API development for cooperative airborne-based sense and avoid in UAS

by

Leonidas Antoniou

A DIPLOMA THESIS

Submitted in fulfillment of the requirements for the degree

'Diploma in Electrical and Computer Engineering'

Department of Electrical and Computer Engineering

TECHNICAL UNIVERSITY OF CRETE

October 2016

Supervisors

Professor Apostolos Dollas

Assoc. Prof. Yannis Papaefstathiou

Asst. Prof. Panagiotis Partsinevelos

Abstract

The opening of the unmanned aircraft system (UAS) market to civil applications has expanded the research interest in the unmanned aviation field. Under a common airspace where multiple and various types of UAS should be airworthy, the use of advanced control for full autonomy is mandatory· conflicts between UAS are a safety hazard to their surrounding environment, including civilians.

A lot of research has been conducted in conflict (collision) detection and avoidance in mobile robots but there is little application in the UAS sector. This is because of the complexity of the surrounding environment, the non-segregated airspace, the variety in UAS, both from a software and hardware prospective, and the wide use of UAS in civil applications.

The purpose of this project is to extend the functionality of a famous open-source SDK for open-platform UAS, Dronekit, in the advanced control spectrum. The infrastructure for a decentralized solution for cooperative conflict detection has been developed, containing support for inter-UAS communication, prioritization and mission save/restore in a resource-conscious and developer-friendly API.

The software developed undergoes both laboratory and field testing for evaluation purposes. The results are thoroughly discussed and the advantages of the system, along with its limitations, are presented.

Keywords: UAS, ABSAA, cooperative, Dronekit

Contents

Glossary

Note: The terminology that satisfies the scope of this thesis, although not used for the first time by the authors, is mostly obtained by the** (ICAO, 2011) ***and** (Angelov, 2012) ***writings. Any terms that do not have a reference are written by the author of this thesis.

Autopilot. *Commercial off-the-shelf (COTS) flight control system which provides the core functionality that enables autonomous flight.* (Angelov, 2012)

Conflict. *The term, although not clearly defined in the aviation domain, is used when the flight trajectory of an aircraft, conventional or unmanned, is blocked by other aircrafts or physical barriers.*

Cooperative collision avoidance. *The procedure during which a conflict, between two or more subjects, takes place and all the subjects have the ability to exchange their state parameters as input for their collision avoidance functionality.*

Full autonomous UAS. *The capacity of the UA to achieve its entire mission without considering any intervention of the human flight crew. Obviously, with this architecture the SAA must be performed exclusively by on-board means. Nevertheless, these kinds of operation are still not contemplated by any regulatory body in the world.* (Angelov, 2012)

Open-source software. *Software with source code that anyone can inspect, modify, and enhance.* (RedHat, 2012)

Remote-pilot aircraft system(RPAS). *A set of configurable elements consisting of a remotely-piloted aircraft, its associated remote pilot station(s), the required command and control links and any other system elements as may be required, at any point during flight operation.* (ICAO, 2011)

See and Avoid. *The capability to physically see conflicting traffic or other hazards and take the appropriate action to comply with the applicable rules of flight.* (ICAO, 2011)

Segregated airspace. *Airspace of specified dimensions allocated for exclusive use to a specific user(s).* (ICAO, 2011)

Sense and Avoid. *The capability to sense, with electronic equipment (sensors), conflicting traffic or other hazards and take the appropriate action to comply with the applicable rules of flight.* (ICAO, 2011)

Unmanned aerial vehicle(UAV). *A pilotless aircraft, a flying machine without an on-board human pilot or passengers. Control functions for unmanned aircraft may be either on-board or off-board (remote control). This term differentiates from UAS. See Unmanned Aircraft System(UAS) for more information.* (Angelov, 2012)

Unmanned aircraft system(UAS). *The official term of an airworthy system with the absence of an on-board pilot, consisting also of a ground control station, communication links and launch and retrieval systems.* (Angelov, 2012)

1. Introduction

1.1 History

Before ICAO's Circular No.328 which was published in 2011, there were no organizations of international impact to publish or even define the terminology and operations of UAS. The main reason was the military nature of such technologies. The first steps towards unmanned aircrafts, as they are known today, were done by the U.S Army during World War I. Even though the results were disappointing, the technology continued to evolve so that, in 1995, the first UAS – military drone was in the production line (Whittle, 2013): The MQ-1 Predator is still used for dangerous military operations and the project currently costs more than two billion U.S. dollars (United States Special Operations, 2011).

Along with the first successful steps in the 1990s, of the U.S. Department of Defense (DoD) for unmanned aviation in the military, the National Aeronautics and Space Administration (NASA) started the research for UAS in civil applications. The first outcome was the Proteus model, after the Greek god Proteus – who, according to the mythology was a shapeshifter. Proteus was designed to accomplish a wide variety of missions, some of which include commercial imaging, reconnaissance/surveillance and high-altitude, long-duration telecommunications relay platform etc. (Gibbs, 2009). (Nonami, et al., 2010)

Figure 1.1 MQ-1 Predator General Atomics (U.S. Air Force, n.d.)

Figure 1.2 Proteus NASA (Gibbs, 2009)

The development of the first reliable unmanned aircrafts, both in military and civil operations, along with the invention of vertical take-off and landing (VTOL) aircrafts, like the helicopter, naturally brought the scientific community to the development of VTOL UAS. Japan and the U.S. have made great contributions to the development of such multi-rotor aircrafts, like rotary-wing and quad tilt rotor (QTR) UAS (Nonami, et al., 2010, p. 19).

Transitioning to the twenty-first century, someone could state that Unmanned Aviation is characterized by multi-rotor technology. UAS exploiting this technology combine ease-of-use with countless applications both in civil (telecommunications, environment, agriculture, geospatial data, commercial, public safety) and military (agility, heighten element of surprise, maneuverability, precision firing) applications (Defense Science Board (DSB) Task Force, 2007).

Rapid development of UAS technology has created an exponential increase in flight hours of UAS through the years, starting from 1996. With the increase of the flight hours, the need to integrate UAS to the U.S. National Airspace System (NAS) and worldwide, was getting stronger. Due to this, UA should gain access into the *non-segregated* airspace *"in support of their operational, training and test and evaluation missions"* (UAS Task Force, Airspace Integration Integrated Product Team, 2011). (Spriesterbach, et al., 2013)

The need for integration to the non-segregated airspace, as mentioned by numerous authors through the years: (UAS Task Force, Airspace Integration Integrated Product Team, 2004), (Lacher, et al., 2010), (UAS Task Force, Airspace Integration Integrated Product Team, 2011), (Anon., 2013) to mention a few, combined with the development of multi-rotor unmanned aircrafts, has created an even stronger potential of unmanned aviation to widespread use.

Figure 1.3 Scout Datron (Crane, 2012)

1.2 Unmanned aircraft systems (UAS)

According to the International Civil Aviation Organization (ICAO), UAS are pilotless aircrafts that operate without a pilot-in-command on-board (International Civil Aviation Organization, 2011, p. 17). They are called systems because they consist of three main segments: Air, Ground and Communications segment. On each segment, there are different subsystems responsible for certain functions (Angelov, 2012):

In the **air** segment, there is the unmanned aircraft: the airframe, avionics and propulsion system. Each component is cooperating for the airworthiness of the UA. Additionally, there are the interfaces for the communication between the other two segments and extra hardware for specific operations by the user. The last is also called payload and usually consists of sensors and imaging components. Included in the payload of the air segment, a computer can provide extra processing power as an onboard companion computer. The onboard computer may provide *advanced control* functionality. This can be a high-level autonomous flight with sense and avoid (S&A) capability and/or a special mission requiring onboard advanced image acquisition and processing. Usually, a UAS is characterized by its gross weight.

The **ground** segment, or ground control station (GCS), is involved in the higher-level operations of the air segment. All the required equipment for flight planning and mission monitoring belongs to the ground segment (Angelov, 2012).

The **communications** segment is responsible for transferring mission-related (Payload) data, messages (External Communications) and commands (Command &Control) between the air and the ground segment.

1.3 Autonomous operations of UAS

A UAS is autonomous when there must be no need for human intervention during a mission. This can be the successful execution of a flight plan where the UAS must fly over certain waypoints. The (fully) autonomous UAS must also be responsible of handling unexpected events and potential hazards during its mission. (Gundlach, 2012)

An unexpected event could be an object or another UA blocking the UAS' trajectory. Or, since the missions of a UAS are usually characterized by the three D's: Dull, Dirty, Dangerous, a potential hazard could be a really toxic or hot physical environment (e.g. telemetry mission over an active volcano). For this reason, a UAS with higher level of autonomy, apart from the necessary avionics that ensure the airworthiness of the UA (flight control computer, inertial navigation system), must communicate with a set of real-time sensors (barometer, GPS, temperature) that provides feedback of the surrounding environment. The UAS then should use the data to take actions for the prevention of a potential system failure.

A subject of autonomous flight is collision avoidance. Since there is no human pilot or crew onboard, the avoidance of conflicts must be relied either on a remote human pilot (see and avoid) or, in case of an autonomous UAS, a detection system (sense and avoid – S&A). An S&A system uses real-time sensors, processes their output, decides a new, conflict-free flight path and gives the command to the autopilot to execute it. This procedure happens continuously at a specified frequency, according to the airspace management rules (such as groundspeed, altitude, safety zone limitations).

The set of sensors can either be onboard the UAS, thus having airborne-based S&A, or in a base station (ground-based). Ground-based sense and avoid (GBSAA) is usually a procedure where data from multiple UAS is acquired by a ground sensor (e.g. a radar). Then the base station processes the received data and transfers the new flight plan back to the UASs. (Spriesterbach, et al., 2013)

In airborne-based sense and avoid (ABSAA), external information acquired by sensors or the communication segment is processed onboard every UAS and, after it is decided that a collision is about to happen, the system commands the autopilot to execute then new flight plan, as shown in Figure 1.4.

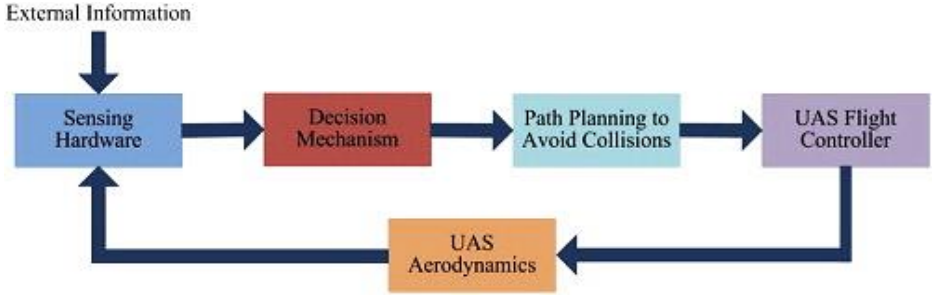


Figure 1.4 The structure of S&A functionality in UASs (Yu & Zhang, 2015)

Either airborne- or ground-based, S&A functionality can also be divided into two other families: cooperative and non-cooperative. During non-cooperative S&A, the UAS can deduce the existence of other UAS or physical barrier through sensors (e.g. sonar, lidar) and follow the rest of the S&A datapath.

In S&A's cooperative counterpart, UAS have the ability to exchange state parameters with others and make decisions based on the data exchanged. Information exchange can happen either with a direct communications link or via a base station.

Each mix of methods, see and avoid or sense and avoid, cooperative or non-cooperative, airborne-based or ground-based have their own applications, advantages and disadvantages in autonomous flight. This thesis will develop and test a cooperative, sense and avoid, airborne-based method and analyze its advantages and disadvantages.

1.4 Problem addressed

Widespread use of multi-rotor UAS into the non-segregated airspace induces high risk of *conflict* between the like and manned aircraft as well. The first complete set of rules and regulations for the flight of small UAS (weight less than 25kg) in the U.S. NAS was published by the Federal Aviation Administration (FAA) on 29th August 2016, rule 107 (FAA, 2016).

Among the statements of rule 107, there is a particular limitation which must be addressed in the future, if we want *full autonomy* in a UAS: *"Visual line-of-sight (VLOS) only; the unmanned aircraft must remain within VLOS of the remote pilot in command and the person manipulating the flight controls of the small UAS. Alternatively, the unmanned aircraft must remain within VLOS of the visual observer"* (FAA, 2016). This means that, even if a small UAS is programmed to perform an operation autonomously, the responsible person for programming the UAS operation is obliged to physically watch over the UAS while performing its mission.

The reason of existence of the aforementioned limitation is the lack of technology that, during a mission, can reduce the safety risk to acceptable levels *"in the presence of anomalies"* (Atkins, n.d.). One great "anomaly" is the case of a conflict.

The term conflict, although not clearly defined in the aviation domain, is used when the flight trajectory of an aircraft, conventional or unmanned, is blocked by other aircrafts or physical barriers. If technology could provide conflict-free autonomous flight in the international airspace, with an acceptable -or no at all- risk, then the aforementioned limitation of FAA's part 107 should be withdrawn.

So far, conflict (or collision) avoidance between conventional aircrafts is being held by the traffic collision avoidance system (TCAS), often combined with an S&A satellite-based system, the Automatic dependent surveillance – broadcast (ADS-B). The installation of the related hardware to the aircraft is being carried out by professionals and has a four to five figure price range. What is more, a global ground station infrastructure that works as a radar is in place, which naturally has great build and maintenance costs.

The option to adopt the ADS-B system by UAS is studied (B., et al., 2013), (Y. & S., 2015), (Lin & Lai, 2015) and the industry (uAvionix, 2016), (Trimble, 2015) tries to produce small-

scale and cheap ADS-B transponders. FAA's "NextGen" program makes it compulsory for any airborne system in the NAS to have ADS-B transponders by 2020, as well (FAA, 2016).

The disadvantages of integrating the ADS-B though are important: UAS need extra hardware and interfaces applied globally, if there is a need for ground controllers, then these base stations will have high maintenance cost and can have fatal downtime due to extreme scaling of UAS populations.

Another approach in airborne-based S&A systems in cooperative collision avoidance is exploiting hardware that is already in the payload of a UAS. Each UAS with minimum hardware requirements can be able to establish a certain communication link with other UAS and perform the necessary actions. The only prerequisite is up-to-date software. Later through the chapters, the advantages and disadvantages of this approach will be explained in detail.

The purpose of this thesis is to develop an API that aids in cooperative, airborne-based sense and avoid (ABSAA), conflict-free, autonomous flights between small UAS of different architectures (heterogeneous) in the non-segregated airspace. For this reason, an extension of an open-source UAS flight controller is developed, that increases its capabilities in cooperative ABSAA, and tested. The newly developed libraries introduce functionality on top of the already developed open-source flight controller. The key milestones of the development go as follow:

- **Creation of interfaces for network communication between different UAS.**
  This step is the most important since different UAS were not able to communicate with each other before. This is the starting point when implementing a system for cooperative ABSAA. Since the writing of this thesis, having as minimum requirement a Wi-Fi interface, network communication is achieved by the interchange of UDP broadcast messages. The project can be easily extended, though, for experimenting with more protocols, potentially safer and lighter.
- **Built-in prioritization algorithm in the case of many conflicting UAS concurrently.**
  It is very important to clarify that a conflict cannot only happen between two subjects. In the future civil airspace, it is very probable to have a conflict between numerous UAS at the same time. In case of a future collision avoidance strategy, a prioritization algorithm should take place at first so that each drone will take action based on a priority number. The prioritization algorithm takes a wide range of variables into account, from mission importance to battery level, so it can be applied fairly and globally to every stakeholder.
- **Saving and restoration of the state of the UAS in case of an emergency or mission takeover.**
  A new concept is introduced where a UAS, operating an autonomous mission, can postpone it in case of a conflict or emergency, and continue later on. A set of necessary variables is saved and then loaded when emergency state is over.
- **Make it portable and flexible.**
  Throughout the project, extra effort has been placed to make the source code extendable. It is clearly stated where in the source code new algorithms or protocols should be deployed, in order for the programmer to test his/her own functions. Also, Python language demands minimal setup and is already installed in most Linux distributions.

- **Make it readable: Detailed documentation both for the user and the programmer.** The main purpose of this project is to be used by the community. For this reason, the project is available to the public under the GNU General Public License, with the name "DK+" (DroneKit-plus) (Antoniou, 2016). Availability, though, is not enough when developing open-source projects. Strong documentation has to be written in order to support it. For this reason, following industry standards (Georg Brandl, 2016), (SourceForge, 2016), a comprehensive documentation can be found in the project's website and in Appendix F: User's and programmer's manual.

1.5 Structure

Chapter One was devoted to introductory material helping understand the history behind the modern UAS, its main architecture and the importance of autonomy in such systems. The need for integration in civil airspace was also stated along with the risks involved. In Chapter Two, the most relevant pieces of work are stated, in order for the reader to understand where are we in research and what products have already been, or are soon to be, available for advanced control in UAS.

Chapters Three to Five explain the implementation of the API and the system that is implemented in. In more detail, Chapter Three explains the parameters and characteristics of the system being modelled so that in Chapter Four and Five, a more precise, actual implementation of the system can take place, with detailed information about the system architecture in the hardware and software prospects.

After the system modeling and implementation described in previous chapters, Chapter Six explains how is the system validated and tested, under both software-in-the-loop and real-life scenarios. Ultimately, the results are reported and commented upon in Chapter Seven.

2. Relevant work

This chapter provides information on the research of unmanned aircraft moving gradually from the more general UA topic to the more specialized cooperative, airborne-based sense and avoid systems. An overview of the current legislations and available products and service providers is also described.

2.1 Initial UAS design

Thousands of articles and papers in the field of UAS have been published, especially by American and Chinese institutions. The first papers were published when the Predator drone came to production, in 1995 (results based on Scopus). During that time, the fundamentals of the modern UAS were being implemented, concerning subsystems like the inertial measurement unit (Humphrey, 1997) and the flight control system (Ozimina, 1995). Some effort in the development of the UAS communications segment in military applications is also observed. The first results on full duplex communication links in UAVs were reported by (Pinkney, et al., 1997). From 1998 until recently, there are publications concerning autonomy in landing (Lin, et al., 1998), attitude optimization (Oshman, 1999), (Kendoul, et al., 2007), (Ducard, 2008) and the integration of automatic control (e.g. use of PIDs) to the flight control system (Salih, 2010), (Pounds, 2012). The aforementioned publications started having great impact, indicating that the engineering sector has expanded its share in the UAS field. The first surveys on COTS autopilots, cited by many, indicate the maturity of the market (Chao, et al., 2010). The modern problems in autopilots concern matters of resilience and adaptiveness (Garcia & Keshmiri, 2013). From year 2010 there is also interesting bibliography on the design of UAS, accumulating all the previous related research. Some useful and educational books are (Nonami, et al., 2010) and (Gundlach, 2012).

2.2 Integration of UAS to the civil airspace

From NASA's first try, with the Proteus model, to exploit the use of UAS in civil applications to the first feasibility analyses of integrating the UAS to the civil airspace, more than a decade has passed. In the first publications concerning the "domestication" of the UAS, the authors tried to report all the issues and risks involved in the integration (Anand, 2007), (Dalamagkidis, et al., 2008), (Finn & Wright, 2012). After year 2010, which is marked by the maturity of the UAS industry, the innovations and standardization procedures take place. The U.S. government (Defense Science Board (DSB) Task Force, 2007), (UAS Task Force, Airspace Integration Integrated Product Team, 2011) and European governmental institutions (Commision, 2014) publish the first proceedings in the rulemaking process and recognize the potential of UAS in civil operations. International organizations also publish concrete terminology on the civil use of UAS, the most important of which is (ICAO, 2011). Risk assessments take place as well, since civil security is put at stake with the introduction of UAS into the non-segregated airspace (Luxhoj, 2013).

Currently, the U.S. has the most organized ruleset of operating UAS in the NAS, which mandates the adoption of its practices by other countries as well. The European Union has also stated the importance of UAS in the civil scope and EASA has also taken measures for operations inside the continent. The most important milestones in UAS integration to the civil airspace are ICAO Circular No.328 and FAA's rule 107.

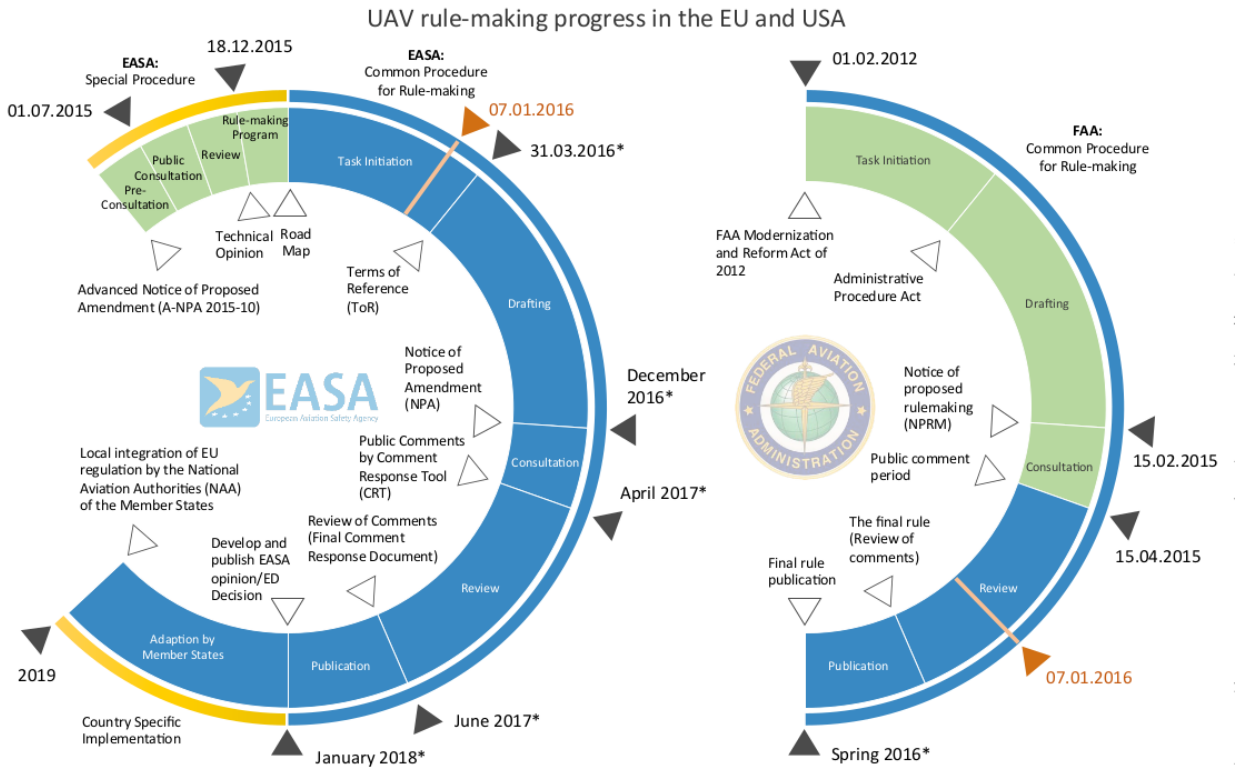


Figure 2.1 UAV rule-making process in the EU and USA for integration in the civil airspace (Drone Industry Insights, 2015)

2.3 Fully autonomous UAS operations

With the introduction of the UAS to the civil airspace, the need for full autonomy is fundamental. Most modern publications are concerned with the safety and airworthiness of heterogeneous sets of UAS in unknown environments. Many methods and algorithms have been adopted in order to tackle the safety issues imposed by the free flight of UAS to the non-segregated airspace. A summary of the S&A methods available can be found in 1.3 Autonomous operations of UAS.

There is a very interesting publication which presents the safety layers for collision avoidance in civil airspace and how the manned collision avoidance procedures can be integrated to the UAS S&A functionality (Korn & Edinger, 2008). The first sound results of S&A in UAS are evaluated by (Shakernia, et al., 2007), with authors mainly from the military industry, though. In Europe, the use of sensors for non-cooperative collision avoidance systems is evaluated and tested (Fasano, et al., 2008). The base of S&A systems is also set by Hutchings et al., by introducing the term Equivalent Level of Safety (ELOS) to compare flight safety between UAS and manned aviation (Hutchings, et al., 2007). A really good, recent report which is frequently referenced in the use of S&A in UAS is also published by ARC AIAA (Prats, et al., 2012). Finally, an overview of potential technologies in small-UAS for S&A systems, both cooperative and non-cooperative is presented in (Ramasamy, et al., 2014) and (Yu & Zhang, 2015).

2.4 Collision avoidance

There has been great research in collision avoidance methods through the years. Modern algorithms contain artificial intelligence elements and complex computation formulas. The issue of collision avoidance in mobile robots is being questioned before the technological advancements in unmanned aviation. Some of the most influential papers for mobile robots include the "dynamic window" approach, where the search zone for collision avoidance is parametrized according to the speed of the robot (Fox, et al., 1997) and the application of AI in real-time obstacle avoidance (Khatib, 1986).

