Robust Ground Control System Development for Unmanned Arial Vehicle Collision Avoidance

Craig Fisher
Colleges of Arts and Sciences
University of Scranton
Scranton, Pennsylvania
Email: craig.fisher@scranton.edu

Jonathan Taylor
School of Software Engineering
Auburn University
Auburn, Alabama
Email:jzt0011@auburn.edu

Dr. Saad Biaz
Associate Professor,
Computer Science and Software Engineering Department
Auburn University
Auburn, Alabama
Email:biaz@auburn.edu

***Abstract*—The rise in Unmanned Aerial Vehicle (UAV) technology necessitates the growth and exploration of algorithms to prevent UAV collisions. Alongside this exploration, software must be developed for testing these algorithms under real-world computational and software constraints. The software discussed in this paper takes the form of a Ground Control Station (GCS) that manages both real and simulated UAVs as they attempt to complete missions and avoid each other. We discuss the software designs decisions we used to improve an existing GCS that is built on the ROS framework. The design and implementation follow an agile process to provide a modular and efficient framework for the needs of UAV collision avoidance research. This procedure was carried out using the design principles of low coupling and high cohesion. To this end, we refactored an existing GCS to reduce its interaction complexity and to strongly define each modules role and definition.**

I. Introduction

With the rise of Unmanned Aerial Vehicles (UAV's) for military and civilian uses, there will be a strong desire to manage many aircraft with minimal human interference. As more UAVs are flown simultaneously, one very real danger is collision with other UAVs. Thus, any truly autonomous UAVs will not only need to complete their own missions. They will also need to run their own algorithms for collision avoidance. Alongside the development of these algorithms, software needs to be developed that can reliably test the algorithms within the context of a larger Collision Avoidance software system. To be valid for real-life use, new algorithms need to be tested both as valid mathematical formulas and as realistic operations in a real-world setting with real-world constraints.

To help test Collision Avoidance for UAV systems, we have worked on improving a Ground Control System (GCS) for managing both real and simulated UAVs. The GCS in question was originally built as part of the Auburn University Aerial and Terrestrial Testbed for Research in Aerospace, Computing, and maThematics (ATTRACT) Project. The goal of our work has been to make the system more robust through proven methods of Software Engineering.

The GCS in question has been built on top of the Robot Operating System (ROS) framework. The ROS framework supplies many features expected from an operating system, including hardware abstraction, multithreading, and message-passing. Projects written using ROS are known as ROS packages, which are written in C++ and Python. The functional units of any ROS packages are called nodes, and they represent processes within the operating system. They will have their own separate main methods and will communicate with each other via a hidden Master Node. In ROS, there are two main ways for nodes to communicate services and topics. In services, ROS nodes can either be a client or a server for the service. In the process, a message is sent from the client to the server, and once the server has completed handling the message, it gives a response to the client. Services are intended for one-to-communication between nodes. For many-to-many communication, ROS provides topics and nodes can either publish to them or subscribe to them. If a node is publishing to a topic, it is repeatedly sending messages whether or not any other node is receiving them. If a node is subscribing to a topic, it is waiting to accept every message it can get (whether or not there are any) and then acting on those message in a callback function. [4] In the context of the GCS, the ROS system allows the system to smoothly divide the labor of managing planes, responding to user input, and running a simulation. The exact division of labor through nodes and classes will be discussed next.

July 17, 2013

II. Architecture

LEGACY GCS: ATTRACTs legacy GCS was designed to produce courses that could run collision avoidance algorithms on real and simulated planes. To that end, a GCS of five processes was developed. GuiInterfacer interfaced the GCS with user input. Coordinator received GPS data and organized planes to be used by other modules. CollisionAvoidance calculated avoidance maneuvers given GPS data. Simulator synthesized GPS coordinates for fake planes. And, Xbee sent and received GPS coordinates over wireless communication. The legacy design and its implementation were flawed. GuiInterfacer was responsible for starting the GCS when it should have only linked the GCS to a specific GUI. GuiInterfacer also only allowed for a single course to be run during GuiInterfacers execution. Coordinator had too many responsibilities including plane creation, plane management, and command creation, portraying itself as a god class. CollisionAvoidance created its own plane object using information kept in Coordinators separate but redundant plane implementation. Simulator created SimPlane objects which held redundant data found in Coordinator and CollisionAvoidance. Simulator also assumed planes were flying at fixed speeds and functioned correctly only when GPS updates were received at constant intervals.

