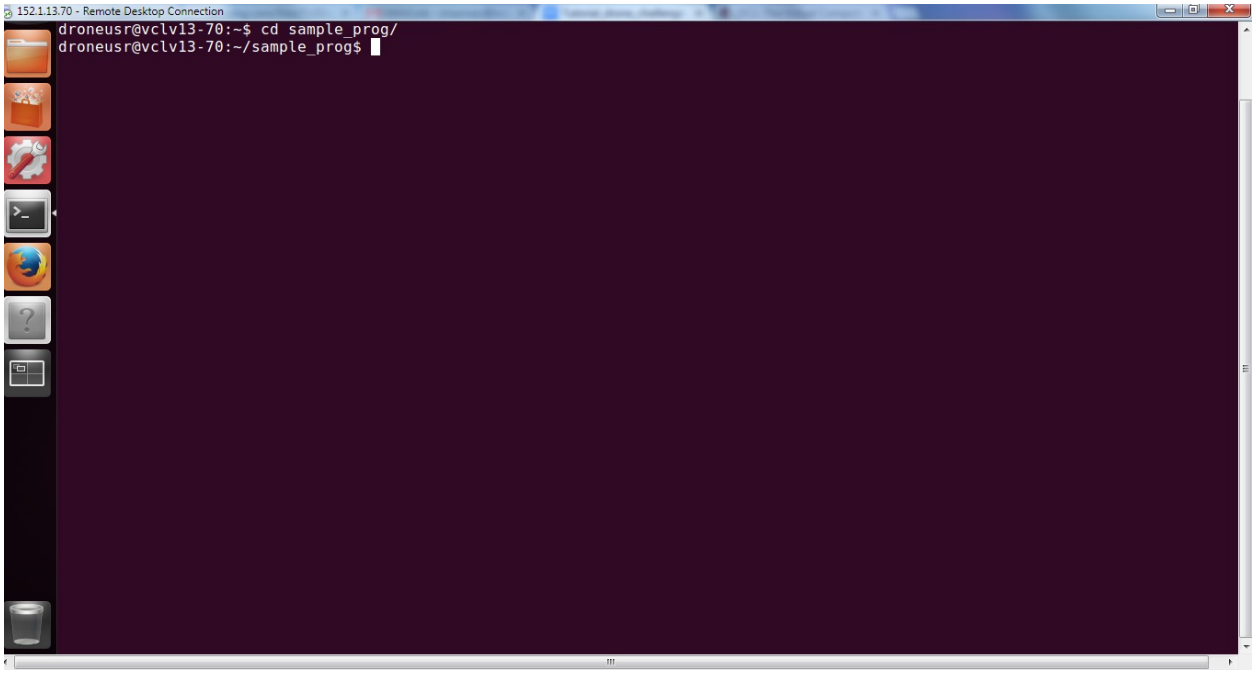


Zoom in using the side-bar on the right so that you have get a decent view of the copter:

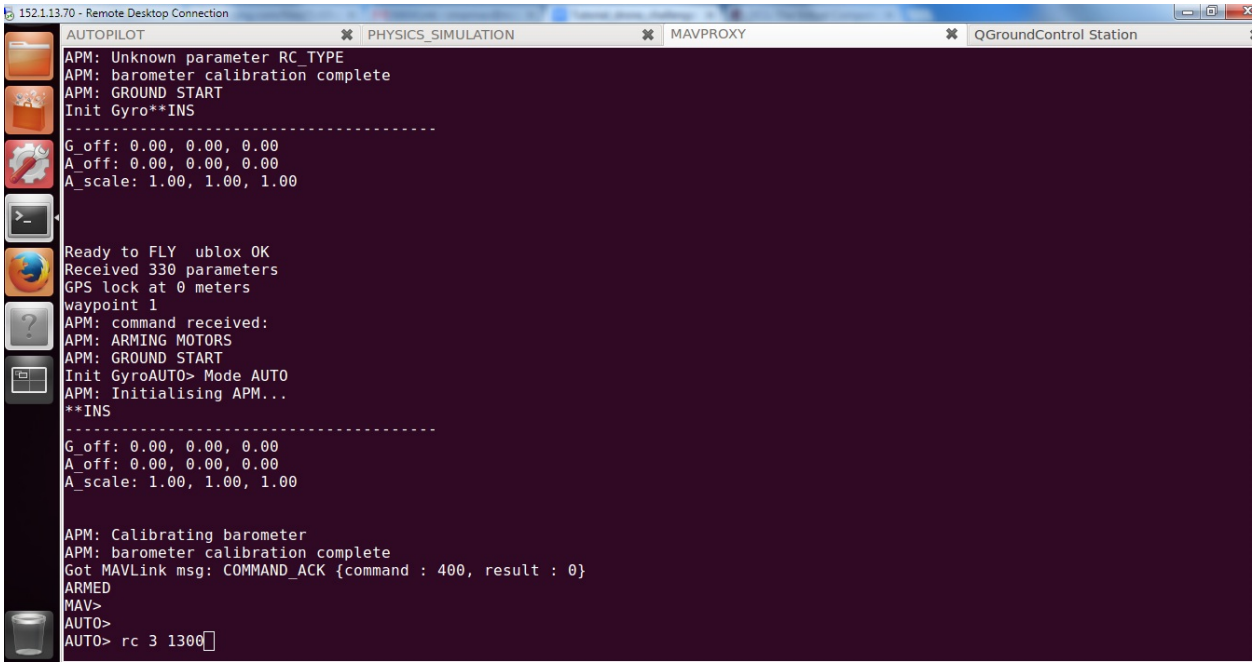
NOTE: This has to be performed manually the first-time you run SITL after reserving an image.For subsequent runs, you can click on ‘Last Pos’ and you will be taken to the last position that you were looking at