The applications of collision avoidance in UAS exploit methods of trajectory planning (Richards & How, 2002), automatic control and swarm intelligence. The problem is recognized to be really complex, especially when the UAS is in a dynamic environment taking full advantage of their maneuvering capabilities (Frazzoli, et al., 2002). An interesting and influential publication describes the coordination of UAS by generating trajectories, taking into account the time domain as well (McLain & Beard, 2000).

2.5 Cooperative airborne-based sense and avoid

The most important stimulant for the research of cooperative ABSAA, a method of collision avoidance towards fully autonomous UAS, is FAA's "NextGen" program. According to NextGen, every airborne vehicle in the NAS must be equipped with an ADS-B transponder by the start of 2020. The same rule applies for the European airspace as well. It is not sure yet if the UAS are an exception to this mandate but there is research available to support their full integration into the NextGen NAS (Martel, et al., 2011), (Pahsa, et al., 2011), (Stark, et al., 2013). (Jill, 2015)

The U.S. Air Force has already set the *"roadmap for airworthiness and operational approval"* of ABSAA systems (Lester, et al., 2014). There is an interesting section in a MITRE Corporation's publication, describing the operation, needs and risks of cooperative ABSAA (Lacher, et al., 2010).

2.6 Relevant products

The theory and the undertaken research by the aforementioned institutions and organizations is directly applied to the UAS industry in the hardware, software and services field. The UAS ecosystem is rapidly growing since applications in the civil scope have arisen. Below is a market research of the most prominent manufacturers and service providers in the field.

Hardware

Most companies sell proprietary systems with a wide range of functions, mainly for imaging and telemetry. The biggest UAS technology exporters are, according to professional reports, China (DJI), France (Parrot) and U.S. (3DR). The need for innovation and industry advancement is supported mostly by U.S. ($41M) and Europe ($10M) through funding of early-stage start-ups. The highest demand in UAS, as of Q4 2015, is in multi-rotor aircrafts, with an exponential increase in this trend, proven by the number of registered multi-rotor UAS in the NAS. Apart from the numerous commercial and governmental platform manufacturers,

a great market share belongs to companies that develop individual components and systems for all the segments of a UAS. (Drone Industry Insights, 2015)

While companies like Parrot and DJI have the greatest market share in the consumer field, hobbyists and researchers who want to implement custom applications and ideas have to focus on companies providing open-source platforms and SDKs, the most important of which is 3DR.

Software

The company with the greatest impact, having a completely open ecosystem of software and hardware is 3DR (3DR, 2015). DJI also provides an advanced SDK but solely for their own systems. 3DR's Dronekit is part of Linux Foundation's collaborative project with codename Dronecode. Dronecode is an open source platform for the development of UAS (The Linux Foundation, 2016). The project members are international companies with great impact not only in the pure UAS industry (3DR, Parrot, Yuneec) but in the global hardware industry as well (Intel, Qualcomm). Many of the project members exploit Dronecode for applications in their proprietary systems (e.g. Parrot).

The Dronecode hub contains a set of open-source projects related to autopilots (PX4, Pixhawk), industry-standard communication protocols (UAVCAN, MAVLink), ground control stations (QGroundControl, APM Planner 2.0), simulation environments (MAVROS) and advanced control APIs (Dronekit). (Dronecode, 2016)

Services

A great share of the market is related to drone services. Since the FAA and EASA mandates the use of small UAS in the civil airspace only upon certification, there are many approved schools providing training to individuals and companies, both for recreational and commercial use. In case of an individual without UAS ownership/certification, there is a wide range of companies providing drone operation. Imaging in drones also plays a vital role in the services market, with a great number of companies in mapping, inspection and agriculture. In support of the above service provider infrastructure, there are suppliers, retailers and companies in the integration and engineering sector. (Drone Industry Insights, 2016)

3. System modelling

In this chapter, an abstract model of the system is going to be presented, along with the physical environment parameters taken into account. The chapter will be divided in four sections, each one modelling the input, processing, communication and output parameters of the system. From now on a multi-rotor UAS will be simply referred to UAS.

3.1 Input

A UAS is a system that interacts with the physical environment. Depending the flight mode/plan the user has set, the UAS continuously configures the rpm of each rotor, and as such the attitude of the system, in order to maintain its flight plan. The management of the UAS attitude (flight control) is handled by the autopilot and sensors are important for this role. More sensors can be equipped on the UAS for specific missions/advanced control, such as optical, gas and sonar, but for the purpose of this thesis, no use of extra sensors was exploited.

Inertial measurement unit – Speed, stability

The autopilot contains sensors the existence of which is vital to the UAS. Since the UAS supports maneuvers in three axes, namely pitch, yaw, roll, there must be information about the inertia in the three-dimensional pane. Also, in order to maintain stability, a gyroscope is needed, again with three-dimensional readings. Most commonly, the three-axis accelerometer and gyroscope are integrated into one sensor called inertial measurement unit (IMU), or motion processing unit (MPU). Usually the IMU is a fourth-generation MEMS sensor, meaning that data is delivered to the CPU already sampled, quantized, corrected/linearized and amplified. Because the IMU contains a temperature sensor for value correction, it can also output temperature data. It is important to mention that the temperature reading is not the atmospheric but the one inside the IMU.

The IMU is responsible for providing real-time measurements of angular velocity and linear acceleration. Usually, the sensors providing motion data have very high sensitivity and the sampling rate, depending the system bandwidth, can be up to 20MHz. The software should define the tolerance of these measurements to avoid message overhead.

Magnetometer – Compass heading

Apart from angular velocity and acceleration, the orientation of the UAS in relation to the north is also of great importance for navigation in a flight plan. This information is also important when detecting a conflict. The output of the magnetometer ranges from 0^o to 360^o degrees, with zero being the geographical (true) north.

Barometer – Altitude

Since a UAS must be able to maintain a user-set altitude, there must a sensor which measures it. The altitude is usually measured based on the difference of atmospheric pressure between sea level and the vehicle's height. It is important to know the relative altitude (height from ground-level) during take-off and landing in order to avoid a ground collision. Altitude is usually measured in meters through the barometric formula (Pixhawk, 2016).

GPS – Location

The GPS is a positioning system which allows the UAS to navigate based on satellite feed. In order to have a "3D fix" – accurate measurements in the 3D geographic coordinate system

(latitude, longitude, altitude), at least four satellites need to feed the GPS. The fourth is needed for the time variable. Precision is also very important in GPS measurements. Two metrics, vertical and horizontal dilution of precision (VDOP, HDOP) are used to define it. The smaller these values are, the better precision there is (Langley, 1999).

Sometimes GPS values differ in accuracy. It is important that the accuracy of GPS when having conflicting UAS be less than one meter. According to the GIS community in Stack Exchange, in order to achieve accuracy of less than about one meter (1.11 m), the reading should have at least five decimal places. See GPS Precision for more information.

Many times, it is needed to derive the distance in meters between two coordinates. This is usually derived from complex formulas taking into account the spherical shape of the earth. The distance between two conflicting small UAS is so small, though, that the area of interest is a flat surface. A simple Euclidean distance in the 2D space of the coordinates, always transformed to meters, is adequate. The third dimension, altitude, can be then compared to decide if there is a conflict.

Wi-Fi – Network messages

Along with the sensor system inside the autopilot, the UAS is also capable of receiving network messages that originate from nearby UAS through a Wi-Fi module. The UAS-specific messages include state parameters and their mission plan for the conflict detection system of the companion computer. More information will be given in System design and implementation.

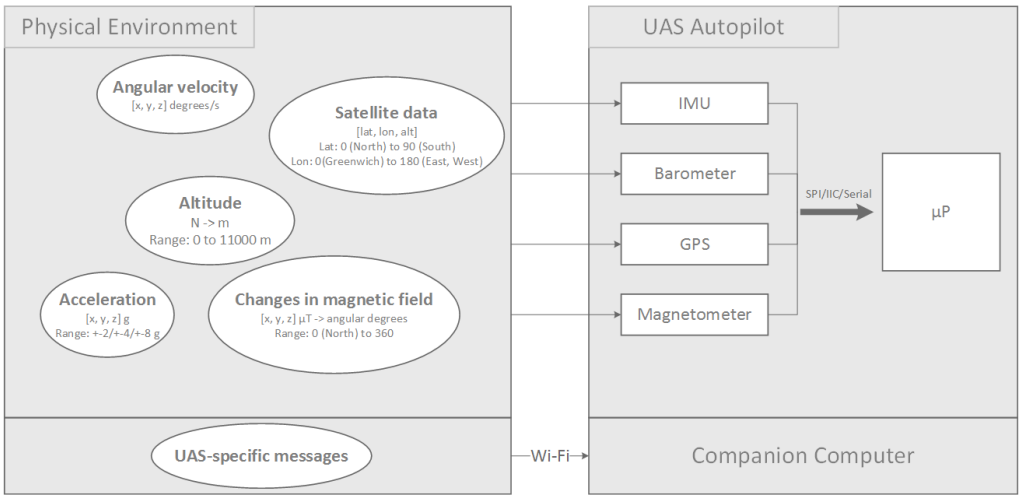


Figure 3.1 UAS autopilot sensing and receiving capabilities

Input listeners and thresholding

Dronekit has the ability to communicate with the autopilot via the low-latency communications protocol, MAVLink. The autopilot can provide vehicle-specific parameters: sensor readings and state parameters (battery level, flight mode and emergency state etc.). Not all parameters are useful for the solution developed so there is the need of choosing the fitting parameters and also choose when old values should be overwritten by new ones. For the first need, Dronekit

supports the necessary functions to add listeners to the chosen parameters. That means that whenever a parameter changes, a user-defined action will take place.

Because of the high sensitivity of the sensor systems, thus the high value change rate, an extra thresholding limit is put for each parameter. In that way, too many writes in the system's memory is avoided. The thresholding limits are determined upon experiments. According to the needs, the sensitivity of the sensors that the system observes is explicitly reduced.

3.2 Processing

The UAS is equipped with a range of different processors, each one with a different task: integrated into the sensor systems for data acquisition and signal processing, in the autopilot for the flight management and one extra for failsafe, inside a companion computer for computing-intensive tasks.

The sensors and the autopilot are black boxes for the purpose of this thesis, since the tasks involved are developed inside the companion computer. The tasks that are processed by the companion computer are:

- Send state parameters through a network socket
- Receive state parameters from nearby UAS
- Process received messages
- Run collision avoidance algorithm

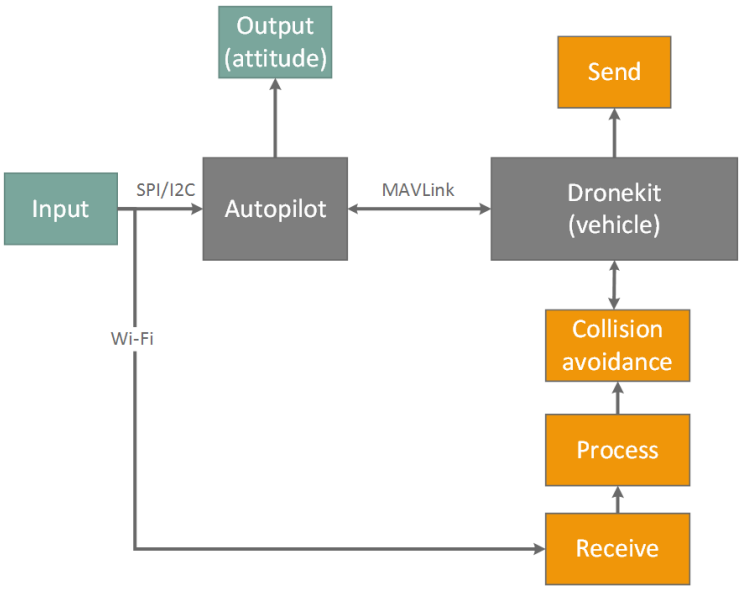


Figure 3.2 Top-level block diagram of the UAS. The orange blocks are the newly implemented modules. All the modules are daemon threads running concurrently at specific rates.

Energy and processing efficiency

The modules listed above and detailed in Figure 3.2 and Figure 3.3 are run as daemon threads in the companion computer. A computer that deals with advanced control of a UAS, especially

when this involves collision avoidance procedures which are important for the airworthiness of the aircraft, is important to be energy efficient, robust and have a high fault tolerance. Although computing-intensive tasks in a companion computer can be energy consuming, the energy that is needed to power the rotors is far greater and this is the bottleneck in short-range UAS.

The companion computer of a UAS (e.g. RPi, Odroid) is not an embedded processor, according to Marwedel, since the processor is of general use. What is more, it is running under a general-use OS (Linux) which is not designated to be real-time or for specific processes. (Marwedel, 2011)

On top of that, Python language is not able to exploit multi-core architectures unless using libraries that actually create a lot of message overhead and unpredictable lock sessions (Beazley, 2010). This concludes that the system is not optimized for these certain tasks. Further timing and testing procedures are described in Validation and performance evaluation, in order to check that the system is functional. Unfortunately, the processor interrupts and the number of system calls are unpredictable.

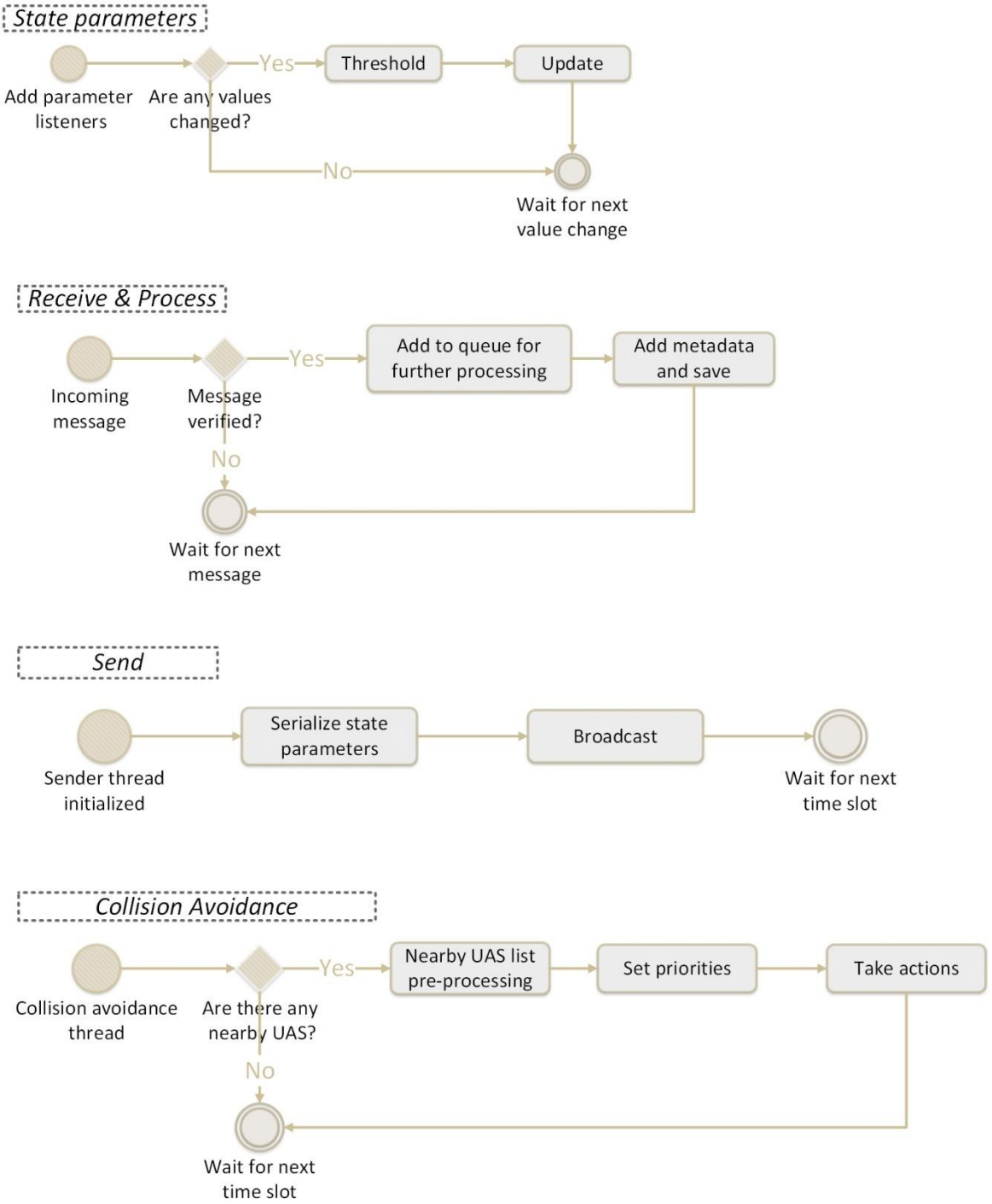


Figure 3.3 Activity diagrams of the task and the state parameters update modules

3.3 Communication

The UAS is a real-time system, in means that it is vital to interact immediately with its surrounding environment. A processor or OS might not be "real-time", e.g. it does not timeout an interrupt for a real-time process to continue its operation (The Linux Foundation, 2013). The communication layer has to be real-time though since it is important to deliver messages in a timely manner and uncorrupted. The main communication links are between:

- Sensors – Autopilot
- Autopilot – Companion computer
- Companion computer – Companion computer (on separate platforms)

In order to "guarantee" the real-time behavior of the system, both point-to-point communication and shared buses are used, depending the application.

Sensors – Autopilot (I^2C, SPI)
The external sensors can communicate with the autopilot via two kinds of protocols: Serial Peripheral Interface (SPI) and Inter-Integrated Circuit (I^2C).

The SPI is the simplest protocol, a single-master communication protocol where every slave SPI device connects to the master SPI device, thus needing slave select logic. The master and slave devices must share the same clock parameters (frequency, polarity and phase) in order to be able to communicate. Because of the binary values of the clock polarity and phase, there are four modes of operation. Physical interface characteristics, maximum data rate and addressing scheme do not need to be specified in this protocol.

I^2C is a multi-master protocol. Any slave can connect to any master as long as the devices have a unique 7-bit address and data is divided into bytes. A communication between any number of devices can be "flawless on just 2 physical wires". The bus speed can be 100kbps, 400kbps or 3.4Mbps.

Since I^2C achieves communication between multiple devices on a single bus, it can have a much more elegant topology than that of the SPI, where slave select logic and multiple wiring is mandatory in case of many devices. The SPI though can achieve much faster speed rates (over 10Mbps), since there are no limitations by the protocol itself. (ByteParadigm, 2016)

Autopilot – Companion Computer (MAVLink)
MAVLink is a low-latency communications protocol between the autopilot and a companion computer or ground control station. It is an open source project that has become very popular due to its byte-level serialization, thus making it lightweight and supported by most radios. MAVLink packets are dependent on the command. The message overhead is just eight bytes. (QGroundControl, 2016)

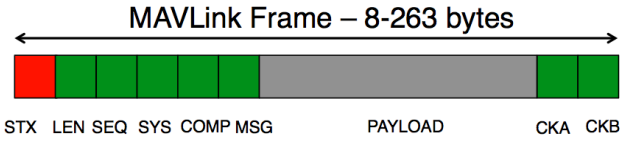


Figure 3.4 MAVLink packet anatomy (QGroundControl, 2016)

Companion computer-to-Companion computer (IEEE 802.11)
The UAS have the ability to communicate with others, in order to exchange information about their state. The transmission is carried out by the IEEE 802.11 (or commonly Wi-Fi) standard, a wireless local area network (WLAN) interface which provides 1 or 2 Mbps of transmission in the 2.4 GHz band. The 802.11 protocol belongs to layer 1 and 2 of the OSI architecture, namely Physical and Data link. That means that 802.11 can be used by any Network (OSI layer

3), Transport (OSI layer 4), and Host Layers (OSI 4 to 7) applicable. Currently, the software supports communication between HTTP sockets with UDP broadcast messages.

UDP has been chosen over TCP since the latter introduces high latency in the handshake of the devices. UDP is not lossless but in 5.1 Networking analysis, the limitations of this protocol are analyzed.

Because the 802.11 protocol can cause a lot of collisions in the receivers, especially if the messages from the UAS are broadcasted, there is the need for ensuring the integrity of the packet. For this reason, there is a checksum verification upon the receipt of a packet. The checksum type is MD5 in order to minimize the complexity of the software. MD5 is a hashing algorithm which is not safe for encryption of data since it is a broken algorithm, but it is good for creating hashes for checksum verification because it is lightweight. MD5 creates a 128-bit digest of the packet and it is sent along with the original value. When the receiver gets the message, it performs a new MD5 checksum of the information and verifies it with the one sent along. If they are equal it means that the data received in not corrupt.

3.4 Output

The output of the system can be considered the final outcome of all the processing in the autopilot and the companion computer. The purpose of all this processing is to finally guide the UAS to a specified location, thus controlling the attitude of the UAS, ultimately controlling the speed (rotations per minute - rpm) of each rotor.

The autopilot outputs control signals to an Electronic Speed Controller (ESC) so that the later can adjust the rpm of the motor accordingly. The ESC's are powered by the UAS power source, usually a Li-Po battery, in order to provide the necessary amperage to the motors. The more advanced controller is preferred over a PCB unit because of the nature of the brushless motors used in quadcopter UAS:

The brushless motors of a multi-copter UAS contain sets of coils in the rotor that operate as electromagnets when current is passed through them. A permanent magnet is the stator (the stator/rotor can be the opposite). The coils are powered each one at a different time slot, thus having a three-phase ESC output, in a manner which allows the continuous movement of the rotor. A sensor is used to give feedback to the ESC about the current position of the rotor. The ESC can then activate the necessary coils to keep a steady rpm rate.

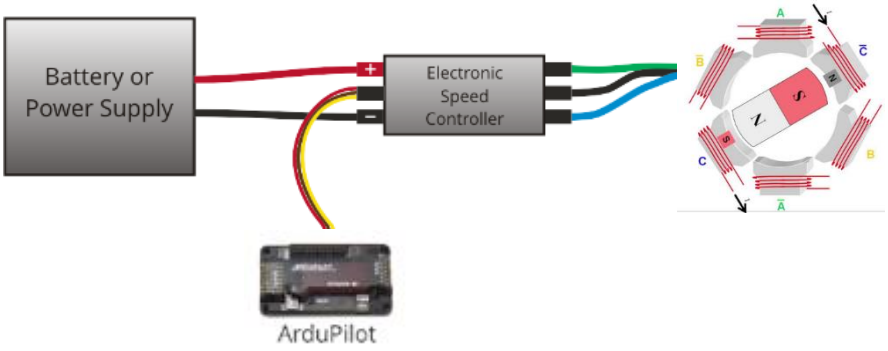


Figure 3.5 Layout of system output (Techno Sainz, 2016)

4. System design and implementation

This chapter provides a detailed description of the system, the design parameters taken into account and the programming practices. There is also going to be detailed description of the timing factor and the handling of events.

4.1 Top-level design

There is a try to create a system with good granularity in order to make easier modifications to the software. In this way the deployment of new protocols, porting to multi-processing functionality instead of multi-threading, addition/removal of new modules is more feasible. A more comprehensive view of the system is from a data flow prospect. For this reason, a DFDM is shown in Figure 4.2, for easier inspection of the design logic. The new library is a set of five classes:

DroneNetwork: handles the communication protocols and initializes the send/receive/process threads in order to keep the nearby UAS state parameter list up-to-date. This class is the central node of the whole system.

Send: Initialized by the drone network class, this thread is responsible for fetching the UAS parameters and transmitting them with the specified protocol.

Receive: Initialized by the drone network class, this thread is responsible for receiving any incoming messages, verifying their integrity and their source and, if coming from a UAS, it is put to a queue for further processing.

ReceiveTask: Initialized by the drone network class, this thread is responsible for getting the messages put in queue by the Receive class, adding metadata and storing to the appropriate list. The metadata stored along with the parameters of the drone are distance between the two UAS and a timestamp indicating the time received. If the distance calculated is beyond the specified safety zone, the message is ignored. If the message comes from a conflicting UAS, then the thread is responsible for updating the nearby UAS list.