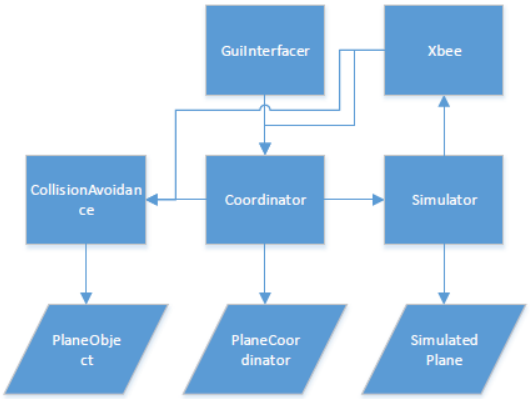

Fig. 1. Legacy Class Diagram

Because of these flaws, the testing of collision avoidance algorithms was difficult. As a result, the legacy GCS is being refactored to reduce the GCSs unreliability and produce an extendable application that allows for easier evaluation of collision algorithms.

REQUIREMENTS: Through collaboration with the collision avoidance research teams, a cursory list of features was created that characterized a more modular and efficient GCS.

- GCS internals to be a standalone application that was not reliant on a GUI
- Ability to load multiple courses without having to exit the application
- Dynamic allocation of plane identifiers
- Simulator that could change GPS update frequency and simulation speed

Based on analysis of the existing GCS, it was determined that an entirely new architecture would be needed to implement these functions.

NEW GCS: It was decided then that low coupling and high cohesion would be the principles to guide the new GCSs modeling process. The first iteration consisted of a MasterNode class and the Coordinator. The aim of the MasterNode was to eliminate any connection the GCS had with outside environments so that its functionality was not defined by any other controller. We encapsulated control of the GCS into services whose client would be a MasterNode. This design allows for a GUI interface without compromising the portability of the GCS and to ensure that the GCSs integrity is not compromised.

Coordinators functionality was also changed. Coordinators previous responsibilities were plane creation and plane management. Having two roles is not excessive; however the complexity of these roles warranted a level of separation between them. Plane management is an involved process and is the foundation of the GCS. Using the same object to create planes that also manages their complex behavior can increase coupling if their functionalities are not kept separate. We decided to delegate plane creation into the class PlaneBuilder using the builder pattern [11]. Previously, plane creation involved only parsing information from course files. We expanded this feature to allow plane creation and waypoint allocation mid execution.

Because planes were being created and managed separately, we could take a more critical look at plane objects . We did not want to have redundant information being stored in three separate places. Coordinator had been remodeled with reducing redundancy between CollisionAvoidance and itself. Now that PlaneBuilder was the only one responsible for a planes creation, we could define common plane classes, SimPlaneObject and PlaneObject, that all modules would be expected to use.

CollisionAvoidance was the next process to refactor after the core components were implemented. In the legacy GCS, CollisionAvoidance would process GPS data drawn from shared memory. Once it received a GPS update, a plane object was updated, then CollisionAvoidance asked Coordinator for information about all other planes, and finally decided necessary avoidance maneuvers. Because Coordinator also processes the GPS coordinates, CollisionAvoidance was changed from a process to a library module. CollisionAvoidance now calculates avoidance commands from inside of Coordinator where it has access to plane data. CollisionAvoidance was not cohesive in its previous state; it had multiple roles that did not benefit itself or the rest of the GCS.

With the foundation of the GCS thoroughly defined, Simulator and Xbee were easy to integrate. Simulators roles and cohesion were not problems in the legacy GCS; its problem was a lack of flexibility. Simulator depended on a constant calculation frequency for each simulated plane; every plane assumed that it was simulating every one second. This was not the case. Even though frequency was close to a calculation a second, it was never exact. Also, testers wanted to be able to not only change the frequency at which each simulate plane updated but to also change the simulation speed. To this end, Simulator was first updated to interface with SimPlaneObject. Secondly, the calculation loop was updated to be able to change its frequency and to calculate plane simulations based on actual time passed since its last execution. This way regardless of the frequency of GPS updates, the plane accurately calculates its position.

Xbees only function was to read and write from a wireless port. The legacy code initiated reads from a thread spawned at the beginning of execution. It would then handle writes in a callback when commands were written to a buffer. We decided to break Xbee into XbeeIn and XbeeOut. The motivation for breaking Xbee into two processes was that Xbees read thread was using a deprecated library. The key features of the newly implemented GCS are to provide an extendable framework for research in collision avoidance to go unhindered by faulty systems and error prone designs. To this end, we developed an interface for system management that allows driver programs to be customizable to the research needs. We created extendable collision avoidance frameworks through modular plane creation, plane simulation, and avoidance calculations. Lastly, we implemented a list of robust features to allow for dynamic course creation and evaluation.