Step 4: Running sample app 1:

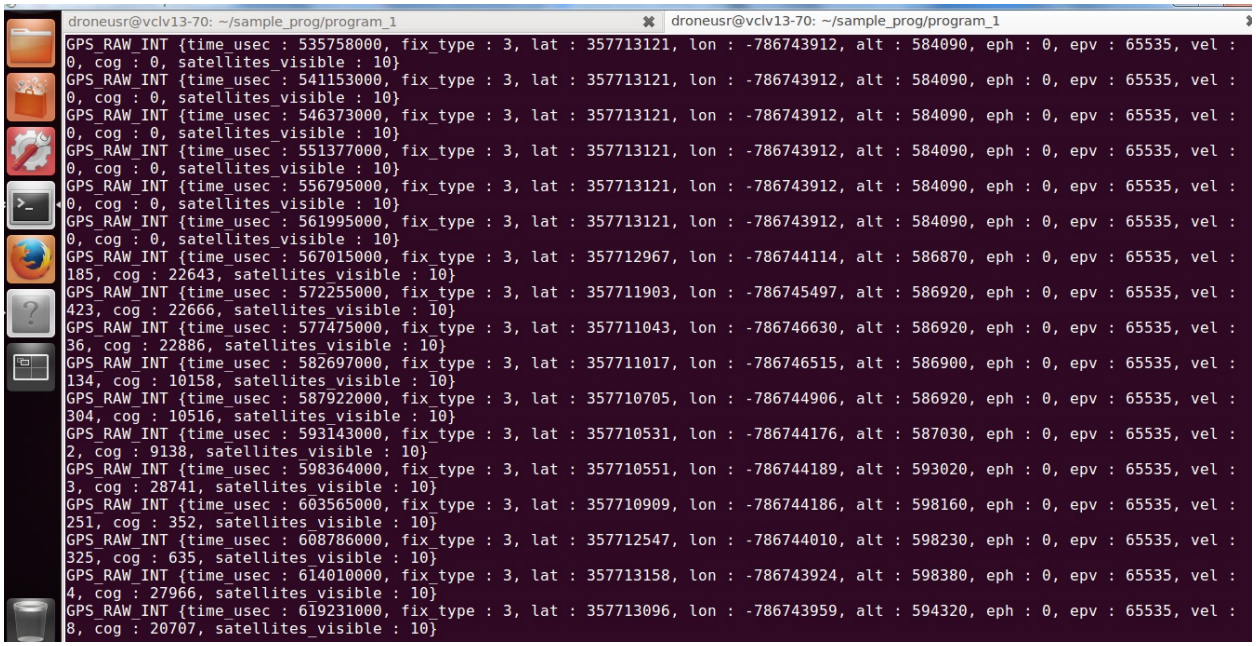
Open a new terminal and enter the ‘sample_prog’ directory:



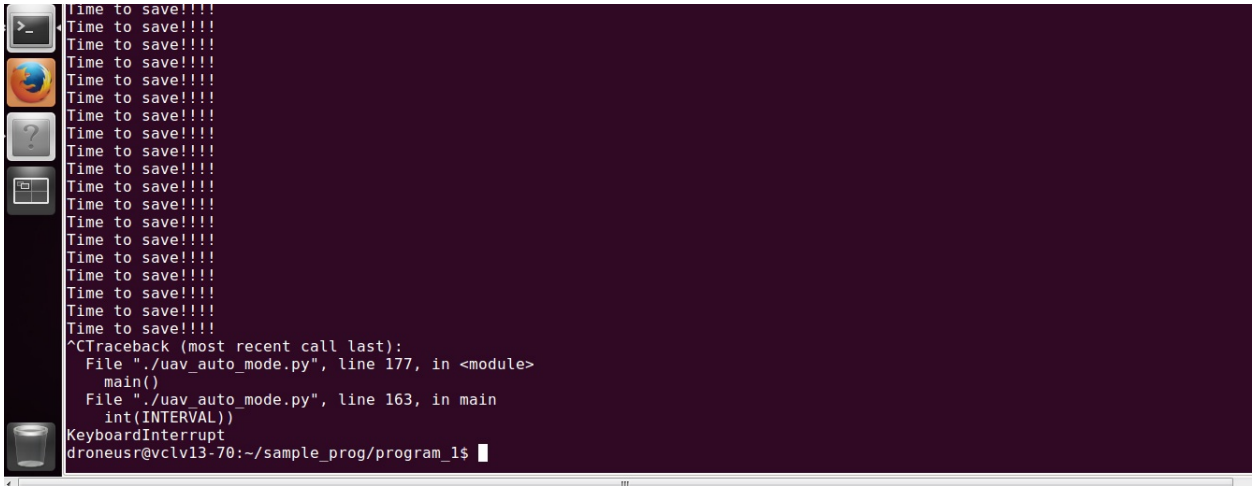
Enter program_1 directory and start the app using the following syntax:
./uav_auto_mode.py <MAVProxy port to connect to> <name of the file>
In our example, the MAVProxy port is always 12345. We specify a file named gps_data to which the readings are saved periodically (every 5 seconds).
Step 5: Provide throttle by entering the following command on MAVProxy terminal:
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):