CollisionAvoidance: This class must be initialized in the main thread, specifying the desired collision avoidance protocol. Currently, a naïve protocol is used which works like the priorities of cars in a crossroad. This class contains methods for prioritization and context switching, which are going to be described later in this chapter

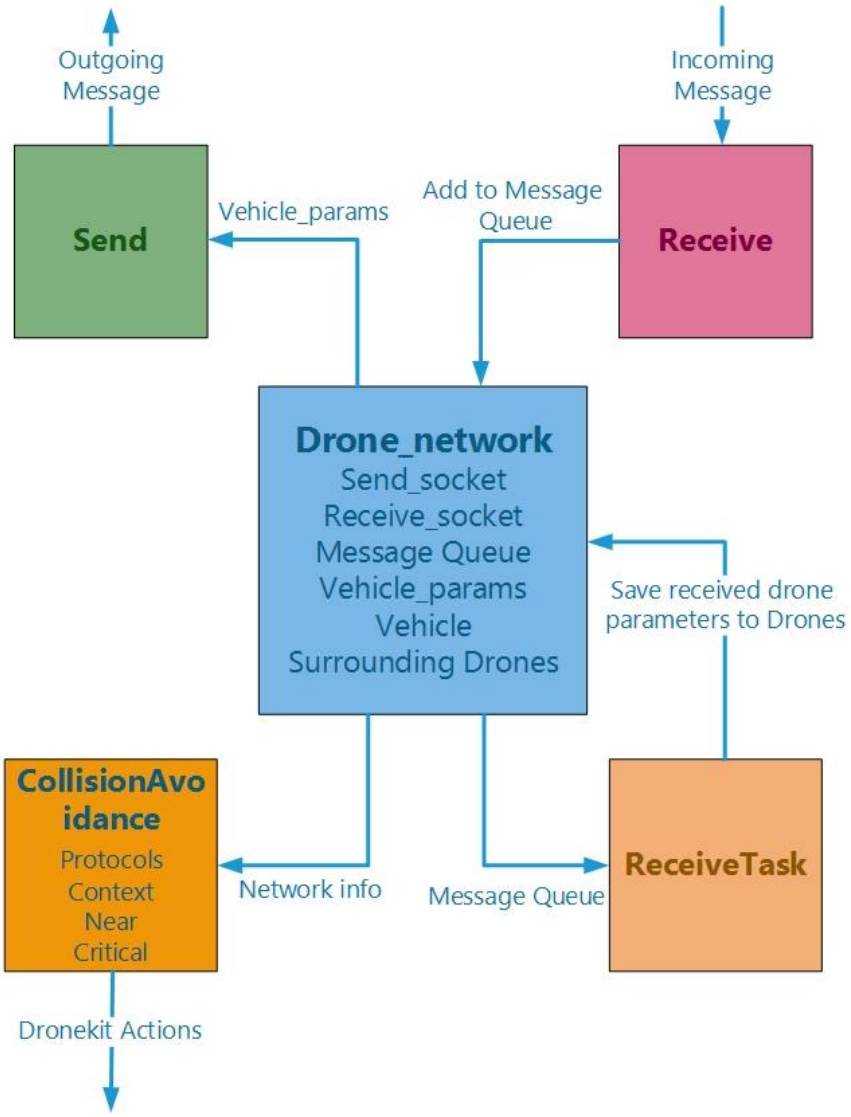


Figure 4.1 Top-level design of the cooperative ABSAA API.

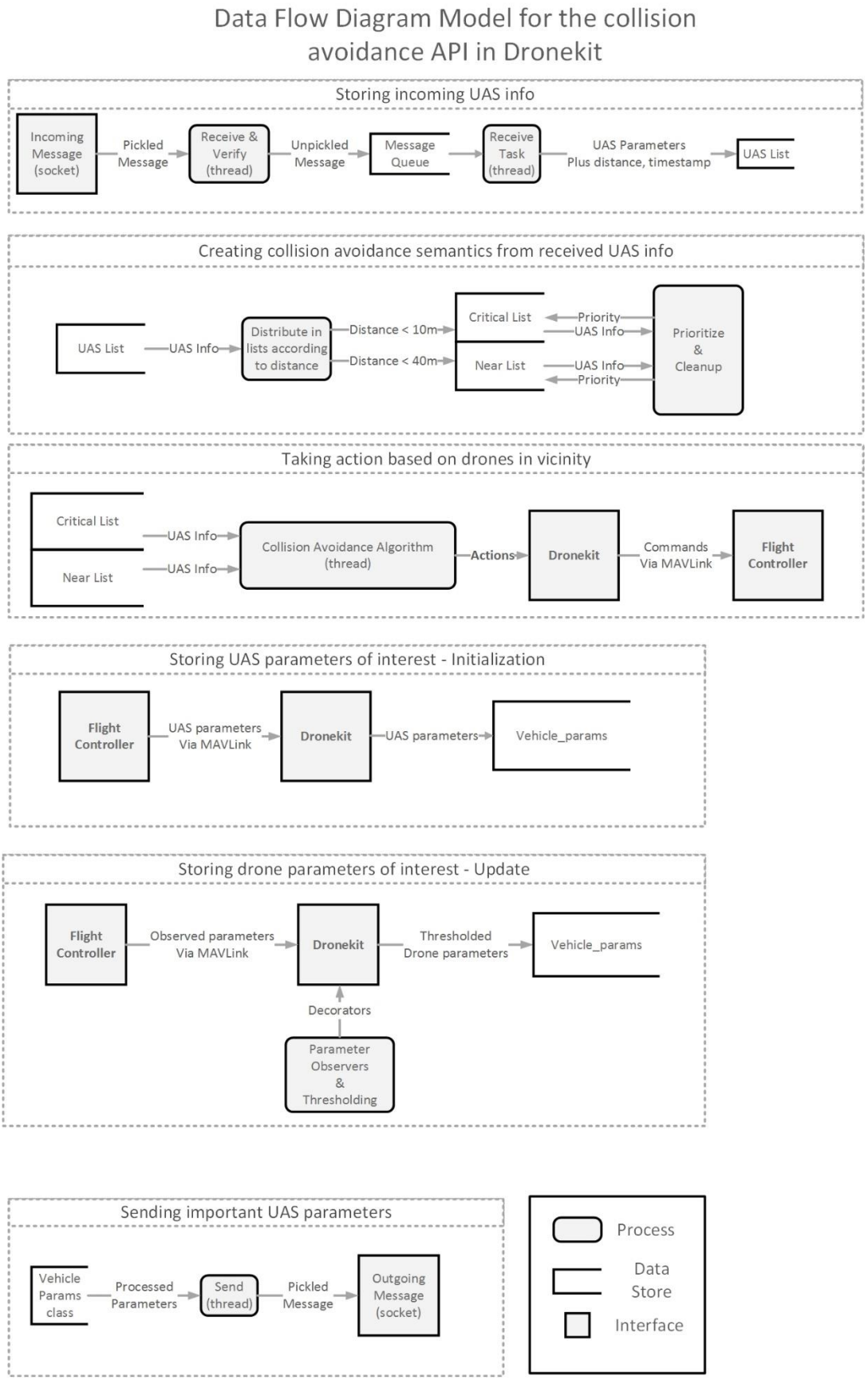


Figure 4.2 DFDM of the implemented system.

Why two threads for receiving?
The parameters received from the network are transferred from the receiver to the rest of the system via a thread-safe Python queue. The collision of messages at the receiver is beyond the matters of this thesis and as such, cannot be predicted or avoided. What can be avoided though is the loss of a message inside the system.

The task of processing a message requires a number of CPU-bound tasks: calculation of inter-drone distance and list update. What is more, if a long interrupt or system call takes place in the processor, a number of messages can be lost. If, while performing calculations or undertaking a long interrupt, a new message arrives, the receiver thread will be busy and will lose the message.

In order to avoid this kind of hazard, a new thread, namely ReceiveTask, is responsible of performing these calculations for every message that arrives. The Receive thread is then responsible only for checking the message integrity and passing it to the ReceiveTask thread through a queue.

The complexity of the Receiver's algorithm is then reduced to O(1), while the former complexity of the subsystem, supposing the multiplications are carried out by the Karatsuba algorithm, was $O(n^{1.585})$.

An asymptotic analysis may not be very convincing when talking about bounded input systems but a timing analysis is described in Chapter Five. In any case, minimizing the workload of the receiver and keeping a message history minimizes data loss. Queuing the received messages for further processing by other threads and performing a minimal checksum verification are measures that reduce data loss and increase data integrity.

4.2 Collision avoidance protocol

As presented in chapter 2.4 Collision avoidance, the advancements in collision avoidance of UAS are very important. The development of a new method, or the deployment of an advanced collision avoidance method in the implemented system would be unreasonable. Due to the system model though, it has become feasible for the developer to deploy his/her own collision avoidance approach for testing. The API created for cooperative ABSAA contains some useful practices that can be exploited by the users. The most important are the prioritization algorithm, context switching and taking temporary control during a mission.

The main idea behind the fundamentals of the presented approach in cooperative ABSAA using mainly GPS for sharing location and the IEEE 802.11 WLAN protocol, is maintaining the nearby UAS lists and deciding the new flight plan to avoid potential conflicts. 'Nearby' is decided explicitly inside the source code. The safety zone is a sphere with a certain radius. In order to decide if a UAS is in the vicinity of another UAS, the Euclidean distance is calculated first and then the altitudes are compared (See Appendix A: Transformations and formulas). An abstraction of the protocol is shown in Figure 4.3.

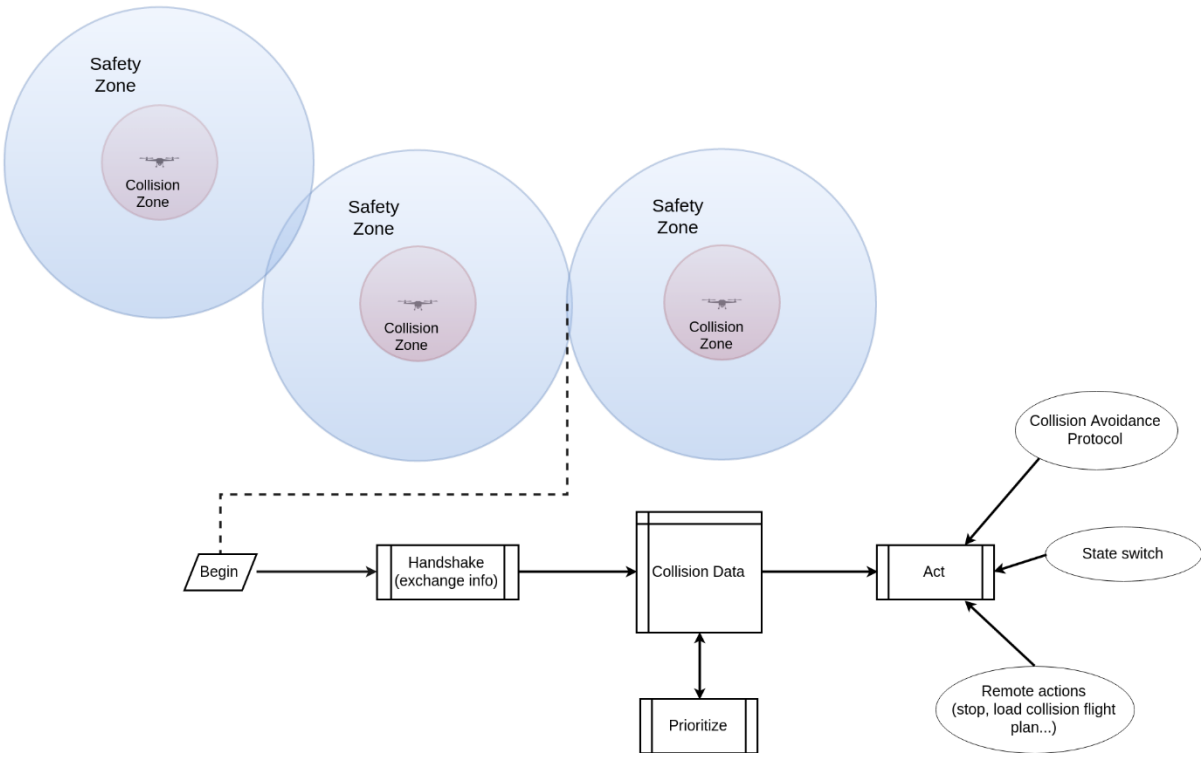


Figure 4.3 An abstraction of the collision avoidance API

UAS momentum

When the collision avoidance protocol requires the vehicle to reduce its groundspeed to zero, one of the most important factors affecting the system is negative acceleration in the x-axis. This is because a UAS cannot stop immediately, as it has momentum from its pre-conflict groundspeed. In order for a UAS to reduce its groundspeed it needs to come at a break angle which gradually causes the system to hold its position. Ideally the angle would be so high that braking could happen immediately, but this would have the tradeoff of unstable or even system failing behavior.

The procedure of 'stopping' can be implemented with various ways by the autopilot and every way has significant differences in terms of acceleration. The decision to follow a certain stopping procedure is decided with trial and error since the factors affecting acceleration are very low-level and airframe/avionics dependable. Most recent ArduCopter firmware (AC3.3) has support for braking but this mode is not backwards compatible with previous versions. A more general solution should be given so that stopping can be feasible in older autopilot versions as well. Since the decision was made with trial and error, more information on the decision-making process is described in chapter 5.2 Software in the loop (SITL).

Prioritization

The collision data which is provided by the conflicting drones is maintained by the CollisionAvoidance thread. It is responsible for cleaning-up obsolete information and giving priorities based on each of the UAS parameters. The prioritization algorithm is implemented inside the thread class and, in order to be functional, must be run by every UAS instance

globally. The priorities model is taking into account four sets of state parameters which have certain importance The sets of attributes are not orthogonal, but some are more important than others.

The *SYSTEM_STATUS* set describes the system status as a whole. It can be divided into subsystems which describe a higher-level state. For example, the system is on the ground if it is in state UNINIT, POWEROFF, BOOT, STANDBY, LOCKED, CALIBRATING. More information about the state parameters set can be found on the Appendix D: Prioritization parameter sets.

The *Capabilities* set describes the feasibility of giving remote commands to attitude and altitude changes. If a UAS does not support such a capability, then it is difficult to change its flight plan, so it needs to have a higher priority in order for a conflict to be avoided.

The *SYSTEM_MODE* set describes the flight mode of the UAS. If it is in a MANUAL mode then it is receiving RC commands, so there is the need for channel overriding in order to change its attitude. If the mode is one in the AUTO set, its mission plan must be saved before the avoidance plan is loaded. See

UAS context switching for more information.

MISSION_IMPORTANCE is a new feature of the UAS, which should be adopted by the industry in order to differentiate a UAS operating for Search & Rescue missions from a UAS operated by hobbyists. The Mission Importance is a number that indicates the UAS' operation importance:

- **Level 2**: Used for governmental, search & rescue, national/civil security operations.
- **Level 1**: For UAS designated for commercial use, businesses and industry-related operations.
- **Level 0**: Used by hobbyists for recreation and entertainment.

The assignment of mission importance in a UAS has potential security threats. Since the code is open source, the developers can edit the variable so as to be of top importance, even if the intended use is for recreation (Level 0). So far, this attribute cannot be safely used or taken into account by the prioritization algorithm but in the future it can provide really useful applications.

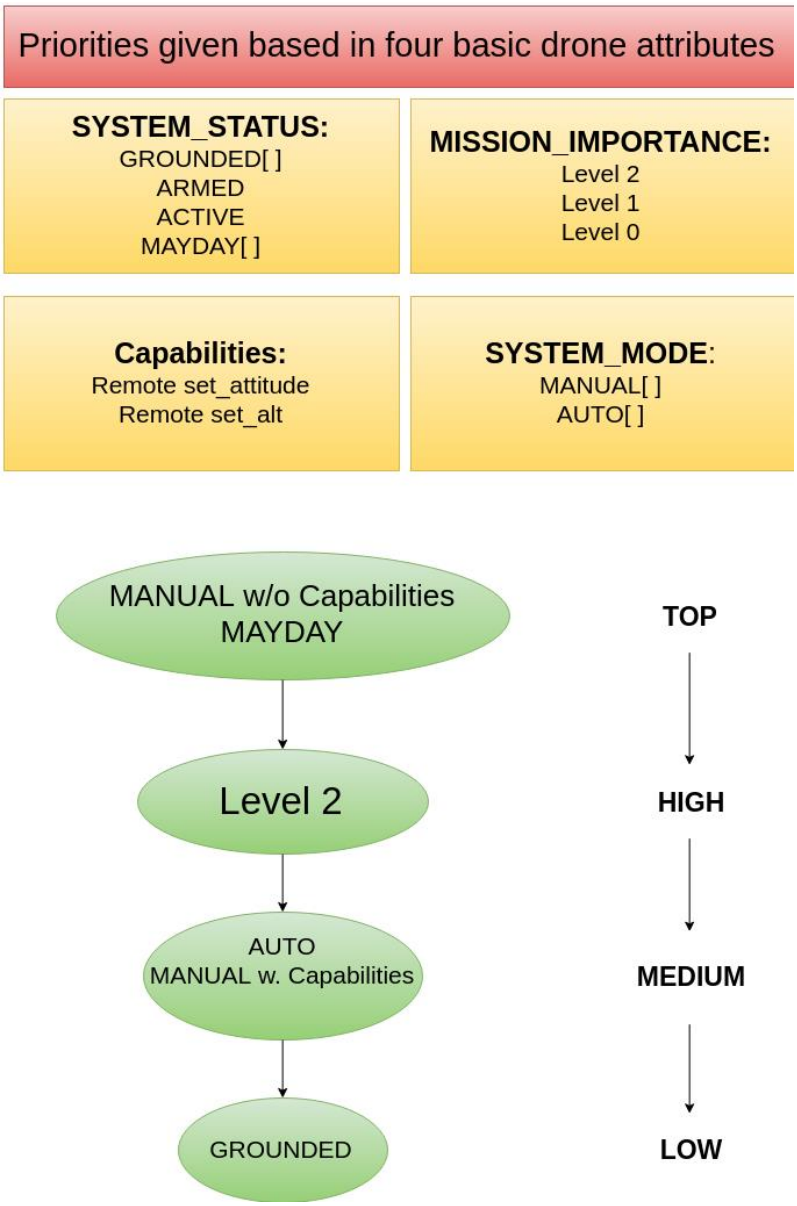


Figure 4.4 The prioritization model.

How globalized can the algorithm be?

There is a worst-case scenario crafted for this question. Imagine three UAS (expandable to more than three with the same logic) in an area where the UAS in the middle can sense the other two, but the other two cannot sense each other. This scenario will be called ‘Chaining of Priorities’. The example of Figure 4.5 explains the scene.

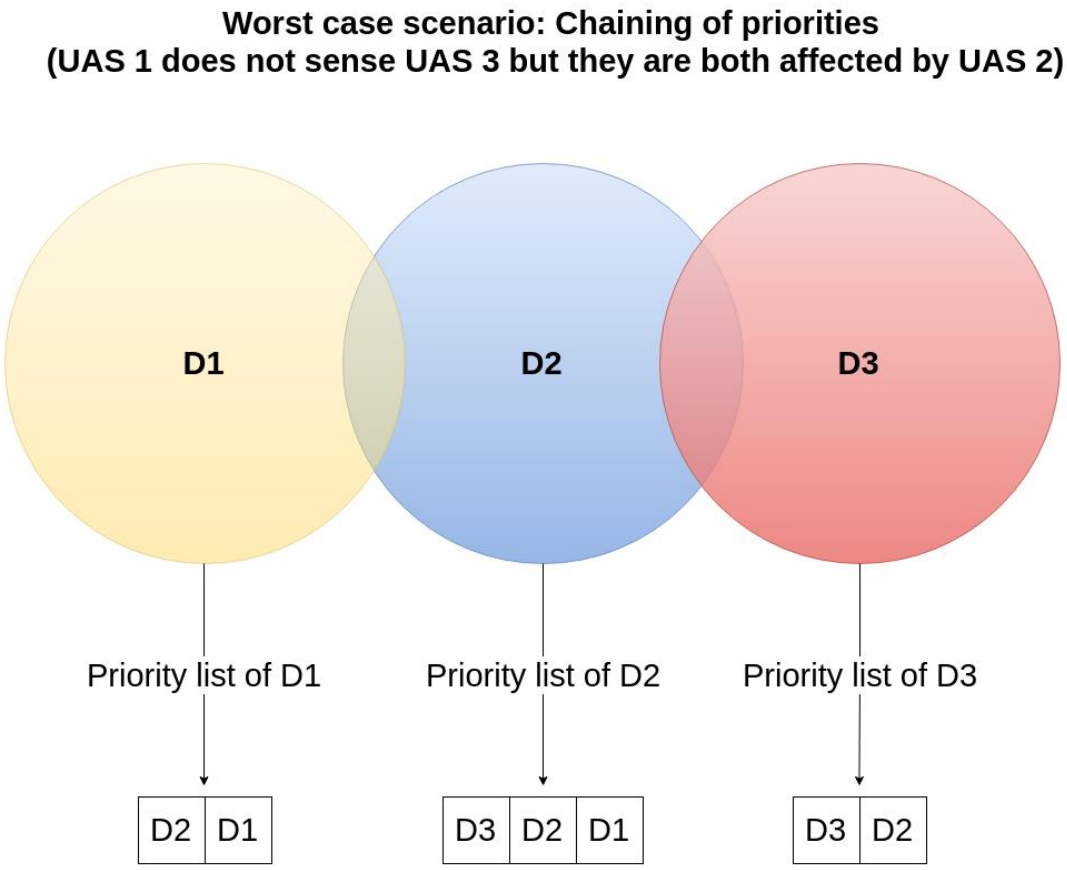


Figure 4.5 Scenario of chaining in the prioritization algorithm

Someone would expect that the priorities would be corrupt. If it is decided that the priority is D3->D2->D1, then D1 would wait for D2 and D3 would act before D2. So, according to the algorithm, D1 might not be sensing D3 but it will wait until D2 has moved away. Implicitly, D1 waits for D3.

The real problem here is not the prioritization algorithm itself but message collisions: If two conflicting UAS are not aware of their counterparts, or if the lowest priority UAS is not aware of its highest priority UAS then the prioritization algorithm will be right, but actual priorities will not be correct.

Another problem which arises when priorities are chained, is that the whole set of stakeholders can expand to a very wide area in a manner that the starting and ending UAS would never have a conflict otherwise. This scenario is visualized in Figure 4.6.

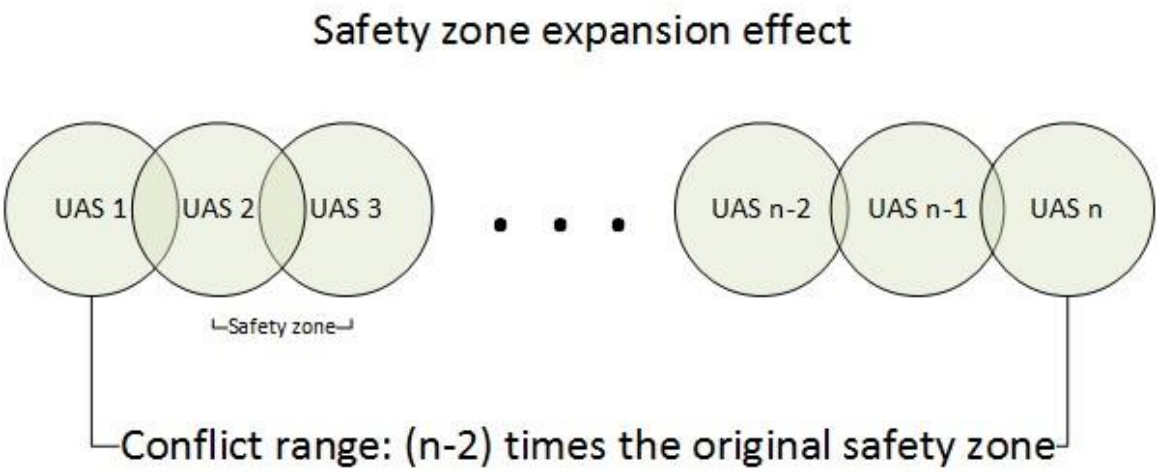


Figure 4.6 The effect of multiple UAS prioritization chaining.

UAS context switching

A new idea is expressed with the term 'UAS context switching'. Most UAS perform autonomous flights and have a preloaded mission plan that follow. During a conflict, there may be a future possibility of changing the flight plan temporarily. The UAS, though, should be bound to finish the mission it is programmed for. For this reason, a set of state parameters along with the mission plan are saved in the memory and when the conflict is over, they are loaded back to the UAS. This operation allows for a temporary change in the UAS state/flight mode in order for it to overcome a conflict and, as such, increases the adaptability of the system.

Taking control of the UAS allows for immediate response in case of an emergency. Currently, taking control means holding the position and altitude of the UAS steady while saving the previous state with the context switching module. In order to give back the control to the pre-programmed mission, a context restore takes place.

Although there is a significant latency, as shown in the simulation, during the context switching procedure, this operation is fundamental not only in collision avoidance but also in other applications discussed in Discussion and future work.

4.3 Timing

Timing is a very important factor when dealing with real-time embedded systems (Marwedel, 2011). Concurrent processes can lock the CPU or I/O for long periods of time if they are not handled correctly, resulting the system to freeze. The scheduler of the system is actually the one in the OS, but inside the code there is the need to suspend the execution of daemon threads at a given frequency, thus releasing their locks for other threads that must run as well.