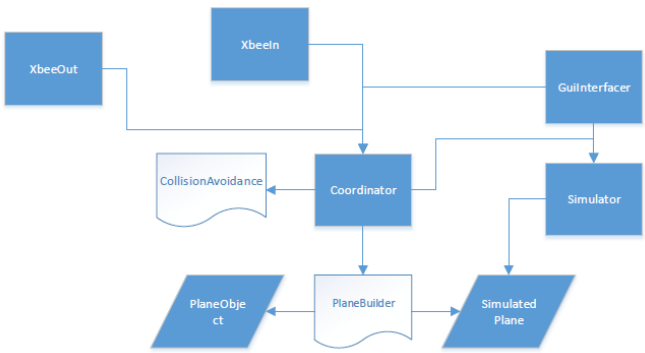

Fig. 2. Current Class Diagram

III. System Analysis

Before we begin a discussion on the functionality and nuances of the system, we feel it important to discuss the overarching dynamics that we hope this system can provide. As stated, the purpose of this system is to test and develop collision avoidance algorithms. To that end, the system should be extendable and modular. For example, a decentralized algorithm should be able run and not get interference from the centralized collision avoidance module. Also, when a user develops a new algorithm he should not have to question or toil to determine its integration into the system. Having to recompile the system to add a new collision algorithm should not be necessary. The framework for adding algorithms or new data representations of planes should be possible during runtime. These dynamics and possible interactions were the driving force during development.

Coordinator

We begin our analysis of the system with Coordinator, the most involved and central object in the system. Coordinators role is to manage plane data and provide this information to working modules. Managing plane data adding and removing waypoints, adding and removing planes, and passing plane information to collision avoidance modules, Xbee communication, and the simulator. We separated these responsibilities into creation and management. Coordinator fulfills the managerial duties and PlaneBuilder creates all plane data. Coordinator uses several data objects to manage plane data. It has a map of PlaneObjects and SimPlaneObjects whose key is the planes identifier. Coordinator also contains a list of the identifiers for all newly created real planes that have not been assigned their data. The list is required so that real planes do not have to be in the air when a course is loaded. A course can thus be created without having to have real planes in a predefined position. Coordinator processes incoming planes so that the fill the empty slots created in the course file. When a telemetry update is recorded from a real plane with an unknown identifier, the plane is sent a command to change its plane identifier to the first identifier in the list. If the list is empty and a new plane sends telemetry, we assign that plane a new identifier but provided it no waypoint information. Instead, we tell it to loiter at its current position. The loiter feature is in an experimental state currently. When a plane starts up, its GPS is not correct; this causes the planes first telemetry updates to broadcast its position at 0 degrees latitude and 0 degrees longitude. To properly implement this feature, this initialization data needs to be ignored or else the plane will attempt to loiter off the coast of Africa. Coordinator uses two data objects, Commands and Telemetry, for inter-process communication. Currently, Commands are used to issue collision avoidance maneuvers and change incoming planes identifier. The Command structure contains a header, plane identifier, simulation flag, command identifier, parameter, latitude, longitude, and altitude. The header itself contains a timestamp and a sequence number. Currently, both these variables are unused. They potentially used to verify packet loss and duplicate plane identifiers; this will be explained in a latter section. The simulation flag denotes if the plane is a simulated plane or a real plane. The command identifier is the index of the command that is being sent. There are four possible commands, COMMAND_NORMAL_WP, COMMAND_AVOID_WP, COMMAND_MANEUVER_WP, and COMMAND_SET_ID. The parameter is only used when a COMMAND_SET_ID is issued. Parameter is the plane identifiers new identifier. For example, if a new plane has the identifier 7 and we want to change that number to 4, the plane identifier would be 7 and parameter would be 4. Commands with the simulation flag cleared will be sent wirelessly to the planes by XbeeOut. Commands with the simulation flag set will be sent to Simulator. Telemetry is simply a status update from a plane. The values in Telemetry include a header, plane identifier, current latitude, current longitude, current altitude, destination latitude, destination longitude, destination altitude, ground speed, air speed, target bearing, waypoint index, and distance to destination. Telemetry is broadcasted by XbeeIn and Simulator. Coordinator receives the broadcasted telemetry and uses it to update plane data that can then be given to Collision Avoidance. Coordinators typical flow of execution is to load a course file containing planes and their waypoints. Then, Coordinator will call PlaneBuilder to create objects from this course file. PlaneBuilder will populate Coordinators list and return the necessary commands needed to begin execution of the course. With this information, Coordinator will pass along any simulated plane data to Simulator and issue commands to both real and simulated planes. Finally, Coordinator will listen for the planes telemetry and update Coordinators member variables. After the update, Coordinator will call CollisionAvoidance. CollisionAvoidance then calculates avoidance maneuvers that Coordinator will issue to the planes. Coordinators most crucial function is to provide planes with their waypoint information. When Coordinator receives a telemetry update and the plane has arrived at its destination within a certain threshold, Coordinator issues the plane its next waypoint. Real planes are only given one waypoint; if they do not receive another after reaching their destination they will circle that waypoint. The need to issue the next command is not mirrored in simulated planes. Simulated planes hold all of their waypoint information and do not require Coordinator to tell them their next waypoint.

