CS407 Group Report

A framework for autonomous drone networks

Authors:
Alex Henson,
Ben De Ivey,
Jon Gibson,
William Seymour

Supervisor:
Dr. Arshad Jhumka
Secondary Marker:
Dr. Andrzej Murawski

Department of Computer Science
University of Warwick
Summer 2016

Contents

List of Figures

List of Tables

Abstract

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

CHAPTER 1. Opening

1.1 Key Words

Autonomous Drones, Sensor Networks, Network Simulation, Pathfinding, Physical Routing, Communications Routing

1.2 Word Count

The document contains 30,000 words. This number was calculated from the document source by Texmaker.

1.3 Acknowledgements

We would like to thank our project supervisor Arshad Jhumka for guiding us and giving us advice on available technologies, methods and tools for wireless sensor network implementation, and for his continued support, even when we decided to change the foundational software basis of our project. Finally, we would also like to thank him for his critique during meetings and the poster presentation on our implementation of routing and research into the field.

1.4 Introduction

This report will provide a comprehensive analysis of the project undertaken by our group on the subject of autonomous drones in sensor networks. There will be a background summary of the key components in this field, as well as a discussion of the ongoing research, development and production being carried out. We will supply an analysis of the potential problems for which a solution can be found in drone networks, and a justification for the resulting aims and objectives of our group. The report will detail the design, implementation and testing of the solution, including considerations for the management of the project. Finally, the project outcome will be evaluated, followed by a conclusion reflecting on the success of the project and considerations for future works.

CHAPTER 2. Background

This section will introduce the components which will be researched into that form the basis of the project. The aim of this section is to provide the reader with definitions for keywords which will appear on numerous occasions throughout the project, as well as helping to lead into a definition of the problem space and the objectives of the project as a result.

2.1 Drones

2.1.1 Definition

Unmanned Aerial Vehicles (UAVs), also known as drones, are aircraft which are either piloted, or perform autonomously using pre-programmed flight path and objectives [2]. Consumer-level drones are typically small in size, and take the form of quadcopters, which are multi-rotor helicopters with four rotors. These types of drones are very lightweight, and powered by batteries; power consumption is almost completely attributed to fight time, although a modified drone will be required to exhaust power on its additional parts. We will be focusing on the use of these drones for the scale of this project.

2.1.2 Sensor Capabilities

Drones are typically equipped with cameras, as well as additional sensors, which vary depending on the type of drone, or its purpose. In the case of military drones, sensors such as multi-spectral targeting systems, night vision, infrared imaging and GPS are an absolute must [6]. However, mounted weaponry may also be included for direct warfare, unless the drone is designed specifically for intelligence, surveillance or reconnaissance, as power consumption is a primary concern, and must be limited. Military drones are controlled via satellite from a military base, although drones may also be controlled by Wi-Fi, radio or remote. Consumer-level drones have a wide range of usages, and the sensors that they require to accommodate these tasks are largely dependent on the price. It is possible to attach a large multitude of different sensors to drones; however the primary function of an aerial (consumer-level) drone is to collect high-quality imagery, often beyond the level of detail that the human eye can process. These types of sensors include stereoscopic, thermal imaging, near-infrared and infrared, but other sensors such as thermal sensors and proximity sensors may also be used [12].

2.1.3 Usages

Consumer-level drones have become increasingly popular in the past few years as they have become more affordable, capable and reliable to use. Due to their ability to capture high-quality imagery from impossible-to-reach locations, drones are incredibly useful in areas such as real estate, to take aerial shots of properties, or as a cheap alternative to huge, expensive helicopters for capturing news, such as high speed chases [4]. Drones can also be employed for services such as delivery; there have been several initiatives for drone-based delivery of food or packaged goods by famous companies such as Amazon and Domino's Pizza [5]. Another possible usage for drones is in emergency services, such as the detection of forest fires, or search and rescue. It is these areas of drone research and development which can be considered to display the strength and importance of drones, as they are able to perform dangerous tasks which humans are incapable of and/or with no physical risk to human beings themselves. Compared to other types of mobile sensing, drones offer direct control over where to sample the environment, such that they can be explicitly told where to move to.

2.2 Sensor Networks

2.2.1 Definition

A wireless sensor network (WSN) is a wireless network consisting of spatially distributed autonomous devices using sensors to monitor physical or environmental conditions. These devices are referred to as nodes, which are able to communicate with each other and with a base node, commonly referred to as a gateway, which provides connectivity between itself, the nodes and the rest of the wired world [8]. Nodes may vary in size and number depending on the network, but will typically contain transceivers, a battery, an electronic circuit for interfacing with sensors and an energy source. The ability to cooperatively pass sensor data to a main location has implications in many different industries, as well as military applications.

2.2.2 Drone Networks

In the context of drones, this refers to a wirelessly connected network of autonomous drones with a base station, which can distribute information or commands to the drones in the network, as well as facilitate communication between them and itself. Given that autonomous drones are emerging as a powerful new breed of mobile sensing system which can carry rich sensor payloads with various methods of control, a collaborative network of drones has considerable potential, and can greatly extend the capabilities of traditional sensing systems [9].

2.3 Network Simulation

2.3.1 Definition

As the name suggests, network simulation is a technique for modelling the behaviour of a network without performing a real, physical deployment, in order to test the effectiveness of the network, and assess how the network will behave under different conditions. Therefore, a simulation refers to software that predicts the behaviour of a network, so that performance can be analysed. By emulating an existing network, unexpected problems can be addressed or prevented prior to the deployment of the network.

2.3.2 Structure and Testing

A network simulation typically produces output in a GUI such that aspects of the network can be interpreted visually, such as to see how nodes interact, how data is sent, where connections go out of range or experience interference; it is possible to study the actual performance of a network and its protocols against the conceptual design. The simulation must be careful to provide an adequate level of detail to test the network without affecting the performance [1].

2.4 Routing

2.4.1 Physical Routing

In order for mobile sensor networks to gather data about the environment, they will be required to navigate freely using predetermined pathfinding algorithms. For a drone network, a drone will be required to navigate 3-D airspace and collect sensing information, whilst being careful to maintain an efficient route and avoiding problems such as collision with its neighbours or the limitations of its physical components, such as battery life. Pathfinding must take into account the possibility of nonlinear dynamics, various constraints and changing environments [10].

2.4.2 Communications Routing

One of the core aspects of a sensor network is the ability for nodes to communicate with each other, relaying data back to the gateway. For an optimal routing algorithm, the exchange of data must be robust, avoiding congestion and maintaining connectivity when faced with mobility, whilst trying to maximise the duration for which the sensing task can be performed [3]. The properties of communications routing which are to be optimised are dependent upon the type of sensor network; a drone network with less than thirty minutes of battery life must be optimised for energy consumption.