Periodic suspension of thread activity

The frequency the threads are suspended is dependent on the average airspeed of the UAS. This happens because the frequency the threads are suspended, especially the receiver and sender ones, is dependent on the frequency the UAS messages are broadcasted. If a UAS has an airspeed of five meters per second, then it should have its position known to the surrounding

UAS two or three times per second. This way, the UAS would be sensed each time it would cover a distance of one to two meters. The frequency would then be set to send the parameters every 0.3 to 0.5 seconds. For optimization issues, the rest of the system would also be active every 0.3 to 0.5 seconds, so that scheduling can be easier.

If a UAS is flying with an airspeed of 44m/s, which is by the way the speed limit set by the FAA (FAA, 2016), then either the safety zone should be expanded or the frequency should be set to at least twenty messages per second. This would of course cause a lot of jamming in the communication channel, but this is discussed in Validation and performance evaluation. The periodic tasks are the CollisionAvoidance and Send threads. Some application would require the suspension in the main thread as well.

Non-blocking operations
Besides the suspension of the above threads, there are also some I/O-bound threads, the Receive and ReceiveTask where they should run continuously in order to receive and process the incoming messages. Continuous run, or else blocking operation, would cause very long system calls and the system would be susceptible to freezing.

For this reason, the network socket of the Receive thread is set to non-blocking, meaning that the thread returns immediately if the object does not have any I/O event. This allows for the shortest system calls possible. The problem that raises is that non-blocking operations return immediately, so it is not known when the socket has an incoming message afterwards. Polling the socket is not an optimized solution since, between the polling windows, an I/O event will be ignored.

An optimum solution to the problem is querying whether the socket is ready for reading with the select() method. That way, the thread is running only when there is an actual I/O event. The ReceiveTask thread, since it receives the messages from a queue, is active only when there is an object in the queue. Since the select() method can wait for an unspecified amount of time, an extra layer of protection is added by adding a timeout of one second. This timeout influences the maximum latency of the receive_task worker since both the receiver and worker are accessing a thread-safe queue. For this reason, a higher timeout in select() would potentially cause intolerable I/O-related latencies.

4.4 Event handling

Through the system, there are numerous operations which could raise an event. If these events are not handled accordingly, the system would be very susceptible to failure. A level of fault tolerance should also be introduced to the system. The events can be internal if they are caused by the components of the UAS or external if they are caused by the environment (Marwedel, 2011).

External
The events caused by the environment can be infinite in the real world: Weather conditions, conflicts with cooperative or non-cooperative objects and human interactions. In such a dynamic environment, the system must be highly adaptable. The handling of external events can be mostly avoided by the sensing mechanisms of the systems, such as the IMU of the autopilot and the collision avoidance module of the companion computer. Research is being

conducted in order to create more resilient systems, being described with more complex models, taking into account more factors of their surrounding environment.

Internal
The events of interest that can be raised internally are mostly exceptions, which must be handled appropriately in order to ensure a safe operation for the UAS and the user. Every error is described by a unique ID. Defensive coding is needed, usually with try-catch blocks, in order to handle the exceptions that might rise in a piece of code. The most usual exceptions raised are:

- **Socket-related errors**
  Most usually if the address is not valid or if the OS cannot open/operate/close a socket.
- **Empty or full queue**
  If there is a try to get an object from an empty queue, the software must simply ignore the error and wait for an entry. If there is a try to write an object to a full queue, then the software must temporarily save all the messages to a new structure.
- **Object serialization (pickling)**
  Pickling is the python algorithm for serialization of data. Serialization is needed in order to pass messages through channels. If an object cannot be serialized, e.g. a custom data type, an error is raised. Usually the software must warn the user that the object was not serialized, and pass an empty string through the channel, in order to avoid further system damage. The received message should then be checked and, if invalid, be ignored.

Failsafe operation
The events that can cause hazards to the system must be handled in a way that makes the system fault tolerant. If an event is an exception which is unhandled by the software, the system can run into an emergency state. In the event of a system failure, a failsafe operation must take place in order to protect the system.

In case of a companion computer failure, the autopilot must still be functional in order to handle the rest of the UAS operation. The autopilot and the companion computer are separate systems, so the failure of one does not cause a failure to the other. If the companion computer fails, the autopilot is put into failsafe and causes the UAS to land as safely as possible.

In the autopilot system, there usually is a spare processor in the event of failure, so that the processor can take over and perform all the necessary operations to ensure the safety of the UAS (assign its state to Emergency and the flight mode to Land).

Along the Dronekit code and the extension developed for this thesis, there is a flag that states that the system is in emergency. For example, if the networking module cannot start the necessary threads for communication and collision avoidance, the system is not allowed to arm its motors and takeoff.

4.5 Hardware and software used

Most of the thesis work was implemented and simulated on a PC running Linux OS, Ubuntu 16.04 Mate distribution, with 8GB DDR3 (1666Mhz) RAM, Intel® Core™ i5-2430M CPU @

2.40GHz × 4. The code was written in Python 2.7 language and tested in a software-in-the-loop (SITL) simulator, provided by Dronekit.

The code was then deployed, on two multi-rotor UAS, on two different companion computers:

- Raspberry Pi 2 model B with a supported Wi-Fi dongle
- Raspberry Pi 3 model B

The aforementioned computers were running lite versions of Linux, 64-bit Raspbian in ARM-based architecture. Some Python 2.7 scripts were deployed in the GCS for pre-flight testing of the network communication link between the systems.

The computer used for the development was of minor importance. As far as a computer with internet connectivity could run a Linux distribution with Python 2.7 installed, along with some project-specific libraries that will be detailed later, any modern system should be adequate.

Multi-rotor UAS

The project was tested on two UAS with different airframes, thus different air-dynamics. Even though the two airframes have about the same weight, the motors in the foam quadcopter are more powerful. This is because the foam is very lightweight and has relatively big dimensions. This frame is not ideal in windy conditions and has proven instability under these circumstances.

The aluminum Phone-Drone, which is designed by the TUC SenseLab members, has smaller dimensions and is denser than its EPP counterpart. This feature allows flight in more extreme conditions and is more stable.

Table 1 Comparison of two UAS specifications.

	Foam quadcopter	*SenseLab Phone-Drone*
Material	Expanded Polypropylene	Aluminum
Size(mm)/Weight(g)	640x640x80/500	250x250x50/600
Motors	Multistar 2209-980	T-Motor MN1806-14
Autopilot	APM 2.5 autopilot	APM 2.6 autopilot
GPS/Compass	3DR GPS LEA-6 v1.1/On-board	3DR uBlox GPS/Yes
Companion Computer	Raspberry Pi 2 model B (year 2015)	Raspberry Pi 3 model B (year 2016)
IEEE 802.11	Odroid Wi-Fi 3	On-board
6-ch 2.4GHz Receiver	Spektrum AR610	Saturn X6

Software

The development of the software was done in Linux Ubuntu 16.04 environment, with Sublime Text 2.0 editing software. A Linux distribution was chosen over a Windows OS since the development and testing is more straightforward in these systems. The developed software leveraged the API of 3DR's open-source project, dronekit-python, and as such, the

programming language was Python. In order for dronekit-python to run, the contributors have already created a python library called droneapi.

Dronekit 2.7

Dronekit is powered by 3DR and is an open-source project aiding in advanced UAS control. Dronekit is a complete drone developer tools ecosystem providing the API for the development of applications in computers (dronekit-python), smartphones (dronekit-android) and the cloud (currently not supported yet). From now on, when referring to dronekit-python, it will be written simply as dronekit.

Dronekit is written in C++ and provides an extra abstraction layer for the communication between an onboard computer and the autopilot. For this reason, it aids in developing processor-intensive tasks that augment the functionality of the autopilot. What is more, the applications are written in Python, a high-level programming language extremely versatile and easy for prototyping. With dronekit, someone can call methods, instead of constructing messages for the autopilot, and store their output directly in the abstract data types of Python. (3D Robotics, 2016)

The number of SDK environments for UAS is very limited since most companies provide proprietary, out-of-the-box solutions. The best rival of dronekit is the SDK of DJI, but the latter is functional only for DJI hardware (DJI, 2016). What is more, dronekit belongs to The Linux Foundation's Dronecode project, a collaborative open-source ecosystem for UAS. There are numerous companies and individuals supporting the project and as such, many of its projects (MAVLink, Ardupilot to name a few) have become UAS industry standards.

Ardupilot 3.2.1

The autopilot is an integral part of a UAS. For this reason, a Dronecode open-source project called ArduPilot is responsible of operating the UAS flight code in conjunction with the companion computer. Figure 4.7 is a very good explanatory top-level architecture, describing the functionality of ArduPilot and its connection with the rest of the UAS subsystems. Because of the hardware abstraction layer (HAL) of Ardupilot and its great support in COTS autopilot hardware, there was no need for further survey on the autopilot project. (ArduPilot Dev Team, 2016)

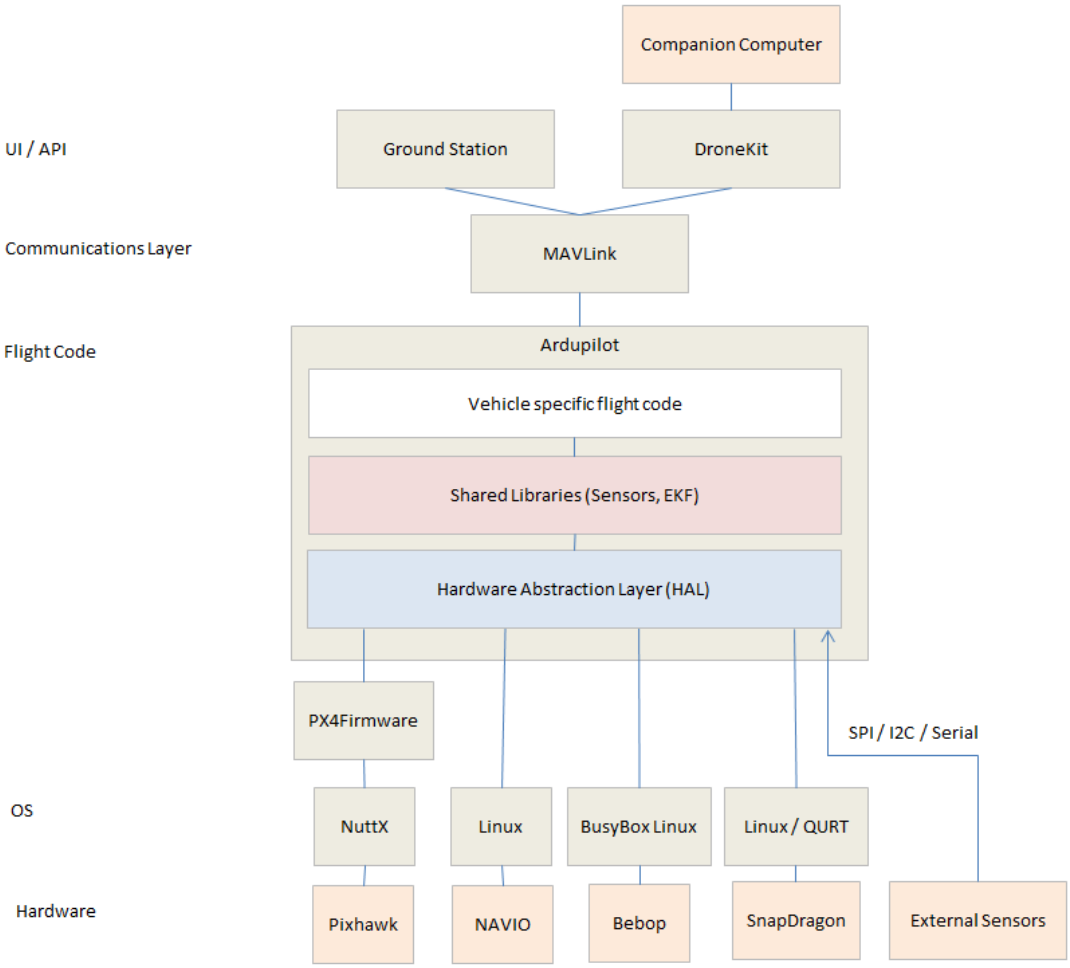


Figure 4.7 Top-level architecture of ArduPilot (ArduPilot, 2016).

MAVLink 1.0

The interface between the autopilot and the companion computer/GCS is the MAVLink communications protocol. MAVLink is another open-source project of Dronecode and has become the industry standard in the communication segment of a UAS. The main reason is that the MAVLink protocol is extremely lightweight since messages are serialized in byte arrays. This means that they are transferred optimally between processes (autopilot/companion computer) and are also appropriate for use with any type of radio modem (autopilot/GCS).

The project currently supports eighteen types of autopilots for twenty-eight types of vehicles. Among the message types, the most common are messages containing information about heartbeat, vehicle state, changes in sensor values and current flight plan. The most useful commands that can be given via dronekit and/or the GCS include change in flight mode, mission load, takeoff/landing, payload operation (e.g. gimbal). If there is a parameter that is not wrapped in a MAVLink message already, there is support for custom message definition by editing xml files (dialects).

Although there is extensive documentation for the message types and commands, creating listeners for the latter can be a complex task. For this reason and in order to increase the

readability of the code, dronekit-python's API provides an extra layer of abstraction in the use of the most common MAVLink structures. (QGroundControl, 2016)

Dronekit-sitl
An ArduPilot simulation environment, software-in-the-loop (SITL) is also being developed by dronekit contributors. The tool is actually a python library and is called dronekit-sitl. There are pre-built vehicle binaries, including quadcopter, but the user can apply custom binaries for local use. The tool is used for extensive testing of the autopilot software and a set of pre-built commands can be sent through MAVLink. Note than Dronekit-SITL is able to run only on processors of the x86 family, thus making it impossible to run on the RPi's or Odroid, which have an ARM-based SoC. (dronekit, 2016)

APM Planner 2
Both real UAS testing and data from dronekit-sitl can also be sent to a user interface for visualization, the APM Planner. The project is part of the Dronecode collaborative project that provides the user interface for mission planning in multiple UAS platforms. The communication between the APM Planner and the SITL simulator is full duplex, meaning that besides reading the UAS parameters, commands can be passed through the MAVLink protocol from the APM Planner to the autopilot, too. (ArduPilot, 2016)

4.6 Online repository and documentation

The purpose of this thesis is to create an API extension for the already developed Dronekit-Python API. For this reason, the project is uploaded to a GitHub repository and is being updated constantly in order to add new functionality and fix bugs. The open-source software is covered with the GNU General Public License version 3.0.

Git
GitHub uses the Git distributed version control system. Any user can clone the repository to his/her system and modify it. The modifications can be done locally and uploaded as a new branch or be uploaded to the main project (master branch). In that way, the project can have many contributors and users.

Sphinx
In order for the project to have many contributors and users, there is the need for extensive documentation both for programmers and users. For this reason, documentation was written with the help of reStructuredText and the Sphinx formatter. Sphinx and reStructuredText are the official documentation tools of Dronekit-Python and that is why they were chosen for this project as well. Sphinx is also the official documentation formatter of Python.

reStructuredText is a scripting language that is later processed by a formatter (in this case Sphinx) and creates documentation interfaces in many formats. The format used for dk-plus documentation is html, so that it can be read by any browser. The documentation can be accessed through the GitHub project page. An offline version of the documentation is presented in Appendix F: User's and programmer's manual.

5. Validation and performance evaluation

The purpose of this chapter is to evaluate the functionality of the system. Both lab testing and customer verification techniques will be applied. The results will then be analyzed to describe the performance limits, advantages and disadvantages of the system. The testing of the subsystems and individual functions is not described in this chapter as the changes in the source code were minor bug fixes.

5.1 Networking analysis

In general, it should be noted that any UAS broadcasts its position in a rate of two messages per second. This rate is determined by the max speed of the UAS which is assumed to be 5m/s. If a message is received after a maximum of half a second, then the position of the UAS will be outdated just by a maximum of 2.5 meters. This assumption is realistic in the average case of small-UAS, the speed limit set by the FAA is much higher though, at about 44m/s (100mph) (FAA, 2016). Further discussion on the limitations is done later in this chapter.

Two of the most important metrics in the performance testing of the network are *jitter* and *roundtrip* time. According to the 0MQ whitepapers, roundtrip is the time interval a message requires to be sent and received back. Jitter is "used to express how much do individual latencies tend to differ from the mean" (iMatix Corporation, 2014). The latencies that are going to be used in the jitter calculation will be the roundtrip time.

Another important factor to be taken into account is the size of the message. The actual message size, as analyzed by the WireShark network traffic analyzer, ranges from 885 to 1064 bytes, including data, UDP and IP overhead. The overhead includes source/destination addresses, flags and the specified protocol parameters. The overhead by sending broadcast UDP packets through IPv4 sockets, without checksum verification field (it is included in the data segment), is a constant of 42 bytes. Data can range from 848 to 1064 bytes depending the serialization of each packet and the fact that some values can also be null.

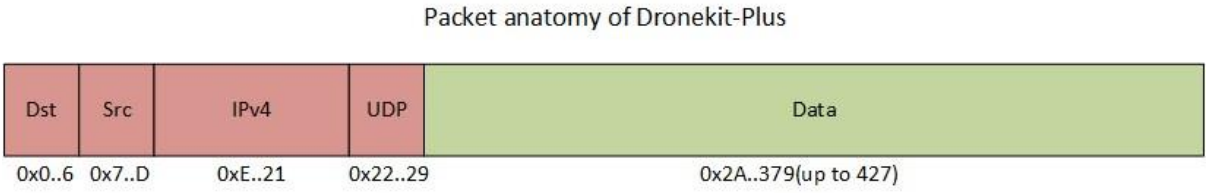


Figure 5.1 Packet anatomy of the UAS messages sent with UDP. Information obtained from the WireShark network traffic analyzer.

The data segment contains vehicle information which can be used in a collision avoidance protocol. More information on the data segment can be found in Appendix C: Information shared between UAS.

The sampling was performed inside the companion computers while running dronekit and the ABSAA system. This was done in order to fully simulate the latency, possible interrupts and available resources. There were two tests to evaluate the performance of message exchange.

The first case was the average case, during which a UAS receives ten messages per second. The purpose of the second case was to jam the network and the receiver by sending a hundred messages per second. The testing code was running for enough time to produce about 200000 samples.

The distance between the antennae was one meter but experiments have shown that data can be transmitted in a roughly 75-meter range, a limit much higher than the actual UAS 40-meter safety zone. An extra counter was operating in both sender/receiver systems in order to determine message loss due to collisions and the UDP non-lossless protocol.

The outcome of the testing was analyzed with the GNU Octave software. In order to have better visualization, data was divided into a hundred bins for charting the histogram. Since the information is packed mostly to the left of the chart (most messages have low roundtrip times) the chart was also presented in logarithmic scale.

Histograms can also easily give the necessary statistics for the analysis. A very useful one is the 99th percentile, which describes where the 99% of the sample belongs to, in terms of latency. It is important to know the latency of 99% of the packets, but it is also important to know the worst case, which fortunately belongs to the rest 1%. This percentage, when analyzed asymptotically, is a huge number for packet loss but with such a bounded system input (2 to 20 messages/s), a 1% packet loss can be tolerated.

Test case: Average (10msg/s)
An average case of 10msg/s rate in the communication channel is tested and evaluated. The results are as follow:

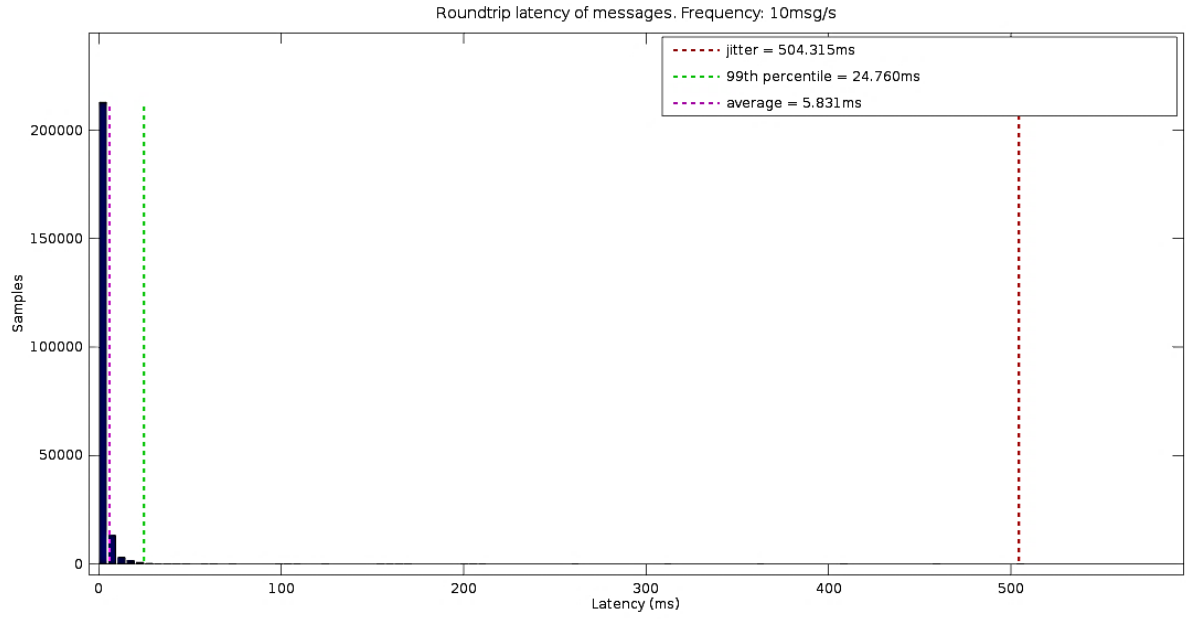


Figure 5.2 Histogram of the latencies of roundtrip messages with a ten message/s rate. As it can be seen, the average latency of these messages is about 5.8ms, most of the messages have a latency of about 24.7ms and the jitter is about a hundred times higher than the average case.

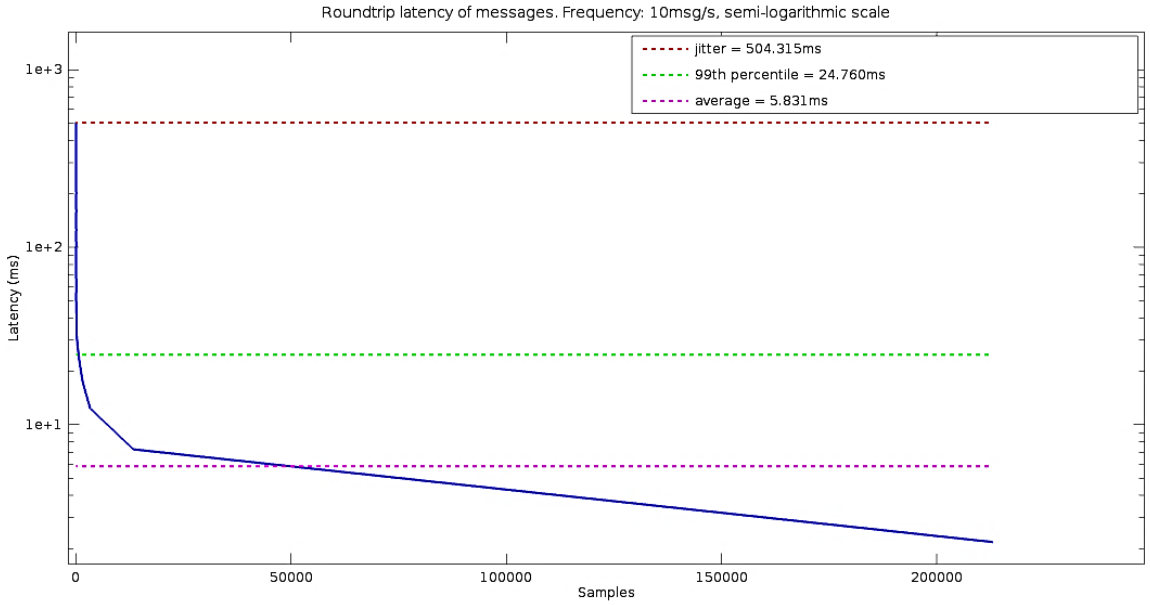


Figure 5.3 The same data as of Fig.5.2, this latency-samples diagram is in semi-logarithmic scale for better visualization. The constants of jitter, 99th percentile and average are also drawn for ease of inspection.

The message loss because of the communication channel was zero but it is interesting to know that the counter on the sender side instead of sending 25000 seconds * 10 messages/s = 250000 messages, it sent 234160 messages.

This is about a 6% of messages that were never sent because of the system. If this rate is evenly distributed during the session, then 0.6 messages are not sent every second. This value is tolerable for the system, taking into account the max airspeed of the UAS which is 5m/s. The loss of information translated to actual position error would then be a maximum of 2.5 metres, considering only the network, without taking into account other real-life parameters such as the GPS error, latency in the CPU-bound processes of the system's processor and latency in the autopilot hardware to take the actions commanded by the companion computer.