PlaneBuilder

Coordinator delegates plane creation to PlaneBuilder. We chose to abstract creation in to a separate object to keep Coordinator cohesive. Removing creation helped to keep Coordinator focused on plane management. When developing PlaneBuilder, we had to ensure that Coordinator remained

removed from the creation process. In an early iteration of PlaneBuilder, Coordinator had an interactive role in creating plane data; this kind of relationship defeats the purpose of abstracting creation. To combat this, the current PlaneBuilder handles all decisions on plane creation. The data is directly populated in Coordinator because PlaneBuilder is a friend class for Coordinator. PlaneBuilder currently provides Coordinator with four features, build a real plane, build a simulated plane, load a course file, and allocate a plane identifier. Building planes requires that a waypoint is associated with that plane. When a request for a plane is made, PlaneBuilder checks to see if there are any available identifiers. The algorithm assigns it first plane the identifier 2 and increments that value for subsequent requests. The max value for a plane identifier is 255. Our reasoning for starting with 2 is that ArduPlane defaults a planes identifier to 1. The algorithm is flawed in its current state because this assumption is not guaranteed. When we change a planes identifier, the plane saves that value to EEPROM instead of storing it only in RAM. Because of this, new planes that have been previously used by our system do not have a plane identifier of 1. The problem arises when we have two planes with the same identifier. A potential solution to solve this dilemma is to use the Telemetrys header index to calculate the discrepancy between Telemetry updates. Loading a course file is the most involved process in PlaneBuilder. PlaneBuilder parses the data in the course file and instantiates all the data. All real planes created in the course file are added to the new plane list mentioned earlier. The course files syntax is as follows, Plane-identifier latitude longitude altitude simulated-flag Each line represents a waypoint for that plane. The first waypoint added will be used as the starting location for simulated planes. The simulated-flag is a 0 for a simulated plane and 1 for a real plane. After a simulated plane is created, Coordinator sends Simulator a copy of this information. The reasoning for this redundancy is that Simulator will be simulating the simulated plane objects frequently. It would be expensive for Simulator to request plane information at the outset of every simulation cycle.

CollisionAvoidance

CollisionAvoidance is a sub system that is currently regulated by Coordinator. CollisionAvoidance is the focal point for extendibility. Currently, CollisionAvoidance requests the identifier of the latest update and a map of all planes. It returns a list of avoidance waypoints that need to be made. The current implementation is sufficient for reactive algorithms like RIPNA but does not handle proactive algorithms well. Avoidance waypoints are only for imminent collisions and are not suitable for path planning. An additional list should be returned from CollisionAvoidance that contains path planning waypoints. After much deliberation, it might be useful to move CollisionAvoidance to its own process. We recommend this so that the computational characteristics of an algorithm can be measured or observed independent of the frequency of Telemetry updates. Other improvements include implementing plugins for CollisionAvoidance and fleshing out decentralized algorithm support. The benefits of plugins include minimal recompilation time and a strict structure for collision avoidance algorithms to follow. Currently, Coordinator handles the switch between centralized and decentralized algorithms. If CollisionAvoidance were to become its own process, CollisionAvoidance would need to be able to respond to a switch from decentralized to centralized or vice versa.