CHAPTER 3. Specification

In this section of the report, there will be an introduction to the possible problem(s) which are solvable through consumer-level drone networks, as defined in the previous section. After describing the problem, the objectives which need to be achieved in order to provide a solution for the problem space will be clearly laid out, to outline the foundation of the project. Justification for why the project solution to the aforementioned problem is both necessary and valid will also be given. There will be an analysis of the stakeholders in the project, followed by a feasibility study, where we provide a brief discussion of the scope of the problem which is to be handled, including the level of depth with which drone networks will be explored and implemented throughout the project.
This study also allows us to identify the possible problems which may arise and analyse the economic implications of the project, as well as give a brief introduction to the management of the project, in order to show that the project is actually feasible to complete with the time and resources available. Having defined the objectives which must be completed to provide a solution for the project, the subsequent functional and non-functional requirements must be identified, to ensure that the project deliverables are measurable and well-defined. Finally, any changes from the original specification at the beginning of the project will be briefly discussed, with justification for these changes.

3.1 Description of the Problem

The problem, in the context of this project, is that there are many situations in which humans are unable to effectively carry out a task, or would be put at risk, and so the situation is too difficult or dangerous to handle. In such situations, a well implemented drone network could not only perform better than a human could in a risk-free environment, but the use of drones as a sensor network provides a better alternative to other possible sensor network solutions, which will be discussed later. We can define an arbitrary problem which is impossible or risky for humans such as search and rescue, or high altitude photography, as well as problems which are feasible for humans, but can be more readily solved by UAVs, such as automated delivery.
For this project, we have decided to focus on the possibility of using a drone network to detect and combat forest fires for the use of emergency services. While problems such as these have

existed for a long time, solutions offered through sensor networks using drones is an area which is still in the early stages of research and development. While alternative solutions to using drone networks already exist, it is possible that the use of drones can expand upon existing solutions. Furthermore, the implementation of a drone network can hypothetically be applied to any similar problem area by altering the drones sensory inputs and outputs arbitrarily, and so a solution can be provided for the general use case, and applied to various other situations. The solution to this problem can be split into four main areas: the network simulation, the physical routing, the communications routing, and the physical deployment. In order to successfully design and implement a drone sensor network, it is necessary to construct a stimulator which will accurately model the performance of the drone network, with associated physical and communications routing algorithms for optimal performance, before finally transferring this model to the physical drones and deploying them. The general solution can then be tested in a real situation, and then possibly extended to handle specific problem-solving tasks with user input.

3.2 Objectives

The aim of this project, then, is to design and implement a general-use drone sensor network in order to solve the arbitrary problem of detecting and counter-acting forest fires based on user input, with the capability to be extended to any potential situation in the problem space. The major components of the project were outlined in the project specification during the early stages of the project life cycle, and can be summarised as follows:
(**THIS SECTION PROBABLY NEEDS WORK**)
(in the sense that tasks could be reordered, bullet points could be added for each enumerated item for more detail, tasks could be added/removed/reworded)

1. **Implement a network simulator**. Either using a pre-existing set of libraries, or by creating our own, which are specifically tailored to our domain
2. **Establish a network of drones with a base station**. Drones can communicate with one another and the base station, sending data between them
3. **Provide autonomy to drones**. Each drone must be able to dynamically control itself, as opposed to remote control, in order to operate autonomously in a network
4. **Implement drone pathfinding**. Drones must carefully navigate through an area of physical space using well-defined rules for physical routing
5. **User input tasking**. The user must be able to define a problem for the network to detect, which is passed from the base station to the drones
6. **Implement problem detection**. Drones use sensory information to collect and pass data, notifying the base station in the event of a problem

These tasks are roughly estimated to take an equal amount of time, where separate tasks can be assigned to each group member for maximum efficiency. Certain tasks, such as communication and pathfinding can be developed in parallel. Delegation of tasks and group roles will be discussed in depth in later sections.

3.3 Justification

The benefits of carrying out a project involved in creating an efficient, risk-free solution to emergency situations is relatively self-explanatory. The project is justifiable in its ability to produce potentially life-saving results and improve the general quality of human life, by implementing a consumer approach to any of the common usages that drones can be applied to and more, through the use of sensor networks. Drone networks themselves are an area of research which is growing in popularity, so the project interacts well with the state-of-the-art, and could have considerable impact on future developments. A general use solution for the project would be easy to use and deploy, and incredibly extensible.

3.4 Stakeholder Analysis

As previously discussed, the implementation of a drone sensor network has implications for a wide variety of industries. This section will therefore give a formal outline of the prospective stakeholders in the project and the justification for them. Firstly, with respect to detection and response to forest fires, emergency services such as firefighters, ambulance services and search and rescue would benefit greatly, by providing the ability to pre-empt danger and respond to crises much more rapidly, as well as reducing the risk to human lives in combating fires. Given that drone sensor networks are a relatively new field of research, those parties interested in research and development of sensor networks would also benefit from the project, as the results may extend the field of research, and the project solution could be adapted for use in other areas. In the same way, commercial businesses such as real estate and delivery services in pursuit of more efficient business may also benefit from the project. Finally, considering the use of drones in military operations, there are possible ramifications for military usage of the project, which will be discussed in further detail in later sections.

3.5 Feasibility Study

While the merits of the project are justifiable, it should also be noted that the project implementation carries a considerable technical difficulty, involving adaptation of individual, consumer-level drones to programmable, autonomous drones which function as a sensor network capable of algorithm-based communications and pathfinding, which are to be simulated and then physically

deployed. Therefore, it is important to analyse the feasibility of the project, given time, hardware and other constraints, which will be discussed in the following sections.

3.5.1 Problem Scope

It is important to consider the scope of the problem, with regards to how far the solution can be extended into the domain of, in this case, search and rescue. Given that drone sensor networks are a relatively new domain for research and development, there is no commonly used and accepted standard for consumer-level drone networks in the context of search and rescue. In other words, the solution to the problem is not an extension of a previously existing solution, or of a set of rules governing how the problem can and should be solved using drone sensor networks. As a result, the scope must be carefully defined such that an effective solution can be reached for the problem, without expanding too far into the problem domain. By implementing a drone sensor network which can be adopted into any general use case, which can be easily extended to take any required sensory information and applied to solve a problem, we can avoid overextending the project. At the same time, in order to show that the network can effectively handle user input tasking, we focus on taking one piece of sensor information, such as thermal, to demonstrate an accurate model for problem detection (as defined by the user), and response through well-established communications.