Finally, a 10-msg/s receive rate with a 2msg/s send rate of each individual UAS, means that the UAS could be able, on the network side, to handle up to at least five UAS. The jitter, which belongs to <1% of the total samples is also tolerable at 0.5s, since it equals the time window of the whole system.

Test case: Jamming (100msg/s)
The following scenario tries to jam the channel with a hundred messages/s rate. The results are very interesting in terms of latency and are shown in Figure 5.4 and Figure 5.5.

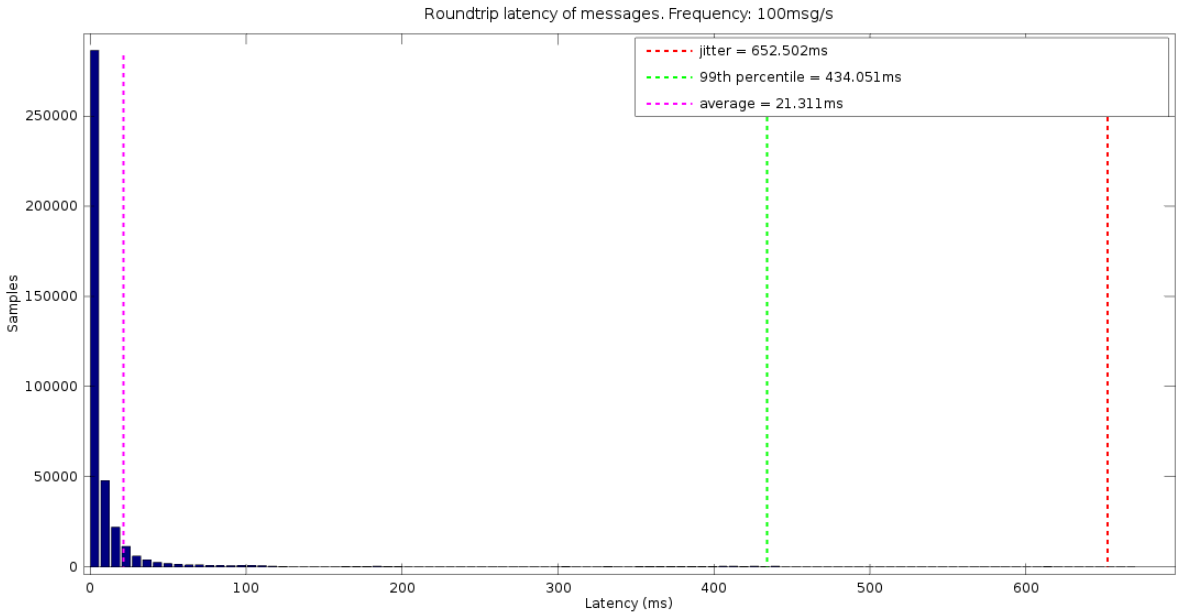


Figure 5.4 Histogram of the latencies of roundtrip messages with a hundred message/s rate. As it can be seen, the average latency of these messages is about 21.3ms, most of the messages have a latency of about 434ms and the jitter is about thirty times higher than the average case.

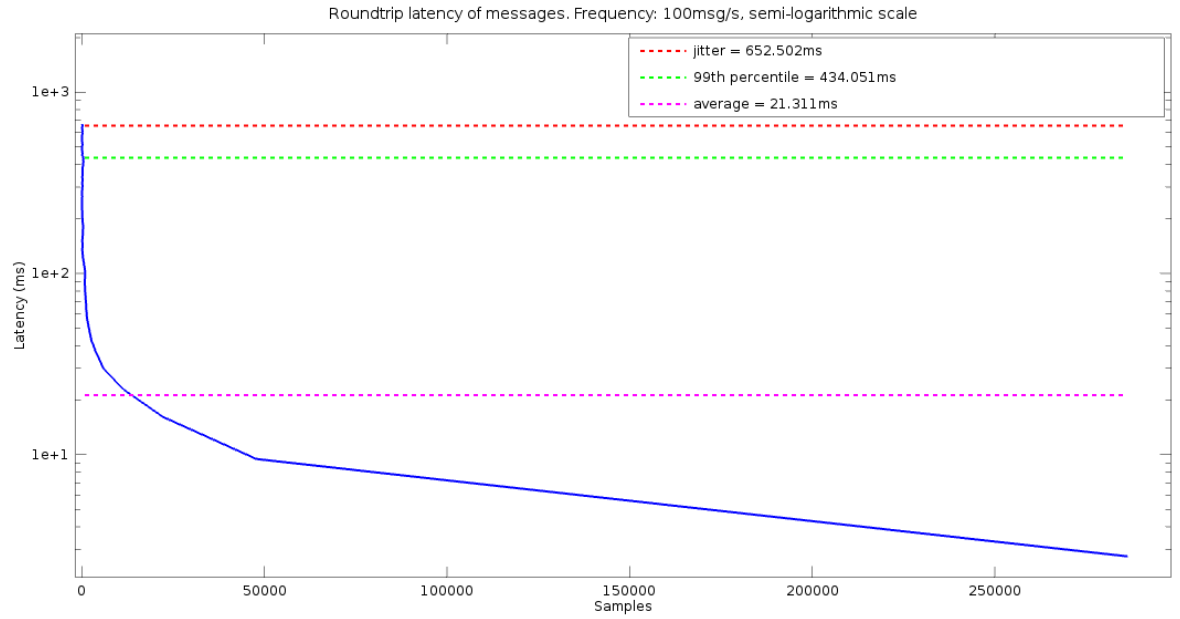


Figure 5.5 The same data as of Fig.5.4, this latency-samples diagram is in semi-logarithmic scale for better visualization. The constants of jitter, 99th percentile and average are also drawn for ease of inspection.

In the second test, jitter is not as high a multiple as in the first case, thus moving the average value way higher in the latency meter. The results should be expected: the receiver has ten times the workload in both I/O- and CPU-bound tasks and, combined with the overhead of the system, the latencies are much higher than this factor. Because of the average latency nearing the timing window, a message loss is also observed between the sender and the receiver of

about 9%. Out of all the messages that should be sent, a 13.5% was never sent, in opposition to the first case's 6%. This concludes that the receiver lost in total about 16% of the messages.

By performing the same statistical analysis as in the average test case, 16 messages out of 100 were lost every second. If this number is distributed to all participating UAS, then eight out of fifty UAS will not be able to share their information during their time slot. In the worst case, a UAS will not be able to share its own data eight consecutive times, meaning four seconds. If this time is translated to position error, with a max speed of 5m/s, the conflicting UAS will be outdated by 20 meters. If the safety zone is 40 meters, in a frontal collision case, the UAS will have already crashed without taking into account any other latency factors!

As it can be seen, the increase in latency from 10msg/s to 100msg/s is not linear but has an exponential increase. A further optimization analysis should be considered in order to define the acceptable limits of the system. It is quite sure though, that, inside a forty-meter sphere, the network channel allows for at least five UAS to participate safely in a conflict session.

5.2 Software in the loop (SITL)

Simulation of the UAS system in the development computer is called software-in-the-loop. The primary, coarse tests of the system functionality are undertaken in the protective environment of a host computer. The necessary simulation software is dronekit-sitl, which contains the binaries for the multi-copter vehicles, the physics engine and the simulation parameters.

Testing scenario

In order to check the functionality of the system, the developer's computer was used to simulate the vehicle and a Raspberry Pi was used as a beacon with hard-coded coordinates. For the visualization needs, APM Planner 2 was used. The home location of the vehicle, the flight parameters and the autopilot version should be defined beforehand. With the help of Google Earth Pro, the beacon's distance from the take-off simulated vehicle's coordinates (called the home location) was hard-coded to be about 23 meters away from safety zone of the simulated vehicle.

This scenario tries to test the whole functionality of the system. What is not tested is scaling (use for more than two stakeholders) and interaction between more than simulated vehicles, since APM planner cannot visualize more than one simulated vehicle instance. What is also not tested is conflict during a non-autonomous flight (manual modes). The prioritization algorithm as standalone has been evaluated with the use of ten dummy vehicles of different parameters and its functionality has been confirmed.

The simulated vehicle's mission was to perform a flight with 10-meter altitude, in a 50-meter square.

The simulated vehicle's mission was as follows:

1. Take off to a target altitude of ten meters
2. Go 50 meters north [First waypoint]
3. Go 50 meters west [Second waypoint]
4. Go 50 meters south [Third waypoint]
5. Go 50 meters east [Fourth waypoint]
6. Return to launch point

The position of the beacon was 63 meters north of the home location and the parameters of the beacon were crafted in a way that it has higher priority than the simulated vehicle (the easiest parameter for this is elevating the beacon's mission importance).

The expected behavior of the flying simulated vehicle was to:

- Start receiving packets from the beacon and ignore it at first. Cross-check with packets received by the beacon.
- When the distance between the beacon and the simulated vehicle would be less than the safety zone, the message should be processed and the console should print a warning message that another vehicle is approaching.
- When detecting the beacon inside the safety zone, the simulated vehicle should immediately reduce its airspeed to 0m/s, since its priority should be lower than the beacon's.
- The beacon would then be stopped and, after 5s which is the timeout of outdated UAS in the conflict zone list, then simulated vehicle should continue its mission.

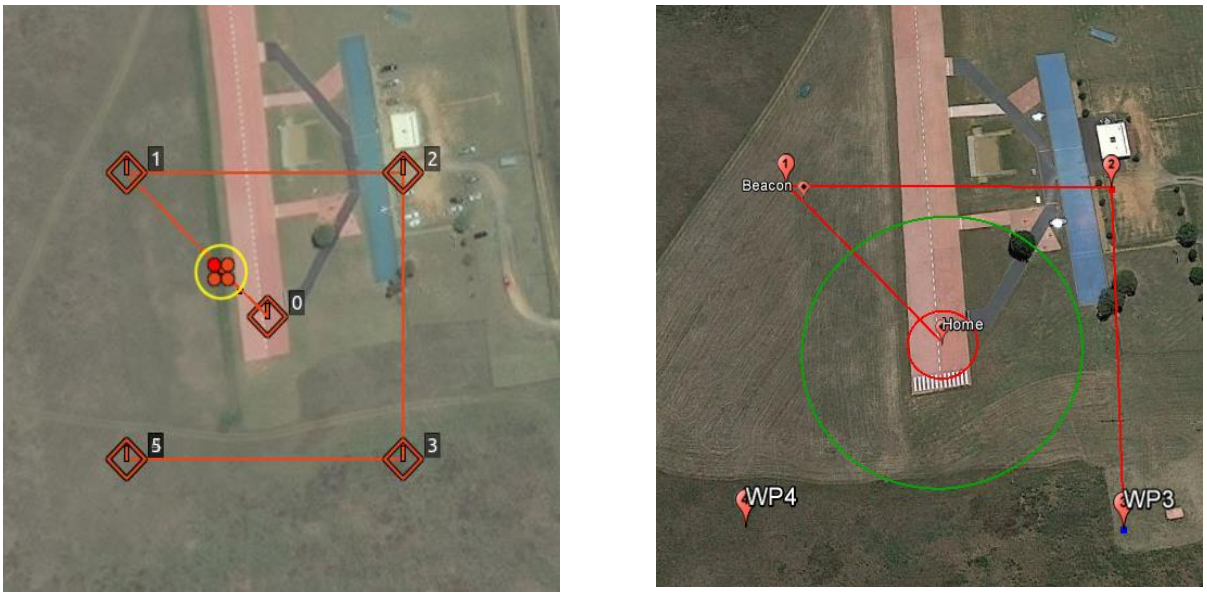


Figure 5.6 The mission waypoints loaded in the simulated vehicle as shown in APM planner (left). The yellow circle with the four dots represent the quadcopter's current position. On the right, the mission and beacon position as shown in Google Earth. The green circle represents the safety zone of the UAS and the red one the critical zone. The beacon is in the way towards waypoint 1.

Environment variables

In order to overcome some limitations of the system, for testing purposes, a slight simplification of the model was configured. The greatest limitation was the power source since the simulation could be running for many minutes as such, the simulation did not take into account the power consumption.

Another feature which was ignored was airspeed. Strong wind would require the autopilot to perform many corrections for stabilization and that would jam the graphs with irrelevant data.

Another important factor was the GPS error. Some autopilot modes require a fully functional GPS, with 3D fix and low HDOP and VDOP. For this reason, the GPS of the simulated vehicle had very accurate GPS measurements.

For fully functional autopilot operation, the barometer is also of great importance and it is usually a sensor that does produce erroneous measurements. In order to verify the actual functionality of the system, the barometer was programmed to deliver accurate results.

Besides the above modifications to create ideal flight conditions, the necessary initialization and pre-arm checks were performed. The code-style should be defensive in order to avoid hazards during a bad boot or faulty avionics.

Observations

The APM Planner visualization pane optically verified the functionality of the system. The console messages also provided ample information in real-time and the vehicle's behavior has been confirmed.

In order to observe the (negative) acceleration of the vehicle during a conflict, data from the flight logs have been analyzed. In more detail, a correlation between the flight mode and the groundspeed has been visualized and the time variable was extracted. Four stopping methods have been tested and the results are as follow:

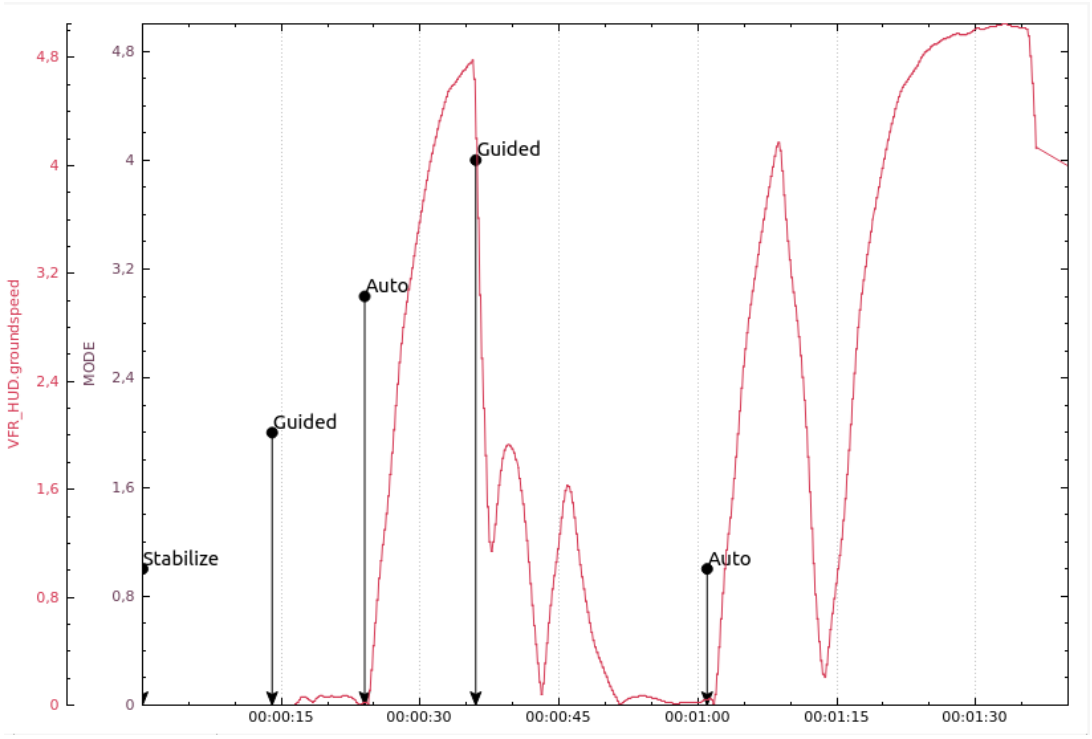


Figure 5.7 UAS follows an autonomous mission (AUTO). When the higher priority beacon is in range it changes to GUIDED mode, saves the mission and holds its position until further commands are sent to the autopilot. Groundspeed stabilizes to zero after almost ten seconds and the distance covered during this period is more than 25 meters.

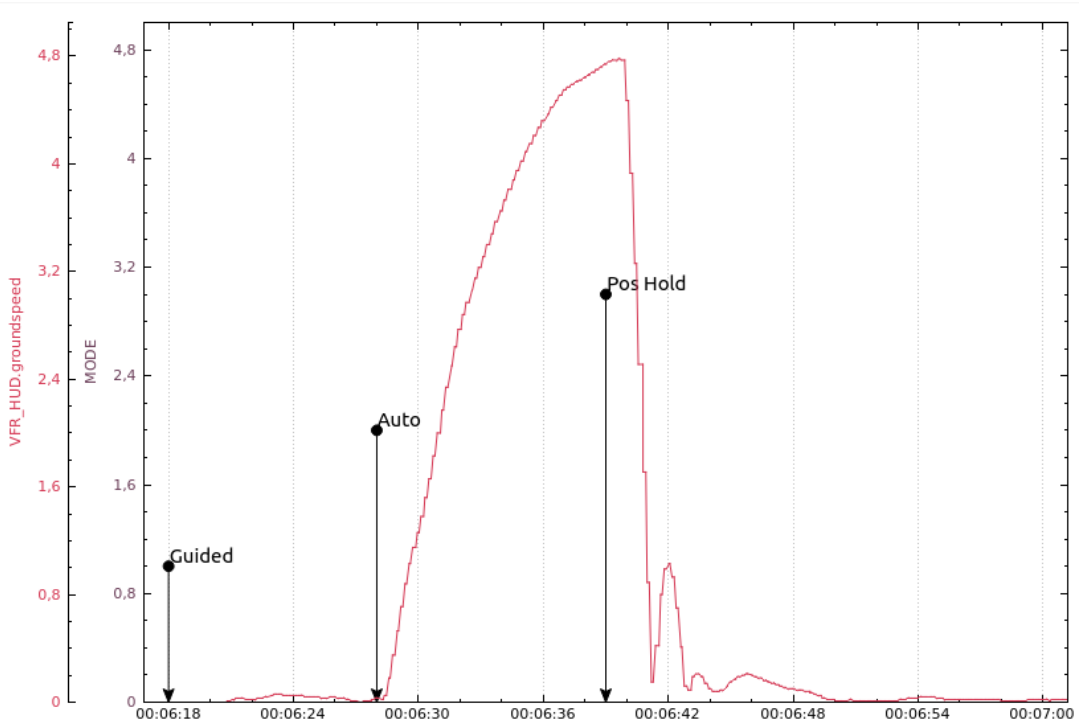


Figure 5.8 POSHOLD contains a settable braking rate parameter which is particularly useful when experimenting with more aggressive braking. The parameter was increased from 8 degrees/s to 20 degrees/s. POSHOLD is a manual mode though, requiring input from the ground pilot. If not received within three seconds, the autopilot protocol demands the UAS to be in failsafe state. For this reason, the RC throttle channel is overridden to keep the UAS in the air. Further modifications of the throttle for stability reasons are undergone by the autopilot. The groundspeed momentum lasted for 8 seconds with significantly lower curves and the distance covered was 14 meters.

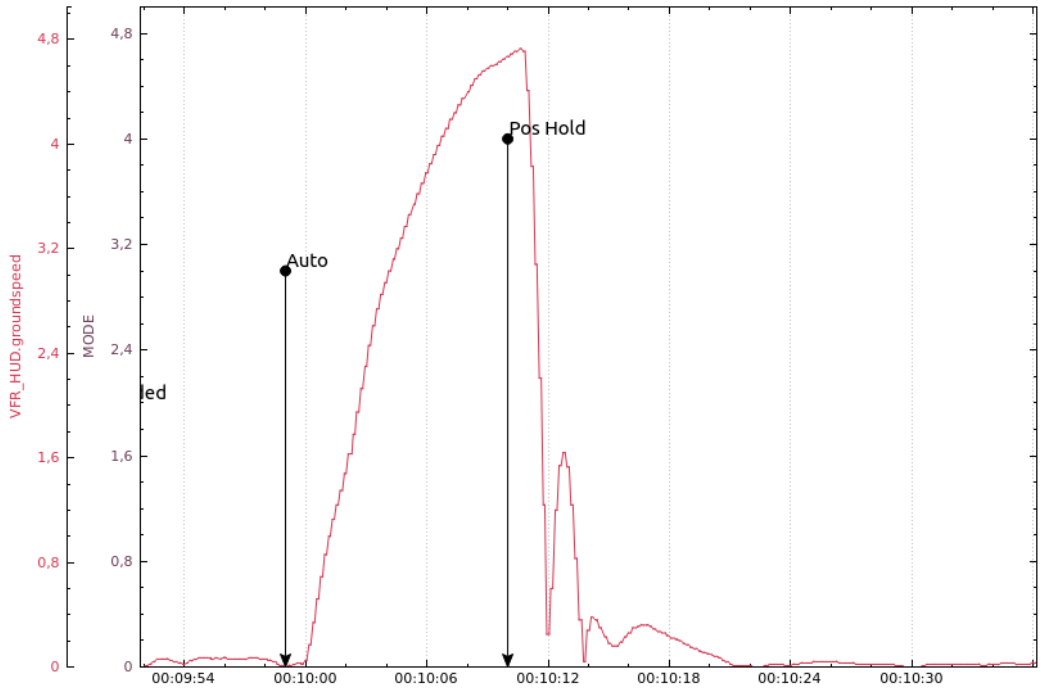


Figure 5.9 The only parameter changed in this solution is POSHOLD_BRAKE_RATE which was increased by ten degrees/s. A 30degrees/s rate in braking is the limit since an increase could result in aggressive behaviour of the system. The results are much better compared to all the solutions presented. Time until zero groundspeed: 10 seconds. Very low groundspeed curves result in a ten-meter distance until zero.

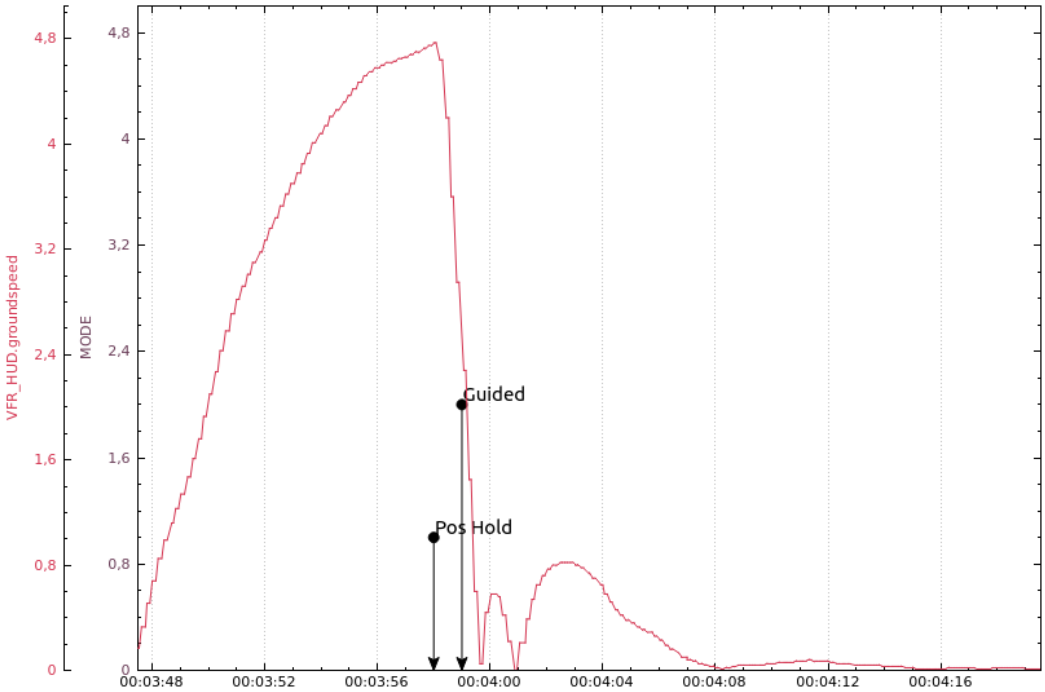


Figure 5.10 After the beacon surpassing the safety zone of the UAS, POSHOLD mode is activated followed by GUIDED mode. This hybrid solution does not need an RC channel override and exploits the POSHOLD_BRAKE_RATE parameter. Latency here is not reduced compared to the GUIDED-only solution but the groundspeed curves are a lot lower and smoother. Distance covered from conflict detection to actual zero groundspeed is 20 meters.

5.3 Hardware in the loop (HIL)

Performing a simulation with the participating hardware connected gives a more realistic feedback for benchmarking, environmental factors to take into account and the physical hardware limitations. The quality of the system is also increased since during the design and implementation process there are details in hardware limitations that demand fine-tuning of the system (Kleijn, n.d.). HIL simulation was used for the 5.1 Networking analysis chapter, too. This testing step is also a preparatory step for the final field testing procedure (See 5.4 Field testing).