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



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

Sample application 2

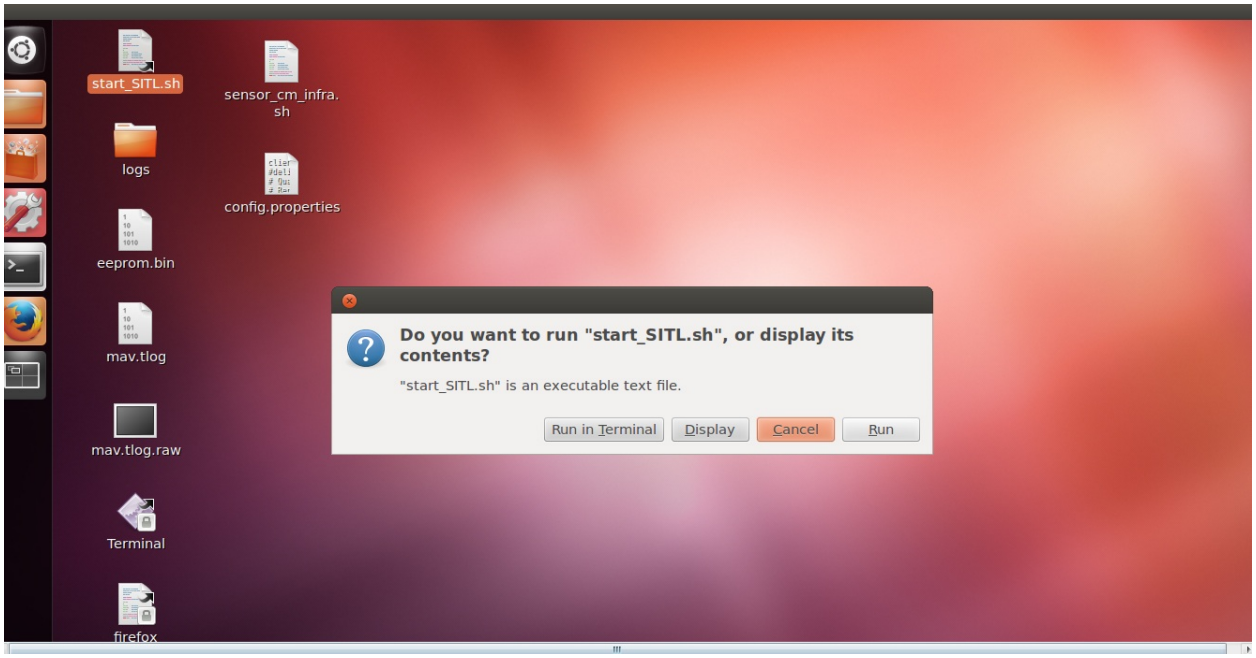
This sample application performs the following operations:

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 reading 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.

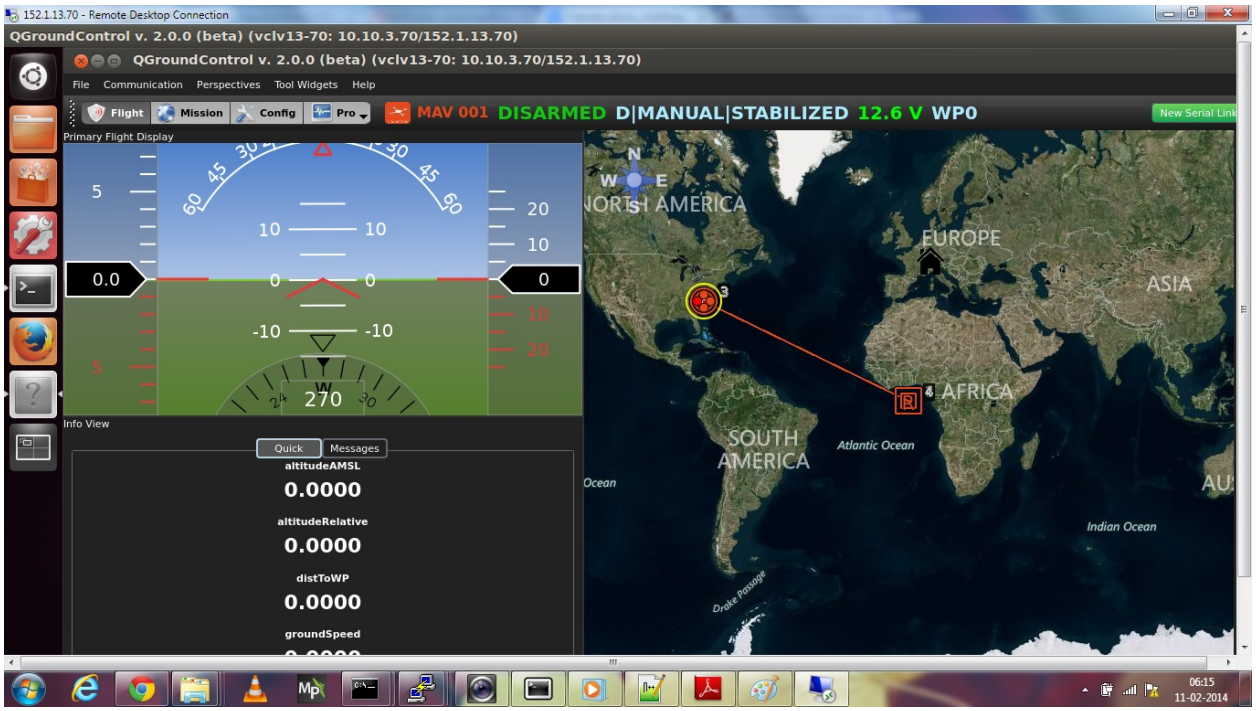
Code for the sample application is available in /home/droneusr/sample_prog/program_2