3.5.2 Project Scope

Having defined the scope of the problem, it is also necessary to examine the scope of the project itself in terms of how far we extend into the domain of drone sensor networks. The implementation of any sensor network requires a lot of concise testing and a solid formation of protocols. There are many areas to consider with physical routing and communications, as well as physical deployment and use tasking. When creating the network simulator, we can tailor it to our specific requirements in order to minimise the amount of considerations for network protocols and possible problems, which will be discussed later, whilst accurately modelling real, physical deployment. Considering the time constraints that are imposed on the project, it will also be necessary to adapt pre-existing algorithms to establish our network communications and physical routing, as opposed to creating our own algorithm. In terms of physical deployment, the project is limited to two drones, so it is not possible to implement a full-blown network to test the solution. Nonetheless, it is possible to show that the network is theoretically scalable and accurately demonstrates the ability for drones to communicate and use pathfinding effectively.

3.5.3 Financial Analysis

The basic requirements for a physical drone network are the drones themselves, the sensors which will be attached to the drones, as well as equipment for programming the drones and

communications. All of these are provided for free by the Department, so there are no financial constraints for physical deployment. While it is possible to purchase more drones, the same result can be achieved, provided that we have at least two drones. Additionally, the project will not include the use of any bespoke software or libraries in the development of the simulator or communications; they will be open source, so there are no costs incurred in the development of the project.

3.5.4 Market Analysis

For the project to be used by third parties, there must be licensing associated with the final product. As the project is intended to be open source and not commercialised, we will be using the GNU General Public License v3.0, which stipulates that our project is open source, with the freedom to use or change the software, as well as distribute it and share changes. However, in the event that the project is used, the software creators are not accountable for any problems with the project software. As the project is not commercial, with no intention to monetise it, the use of this license allows us to appease any and all stakeholders whilst protecting ourselves from potential threats.

3.5.5 Project Management

There are many challenges associated with undertaking a project such as this, particularly with regards to administration, scheduling and the division of tasks. Therefore, it is crucial that the project is managed efficiently, and that progress is carefully monitored and assessed throughout the life-cycle of the project. The issues associated with project management and how they were handled throughout the project will be discussed during the evaluation stages of the project. However, there are several challenges that the project group will be presented with in the context of project management, which are summarised below:

- **Development methodology**. There must be consideration for a development methodology which accommodates the size of the group and the time each group member has available, as well as how to track the progress of the project
- **Deadlines and scheduling**. Meetings and deadlines must be scheduled such that the group is fully aware of what stage of the project must be completed and when. Interested parties such as the project supervisor must also be regularly informed of the progress of the project to allow for appropriate advisory actions where necessary
- **Group roles**. Each member of the group must have a clear and distinct role in the group, with one member taking the role of project manager, dedicated to handling each group member and their associated tasks, as well as managing the state of the project to ensure that it is completed effectively and within constraints

- **Managing project materials**. It is necessary to manage the project material to ensure consistency amongst each members work, as well as maintaining previous iterations of code to avoid problems such as losing previous functionality

3.6 Requirements Identification

This section will contain a list of the functional and non-functional requirements identified at the beginning of the project which will be necessary to implement to ensure that the project software adequately represents a solution to the problem as defined in this chapter. A functional requirement refers to quantitative requirements which can be easily measured and tested to ensure correct functionality. Non-functional requirements are those which cannot be measured, as they are subjective, but are nonetheless important for a successful project outcome.

(**THIS SECTION PROBABLY NEEDS WORK**)

3.6.1 Functional Requirements

#	Description
R1	User must be able to run an executable which will set up the simulation environment and communications module
R2	A class must be instantiated for the nodes of the simulator (i.e. drones and base station) and a communications module must be installed on each of them
R3	The simulation must run, such that each node begins to operate and travel within the environment
R4	Each node must be capable of sending, listening for and receiving messages over Wi-Fi/radio
R5	The environment must incorporate and handle multithreading for each node
R6	The simulation must be able to model and be robust to interference between message passing
R7	The user must be able to see output from each node as messages are sent and received
R8	The user must be able to visualise the output of the simulation on a graphical interface
R9	The base station node must be able to take user input which can be broadcast to other nodes
R10	The simulation must be capable of communications from any (preselected) algorithms
R11	The environment and its respective nodes should stop running when a KILL command (message) is received
R12	Nodes must be capable of moving through the environment autonomously using pathfinding algorithms
R13	Nodes must be able to recognise useful sensory information, and broadcast the information back to the base station
R14	The simulation code must be adaptable to the physical drones for real deployment

3.6.2 Non-functional Requirements

#	Description
R1	Connections between nodes must be stable and resistant to interference
R2	Passing of messages must be within a reasonable timeframe
R3	The broadcasting of information from drones or user input must be accurate
R4	The user must be satisfied with the output from the environment
R5	The final code must be extensible for more specific use cases

3.7 Project Deliverables

Throughout the course of the project, there are several deadlines that must be met for which the project will be required to deliver a product. At the beginning of the project, the project specification must be submitted, which introduces the project and its initial planning and methodology, which will be appended to the report for reference. At the end of Term 1, a poster presentation will be given by the project group to select supervisors and invigilators, and a live demo of the project software and the final report are delivered at the beginning of Term 3. The demo will display the proposed solution to the problem defined at the start of the project, and showcase the features of the working solution. Alongside the final report, the code for the project, including the simulation, communications routing, physical routing and deployment code will also be submitted.

3.8 Changes From Original Specification

In the original specification, it was stated that NS3, a set of network simulator libraries, would serve as the core implementation of the simulator, and that the code for physical deployment would then be built off of the simulator as a result. However, we have decided to instead build our own simulator, after discovering that NS3 is relatively bloated and difficult to use. The benefits to creating our own stimulator are that the code can then be directly translated onto the drones during deployment, as well as being directly tailored to the project domain. As a result, the functionality provided by NS3 must be manually generated, which may increase the complexity of the project.

CHAPTER 4. Literature Review

CHAPTER 5. Design

5.1 Methodology

5.2 System Architecture

5.2.1 Network Simulation

Fundamental Structure

Environment

Application Programming Interfaces (API)

5.2.2 Communications Modules

The simulator itself should not prescribe a set way of performing communications routing, rather it is up to the user to specify the routing algorithm they wish to use. This needs to be done in a way which makes it easy to specify different routing algorithms for different nodes in a simulation, i.e. there is not one set algorithm for each simulation. This is useful, for example, if one wishes to simulate a network of quadcopters backed up by a series of stationary nodes on the ground. In terms of extensibility, it should also be possible to package, distribute, and import implementations for different communications algorithms.