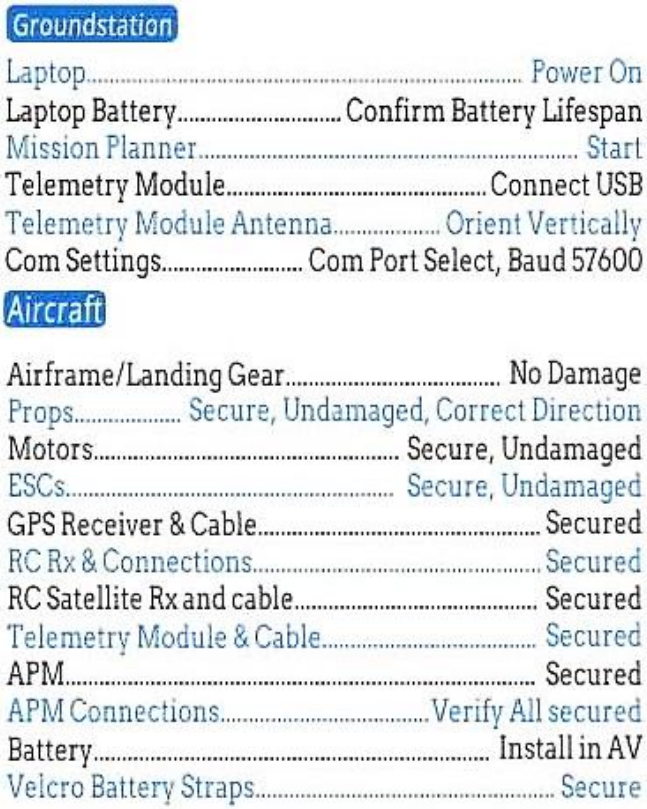
Testing scenario

The foam quadcopter was fully assembled and booted indoors. The propellers were removed for safety reasons. The PhoneDrone was used as a beacon, but also in full assembly and with hard-coded coordinates in a way that it is in the vicinity of the foam quadcopter. A laptop was used as a Ground Control Station (GCS) for uploading the necessary files and observing system changes on the remote companion computer consoles. The scenario contains the following actions:

1. Boot the two UAS.
2. Perform the pre-flight checklist test
3. Connect to the UAS ad-hoc network with a laptop for access.
4. Upload the necessary files to both systems.

5. Start the beacon program.
6. Start the autonomous mission on the foam UAS.
7. Observe the UAS reactions.
8. Stop the beacon.
9. Observe the UAS reactions again.
10. Exit the foam UAS mission.

Table 2 APM-Copter Flight Ops Checklist v1.0 (ArduPilot Dev Team, 2016). The telemetry unit was not mounted since the necessary information would be exchanged via WLAN connection. The propellers were also not on-board because of the indoor testing.



Observations – Fine tuning

Based on the logging and the behavior of the foam UAS, the following observations are stated. Any issues that prevented the accomplishment of the scenario are addressed and stated as well:

- Some features of Dronekit were not supported by older firmware versions of the autopilot and as such the mission was not able to start. Some lines in the code are changed for compatibility reasons. The functionality is kept the same. The modifications are not stated since they are minor bug fixes.
- The GPS was not able to have an HDOP of under two meters and sometimes it could not even get a 3D fix. The UAS was relocated or else the pre-arm checks would not be accomplished.

- There was a significant latency during the on-the-fly mode change. Some while-loops that were used for defensive coding were locking the thread for longer-than-expected sessions so they were put to sleep at a 200ms rate. A longer rate would cause bigger delay in the execution of the collision avoidance thread which sleeps every 500ms anyway.
- The WLAN protocol has an average range of 75 meters without physical barriers.
- Some flight modes are not supported so cases are added for backwards compatibility.
- The autonomy of a fully charged 11.1V, 2200mAh battery is approximately 15 minutes of flight time.

5.4 Field testing

Having addressed the issues incurred during the SITL and HIL simulations, the final field testing was performed. The mission was fully autonomous so a remote human pilot was not needed. For safety reasons though a pilot was ready for a UAS takeover in case of emergency. Because of limitations in resources, no more than two UAS were able to interact with each other. The biggest issues during the testing were weather conditions, battery autonomy and to find a suitable takeoff platform.

In order to increase the autonomy of the battery, a shorter flight plan was uploaded to the UAS that served the same testing purpose. A set of fully charged spare batteries were also taken along.

The testing area would need to be open, apart from buildings and not crowded, for safety reasons. It should also be easily accessed and close to the laboratory. The most suitable area was the farmost parking area of the TUC ECE School. The mission plan is presented in the testing area in Figure 5.12.

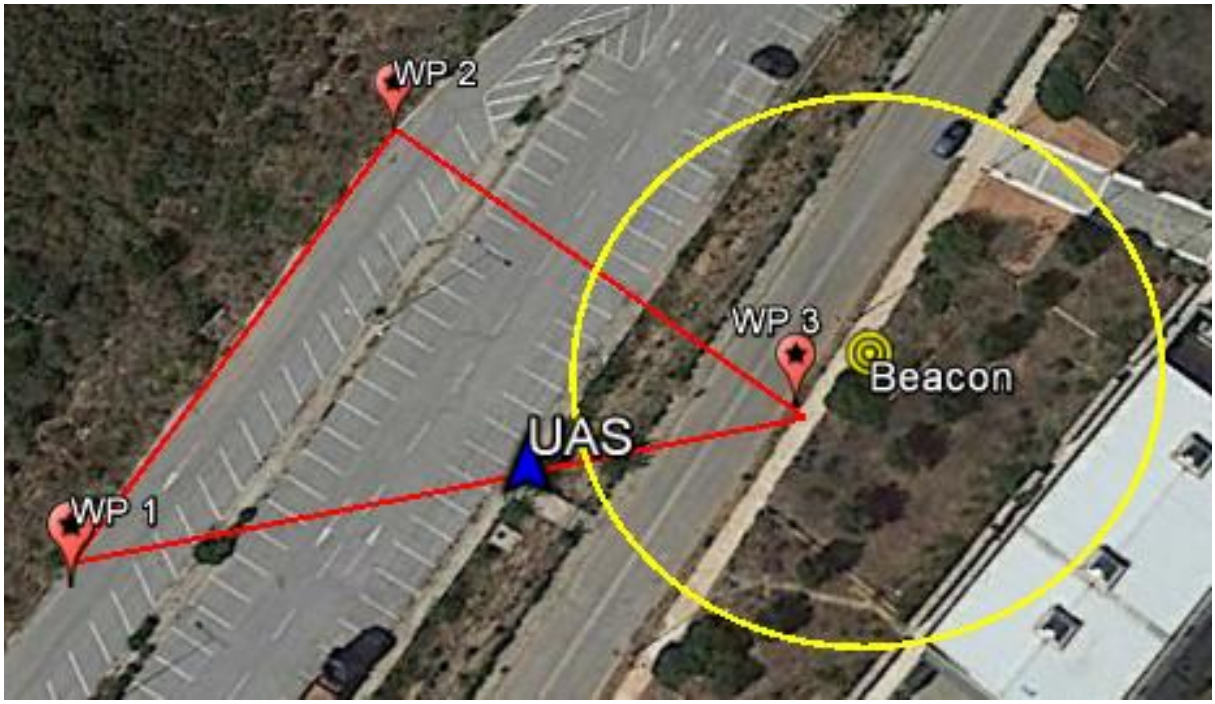


Figure 5.11 The field testing mission plan. The UAS will fly towards WP1, WP2 and WP3 and it should stop at the first yellow-red line intersection. After the beacon stops sending messages, the UAS should continue its mission.

The first tests were performed with the foam UAS flying and the higher-priority PhoneDrone on the ground. The wind speed was too high for the aerodynamics of the foam UAS though (approximately 20-25km/h with gusts), and the system went in emergency status.

For the next tests the flying-beacon roles were changed since the PhoneDrone is less susceptible to wind speed because of its shape and density. Weather conditions were also more thoroughly checked. This resulted in delays because of bad weather conditions (rain, high wind speed). The test was finally a success and the ‘braking’ techniques of 5.2 Software in the loop (SITL) were tested in order to see which one is most favourable for the real system.

6. Discussion and future work

This final chapter is devoted to the presentation of the advantages and limitations of the system. Possible future development ideas are also stated.

6.1 Advantages of the system

Open-source

The system designed in this thesis provides an extra communication and conflict-related interface for an open source UAS developer platform, Dronekit. Dronekit is used by a great proportion of UAS developers who want to experiment with advanced control methods and find industry-related applications. Being part of an industry-standard developer ecosystem helps in evolving the open UAS platform community and gives a boost to create more advanced techniques on a more versatile and robust basis. The designed system can also be reviewed by anyone in the community for further improvement and robustness. Since the deployment of the DK+ API can still be tricky since it is a prototype, the experienced community can transform it to a much more fault-tolerant and robust system.

Minimum hardware and cost

The architecture of the system has followed a minimum hardware approach. It must be expected that a UAS functioning fully autonomously would require extra processing power, other than the autopilot microcontroller. Since the software has to be run on a companion computer with Linux OS because of the Dronekit requirements, the target was to limit any other hardware requirements to zero. In most cases, an external WLAN module has to be connected, like in the RPi 2 or Odroid, but sometimes the computer has embedded WLAN support, like in the RPi 3.

The most important factor that limits hardware requirements is the cooperative nature of the UAS and the fact that the cooperation is de-centralized, or peer-to-peer. That means that no extra sensors are needed between the UAS to sense another UAS in the region and also that no ground stations, radars or antennae are needed for the conflict detection and avoidance.

Combining the global use of open source software and the minimum hardware requirements for advanced control in a UAS platform, a ‘cheap’ solution to airborne-based sense and avoid has been implemented. The system introduces low costs in extra hardware installation and interfacing since it is plug-and-play. It is also energy efficient since no extra sensors, gimbal or image processing is needed. The software itself is lightweight in CPU-bound operations with a complexity of O(n) while traversing lists. N can be a maximum of ten conflicting UAS, since extra scaling will be most likely to decrease the system reliability dramatically in the receiver section. The arithmetic calculations are also elementary algebra, which also reduces the complexity. In addition, the payload is not increased relatively a lot, since a mini companion computer, along with an external WLAN module and casing, weighs no more than a hundred grammars, less than ten percent of the average payload a small/micro-scale UAS can hold. The biggest energy consumers are still the motors.

Portability
The minimum hardware approach and the open-source culture of the system make the system portable. Any small-scale computer running a Linux distribution and having WLAN support by any means, can have the Dronekit and DK+ installed for ABSAA support. The software requirements are also easy to meet since Python 2.x is installed in almost any Linux distribution and the extra dependencies can be installed via pip.

Easy prototyping
The Python language was also chosen because it makes prototyping easier than the use of C++ or Java. Python is a very high-level language where data types are abstract and there are libraries for thread-safe operations that minimize the lines of code that need to be written, in opposition to a C++ code where debugging and writing would take a lot more time. There are serious tradeoffs in optimization, though which will be analyzed in the Limitations section. If a truly OO language was to be chosen, C++ would be the preference instead of Java. Java has the same optimization problems with Python while C++ gives the freedom to fine-tune and go as low-level as someone wants. If someone wants to stitch C++ code though, Python also supports this feature.

6.2 Limitations of the system

Susceptibility to the physical environment
The UAS alone have the limitation that they cannot fly under any weather condition and cannot be autonomous in any place. Depending the airframe, high wind speed, humidity and extreme temperatures are usually no-fly situations. The avionics necessary for a fully autonomous flight necessarily contain GPS readings. An open sky and at least four satellites are needed by default before the arming of the UAS, while position error needs to have a low threshold of approximately two meters. Since the UAS is susceptible to these uncontrollable conditions, the system developed inherits this disadvantage as well.

Not optimized
While the Python language is very good for prototyping, there is a great tradeoff that must be mentioned, the one of low optimization. Problems like memory garbage and the use of large memory space for the abstract data types, running always in a single core due to the Global Interpreter Lock (GIL) even in multi-threading programs, automatic scheduling and big message overhead in the OS kernel cannot be addressed by the Python programmer because of the language's high level of abstraction. What is more, whatever the implementation and the granularity of the python software, the python interpreter can run only in one core, keeping the remaining cores of the companion computer unexploited. (Coghlan, 2011), (Beazley, 2010)

The only way to overcome this major problem is the multi-processing library that is currently being used in a separate test branch. The problem with multi-processing software is high message overhead due to inter-process communication, since the use of shared memory is not functional as it is between threads (Danilov, 2014). Testing has shown that latency in a single-core system is at acceptable rates, but with the explicit introduction of latencies, ranging from

100ms to 500ms, throughout the program, the system stops being real time. A possible multi-core solution maybe would address this issue.

Low scaling

Scaling in the context of how many conflicting UAS can use the software in the same time and place. The most important factors limiting the conflict detection and avoidance ability of the system is the actual number of conflicting UAS inside the safety zone. The greatest bottleneck is on the receiver side, where message collisions can cause great data loss and latency. Another very important aspect is the range of the WLAN protocol which limits the maximum groundspeed of the conflicting UAS to a very low threshold.

Networking tests of chapter 5.1 Networking analysis have shown that five UAS can safely communicate with each other while fifty UAS will introduce great latency to the system and a big data loss rate. It is intuitively believed that ten UAS will also be a safe limit for the receiver subsystem.

Also, since the IEEE 802.11 protocol has a range of maximum 90 meters in the open sky (SpeedGuide, 2016), evaluated by lab testing, the safety zone can be of maximum 90 meters as well. If both UAS need 1s for processing and moving with the opposite direction, leaving a five-meter critical distance, it means that the UAS can stop at a maximum distance of 40 meters after the conflict detection.

The braking rates of chapter Validation and performance evaluation show that on average case it takes 2s from 5m/s to 0m/s, yielding a rough estimate of 2.5m/s^2 brake rate. By solving the following simple kinematics formula (The Physics Classroom, 2016):

$$d = \frac{1}{2} a * t^2 \quad (1)$$

Where:

d = 40m distance,
a = 2.5m/s^2 deceleration.

Solving by the time variant, t, we have:

$$t = 5.66s$$

Replacing to the velocity formula (The Physics Classroom, 2016):

$$V = V_0 + a * t \qquad (2)$$

Where:

V = 0m/s target velocity
t = time needed to cover the 40-meter distance, found through eq.1

We finally have:

$$\xrightarrow{(1),(2)} V = 14.15 \frac{m}{s}$$

This means that the maximum speed should be approximately 14m/s, opposed to the NAS legal maximum speed of 44m/s. This is a great limitation for the system. Any UAS with maximum groundspeed greater than the 14m/s limit will cause the system to fail for the other conflicting UAS as well.

A solution to this problem would be the use of other communication protocols, other than the IEEE 802.11, that have higher range. These protocols would have to be benchmarked again to see the new limitations on the receiver side that they impose and the extra hardware costs and interfacing required.

Non-dynamic for different airframes

The system needs to be adaptive to the airframe of each UAS for fine tuning some important parameters, the most important of which are:

- Autopilot version
  The version of the autopilot can sometimes determine the abilities the UAS has in terms of available flight modes and MAVLink commands. Since not every MAVLink command and flight mode is supported by all autopilot hardware and firmware, different procedures need to take place for the same purpose. For example, the ArduCopter 3.3 supports the BRAKE flight mode which automatically stops the UAS at the current position. Older versions don't have this kind of support so different methods need to be applied, like the ones presented in Validation and performance evaluation.
- Maximum groundspeed
  As mentioned before, if a UAS can achieve a maximum groundspeed of more than 14m/s it should take other measures for conflict detection and avoidance since it imposes serious risks for the rest of the UAS, if communication is achieved via the wireless internet protocol.
- Braking rate
  The braking rate calculated in this thesis via testing (see 5.2 Software in the loop (SITL)) is applied only to the certain airframe tested. Each UAS has got different avionics, motors and aerodynamics and as such different braking rate.

The safety zone of each UAS is currently hard-coded in the system and as such, it does not take into account these parameters. This means that the safety zone distance is not optimized for each UAS individually thus making the conflicting set of UAS more rigid.

In conclusion, the system developed has opened a new field, the cooperative airborne-based sense and avoid method, in the Dronekit ecosystem. The system requires little, cheap and easy hardware additions to an existing open-platform UAS and contains all the necessary documentation to get someone started. On the other hand, there is still a lot of work to be done in order to make it more resilient in the dynamic environment of the UAS industry and more robust to the already developed software. Currently, it is safe to assume that the system works as it should for UAS with maximum groundspeed of less than 14m/s and for five to ten concurrently conflicting UAS.

Not secure
Since the software is open-source, any developer can download and modify it to his/her likings. This imposes a high risk in hacking the UAS in order to be seen as a top-priority UAS (see Prioritization) even if it is for recreational use. Also, the connection is not secure. This implies that in a future mission takeover feature, it will be feasible for other UAS to hijack their surrounding ones.

6.3 Future work
As inspired from the disadvantages of the system, there are great ways for further development of the system for extra robustness and functionality. The following sections mention the most important ones.

Real-time
There is a Linux kernel patch, RT_PREEMPT (The Linux Foundation, 2016), that promises to transform the system into a real-time one. This is done by inserting timeouts for signals interrupting a high-priority process. The deployment of the patch is tricky to the average user and should come pre-installed in order to be widely used. For benchmarking purposes, though, the patch could be installed by a developer and see if the timing differences are high enough to start the procedure of making it available to all the users.

Optimization
Currently there are two big optimization issues. The first is the communications interface and the interconnection of UAS as nodes in a dynamic network. There has to be serious research on dynamic ad-hoc networks and how they can be applied in the UAS case. The problem with the internet protocol communication is the interfacing since any time the UAS can be both routers (give IP addresses to the UAS needed to connect in the subnet) and members (be able to join other subnets). In order to avoid this rather complicated issue, maybe other communication methods should be researched. The decision on the communication protocol has to take into account the tradeoffs between range and bandwidth or cost in resources (installation, money, implementation time).

If the WLAN protocol seems an acceptable solution, further optimization can be achieved by using some network-specific packages that are network-related, like 0MQ (iMatix Corporation, 2016) and Twisted (Twisted Matrix Labs, 2016).

The second optimization issue is the programming language which does not allow low-level programming, thus limiting the source code efficiency. Porting the software to C++ and embedding it to the Dronekit core could be a possible solution to this but again the debugging/implementation time tradeoff has to be taken into account. The transformation of the system to multi-processing instead of multi-threading, in the Python language, could also be very useful in the full exploitation of the resources of a multi-core companion computer, but this again could mean the change in the whole architecture of the system.

Resilience

As stated in the Limitations section of this chapter, the system does not take into account airframe-specific parameters to set its safety distance and the braking methods. A dynamic distance window and extra support for more autopilot versions would make the system more adaptive to the individual UAS.

Also, better precision in the distance between UAS can be achieved by considering the current coordinates for error correction in the distance calculation function.

The 'Drone Language'

A very useful concept that is based on the cooperative nature of the system and the 'context-switching' (see UAS context switching) is to create a messaging interface with a set of supporting commands for UAS to be able to remotely control other UAS. The application of this concept would be, for example, a mission takeover in case of an emergency: A UAS which is registered as a volunteer receives a message from a nearby top-priority, search-and-rescue UAS, that it needs help in scouting. Then the volunteer UAS saves its state and follows the governmental UAS to the mission.

Collision avoidance methods

Currently, the collision avoidance protocol used is pretty naïve and it does not consider much of the information that is shared between the UAS. This is because the purpose of the thesis was to create the infrastructure for the support of such algorithms and not research the algorithms themselves. The deployment of Artificial Intelligence algorithms, like the A* path planning, with the aid of trajectory planning techniques can create very powerful collision avoidance protocols. The time and space complexity are of course increased dramatically and further optimization is needed in order for them to be functional.

References

3D Robotics, 2016. *Welcome to Dronekit-Python's Documentation!*. [Online] Available at: http://python.dronekit.io/ [Accessed 16 September 2016].

3DR, 2015. *Dronekit.* [Online] Available at: dronekit.io [Accessed 15 September 2016].

Anand, S., 2007. Domestic use of unmanned aircraft systems: An evaluation of policy constraints and the role of industry consensus standards. *Standardization News*, 35(9).

Angelov, P., 2012. *Sense and Avoid in UAS.* 1st ed. s.l.:John Wiley & Sons.

Anon., 2013. Unmanned Aircraft System Airspace Integration in the National Airspace Using a Ground-Based Sense-and-Avoid System. *Johns Hopkins APL Technical Digest*, p. 573.

Antoniou, L., 2016. *DK+*. [Online] Available at: https://leonidasantoniou.github.io/dk-plus/ [Accessed 10 September 2016].

Antoniou, L., 2016. *dk-plus.* [Online] Available at: https://github.com/LeonidasAntoniou/dk-plus [Accessed 9 September 2016].

Antoniou, L., 2016. *dk-plus/License*. [Online] Available at: https://github.com/LeonidasAntoniou/dk-plus/blob/master/LICENSE [Accessed 22 September 2016].

ArduPilot Dev Team, 2016. *APM 2.5 and 2.6 Overview*. [Online] Available at: http://ardupilot.org/copter/docs/common-apm25-and-26-overview.html [Accessed 16 October 2016].

ArduPilot Dev Team, 2016. *ArduPilot.* [Online] Available at: http://ardupilot.com/ardupilot/index.html [Accessed 16 September 2016].

ArduPilot Dev Team, 2016. *Flight Modes.* [Online] Available at: http://ardupilot.org/copter/docs/flight-modes.html [Accessed 25 September 2016].

ArduPilot Dev Team, 2016. *Pre-Flight Checklist (Copter).* [Online] Available at: http://ardupilot.org/copter/docs/checklist.html [Accessed 17 October 2016].

ArduPilot, 2016. *APM Planner 2.* [Online] Available at: http://copter.ardupilot.com/planner2/index.html [Accessed 16 September 2016].

ArduPilot, 2016. *Learning ArduPilot - Introduction.* [Online] Available at: http://ardupilot.org/dev/docs/learning-ardupilot-introduction.html [Accessed 16 September 2016].

Atkins, E. M., n.d. *Certifiable Autonomous Flight Management for Unmanned Aircraft Systems,* Ann Arbor, MI: University of Michigan.

Atmel, 2014. *Atmel ATmega640/V-1280/V-1281/V-2560/V-2561/V.* [Online] Available at: http://www.atmel.com/Images/Atmel-2549-8-bit-AVR-Microcontroller-ATmega640-1280-1281-2560-2561_datasheet.pdf
[Accessed 16 October 2016].

B., S., B., S. & Y., C., 2013. *ADS-B for small Unmanned Aerial Systems: Case study and regulatory practices.* Atlanta, GA, International Conference on Unmanned Aircraft Systems, ICUAS.

Beazley, D., 2010. *Understanding the Python GIL.* [Online] Available at: http://www.dabeaz.com/python/UnderstandingGIL.pdf
[Accessed 2 November 2016].

ByteParadigm, 2016. *Introduction to I2C and SPI protocols.* [Online] Available at: http://www.byteparadigm.com/applications/introduction-to-i2c-and-spi-protocols/
[Accessed 20 September 2016].

Chao, H., Cao, Y. & Chen, Y., 2010. Autopilots for small unmanned aerial vehicles: A survey. *International Journal of Control, Automation and Systems,* 8(1), pp. 36-44.

Coghlan, N., 2011. *Efficiently Exploiting Multiple Cores with Python.* [Online] Available at: http://python-notes.curiousefficiency.org/en/latest/python3/multicore_python.html
[Accessed 2 November 2016].

Commision, E., 2014. *Opening the aviation market to the civil use of remotely piloted aircraft systems in a safe and sustainable manner,* Brussels: Communication from the Commision to the European Parliament and the Council.

Crane, D., 2012. *Datron Scout Air Reconnaissance System (ARS).* [Online] Available at: http://www.defensereview.com/datron-scout-air-reconnaissance-system-ars-lightweight-quad-rotor-vtol-micro-air-vehicle-mav-for-tactical-surveillance-missions-video/
[Accessed 9 September 2016].

Dalamagkidis, K., Valavanis, K. & Piegl, L., 2008. On unmanned aircraft systems issues, challenges and operational restrictions preventing integration into the National Airspace System. *Progress in Aerospace Sciences,* 44(7-8), pp. 503-519.

Danilov, K., 2014. *Edge of the Stack: Improve Performance of Python Programs by Restricting Them to a Single CPU.* [Online] Available at: https://www.mirantis.com/blog/improve-performance-python-programs-restricting-single-cpu/
[Accessed 2 November 2016].

Defense Science Board (DSB) Task Force, 2007. *Future Need for VTOL/STOL Aircraft,* Washington: DSB.

DJI, 2016. *DJI SDK - Develop solutions for different industries upon your needs.* [Online] Available at: https://developer.dji.com/ [Accessed 12 October 2016].

DJI, n.d. *Phantom-4.* [Online] Available at: http://www.dji.com/phantom-4 [Accessed 9 September 2016].

Drone Industry Insights, 2015. *Products.* [Online] Available at: https://www.droneii.com/products [Accessed 15 September 2016].