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:



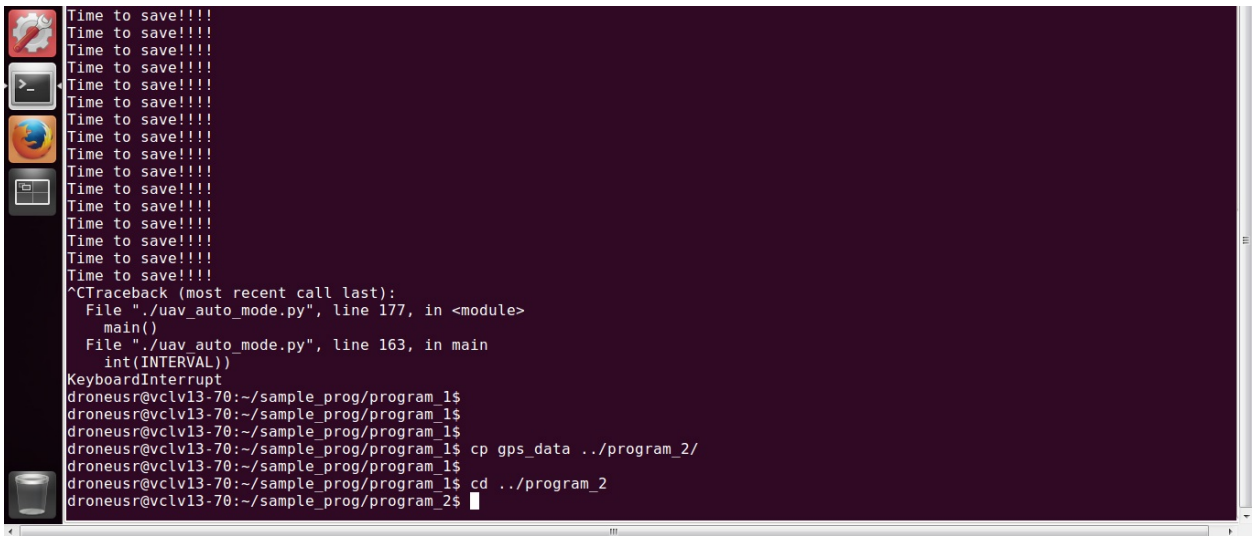
This should open 4 tabs one of which runs the qgroundcontrol (the Ground Control Station that is bundled with this image):



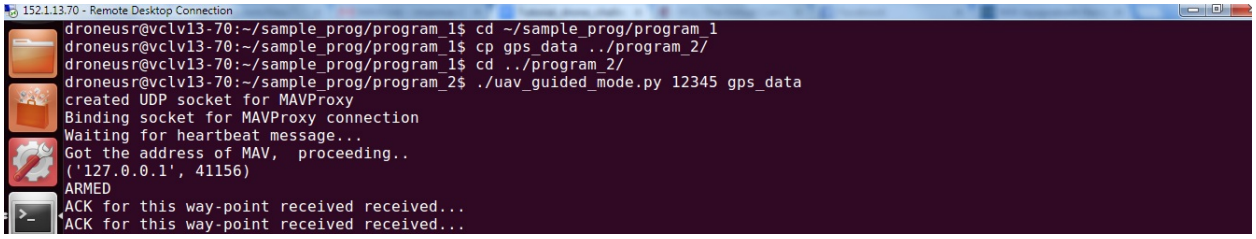
Zoom in using the side-bar on the right so that you have get a decent view of the copter:

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

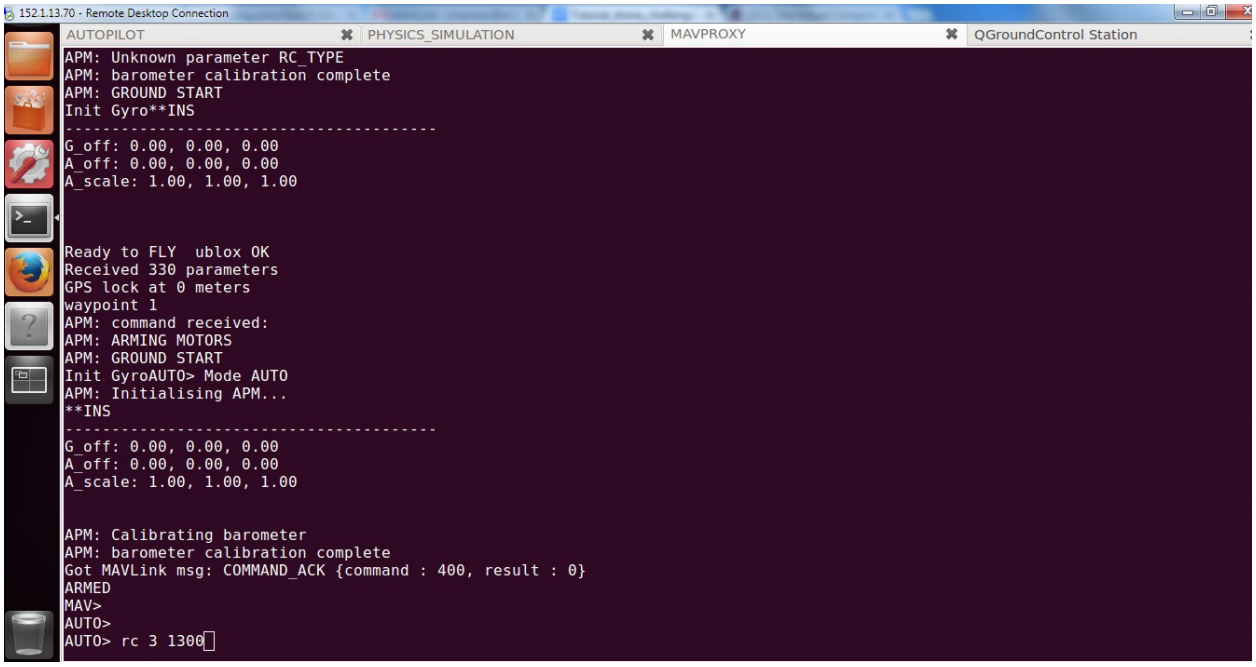
Step 4: Running sample application 2: We make use of the file that was generated by sample application 1. Copy the file to ‘program_2’ directory and enter ‘program_2’ directory:



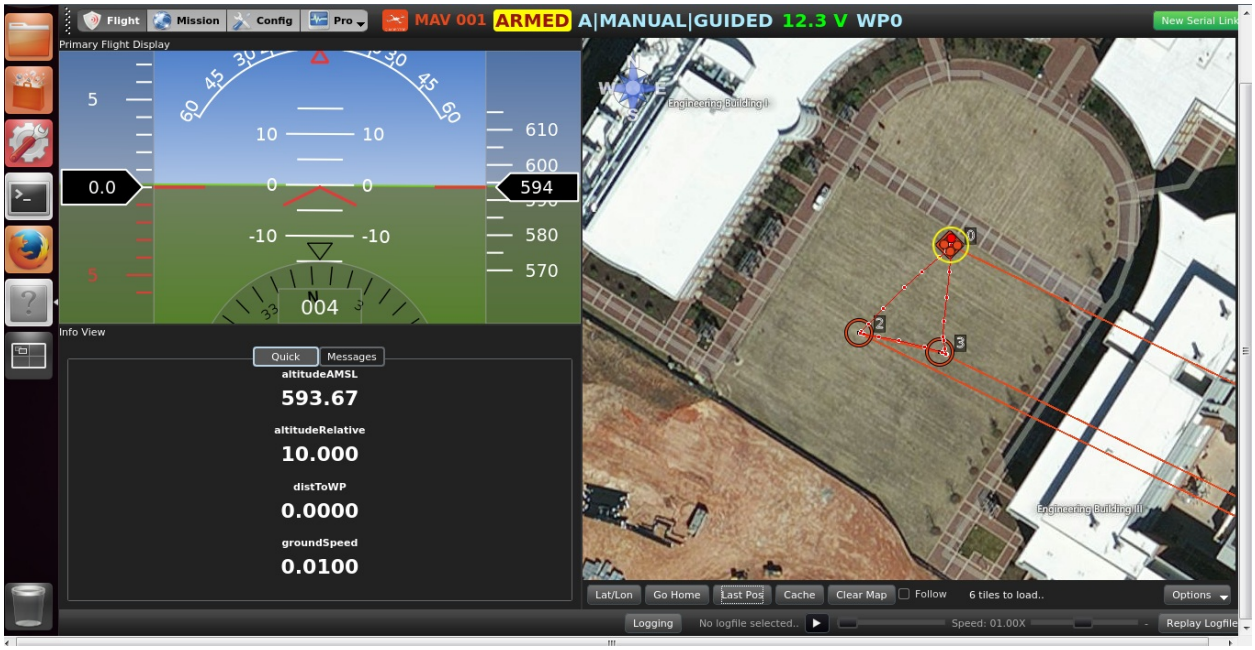
Step 6: Start the sample application:



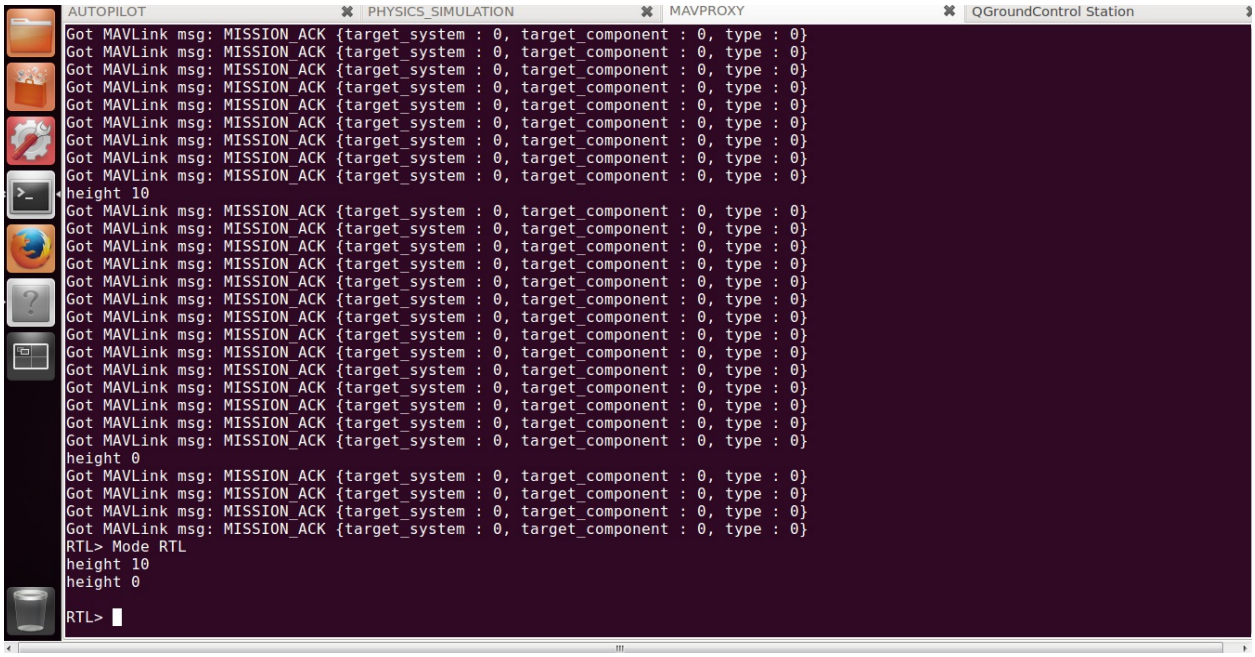
Step 7: Provide throttle by entering the following command on MAVProxy terminal:
AUTO>rc 3 1300



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.

Saving sensor data

Any Sensor data that is read by applications running on the UAV can be saved to periodically to a repository. The VCL image has a basic client and server combination that achieves this functionality.

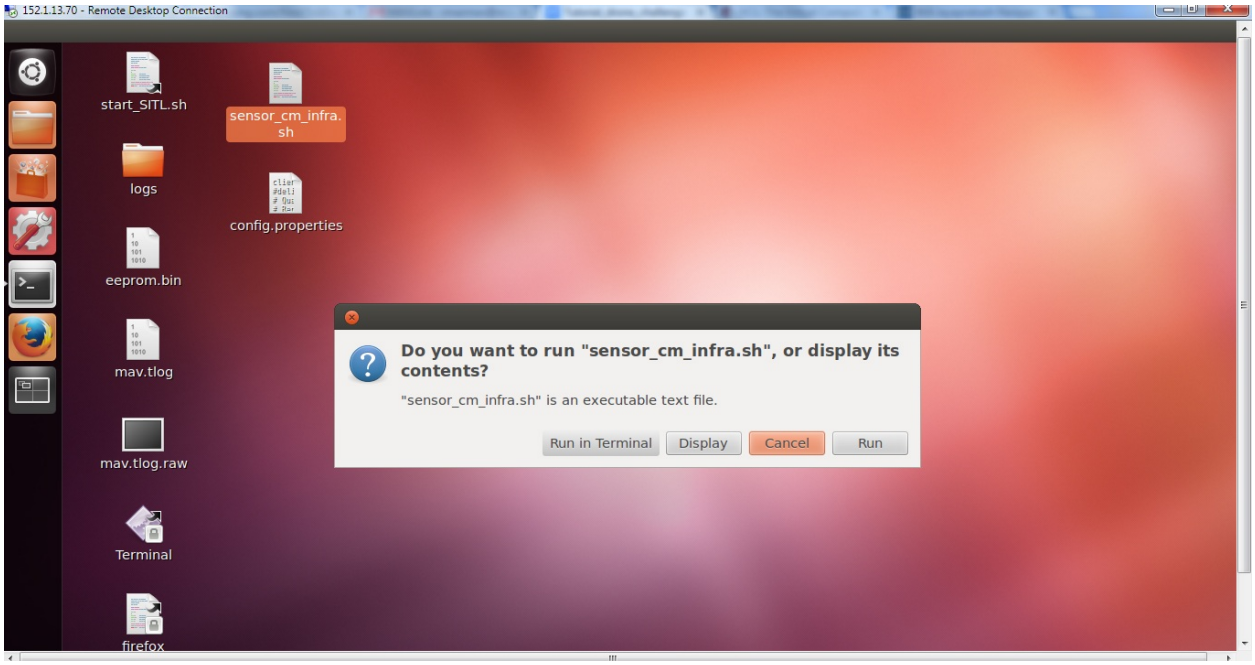
Sensing application (client): This reads sensor data periodically and reports the readings to a server. The type of sensor, the frequency of reading etc. are specified in a file ‘config.properties’.
Server: The server receives readings for various data from different clients and saves this to a repository.