Starting with the first problem, that is the ability to specify different routing algorithms for different nodes in the same simulation, it is clear that the specification of the routing protocol should apply to individual nodes rather than to simulations. To this end, it was decided to break out the communications functions for each node to a separate communications module. This provides a level of abstraction for messageable programs, as they can then make use of generic send and receive functions at the application level of the OSI model, while the communications module has fine grain control over the networking layer. In more practical terms, each messageable must have a communications module associated with it at instantiation, and it is functions of this object that will be called when the unit wishes to send and receive messages. The main benefit of this approach is that it allows flexibility when assigning communications modules - it is possible to either have every node in the simulation use a different type of communications object, or for each node to use an instance of the same object. This provides a very obvious standard for

reusing communications code between simulations.

When considering the second problem mentioned above regarding the packaging, distribution, and importing of routing functions, it was determined that this could be best achieved by combining collections of implementations into libraries and allowing the user to make use of these libraries as appropriate when writing a drone program. This makes it easy to mix and match different algorithms from different sources, and means that in order to use a particular approach, one need only link a program against the appropriate communications library. It also allows different contributors to use identical or overlapping namespaces.

It is necessary at this stage to determine exactly what responsibilities a communications module has, as well as the interface it should expose to its messageable. The most obvious function required of a component designed to communicate is to send information. To this end the communication module must be able to take a message and broadcast it to other nodes in the network. Exactly how this is achieved is dependant on the routing algorithm involved, but it is expected that most implementations will receive a message from the messageable, do some intermediate processing (perhaps to determine the best route to take) and then call the environments *broadcast* function to pass the message to hardware (be that real or simulated).

If we are able to send messages via a communications module then it stands to reason that we should also be able to receive messages as well. Thus, there must also be a way for the communications module to transfer messages it has received from the environment to its messageable. This can be achieved by providing a callback function to the environment, which can be invoked when there is a message to deliver. An alternative method which instead provides asynchronous message passing is for each communications module to have a queue of incoming messages for it to process, which it should check at regular intervals. Both of these messages have merit, given that interrupt driven message delivery can become problematic when the network is flooded with messages (no other tasks can be done as the message interrupt is constantly called over routine code), and the asynchronous approach can lead to problems when it is paramount that a message be delivered immediately (such as a command to ground a malfunctioning drone).

The module also needs some way of performing tasks which are not triggered by the arrival of new messages. In order to facilitate time or state driven processing, communications modules need to have a function which is run independently of the send/receive functions which either loops or is called continuously. It is also clear that one should be able to trigger the pushing out or pulling in of messages from this function, to allow for the use of non-data packets (perhaps, again, to determine the optimum route to the destination).

But if the above function is to be run as a loop, the must be some condition upon which it terminates, lest the simulation run indefinitely. If the threads for a messageable and its communication object were associated with each other then it would be possible to terminate both when the former exits. This can also be achieved by having the program notify the communications module that execution is complete and that it should terminate. The latter option was chosen

due to the fact that it simplified the construction of the simulator, and made it easier to modify the relationship between messageable objects and communications modules in the future, for example to change the relationship from one to one to one to many (so that a base station could have separate communication interfaces to other stationary nodes and to aerial units).

Underpinning all of the above design decisions is the idea of a common specification of a message. It would be sensible to expect all messages to be serialisable as text so that they can be safely transmitted across a real world network. Beyond that it is useful to be able to label messages so that perfunctory filtering can be carried out on messages without the need for in-depth inspection. Each routing model will require more than this, as it does not even include space for a payload or addressing which are both quite useful when delivering messages. To this end it must be possible for the base message model to be extended by individual implementations, but the basic object should remain simple enough that every included feature is required by all implementations.

5.2.3 Physical Routing

5.2.4 Physical Deployment

Libraries

CHAPTER 6. Simulation Software

6.1 Existing Software

6.1.1 NS3

6.1.2 NS2

6.1.3 Some

6.1.4 other

6.1.5 stuff

6.2 Summary of Existing Software

6.3 Development Methodology

6.4 Code Structure

6.4.1 The Environment

6.4.2 Communication Modules

6.4.3 Drone

6.4.4 Base Station

6.5 Results

6.6 Optimisation

6.7 Review Against Original Objectives

6.8 User Manual

CHAPTER 7. Physical Routing

CHAPTER 8. Communications Modules

8.1 Overview

The communications module class, *CommMod*, is the part of the simulator solution that is responsible for performing tasks at level three of the OSI model (the networking layer). Depending on the noise function used by the simulator, it is also possible to have communications modules perform tasks in layer two (such as collision detection and avoidance). Whilst the base class is a part of the simulator itself, the actual implementation of routing algorithms are part of a separate library in their own right. In this way, the communications library is intended to be distributed with **octoDrone** without being a required constituent part.

8.2 Structure

8.2.1 Structure of an Individual Module

Sending and Receiving Messages

When a communications module wishes to transmit a message to other units it invokes the broadcast method exposed by the simulation environment. Messages that originate from the communication modules messageable collect in a queue, which must be checked at regular intervals. When there is a message that the environment determines should be delivered to the communications module, it is deposited in a similar queue. When the module has a message it needs to pass to its messageable, it stores it in a third queue.

In addition to asynchronous messaging, the project design also called for synchronous message passing. This is achieved by having a callback function in the messagable which can be invoked upon receipt of an urgent communication. It is left to the communication implementation to decide which messages are important to interrupt the normal operation of the messageable for.

Intermediate Processing

The virtual function *comm_function* is called when the communications module is started and is expected to return when the associated program is ready to terminate. It should be used to perform any intermediate and non-delivery driven processing such as routing or time based

commands.

8.2.2 Structure of a Collection of Modules

In order to facilitate easy distribution and use, collections of communication implementations are combined into libraries. Given the small size of a communications modules source code (often tens of kilobytes) it might be tempting to statically link them to simulation executables. The problem with this approach, however, is that it precludes the very thing that prompted the creation of communications modules in the first place - modularity. As such, communications code is packaged into a shared library which simulator executables are then linked against dynamically at compile time. This reduces the size of produced executables and makes updating communications code possible without requiring simulations to be recompiled.

8.2.3 Access to Other Simulation Elements

The architecture of the simulator requires that communications modules are able to interact with the simulator environment (to dispatch messages), as well as with the messageable with which they are associated. Since it is impossible to have all of this information available when the communications module is instantiated (creating a messageable requires a reference to a communications module as well), it is necessary to set the messageable associated with a communications object at a later time (but before the simulation is started). Thus, the procedure for bootstrapping a simulations is broadly as follows:

1. Load the sensor data
2. Create a simulation environment referencing the sensor data
3. Create a communications module referencing the environment
4. Create a messageable referencing the environment and the communications module
5. Add a reference to the messageable to the communications module

How these references are used is covered in sections 8.3.1 and 8.3.2.