Drone Industry Insights, 2016. *The Drone Market Environment Map 2016.* [Online] Available at: https://www.droneii.com/drone-market-environment-map-2016 [Accessed 16 September 2016].

Dronecode, 2016. *GitHub - Dronecode_Repository-List.* [Online] Available at: https://github.com/Dronecode/_Repository-List [Accessed 16 September 2016].

dronekit, 2016. *dronekit-sitl.* [Online] Available at: https://github.com/dronekit/dronekit-sitl [Accessed 16 September 2016].

Ducard, G. G. H., 2008. Efficient nonlinear actuator fault detection and isolation system for unmanned aerial vehicles. *Journal of Guidance, Control and Dynamics,* 31(1), pp. 225-237.

FAA, 2016. *Federal Register: Operation and Certification of Small Unmanned Aircraft Systems.* [Online] Available at: https://www.federalregister.gov/documents/2016/06/28/2016-15079/operation-and-certification-of-small-unmanned-aircraft-systems [Accessed 9 September 2016].

FAA, 2016. *NextGen Programs.* [Online] Available at: https://www.faa.gov/nextgen/programs/ [Accessed 9 September 2016].

FAA, 2016. *SUMMARY OF SMALL UNMANNED AIRCRAFT RULE (PART 107).* [Online] Available at: https://www.faa.gov/uas/media/Part_107_Summary.pdf [Accessed 9 September 2016].

Fasano, G. et al., 2008. Multi-Sensor-Based Fully Autonomous Non-Cooperative. *Journal of Aerospace Computing, Information and Communication,* Volume 5.

Finn, R. & Wright, D., 2012. Unmanned aircraft systems: Surveillance, ethics and privacy in civil applications. *Computer Law and Security Review,* 28(2), pp. 184-194.

Fox, D., Burgard, W. & Thrun, S., 1997. The Dynamic Windows Approach to Collision Avoidance. *IEEE Robotics & Automation Magazine,* p. 24.

Frazzoli, E., Dahleh, M. A. & Feron, E., 2002. Real-Time Motion Planning for Agile Autonomous Vehicles. *Journal of Guidance, Control and Dynamics,* 25(1), pp. 116-129.

Garcia, G. & Keshmiri, S., 2013. Adaptive and resilient flight control system for a small unmanned aerial system. *International Journal of Aerospace Engineering*, Volume Article Number: 289357.

Georg Brandl, S. T., 2016. *Sphinx.* [Online] Available at: http://www.sphinx-doc.org/en/stable/index.html [Accessed 10 September 2016].

Gibbs, Y., 2009. *Proteus.* [Online] Available at: http://www.nasa.gov/centers/dryden/history/pastprojects/Erast/proteus.html [Accessed 9 September 2016].

Gundlach, J., 2012. *Designing Unmanned Aircraft Systems: A Comprehensive Approach.* 1st ed. Manassas, Virginia: AIAA EDUCATION SERIES.

HobbyKing, 2015. *Extra Large EPP Quadcopter Frame.* [Online] Available at: http://www.hobbyking.com/hobbyking/store/__25420__Extra_Large_EPP_Quadcopter_Frame_450mm_835mm_total_width_.html [Accessed 16 October 2016].

Humphrey, I., 1997. *Inertial navigation system for a micro unmanned air vehicle.* Canton, MA, Guidance, Navigation, and Control Conference.

Hutchings, T., Jeffryes, S. & Farmer, S., 2007. *Architecting UAS sense & avoid systems.* s.l., IEEE.

ICAO, 2011. *Unmanned Aircraft Systems (UAS),* Quebec: International Civil Aviation Organization.

iMatix Corporation, 2014. *Measuring Jitter.* [Online] Available at: http://zeromq.org/whitepapers:measuring-jitter [Accessed 12 October 2016].

iMatix Corporation, 2016. *ZeroMQ.* [Online] Available at: http://zeromq.org/community [Accessed 1 November 2016].

International Civil Aviation Organization, 2011. *Unmanned Aircraft Systems (UAS),* Quebec: s.n.

Jill, B., 2015. *NextGen and UAS.* [Online] Available at: http://www.droningonandon.com/blog/nextgen-and-uas [Accessed 14 September 2016].

JSP, 2016. *EPP.* [Online] Available at: http://www.epp.com/ [Accessed 16 October 2016].

Kendoul, F., Lara, D., Fantoni-Coichot, I. & Lozano, R., 2007. Real-time nonlinear embedded control for an autonomous quadrotor helicopter. *Journal of Guidance, Control and Dynamics,* 30(4), pp. 1049-1061.

Khatib, O., 1986. Real-Time Obstacle Avoidance for Manioulators and Mobile Robots. *The International Journal of Robotics Research,* 5(1), pp. 90-98.

Kleijn, C., n.d. *HIL-Simulation.* [Online] Available at: http://www.hil-simulation.com/images/stories/Documents/Introduction%20to%20Hardware-in-the-Loop%20Simulation.pdf
[Accessed 17 October 2016].

Korn, B. & Edinger, C., 2008. *UAS in civil airspace: Demonstrating "sense and avoid" capabilities in flight trials.* s.l., IEEE.

Lacher, A. et al., 2010. *Airspace Integration Alternatives for Unmanned Aircraft,* Singapore: The MITRE Corporation.

Lacher, A. et al., 2010. *Airspace Integration Alternatives for Unmanned Aircraft,* Singapore: The MITRE Corporation.

Langley, R. B., 1999. *Dilution of Precision.* [Online] Available at: http://gauss.gge.unb.ca/papers.pdf/gpsworld.may99.pdf
[Accessed 20 September 2016].

Lester, T., Cook, D. & Noth, K., 2014. *USAF Airborne Sense and Avoid,* Bedford, Massachusetts: MITRE.

Lin, C.-F., Bibel, J., Hecht, N. & Serakos, D., 1998. *Autonomous integrated air vehicle control.* Anaheim, AIAA and SAE World Aviation Conference.

Lin, C. & Lai, Y.-H., 2015. Quasi-ADS-B based UAV conflict detection and resolution to manned aircraft. *Journal of Electrical and Computer Engineering,* Volume 2015.

Luxhoj, J., 2013. Predictive Analytics for Modeling UAS Safety Risk. *SAE International Journal of Aerospace,* 6(1), pp. 128-138.

Martel, F., Schultz, R., Wang, Z. & Semke, W., 2011. *Flight Testing of an ADS-B-based Miniature 4D Sense and Avoid System for Small UAS.* St.Louis, Missouri, Infotech@Aerospace.

Marwedel, P., 2011. *Embedded Systems Foundations of Cyber-physical Systems.* 2nd ed. Dortmund: Springer.

McLain, T. W. & Beard, R. W., 2000. *Trajectory planning for Coordinated Rendezvous of Unmanned Air Vehicles.* Denver, CO, American Institute of Aeronautics & Astronautics.

Nonami, K. et al., 2010. *Autonomous Flying Robots: Unmanned Aerial Vehicles and Micro Aerial Vehicles.* s.l.:Springer.

Oshman, Y., 1999. Mini-UAV attitude estimation using an inertially stabilized payload. *IEEE Transactions on Aerospace and Electronic Systems,* 35(4), pp. 1191-1203.

Ozimina, C. D. T. S. K. C. H. E., 1995. *Flight control system design for a small unmanned aircraft.* Seattle, WA, Proceedings of the American Control Conference.

Pahsa, A., Kaya, P., Alat, G. & Baykal, B., 2011. *Integrating navigation & surveillance of Unmanned Air Vehicles into the civilian national airspaces by using ADS-B applications.* Herndon, VA, ICNS 2011.

Pinkney, F. et al., 1997. *UAV communications payload development.* Monterey, CA, Proceedings - IEEE Military Communications Conference MILCOM.

Pixhawk, 2016. *Atmospheric Pressure.* [Online] Available at: https://pixhawk.org/dev/know-how/atmospheric_pressure [Accessed 19 September 2016].

Pounds, P. ,. B. D. ,. D. A., 2012. Stability of small-scale UAV helicopters and quadrotors with added payload mass under PID control. *Autonomous Robots,* 33(1-2), pp. 129-142.

Prats, X. et al., 2012. Requirements, Issues, and Challenges for Sense and Avoid in Unmanned Aircraft Systems. *Journal of Aircraft,* 49(3), pp. 677-687.

QGroundControl, 2016. *MAVLink Micro Air Vehicle Communication Protocol.* [Online] Available at: http://qgroundcontrol.org/mavlink/start [Accessed 16 September 2016].

Ramasamy, S., Sabatini, R. & Gardi, A., 2014. *Avionics sensor fusion for small size unmanned aircraft Sense-and-Avoid.* s.l., IEEE.

RedHat, 2012. *What is open source?.* [Online] Available at: https://opensource.com/resources/what-open-source [Accessed 10 September 2016].

Richards, A. & How, J., 2002. *Aircraft trajectory planning with collision avoidance using mixed integer linear programming.* s.l., s.n.

Salih, A. ,. M. M. M. H. G. K., 2010. *Modelling and PID controller design for a quadrotor unmanned air vehicle.* Cluj-Napoca, Romania, 17th IEEE International Conference on Automation, Quality and Testing, Robotics, AQTR 2010.

Shakernia, O., Chen, W.-Z. & Graham, S., 2007. *Sense and Avoid (SAA) Flight Test and Lessons Learned.* Rohnert Park, CA, AIAA.

SourceForge, 2016. *reStructuredText.* [Online] Available at: http://docutils.sourceforge.net/rst.html [Accessed 10 September 2016].

SpeedGuide, 2016. *What is the typical range of a Wireless LAN?.* [Online] Available at: http://www.speedguide.net/faq/what-is-the-typical-range-of-a-wireless-lan-330 [Accessed 2 November 2016].

Spriesterbach, T. P., Bruns, K. A., Baron, L. I. & Sohlke, J. E., 2013. Unmanned Aircraft System Airspace Integration in the National Airspace Using a Ground-Based Sense-and-Avoid System. *Johns Hopkins APL Technical Digest,* p. 573.

Stack Exchange, 2011. *Algorithm for offsetting a latitude/longitude by some amount of meters.* [Online]
Available at: http://gis.stackexchange.com/questions/2951/algorithm-for-offsetting-a-latitude-

longitude-by-some-amount-of-meters
[Accessed 14 October 2016].

Stark, B., Stevenson, B. & Chen, Y., 2013. *ADS-B for small Unmanned Aerial Systems: Case study and regulatory practices.* Atlanta, GA, ICUAS 2013.

Techno Sainz, 2016. *Rc Car Wiring Diagram.* [Online] Available at: http://astrapt.com/wiring/rc-car-wiring-diagram.php [Accessed 11 10 2016].

The Linux Foundation, 2013. *Intro to Real-Time Linux for Embedded Developers.* [Online] Available at: https://www.linux.com/blog/intro-real-time-linux-embedded-developers [Accessed 20 September 2016].

The Linux Foundation, 2016. *Dronecode.* [Online] Available at: https://www.dronecode.org/ [Accessed 16 September 2016].

The Linux Foundation, 2016. *Real-Time Linux.* [Online] Available at: https://wiki.linuxfoundation.org/realtime/start [Accessed 1 November 2016].

The Physics Classroom, 2016. *1-D Kinematics - Lesson 6 - Describing Motion with Equations.* [Online]
Available at: http://www.physicsclassroom.com/class/1DKin/Lesson-6/Kinematic-Equations [Accessed 2 November 2016].

T-Motor, 2013. *Navigator Series Motors > MN1806.* [Online] Available at: http://www.rctigermotor.com/html/2013/Navigator_0910/34.html [Accessed 16 October 2016].

Trimble, S., 2015. *FlightGlobal - Google targets low-cost ads-b out avionics market.* [Online] Available at: https://www.flightglobal.com/news/articles/google-targets-low-cost-ads-b-out-avionics-market-410473/
[Accessed 9 September 2016].

Twisted Matrix Labs, 2016. *What is Twisted?.* [Online] Available at: https://twistedmatrix.com/trac/ [Accessed 1 November 2016].

U.S. Air Force, L. C. L. P., n.d. *Air Force Photos.* [Online] Available at: http://www.af.mil/News/Photos.aspx?igphoto=2001601881 [Accessed 9 September 2016].

UAS Task Force, Airspace Integration Integrated Product Team, 2004. *OSD Airspace Integration Plan for Unmanned Aviation,* s.l.: Department of Defense.

UAS Task Force, Airspace Integration Integrated Product Team, 2011. *Unmanned Aircraft System Airspace Integration Plan,* s.l.: Department of Defense.

uAvionix, 2016. *ping2020 - uAvionix.* [Online] Available at: http://www.uavionix.com/products/ping2020/ [Accessed 9 September 2016].

United States Special Operations, C., 2011. *Exhibit R-2, RDT&E Budget Item Justification,* s.l.: s.n.

Whittle, R., 2013. *The Man Who Invented the Predator.* [Online] Available at: http://www.airspacemag.com/flight-today/the-man-who-invented-the-predator-3970502/?no-ist

Williams, E., 2011. *Aviaton Formulary v1.46,* s.l.: s.n.

Y., L. & S., S., 2015. *Sense and avoid for Unmanned Aerial Vehicles using ADS-B.* Seattle, U.S., IEEE International Conference on Robotics and Automation, ICRA.

Yu, X. & Zhang, Y., 2015. Sense and avoid technologies with applications to unmanned aircraft systems: Review and prospects. *Progress in Aerospace Sciences,* Volume 74, pp. 152-166.

Appendix A: Transformations and formulas

GPS Precision

The GPS reading of the UAS represents the latitude and longitude in decimal degrees. In order to get a centimeter precision, the following formulas are used. The distance from the equator to the poles along the Paris meridian is defined as:

$$D = 10^7\ meters$$

The angle from the equator to the pole is known to be 90 degrees thus:

$$1^o{}_{latitude} = 111111\ meters_{x-latitude} \quad (1)$$

$$1^o{}_{longitude} = 111111\ meters_{y-longitude} * \cos(latitude) \quad (2)$$

As such, a reading with zero decimal places equals a 111111-meter radius, one decimal place equals a 11111.1-meter radius and so forth. In order to have a centimeter-scale precision (1.11111 cm), there needs at least a reading with 6 decimal places. It has to be noted that the error gets bigger as approaching the poles.

(Stack Exchange, 2011)

The same method, but with a more precise reading, is to calculate the circumference of the earth and then divide it by 360 degrees. The assumptions of this model is that the earth is a perfect sphere with 6378137-meter radius. In that way, one degree of latitude equals:

$$\frac{2*\pi*6378137}{360} = 111319.9\ meters \quad (3)$$

With this transformation, 6 decimal degrees equal a precision of 1.113199 cm.

Ground distance between two coordinate sets

Since the two coordinate sets are relatively close to each other, there is no need for great circle navigation (Williams, 2011). A simple flat-earth model (use of the Euclidean distance) can be used.

The transformation from degrees to meters follows the model of eq.3:

$$\Delta\chi = \sqrt{(lat_2 - lat_1)^2 + (lon_2 - lon_1)^2} * 111319.5\ meters \quad (4)$$

Appendix B: Airframe characteristics

The two airframes that were used for the hardware-in-the-loop simulation have the following characteristics:

Foam quadcopter

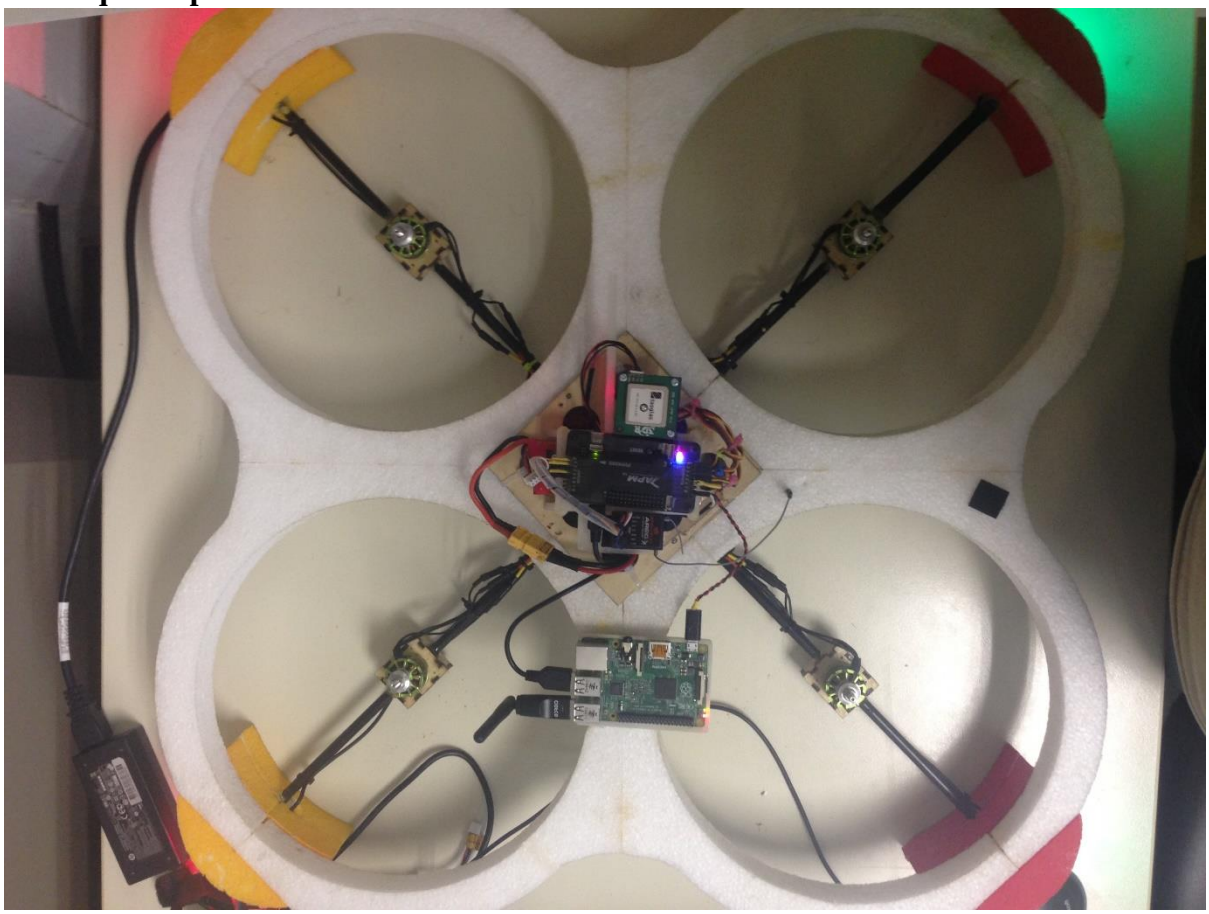


Figure 0.1 Full assembly of the EPP quadcopter. The autopilot must be in the centre of mass for good IMU calibration. Along with the GPS and compass (top of autopilot), they must be as far as possible from power sources and the motors for minimizing magnetic interference (ArduPilot Dev Team, 2016). The companion computer (RPi 2) with the WLAN module can be seen on the centre bottom of the UAS. The propellers have been removed for safety reasons in indoor testing.

1. Airframe material

   High-density expanded poly-propylene (EPP foam). Not resistant to oil and oil by-products including gasoline and kerosene. Lightweight material and energy absorbing, very good for weight-saving applications and withstands multiple impacts (JSP, 2016). The UAS can be unstable at medium-high wind speed so extra care has to be given in the aerodynamics at the design process.

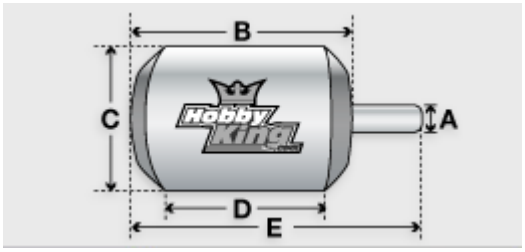
2. Dimensions
   640x640x80 mm. Approximately 250gr without electronics.

Figure 0.2 EPP quadcopter frame in full assembly. Under the wooden cap there is room for the PDB and ESCs. The wooden boxes provide a base for the motors. (HobbyKing, 2015)

3. Motors
   4x Turnigy Multistar 2209-980Kv 14Pole Multi-Rotor Outrunner V2. More poles give better control to the ESCs for better stability of the motors rpm. The motors of the foam must be strong enough to keep better stabilization of the UAS, since the airframe can be too lightweight for windy conditions.

Table 3 Characteristics and dimensions of the foam quadcopter motors.

Kv(rpm/v)	980
Weight (g)	48
Max Current(A)	9
Resistance(mh)	0
Max Voltage(V)	11
Power(W)	99
Shaft A (mm)	-
Length B (mm)	27
Diameter C (mm)	28
Can Length (mm)	12
Total Length E (mm)	31



4. Autopilot:
   ArduPilot Mega (APM) 2.5 is an autopilot that supports the open-source ArduCopter firmware. Hardware version 2.5 is not supported any more by the more modern

ArduCopter 3.3 firmware. The latest firmware supported is ArduCopter v.3.2.1. This induces some minor changes in the software design for backwards compatibility with the earlier versions. The APM autopilot is a cheap alternative to the more advanced Pixhawk one. This is challenging when dealing with advanced UAS functionality since the resources of APM are very limited. As such, on-the-fly operations must be handled with extra care.

The layout of the APM 2.5 is shown in Figure B.3. The APM 2.5 features an on-board compass and the microcontroller is Atmega2560 (Atmel, 2014). Since it contains an AVR CPU, the autopilot is also compatible with Arduino.

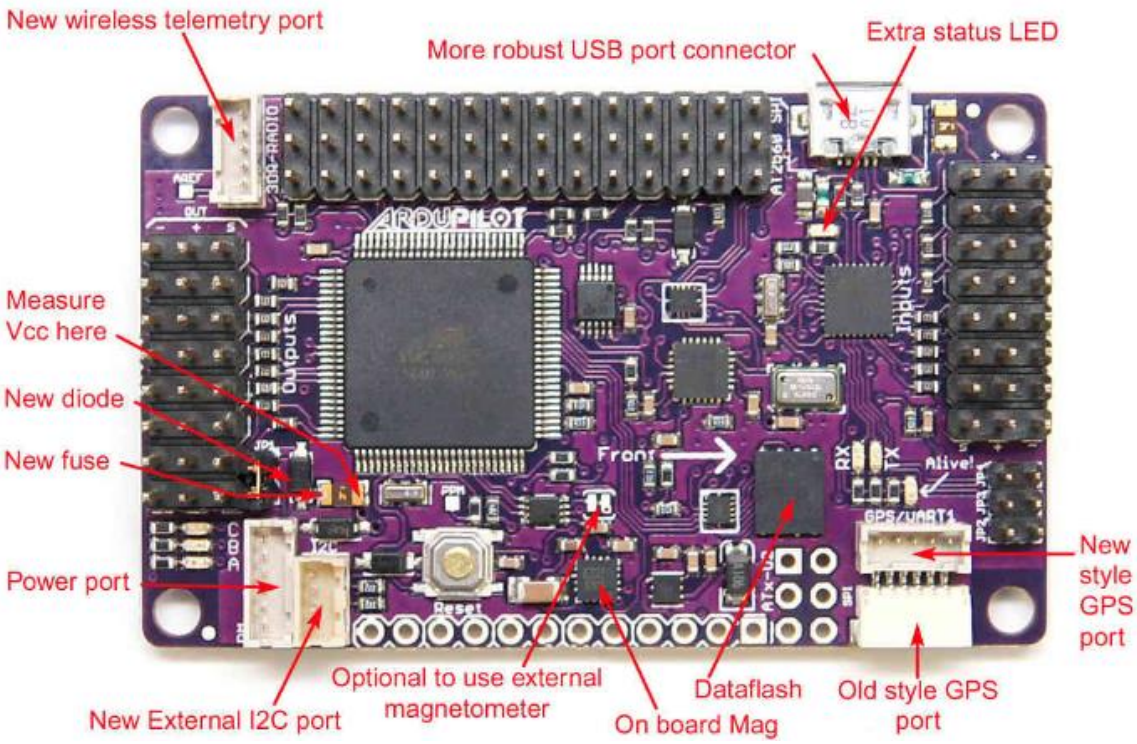


Figure 0.3 ArduPilot Mega 2.5 layout. (ArduPilot Dev Team, 2016).

5. GPS/Compass:

Since the autopilot does not contain GPS, an external module must be used. The supported module that works with APM 2.5 is 3DR's LEA-6 GPS.

Figure 0.4 3DR's GPS LEA-6 v.1.1 mounted on the foam quadcopter, next to the APM2.5 autopilot.

6. Companion computer:
   The computer used for extra processing power in the advanced UAS control side was the ARM-based Raspberry Pi 2 model B with the Raspbian, debian-based Linux distribution. Raspbian is a lightweight OS optimized for the RPi.