Working of the sensing platform:

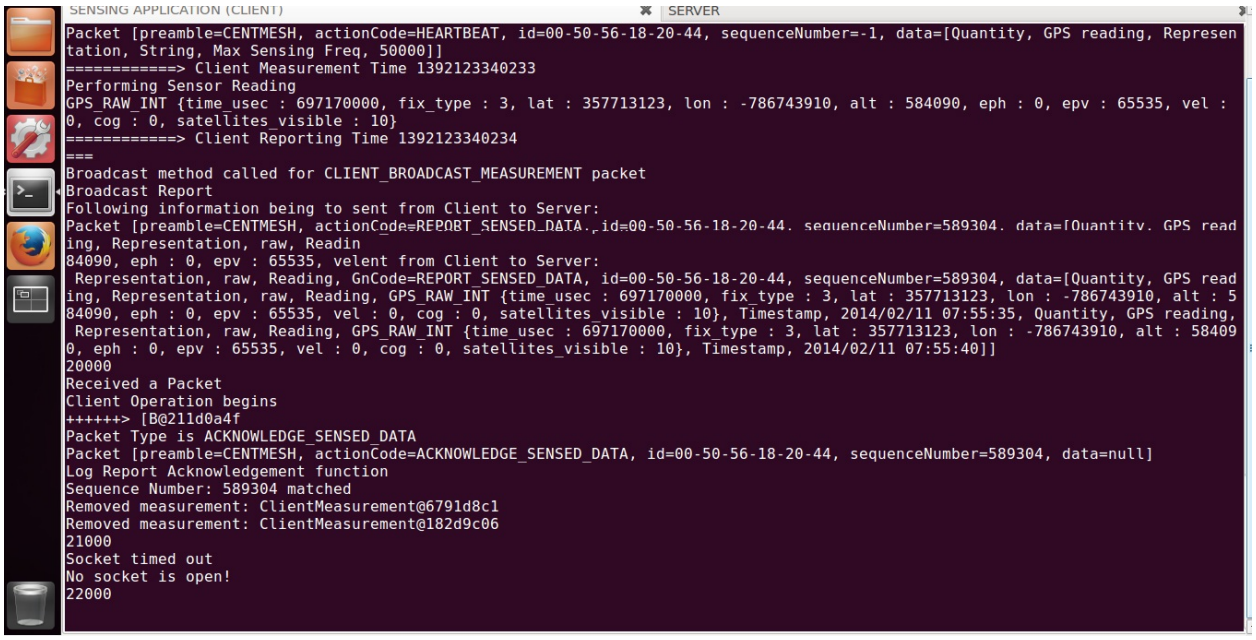
The VCL image has a client-server combination that works with sample application 1. We use GPS readings as sensor data. To recall, the sample application keeps saving the GPS readings to a file every 5 seconds. In addition to that, it also writes the current GPS reading to a file “/tmp/gps_data”. So each time the UAV receives a message with 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).

Running the sensing platform:

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



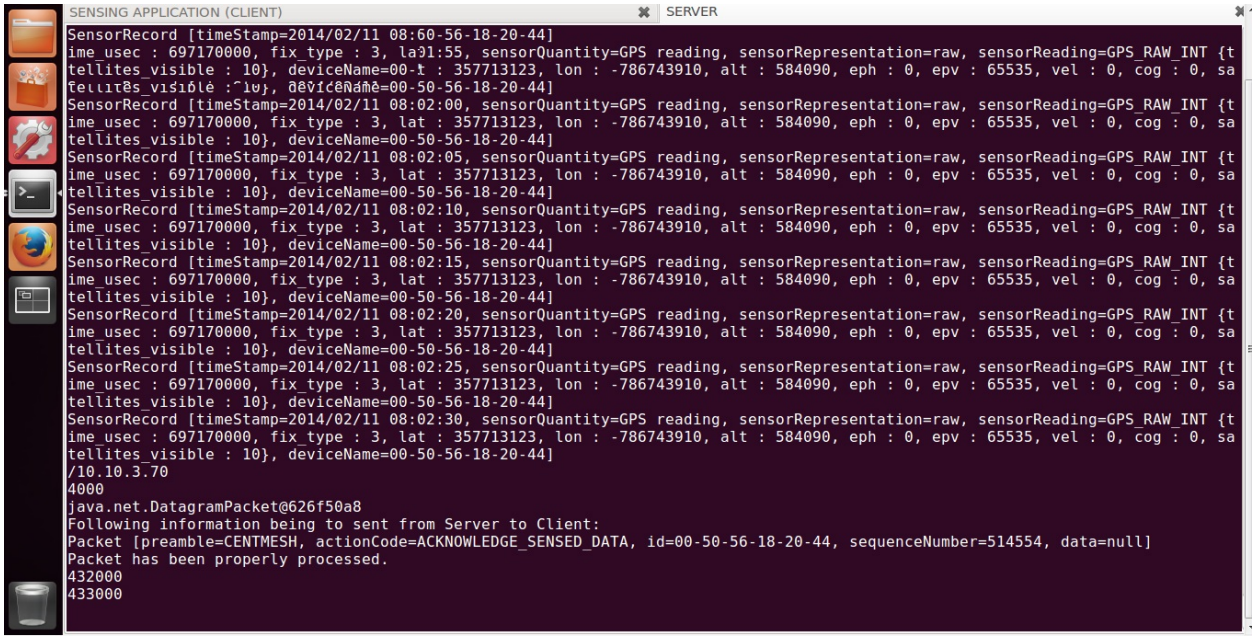
It spawns a new terminal with two tabs - one tab has the client running and the other has server running:



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:


SENSING APPLICATION (CLIENT)
SERVER
SensorRecord [timeStamp=2014/02/11 08:60-56-18-20-44]
ime_usec : 697170000, fix_type : 3, la31:55, sensorQuantity=GPS reading, sensorRepresentation=raw, sensorReading=GPS_RAW_INT {t
tellites_visible : 10}, deviceName=00-1 : 357713123, lon : -786743910, alt : 584090, eph : 0, epv : 65535, vel : 0, cog : 0, sa
SensorRecord [timeStamp=2014/02/11 08:02:00, sensorQuantity=GPS reading, sensorRepresentation=raw, sensorReading=GPS_RAW_INT {t
ime_usec : 697170000, fix_type : 3, lat : 357713123, lon : -786743910, alt : 584090, eph : 0, epv : 65535, vel : 0, cog : 0, sa
tellites_visible : 10}, deviceName=00-50-56-18-20-44]
SensorRecord [timeStamp=2014/02/11 08:02:05, sensorQuantity=GPS reading, sensorRepresentation=raw, sensorReading=GPS_RAW_INT {t
ime_usec : 697170000, fix_type : 3, lat : 357713123, lon : -786743910, alt : 584090, eph : 0, epv : 65535, vel : 0, cog : 0, sa
tellites_visible : 10}, deviceName=00-50-56-18-20-44]
SensorRecord [timeStamp=2014/02/11 08:02:10, sensorQuantity=GPS reading, sensorRepresentation=raw, sensorReading=GPS_RAW_INT {t
ime_usec : 697170000, fix_type : 3, lat : 357713123, lon : -786743910, alt : 584090, eph : 0, epv : 65535, vel : 0, cog : 0, sa
tellites_visible : 10}, deviceName=00-50-56-18-20-44]
SensorRecord [timeStamp=2014/02/11 08:02:15, sensorQuantity=GPS reading, sensorRepresentation=raw, sensorReading=GPS_RAW_INT {t
ime_usec : 697170000, fix_type : 3, lat : 357713123, lon : -786743910, alt : 584090, eph : 0, epv : 65535, vel : 0, cog : 0, sa
tellites_visible : 10}, deviceName=00-50-56-18-20-44]
SensorRecord [timeStamp=2014/02/11 08:02:20, sensorQuantity=GPS reading, sensorRepresentation=raw, sensorReading=GPS_RAW_INT {t
ime_usec : 697170000, fix_type : 3, lat : 357713123, lon : -786743910, alt : 584090, eph : 0, epv : 65535, vel : 0, cog : 0, sa
tellites_visible : 10}, deviceName=00-50-56-18-20-44]
SensorRecord [timeStamp=2014/02/11 08:02:25, sensorQuantity=GPS reading, sensorRepresentation=raw, sensorReading=GPS_RAW_INT {t
ime_usec : 697170000, fix_type : 3, lat : 357713123, lon : -786743910, alt : 584090, eph : 0, epv : 65535, vel : 0, cog : 0, sa
tellites_visible : 10}, deviceName=00-50-56-18-20-44]
SensorRecord [timeStamp=2014/02/11 08:02:30, sensorQuantity=GPS reading, sensorRepresentation=raw, sensorReading=GPS_RAW_INT {t
ime_usec : 697170000, fix_type : 3, lat : 357713123, lon : -786743910, alt : 584090, eph : 0, epv : 65535, vel : 0, cog : 0, sa
tellites_visible : 10}, deviceName=00-50-56-18-20-44]
/10.10.3.70
4000
java.net.DatagramPacket@626f50a8
Following information being to sent from Server to Client:
Packet [preamble=CENTMESH, actionCode=ACKNOWLEDGE_SENSED_DATA, id=00-50-56-18-20-44, sequenceNumber=514554, data=null]
Packet has been properly processed.
432000
433000