8.3 Integration with Other Components

8.3.1 Integration with the Simulator

The communications module class makes use of the following functions from the simulator:

- *broadcast*

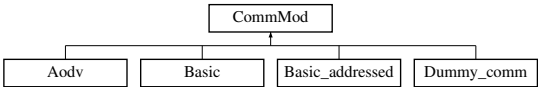

Figure 8.1: Inheritance diagram for the *CommMod* class

8.3.2 Integration with a Program

Programs for drones and base stations can make use of the following functions from *CommMod*:

- *push_in_message*
- *push_out_message*
- *comm_function* (to start the CommMod)
- *set_messageable* (used when defining simulations)

8.4 Provided Implementations

8.4.1 Basic Messaging

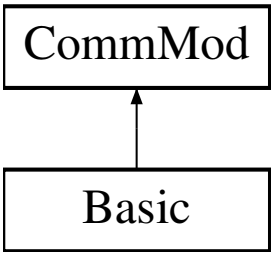

Figure 8.2: Inheritance diagram for the *Basic* class

The basic messaging implementation provides the simplest possible way of sending and receiving messages. This can be useful when creating programs for messageable units, as it removes the need to consider routing problems, units being out of range, and a whole host of other potential issues. Additionally, it provides a good method for simulating existing programs which were not written with **octoDrone** in mind and may perform their own addressing.

In this implementation, all messages that are sent are received by all nodes, regardless of range. Messages sent include payload information, but do not store information on the sender or receiver.

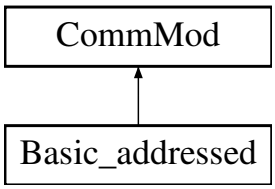

Figure 8.3: Inheritance diagram for the *Basic_addressed* class

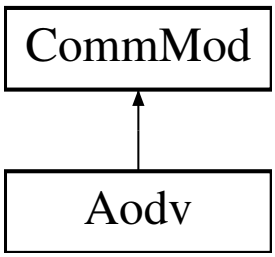

Figure 8.4: Inheritance diagram for the *Aodv* class

8.4.2 Addressed Basic Messaging

The addressed basic messaging implementation extends the above algorithm to additionally take into account the sender and intended recipient of a message when choosing which packets to deliver and which to drop. It is intended to be used for passing messages between programs and communication modules in a way which prevents the need to include code to serialise and deserialise addresses. If appropriate, it can also be used with external programs as described above.

In this implementation, all messages that are sent to an IP address are received by all nodes with that IP address, regardless of range. Messages include payload information as well as the ip addresses of the sender and intended recipient. Messages sent to the broadcast address 255.255.255.0 will be received by all nodes in the simulation.

8.4.3 Ad hoc On-Demand Distance Vector Routing

The Ad hoc On-Demand Distance Vector Routing (AODV) implementation provided in the communications library provides an example of how a more sophisticated routing algorithm can be used within the simulator framework. AODV is designed to be used in mobile ad hoc networks and offers loop free, just in time routing that is fault tolerant and capable of reacting to the movement of nodes.

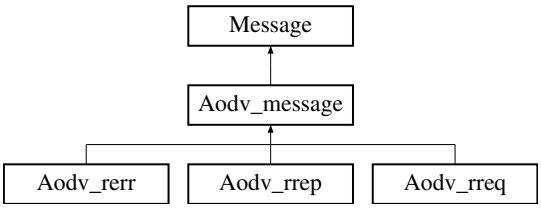

Figure 8.5: Inheritance diagram for the *Aodv_message* class

Overview

In order to explain what is happening under the hood of the AODV implementation provided with **octoDrone** it will be useful to first cover the basics of how the algorithm operates. A node A which wishes to send a data packet to node D transmits a Route Request (RREQ) to its neighbours, which serves the dual purpose of notifying it of the next hop in the most efficient path to D, but also of notifying the routes along the way where they should direct messages addressed to D. If a unit receives a route request for which it does not have a valid route it broadcasts the route request to each of its neighbours in order to propagate the route discovery.

When the RREQ is received by a node which has a route to D, either because (as in this case) it *is* node D or because it is already part of a route to D, that node creates and returns a Route Reply (RREP) packet. This packet is used by every node on the reverse route to store the next hop to the destination, D, and is forwarded to the nodes that sent the RREQ. Incremented sequence numbers are used to make sure that information remains fresh as for loop prevention.

Once the sending node, A, receives an RREP it can send out its data packets via the next hop on the route that was just discovered. This route will be cached for a period of time before a new discovery period must take place. If at any time there is an error during transmission, the node encountering the issue sends out a Route Error (RERR) packet to notify other nodes that cached routes using this hop should be invalidated.

The Routing Table

The AODV implementation has a special data structure to represent entries in an AODV routing table. As per the RFC, this contains the sequence number of the destination node, the time after which the route is no longer considered active, the hop count to the destination node, and the next hop on the route to the destination node. Each AODV communications module keeps a map of destination IP addresses to route objects, which is known as its routing table.

Message Types

In order to easily handle the different types of messages handled by AODV (RREQ, RREP, etc.), the implementation defines a basic AODV specific message class which contains the fields common

to all AODV messages. This is then extended by individual classes in order to completely define which fields are required by each message type. This is shown in figure 8.5.

Sending Messages

With the above classes in mind, we turn to the question of how octoDrone goes about sending messages. Initially, messages which have been deposited into the outbound queue must be deserialised into a structure containing the message payload, the destination address of the message, and the source address of the message (in some situations this may be different from the address of the communications module). This happens because messages are always sent to and received from the environment in string form. While it may at first seem inconvenient, it is necessary in order to preserve the integrity of messages if they are sent over a network (which is the case if we are running a deployment on real hardware). After this the routing table is queried to find out if the node already has an active route to the destination. If it does, then a data packet is constructed immediately and dispatched via the next hop on the route to the destination.

If there does not exist a route in the cache, things become a little more complicated. Because the route discovery process requires that we send and receive a number of different packets before sending the message is considered done, we need to keep track of internal state. In octoDrone's implementation of AODV, this is done by keeping track of the current message we are sending, as well as the stage in the route discovery process we are currently at. This means that whenever something is triggered by a message being received, it can access this information (and update it if required), even when the original message object has long since gone out of scope. After the state information is created, a helper function is used to create the RREQ which will initiate the AODV protocol on other nodes. This is then sent using a call to the simulation environment. At this point there is nothing else to do but wait for messages to be received.

Receiving Messages

When a message is received, the communications module unwraps just enough of it to determine the type of message that has been delivered, which will be one of the AODV types, such as RREQ (this type is internal to AODV, and should not be confused with the type property of the base message class which is used to denote emergency packets). This information is then used to select an appropriate deserialisation function to apply to the message in order to transform it from a string into an AODV data type. Each of the types has its own processing method, which determines what action (if any) to take given the message contents and the internal state of the communications module.