   - CPU: ARM quad-core Cortex-A7 @ 900MHz
   - RAM: 1GB LPDDR2 (synchronous with the CPU clock)
   - Graphics: VideoCore IV 3D graphics core
   - Interfaces: SD card slot, Ethernet, 4xUSB2 ports, Type-A HDMI, CSI, DSI, 40 GPIO pins, 3.5mm audio jack and composite video.

   The Raspberry Pi 2 does not have an onboard WLAN module, so a plug-and-play Odroid Wi-Fi module was used instead.

Figure 0.5 The Raspberry Pi 2 model B with the Odroid Wi-Fi module installed, as part of the full assembly of the foam quadcopter. It is powered by one of the 5V outputs of the autopilot through a micro-USB port.

SenseLab PhoneDrone

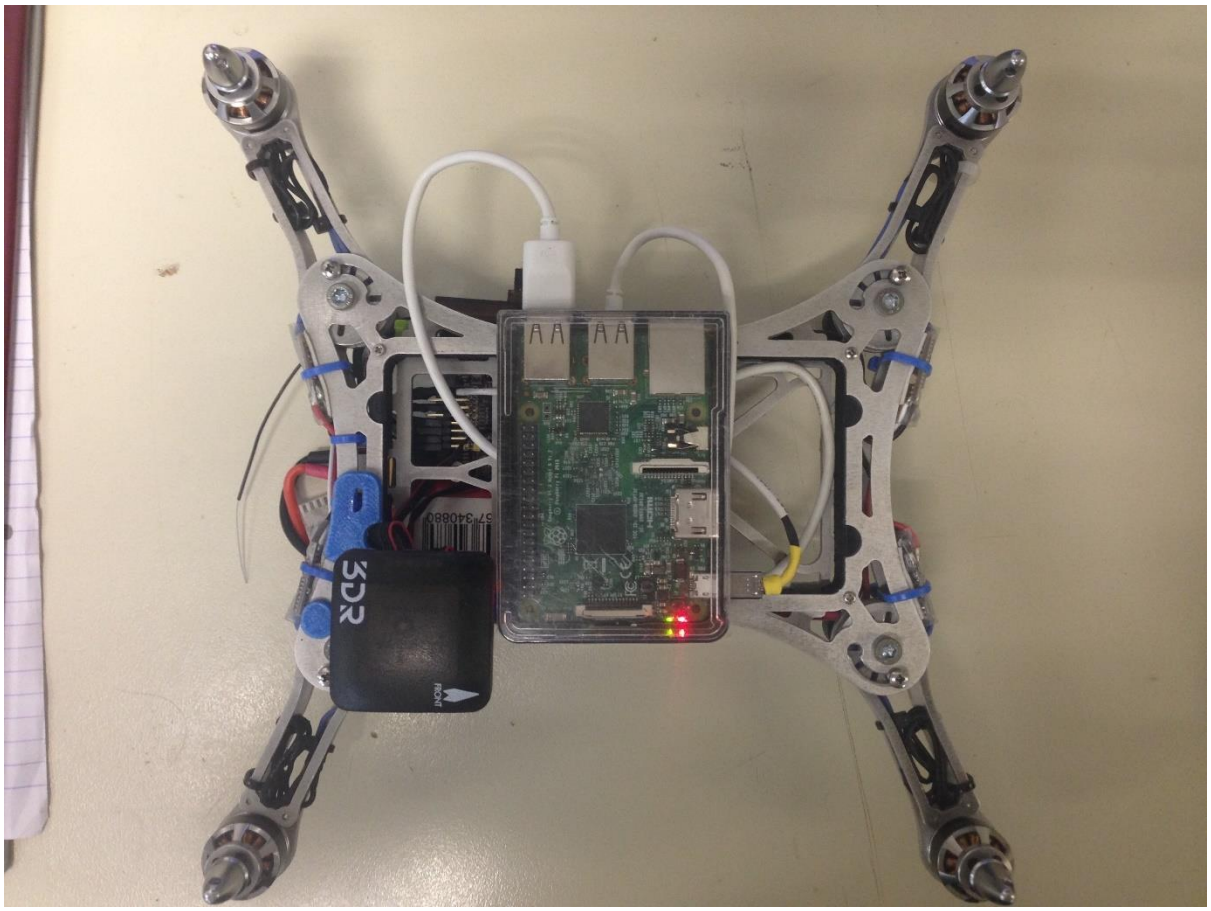


Figure 0.6 The SenseLab PhoneDrone in full assembly. The propellers were removed for safety reasons during indoor testing. The autopilot is under the companion computer which is seen at the centre of the UAS. This UAS originally takes a smartphone as a companion computer but it received slight modifications to accept a Raspberry Pi.

1. Airframe material
   Aluminum is a widely used material for multi-copters since it can be easily fabricated. It is not as energy-absorbing as EPP but after a crash it is feasible to be bent back in shape. Two of the most important drawbacks is that aluminum is RF blocking and transfers vibrations. As such, extra care has to be taken on the placement of the RF receiver and the autopilot, since vibrations can cause erroneous output of the IMU/barometer.

2. Dimensions
   250x250x50 mm, 500gr including all equipment.

3. Motors
   4x T-Motor MN1806-14. Since this airframe was designed to be lightweight and portable, small-sized but powerful motors were needed to be used. For this reason, the dimensions of these motors may be small but the 2300kV output is adequate to easily

lift it. Because a smaller 1800mAh 7.4V battery may be used for portability reasons, the company's datasheet (see Table 4) proves that even with this capacity it can provide enough thrust.

Table 4 Specifications of the Tiger Motors MN1806 2300KV motor. (T-Motor, 2013).

Item No.	Volts (V)	Prop	Throttle	Amps (A)	Watts (W)	Thrust (G)	RPM	Efficiency (G/W)	Operating temperature(℃)
MN1806 KV2300	7.4	T-MOTOR 6*2CF	50%	2.5	18.50	144	9600	7.78	42
			65%	3.3	24.42	183	10500	7.49	
			75%	3.9	28.86	209	11400	7.24	
			85%	5.1	37.74	250	12300	6.62	
			100%	6.2	45.88	290	13300	6.32	
		T-MOTOR 7*2.4CF	50%	3.1	22.94	179	7800	7.80	64
			65%	4.4	32.56	232	8700	7.13	
			75%	5.9	43.66	285	9700	6.53	
			85%	7.8	57.72	341	10500	5.91	
			100%	9	66.60	380	11100	5.71	
	11.1	T-MOTOR 5*3CF	50%	3.6	39.96	195	13600	4.88	50
			65%	4.9	54.39	248	14900	4.56	
			75%	6	66.60	290	16500	4.35	
			85%	7.9	87.69	345	17600	3.93	
			100%	9.5	105.45	395	18800	3.75	
		T-MOTOR 6*2CF	50%	3.9	43.29	259	12800	5.98	65
			65%	5.6	62.16	335	14000	5.39	
			75%	7.3	81.03	410	15600	5.06	
			85%	9.4	104.34	486	16800	4.66	
			100%	11.1	123.21	535	17500	4.34	
Notes:The test condition of temperature is motor surface temperature in 100% throttle while the motor run 10 min.									

Figure 0.7 The 1400KV variation of the MN1806 series (T-Motor, 2013)

4. Autopilot

The autopilot used is the next version of the one in the foam quadcopter, APM 2.6. This version does not have an on-board compass and GPS and as such the supported hardware was used. Since the autopilot in the PhoneDrone does not have a casing for design reasons, a piece of foam should be put on top of the barometer in order to absorb vibrations.

5. GPS/Compass
   The APM 2.6 supported 3DR uBlox GPS LEA-6 with compass module was used. The module was placed in a 3D-printed heightened platform on top of the aluminum airframe in order to reduce vibrations and magnetic interference.

Figure 0.8 The GPS and compass module, mounted on the 3D-printed base (blue).

6. Companion computer
   The computer used was the ARM-based Raspberry Pi 3 model B running the Raspbian OS.

   - CPU: ARMv8 64-bit quad-core @ 1200MHz
   - RAM: 1GB LPDDR2 (synchronous with the CPU clock)
   - Graphics: VideoCore IV 3D graphics core
   - Interfaces: SD card slot, Ethernet, IEEE 802.11n WLAN, Bluetooth 4.1+Low-Energy, 4xUSB2 ports, Type-A HDMI, CSI, DSI, 40 GPIO pins, 3.5mm audio jack and composite video.

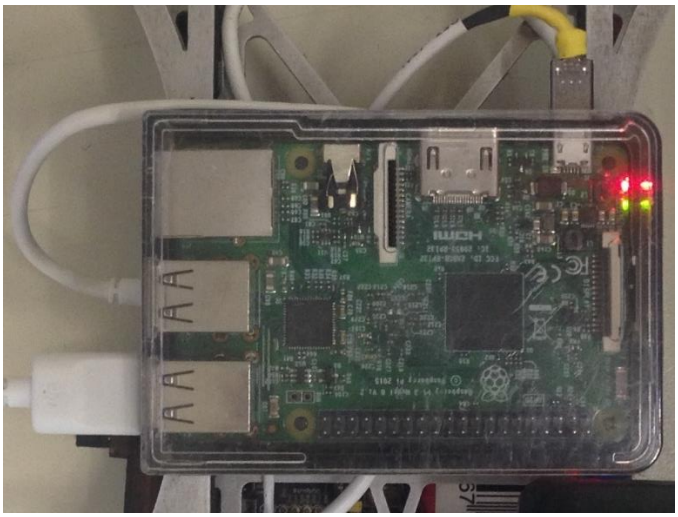


Figure 0.9 The Raspberry Pi 3 model B mounted on the PhoneDrone and powered by one of the 5V autopilot outputs. Casing has been used for the protection of the system.

Appendix C: Information shared between UAS

The vehicle state parameters which are found useful during a conflict involve the system's avionics capabilities, flight plan, remaining power, system state. Not all attributes can be accessed by the system, since some of which are hardware and software dependable. The parameters exchanged are:

Metadata:

- ID: Int representation of a code-generated unique id.
- last_recv: Timestamp added upon message receipt.
- distance_from_self: calculated upon receipt.

GPS-related:

- gps_fix: Needs to have value 3 for 3D fix.
- gps_sat: Number of satellites
- gps_eph: horizontal position error in meters
- gps_epv: vertical position error in meters

Autopilot:

- set_global_alt: Altitude remotely settable. Not supported with AC3.2 versions or older.
- set_attitude: Attitude remotely settable (pitch, yaw, roll). Not supported with AC3.2 versions or older.
- version: Autopilot firmware version.

Flight plan:

- mission_importance: See System design and implementation.
- mode: Flight mode. See Appendix D: Prioritization parameter sets.
- heading: Heading in degrees.
- next_wp: A number indicating the next waypoint in the mission list.
- next_wp_lat: next target latitude.
- next_wp_lon: next target longitude.
- next_wp_alt: next target altitude.

Flight parameters:

- groundspeed: Speed in m/s relative to the ground.
- airspeed: Actual speed in m/s.
- velocity: Three-axis speed in m/s.
- global_alt: Altitude relative to the ground.
- global_lat: Latitude relative to the global frame.
- global_lon: Longitude relative to the global frame.

System:

- ekf_ok: Indicator that the Extended Kalman Filter is ok.
- system_status: See Appendix D: Prioritization parameter sets.
- battery_level: Indicates percentage left. Only if power module is equipped.

Appendix D: Prioritization parameter sets

During the prioritization algorithm, four parameter sets are considered:

1. **System status**
   The status is defined inside the autopilot software and describes the UAS state. The list of states can be categorized to no-fly or fly-mayday and fly-normal.

   The no-fly states have the lowest priority since they wait for clearance before they start flying. The only exception is the UAS with top level mission importance, which can take-off with higher priority. The **no-fly** states are:
   - UNINIT: system uninitialized.
   - BOOT: system during booth.
   - POWEROFF: system during power off.
   - STANDBY: ready to accept commands.
   - CALIBRATING: calibration of the system's avionics (IMU, GPS etc.).
   - LOCKED: does not accept any commands.

   There are also two flight modes that declare a flying UAS in danger. These systems have the greatest priority since it is most probable that they have lost control. These **fly-mayday** states are:

   - EMERGENCY: something is going wrong but the UAS can still be controlled.
   - CRITICAL: The system is in mayday and cannot be controlled.

   The other states belong to the **fly-normal** set and are not weighted in the prioritization algorithm.

2. **System mode**
   The system mode is the flight mode of the UAS. The modes can be categorized as manual and automatic, according to their need in a ground pilot or not. This categorization is important since a UAS in manual flight mode may not be capable of performing flights in auto mode. For more information, see the reference (ArduPilot Dev Team, 2016).

   Ardupilot: ArduCopter flight modes:
   - **Manual**
     - ALT_HOLD
     - STABILIZE
     - MANUAL
     - ACRO
     - SPORT
     - DRIFT
     - POSHOLD
     - SIMPLE
     - SUPER_SIMPLE
   - **Auto**
     - RTL

- LOITER
- AUTO
- GUIDED
- CIRCLE
- LAND
- BRAKE
- LIFTOFF

3. **Mission importance**

   Mission importance is described in detail in Since the decision was made with trial and error, more information on the decision-making process is described in chapter 5.2 Software in the loop (SITL).

   Prioritization.

4. **Capabilities**

   Some autopilots contain information about their capability or not to remotely change the altitude and attitude of the UAS. If there is no such capability, then the UAS must get a top priority since it is non-cooperative. The parameter can be accessed via the vehicle's attributes:

   - bool set_global_alt
   - bool set_attitude

   If one of these capabilities is set True, then the UAS is able to make a remote maneuver, thus having the potential to lower its priority number. Some autopilots do not contain this field, so if the values are null, they are considered True.

Appendix E: Source code mechanics

The steps for initializing and using the API inside the user's dronekit code. Some sample code is shown that helps in the understanding of the system's mechanics. For the whole code please visit: https://github.com/LeonidasAntoniou/dk-plus.

Initialization inside the dronekit code:

```
# Connect to the Vehicle
logging.info('Connecting to vehicle on: %s', connection_string)
vehicle = connect(connection_string, wait_ready=True)
vehicle.parameters['PHLD_BRAKE_RATE'] = 30

#Create the interface with UDP broadcast sockets
address = ("192.168.2.7", 54545)
network = Networking(address, "UDP_BROADCAST", vehicle)
```

Figure 0.10 The Networking class is the central node of the system. A vehicle object from dronekit and the specification of networking protocol is needed for the initialization of the whole system's functionality. The initialization of the Networking class adds automatically the necessary listeners for the keeping the parameters up-to-date.

Behind the Networking class

By calling network.start() inside the main module, the system is initialized. A separate .start() function needs to be called for the CollisionAvoidance class, in order for it to begin functioning.

```
def run(self):
        """
                1.Opens the network sockets depending the explicitly stated protocol
                2.Starts the threads responsible for sending (t_send) and receiving messages (t_receive, t_task)
        """

        #Start networking protocol
        if self.protocol == "UDP_BROADCAST":
                if self.create_udp_broadcast(self.address):
                        logging.info("Network interface started succesfully")
                else:
                        logging.error("Network interface not started, exit...")
                        #status -> critical
                        #run failsafe
        else:
                #You can use whatever protocol you want here
                logging.warning("Unsupported protocol")
                #raise signal

        #Start threads
        self.t_receive.start()
        self.t_send.start()
        self.t_task.start()
```

Figure 0.11 The code inside the Networking class responsible for calling the necessary networking functions and spawning the daemon threads of sending/receiving/processing the vehicle's status parameters.

```
if network == None:
    logging.warning("No listeners added due to unknown network")
    return

#State of System (Initializing, Emergency, etc.)
@vehicle.on_attribute('system_status')
def decorated_system_status_callback(self, attr_name, value):
    network.vehicle_params.system_status = value.state
    logging.info('System status changed to: %s', network.vehicle_params.system_status)

#Battery information
@vehicle.on_attribute('battery')
def decorated_battery_callback(self, attr_name, value):
    if network.vehicle_params.battery_level == value.level:
        pass
    else:
        network.vehicle_params.battery_level = value.level
        #logging.info('Battery level: %s', network.vehicle_params.battery_level)

    #Velocity information (m/s)
    #return velocity in all three axis
@vehicle.on_attribute('velocity')
def decorated_velocity_callback(self, attr_name, value):
    if network.vehicle_params.velocity == value:
        pass
    else:
        network.vehicle_params.velocity = value
        #logging.info('Velocity changed to: %s m/s', network.vehicle_params.velocity)

"""
                Airspeed and groundspeed are exactly the same in the simulation but
                this is not applicable in real-life scenarios.
                Tolerance is added to cm scale
                Return: speed (m/s)
        """

@vehicle.on_attribute('airspeed')
def decorated_airspeed_callback(self, attr_name, value):
    if network.vehicle_params.airspeed == round(value, 2):
```

Figure 0.12 Some of the many listener functions responsible for keeping the UAS status parameters updated, with the correct tolerance to keep writes to minimum.

```
def create_udp_broadcast(self, address):
        """
                If drone_network class is initialized with the UDP_Broadcast protocol
                it opens a UDP socket in the broadcast address
                If anything goes wrong an error appears which is handled out of the function
        """
        try:
                #Setting up a UDP socket for sending broadcast messages
                self.sock_send = socket.socket(socket.AF_INET, socket.SOCK_DGRAM, socket.IPPROTO_UDP)
                self.sock_send.setsockopt(socket.SOL_SOCKET, socket.SO_BROADCAST, 1)

                #Setting up a UDP socket for receiving broadcast messages
                #Set to non-blocking because we have select() for that reason
                self.sock_receive = socket.socket(socket.AF_INET, socket.SOCK_DGRAM, socket.IPPROTO_UDP)
                self.sock_receive.setsockopt(socket.SOL_SOCKET, socket.SO_BROADCAST, 1)
                self.sock_receive.setblocking(0)
                self.sock_receive.bind(self.address)
                exit_status = 1

        except socket.error, e:
                logging.debug("Error in socket operation: %s", e)
                exit_status = 0

        finally:
                return exit_status

def populate_drones(self, scenario):
        """
                For experimenting purposes, can be ommited along with the self.populate_drones() call
                inside the __init__ function
        """
        entries = []

        if scenario == 'test_priorities':
                """
                        Gives dummy values to test priorities
                        Correct priority is: 9 7 4 5 2 3 1 8 0 6
                """
```

Figure 0.13 How the networking with UDP in broadcast is started and a fragment of a helper function – populate_drones() for testing purposes.

The Send, Receive and Process daemons

```
ready = select.select([self.network.sock_receive], [], [], 1.0) #wait until a message is r

if ready[0]:

        d = self.network.sock_receive.recvfrom(4096)
        raw_msg = d[0]
        sender_addr = d[1]
        sender_ip = (d[1])[0]

        #Keep verified messages, ignore the rest
        try:
                msg = pickle.loads(raw_msg)
                if self.verify_md5_checksum(msg):
                        data = pickle.loads(msg[0])
```

Figure 0.14 The receiver daemon waiting for a message non-blocking and verifying the integrity of the message with MD5 checksum.

```
#Calculate distance between drones
lat1 = self.network.vehicle_params.global_lat
lon1 = self.network.vehicle_params.global_lon
lat2 = message.global_lat
lon2 = message.global_lon
message.distance_from_self = geo.get_distance_metres(lat1, lon1, lat2, lon2)

#Ignore drones that are beyond safety zone
if message.distance_from_self > self.network.SAFETY_ZONE:
        #print "Drone ", message.ID, " not dangerous"
        pass

else:
        #Add timestamp
        message.last_recv = time.time()

        #Add extra fields for collision avoidance purposes
        message.priority = None

        #Update drones list
        found = False
        if len(self.network.drones) == 1: #first element apart from us
                self.network.drones.append(message)

        #Update values if message comes from the same sender or insert new sender
```

Figure 0.15 The receive_task daemon adding metadata to the received message.

```
while True:
        try:
                data = pickle.dumps(self.network.vehicle_params)
                checksum = self.create_md5_checksum(data)

                msg = (data, checksum)
                pickled_msg = pickle.dumps(msg)

        except pickle.UnpicklingError, e:
                data = " "
                logging.debug("Pickling Error: %s",e)

        try:
                self.network.sock_send.sendto(pickled_msg, ("192.1

        except socket.error, e:
                logging.debug("Failed to broadcast: %s", e)
                #Failsafe
                break

        time.sleep(self.network.POLL_RATE) #broadcast every POLL_R
```

Figure 0.16 Adding a checksum field and pickling the parameters for sending.

Collision avoidance functions

A collision avoidance protocol can be specified inside the CollisionAvoidance class. The samples of some helper functions are shown below.

```
def take_control(self):
        """Changes speed to zero by changing mode and overriding the RC3 channel"""

        if self.network.vehicle.mode.name == 'POSHOLD':
                #Already in POSHOLD mode
                pass
        else:
                #Save context
                self.save_context()

                #Change mode and assert
                self.network.vehicle.mode = VehicleMode("POSHOLD")
                #while self.network.vehicle.mode.name != "POSHOLD":
                        #time.sleep(0.2)

                #self.network.vehicle.mode = VehicleMode("GUIDED")
                #while self.network.vehicle.mode.name != "POSHOLD":
                #       time.sleep(0.5)

        #Give RC command so that we can bypass RC failsafe, 1500 means stay steady
        self.network.vehicle.channels.overrides['3'] = 1500     #throttle
        logging.info("Control taken!")

def give_control(self):
        """Gives control by restoring to pre-avoidance state"""

        if self.context != None:
                self.restore_context()

        #Cancel RC override
        self.network.vehicle.channels.overrides['3'] = None

        #End session
        self.in_session = False
        logging.info("Control given!")
```

Figure 0.17 Taking and giving control for the mission switch/takeover concept.

```
def restore_context(self):
        """Returns to the state before collision avoidance handling"""

        #Set to GUIDED mode to add any new commands
        if self.network.vehicle.mode.name != 'GUIDED':
                self.network.vehicle.mode = VehicleMode("GUIDED")
                #while self.network.vehicle.mode.name != "GUIDED": #Wait until mode is changed
                        #time.sleep(0.5)

        #Save x, y, z values of mission waypoints in lists since commands.clear()
        #seems to clear every instance of CommandSequence object
        x = []
        y = []
        z = []

        for wp in self.context.mission:
                x.append(wp.x)
                y.append(wp.y)
                z.append(wp.z)

        cmds = self.network.vehicle.commands
        cmds.clear()

        #Add pre-avoidance context:
        #Waypoints
        for i in range(0, len(x)):
                cmds.add(Command(0, 0, 0, mavutil.mavlink.MAV_FRAME_GLOBAL_RELATIVE_ALT, mavutil.mavlink.MAV_CMD_NA
                        0, 0, 0, 0, 0, 0, x[i], y[i], z[i]))

        cmds.upload()

        #Next waypoint
        self.network.vehicle.commands.next = self.context.next_wp

        #Flight mode
        self.network.vehicle.mode = VehicleMode(self.context.mode)
        #while self.network.vehicle.mode.name != self.context.mode:
                #time.sleep(0.5)
```

Figure 0.18 Restoring the previous flight plan and state of the UAS, after a conflict or mission takeover.

```
#Empty previous list components
self.near[:] = []
self.critical[:] = []

#1.Remove entries that have not been updated for the last MAX_STAY seconds
self.network.drones = [item for item in self.network.drones if \
        (item.last_recv == None)\
        or (time.time() - item.last_recv <= self.network.MAX_STAY)]

#2.Update own vehicle parameter entry in the right position
for i in range(0, len(self.network.drones)):
        if self.network.drones[i].ID == self.network.vehicle_params.ID:
                self.network.drones[i] = self.network.vehicle_params
                break

#3.Update near-zone and critical-zone lists
drone_list = self.network.drones

#Only our drone is in the list
if len(drone_list) == 1:
        pass

else:
        #This value is slightly not concurrent
        own = self.network.vehicle_params

        #From the detected drones, add any within a SAFETY-to-CRITICAL-metre range
        self.near = [item for item in drone_list if \
                (item.ID != own.ID)
                &(geo.get_distance_metres(own.global_lat, own.global_lon, item.global_lat, item.global_lon
                &(geo.get_distance_metres(own.global_lat, own.global_lon, item.global_lat, item.global_lon
                & (abs(own.global_alt - item.global_alt) <= self.network.SAFETY_ZONE)]


        #From the near drones, add any within a CRITICAL-metre range
        self.critical = [item for item in drone_list if \
                (item.ID != own.ID) \
                &(geo.get_distance_metres (own.global_lat, own.global_lon, item.global_lat, item.global_lo
                & (abs(own.global_alt - item.global_alt) <= self.network.CRITICAL_ZONE)]
```

Figure 0.29 Some of the code inside update_drone_list(). This function is responsible for keeping correct information on the whereabouts of other UAS in its vicinity, taking into account the safety zone.

Appendix F: User's and programmer's manual

DK+ Documentation

Release 0.2.4

Leonidas Antoniou

Oct 10, 2016