For RREQ messages, we check to see if the message was a hello message (described in section 8.4.4), and if so we check if we have an active route to the sender (who will be one of our

neighbours), and add a new route if we do not. A reply is sent to the sending node so that it can compile route information on us. If it is not the case that the message is a hello message, then it must be route discovery from another node. If the communications module has an active route to the destination mentioned in the RREQ, then a helper method is used to generate an RREQ packet to reply with. This is sent via a call to the simulation environment. Otherwise, we update our internal state to record that we are participating in remote initiated route discovery, and forward the RREQ packet after decrementing the time to live field and incrementing the hop count. In order to keep the implementation small and simple, nodes do not respond to RREQ requests when they are part way through route discovery for themselves or another node.

When an RREP is received, the communication module first check to see if the route it has to the sender (previous hop) is up to date, modifying the routing table if appropriate. Afterwards, we check to see if the route was for us (to send a data packet) or for another node (remote initiated route discovery). Multiple RREP packets for a single route indicate that there exists more than one possible path. In this case, the route with the lowest hop count is chosen (the reply window is defined by the path discovery time variable which is set at instantiation). If the former, then a data packet is constructed using the new routing information, and sent via a call to the simulation environment. Otherwise, the information is passed on (with a decremented TTL). Keeping track of hop counts is done by the helper function.

Upon receipt of an RERR packet, the communication module marks the route as invalid (achieved here by setting the active route timeout to zero), before broadcasting this information so that affected nodes can update their local routing tables.

Broken Links

But when do these RERR packets come from? It may be that a node receives a message for which it does not have a route. It may also occur that a node loses connectivity with one of its neighbours. If either of these scenarios come to pass then the receiving node will broadcast an RERR message to the node which precedes it in the route. Given that the octoDrone implementation of AODV only stores forward routes for simplicity, all RERR messages are sent upon message receipt instead of also being sent upon link failure. This makes sense given that because octoDrone does not emulate the lower layers of the OSI model, nodes do not have access to information about link statuses. One way of mitigating this is to use hello packets as described below in section 8.4.4. When a node fails to receive a hello packet for a neighbour which is listed as the next hop for one of the route in its routing table (this includes neighbours which are the next hop and destination for a route), it will mark that route as invalid and propagate an RERR the next time that route is used.

8.4.4 Extension: Hello Messages

The RFC for AODV also mentions the use of hello messages as a means of broadcasting connectivity information. While the document suggests that nodes should only broadcast hello messages if they are part of an active route, to account for the differences mentioned above, all nodes using the octoDrone implementation of AODV will send hello packets on a regular basis.

CHAPTER 9. Physical Deployment

9.1 Objectives

The main objective of this part of the project was to enable the running of octoDrone simulations on real hardware. While at first it may seem odd that we are attempting to move from a simulated programs to real ones when the overall goal of octoDrone is to allow for the simulation of real programs on virtual hardware. In a sense this a decision that logically follows the move from real to simulated - once I have built a program within the simulator framework and proved that it operates as I expect, it is infuriating to then have to implement this using a different set of frameworks on real hardware. It necessitates rewriting the same software in a way which effectively prohibits simulated benchmarks insomuch as there is no guarantee that simulated programs would exhibit the same performance characteristics as their real world counterparts given that they may be written on top of different libraries and possibly even in different languages. To this end it makes sense that one should want to run the same program in the simulator as in a real deployment.

There were a number of options that were explored in terms of the actual components that were used for our example deployment, similarly with the ways in which we adapted the simulator software to make the process as seamless as possible. The overall goal was to obtain a minimal working example, given that how well the hardware we could acquire would interact with what we had made was an unknown quantity. It is important to distinguish our efforts to make the simulator operate on real hardware units and the actual deployment we carried out. The former is a part of the product that is octoDrone, and was done to a high, professional standard. The latter was implemented using the components which were available to us through the department and, as such, does not represent what would be expected of a deployment carried out in industry. With that said, it is a testament to the flexibility of the simulator that it is possible to coerce it to run on distributed hardware components that it was never intended to support.

9.2 Equipment and Feasibility

The first decision to make with regards to the example deployment was to determine the type of hardware we would be targeting. There were two drones made available by the Department of Computer Science for our use - a pair of Parrot AR 2.0 Power Edition units.

Parrot AR 2.0 specifications[11]

- 1GHz 32 bit ARM Cortex A8 processor with 800MHz video DSP TMS320DMC64x
- Linux kernel version 2.6.32
- 1Gbit DDR2 RAM at 200MHz
- Wi-Fi b,g,n
- 3 axis gyroscope, accelerometer, and magnetometer
- Ultrasound sensors for ground altitude measurement
- 4 brushless inrunner motors. 14.5W 28,500 RPM
- 30fps horizontal HD Camera (720p)
- 60 fps vertical QVGA camera for ground speed measurement
- Total weight 380g with outdoor hull, 420g with indoor hull

9.2.1 Micro-controllers

The noteable thing missing from the above specifications is the ability to program the quadcopter with custom code. In a commercial or industrial deployment of this type, one would expect to use drones that are capable of being programmed themselves. Unfortunately these models are rather expensive, and as such we had to find a way to make the drones autonomous without being able to change the software running on them. The obvious solution was to use a micro-controller to pilot the drone because it would be easy to run our own custom software and communicate with the drone via its WiFi network. Below is a short summary of the investigation we did into the two major micro-controllers aimed at the consumer market (and thus within budget).

Arduino

Raspberry Pi

9.2.2 Inter-node Communication

In addition to communicating with the drone they were paired with, nodes would need to communicate with each other in order to pass messages and communicate with the base station. There were a number of options here which varied from true to life to more synthetic conditions.

Radio

In a real scenario, all communication done between nodes would be done using uni or bi-directional radio links. This allows for the use of software and protocols aimed at mobile sensor networks to facilitate the saving of power. It is obviously beneficial to create an environment which is as close as possible to real conditions, but we felt that this was less applicable for our example deployment.

When we considered radio in the context of octoDrone, the abstraction of the physical and data link layers for communications meant that programs would not be able to tap into the extra power afforded by using it. In addition to this, many commercially available radio modules are designed to be used with the Arduino, which we had elected not to use.

WiFi

The second option we investigated was IEEE 802.11. The advantages over radio links were clear - it was easier to set up and use in in terms of program code, with many common protocols working automatically thanks to support and drivers for the linux kernel. In addition to this, there are many easily available USB adapters that work out of the box with the Raspberry Pi and Raspbian (a spin off of the Debian Linux distribution). This comes at the cost of the ability to make as many decisions at the lower OSI levels, but on balance this would make the process of developing and testing the deployment much easier. We concluded at this point that WiFi was preferred as a communication method over radio.

Ethernet

At this point the net was cast wider, in an attempt to see if there were any other solutions to the problem that were no immediately obvious. At this point we realised that the assumption that the micro-controllers would have to be mounted on the quadcopters themselves was a false one. Not only would having the Raspberry Pi units situated on the ground make flight more stable by reducing weight, it would also mean that we could use Ethernet to connect nodes together. This also sidestepped the issue we were encountering where the model A Raspberry Pis that the Department had given us only had two USB ports. Using two of these for WiFi adapters would have left no room for a keyboard. While not an insurmountable problem, it would have made debugging in particular more difficult than it needed to be. For the reasons above, as well as the aforementioned lack of necessity for accurate real world deployment conditions (due to the lack of budget) we chose Ethernet as the means of connectivity between nodes.

9.2.3 Sensor Data

We also came across the problem of how we would deal with nodes sensing the environment. This largely came down to the decision to install sensors on the quadcopters or to have this data simulated as it would be done in a simulated environment.

Installing Physical Sensors

The best way of testing how a real life deployment would perform would be to get real life sensor data from mobile units as a program is running. There were, however, a number of problems with this approach - because it required a processing unit to be mounted on the quadcopter to read

the sensor data, choosing it ruled out Ethernet as a communication option. Another problem was that real data values are harder to reason about in terms of software correctness. Especially for a trial deployment such as this one, not being able to repeat experiments (even on the same day) and expect the same results was a real problem. The final issue with this method was that the temperature sensors provided to us by the department came only with a datasheet. A large amount of time would have had to be spent in order to get the sensors soldered correctly and interfacing with the Pi. Even then, anecdotal experience with these devices has shown that they can be finicky and inconsistent, especially given that they were almost certainly not originally intended to work with a Raspberry Pi.

Synthetic Data

The alternative to the above was to use the same synthetic data which we had used in the simulated versions of our programs. This had the benefit of keeping some part of the experiment simple while we measured the effects of other variables on the performance of octoDrone (such as wind speed). As the project matures it will make sense to switch to the use of real data at some future juncture, but that is outside the scope of the project in its current state. What this decision did allow us to do was to test a number of edge cases on the real quadcopters which would have only become apparent otherwise after hundreds of hours of testing. This approach of fixing some aspects of the deployment whilst experimenting with others has undoubtedly saved many hours of testing and debugging over the past few months.

9.3 Adapting the Simulator Code

With the equipment selection locked in (Parrot AR 2, Raspberry Pi model A, Ethernet connectivity and no sensors), we set about adapting the simulator to run distributed across a number of hosts. A large amount of thought had been put into the design of the simulator from the beginning to make this possible, but actually making it a reality was a difficult task. Certain pieces of information could be safely used in a distributed application, such as the locations of individual nodes. Nodes cannot access the locations of other nodes in the simulation anyway, and there is no need to check positions for collisions as this can safely be delegated to physics.

Other properties generate problems when run as part of a distributed application - the environment iterates over the list of messageables when broadcasting messages. We therefore needed to find a way to disseminate packets without knowing the other nodes present in the network (supplying each node with a list of other nodes and their addresses would be missing the point by a country mile). Movement also becomes problematic when one realises that mobile units will often move at a given speed for a given amount of time, as opposed to the speed/direction instructions that programs use to make calls to Drone::move.

In order to attempt to solve these problems we first had to decide how the simulator could be

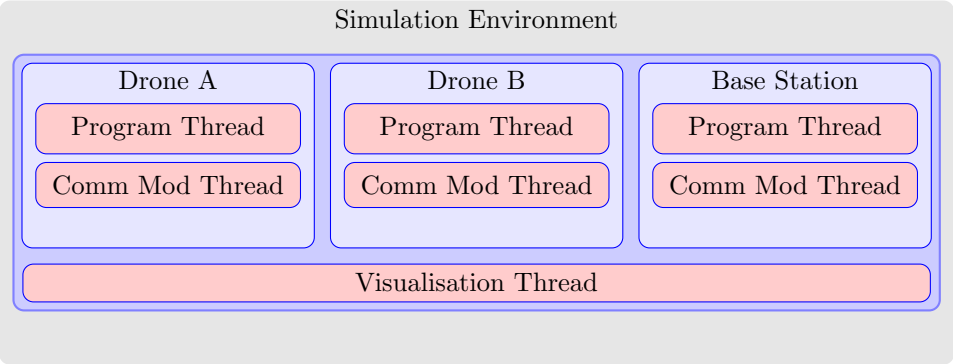

Figure 9.1: Thread diagram for octoDrone simulations

run on multiple hosts. By far the simplest way to do this was to run individual "simulations" on each physical machine and have these communicate somehow. Users would use the same simulation structure as for conventional use (an environment with drone programs and communications modules), but each environment would contain only one messageable. This meant that each hardware unit could run an individual simulation, and that we could hook in to the functions exposed by the simulator API. As such, each time a a call was made to a function that required a call to hardware (such as sending messages), this could be intercepted and run on hardware without any change to the simulator API with the exception of instantiation, which necessarily required information on network interfaces.

It was useful to visualise how the sharded environment would look - figure 9.1 shows the thread structure of a simulated octoDrone run, which can be compared to the distributed version in figure 9.2. Note that the extra threads present in the distributed version of the software hook into the functions exposed by the simulator (a call to Environment::broadcast actually ends up going to the Comm server thread to be broadcast. This meant that the difference between compiling a simulation and a deployment application is as simple as linking against a different library.

9.3.1 Sending and Receiving Messages

Because nodes must communicate with each other over a network (in this case Ethernet), they must be able to actually send packets to each other - a connection must be established between them. This introduces a problem in as much as nodes cannot know the addresses of other nodes in the network (as determined earlier in this section) they must use multicast or broadcast. Broadcasting is effectively deprecated in modern networks, with IPv6 removing it entirely. Add to this the fact that some devices will refuse to let broadcast packets cross the device boundary, it became clear that multicast was the only viable choice. In terms of implementation, raw C sockets were chosen due to their simplicity and speed. It would have been possible to use a wrapper on top of these (such as Boost), but would not have alleviated a significant amount of

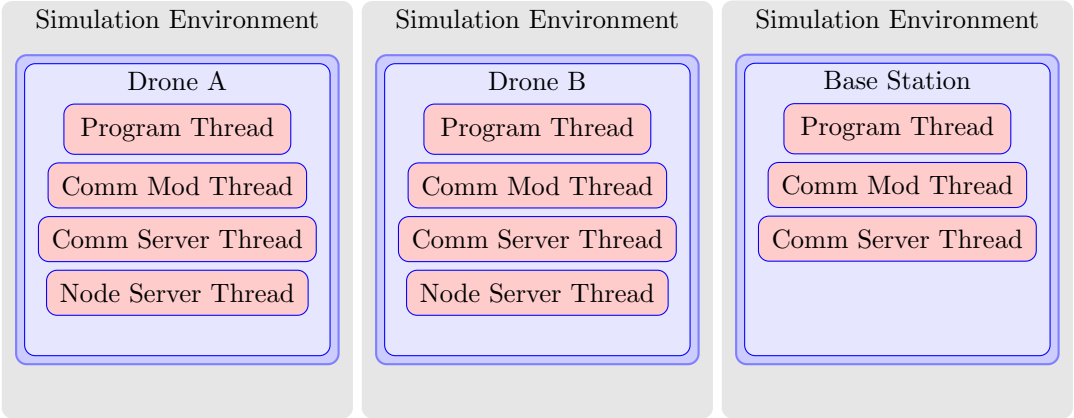

Figure 9.2: Thread diagram for octoDrone deployment

work during development and would not have made the application cross platform unless the threading library used had been changed from the linux POSIX threads (pthreads) library.

Because the receiving of messages was driven by the sending of messages in the original simulator, these actions had to be decoupled for the distributed version. Because listening for an incoming connection is a blocking action it would require another thread to be run whilst the remainder of the application was executing. This thread would take a reference to the local messageable on instantiation that would later be used to deliver messages. Because the application would multicast packets, any outgoing messages would have to be sent to a multicast address that was known to all nodes (specified during the compilation of either the environment library or the simulation).

9.3.2 Movement

As there are no native C or C++ bindings (or indeed, any bindings) for the Parrot AR 2, making the quadcopters move involves using the node js package *ardrone*. The *ardrone* package handles connecting to a drone on the same network as the device it is run on (parrot drones use a default IP address, making them easy to find), as well as sending packets to the drone which it can understand. We created a javascript server application that could be run alongside the distributed simulator that would take commands and use the package to transmit them to the drone to be carried out. This server was passed messages by the environment through a UNIX socket in the filesystem of the host device.

As mentioned above, there was an issue with how movement would be translated from the speed, distance format for used by the simulator and the speed, time format used by the drones. The first attempt at solving this problem assumed that the application would have easy access to GPS coordinates from the paired drone, and used this to perform location checks in the same way

as for simulated runs. It was later learned that there was only one GPS unit available for the two quadcopters obtained, and extracting information from it at runtime was non-trivial. The fallback plan developed involved estimating the time taken for a drone to travel a given distance based on the normalised speed which the *ardrone* library requires. This had to be based on experimental evidence and was not consistent (see notes about drone movement in section 9.3.2).

Quadcopters move by inclining themselves relative to the horizon. This is achieved by having two rotors spinning clockwise, and two rotors spinning anticlockwise. Moving forward entails having one of the rotors which is spinning in a certain direction (say clockwise) move faster and the the opposing rotor of the same direction move slower. This has the effect of tilting the drone on the Y axis (see figure 9.3). With the drone thus inclined, the air displaced by the rotors now causes it to move forward. The same technique can be used in reverse to move the drone backwards, and applied to the rotors spinning in the opposite direction (in this case anticlockwise) to bank the drone left or right on the X axis.

This method of movement results in a period of acceleration at the start of movement as the drone inclines to its target tilt. This means that it takes a few seconds to reach a stable speed, unlike with a ground based robot where the period of acceleration is much shorter. Tilting at too steep an angle will cause the drone to flip (or cut out if there is a hardware tilt-meter and firmware support). Stopping follows a similar transition period as the motor speeds are brought back in sync.

Finally, a drone is turned on the Z axis by increasing the rotation speed of a pair of rotors operating in the same direction and altitude is controlled by increasing or decreasing the rotational speed of all motors in unison.

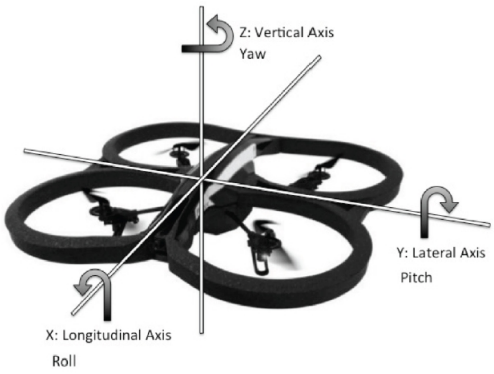

Figure 9.3: Pitch, yaw, and roll in the context of a quadcopter[7]

9.4 Results

9.5 Review Against Original Objectives

CHAPTER 10. Testing

10.1 Unit Testing

10.2 Client Application Testing

10.3 Usability Testing

10.4 Performance Testing

10.5 System Testing

10.6 User Acceptance Testing

10.7 Risk Assessment

CHAPTER 11. Project Management

CHAPTER 12. Project Outcome

CHAPTER 13. Conclusion

Bibliography

[1] Lee Breslau, Deborah Estrin, Kevin Fall, Sally Floyd, John Heidemann, Ahmed Helmy, Polly Huang, Steven McCanne, Kannan Varadhan, Ya Xu, and Haobo Yu. Advances in network simulation. *Computer*, (5):59–67, 2000.

[2] Jim Wright Chris Cole. What are drones?, jan 2010.

[3] Mani B. Srivastava Curt Schurgers. Energy efficient routing in wireless sensor networks. Technical report, University of California at Los Angeles (UCLA), 2010.

[4] Joseph Dussault. 7 commercial uses for drones, mar 2014.

[5] Marcus Faires. Domino's launches world's first driverless pizza delivery vehicles, apr 2015.

[6] U.S. Air Force. Mq-9 reaper, sep 2015.

[7] John Paulin Hansen, Alexandre Alapetite, I. Scott MacKenzie, and Emilie Møllenbach. The use of gaze to control drones. In *Proceedings of the Symposium on Eye Tracking Research and Applications*, ETRA '14, pages 27–34, New York, NY, USA, 2014. ACM.

[8] National Instruments. What is a wireless sensor network?, may 2012.

[9] Kamin Whitehouse Luca Mottola, Mattia Moretta and Carlo Ghezzi. Team-level programming of drone sensor networks. Technical report, SICS Swedish ICT, 2014.

[10] Khanh Pham Robert Sivilli, Yunjun Xu. Pathfinding for mobile sensor networks on the fly. *SPIE Newsroom. DOI: 10.1117/2.1201209.004486*, oct 2012.

[11] Parrot SA. Ar drone 2 techincal specifications, 2012.

[12] Quest UAV. Uav sensors, jun 2015.