PlaneObject

PlaneObject is the data representation of a plane. PlaneObject holds the list of remaining waypoints, the next avoidance waypoint, the planes previous location, and the plane current location. It is important to note that a plane must always have a waypoint. If a plane has reached its final destination, the last waypoint will not be removed. No more commands will be sent to the plane but the last waypoint will remain in the list. Previous location and current location are used to calculate the planes bearing so that it can be available for collision avoidance algorithms. There are two important functions that PlaneObject provides, update and getPriorityCommand. Update is the function that gets called when Coordinator receives a Telemetry update from that plane. This function takes that Telemetry and updates PlaneObjects members and produces any necessary command that Coordinator should issue. GetPriorityCommand provides the most urgent waypoint for that plane. The precedence is, avoidance waypoints are the most urgent, maneuver waypoints are the second most urgent, and normal waypoint are the least urgent. GetPriorityCommand is used to issue Commands when planes are first created and GetPriorityCommand is used in Update if a plane has reached its current destination. PlaneObject has room for improvement. Path planning/proactive algorithms provide a detail path for a plane to follow. This path cannot be realistically implemented only using a single avoidance waypoint and also needs to be separate from normal waypoints. We have added a maneuver list; however this list has not been integrated with CollisionAvoidance. Maneuver list would ideal for crafting a collision avoidance path to its next waypoint. The list would be erased once it reaches its destination. Ideally, a new path would be generated for each normal waypoint.

SimPlaneObject

SimPlaneObject is an extension of PlaneObject that provides functionality for simulation. SimPlaneObjects extensions include simulation speed and a map of all other planes. Simulation speed allows simulations to run at variable speeds. This can reduce the time spent gathering data and testing algorithms. The map of other planes is currently not being used. Its purpose is to allow SimPlaneObject to simulate decentralized algorithms in which they map out their own environment. The simulation algorithm is contained in SimPlaneObject. When Simulator calls SimPlaneObjects simulate function, Simulator provides SimPlaneObject the time since the last function call and SimPlaneObject returns a Telemetry Update. SimPlaneObject adheres to the same rules as PlaneObject for waypoint precedence and waypoint list size. The simulate function is area that we like to see extended. Currently, the simulation algorithm uses the Law of Haversines to determine latitude and longitude. If other techniques were employed the system should support them. This feature could be maximized if the simulation technique was chosen at run time.

Simulator

Simulator controls when SimPlaneObjects simulated their Telemetry. After a SimPlaneObject is created, its data is sent

to Simulator using the service manage_simplanes. Coordinator sends Simulator a list of all planes to be added and their waypoints. This service is also used to add and remove waypoints. After receiving planes, Simulator operates on a timer. This timer is based upon the simulation frequency. A simulation frequency of .5 will trigger 2 simulation callbacks per second. In each callback, Simulator calls each SimPlaneObjects simulate function. The Telemetry that it receives from simulate is broadcasted so that Coordinator can update its copy of the plane and call CollisionAvoidance with the updated state of the planes. Simulator is an important module in the system. It is dangerous to test newly developed collision avoidance algorithms with real planes. Potentially, Simulator can improve by devoting threads to a subset of SimPlaneObjects and allow them to simulate their Telemetry concurrently. Simulator would then function as a process manager that would synchronize Telemetry updates. There is support for threading in ROS via nodelets. We believe that implementing nodelets can add to the Simulators robustness in that the Simulator can provide a more realistic approach to how planes simulate their position.

XbeeIn and XbeeOut

XbeeIn and XbeeOut are the communication processes. XbeeIn receives mavlink messages from planes and converts them into Telemetry, while XbeeOut receives Commands and converts them into mavlink messages. XbeeIn and XbeeOut share the same port. We lock reading and writing to the file descriptor with a mutex. We also ensure that XbeeIn will not hang when no data is available to be read by setting up the port as non-blocking. Currently, XbeeIn starts in a busy wait loop while the port is being set up. A potential solution is to have XbeeIn have a custom callback queue for commands and to address that queue in serial_wait. Improving XbeeIn and XbeeOut can be done by processing acknowledgements sent from the planes. The framework for acknowledgements is already established; however, they are not being used to ensure that data is not lost. A solution we discussed was to keep record of sent commands in XbeeOut. When XbeeIn receives a mavlink message for an acknowledgement, we communicate with XbeeOut that an acknowledgement was received. If XbeeOut does not receive an acknowledgement within a certain time, XbeeOut will resend the command. This problem becomes complex because we do not have a way to uniquely identify which command the acknowledgement belongs to.

MasterNode/Interfacing

To interact with the system, services have been made available for each feature. Adding and removing planes are a service offered by Coordinator. Setting the simulation speed or the simulation frequency are services offered by Simulator. We chose the client server model to make it easy for a user interface or an automation program to interact with the system. Essentially, a master node can establish a link with the system through these services. Please see the documentation on the specifics.

IV. Performance and Reliability

Since the GCS will be used as a platform for testing Collision Avoidance algorithms, there is a great need to ensure that its behavior matches the users expectations. A poorly performing GCS could produce arbitrary results that do not reflect the performance of different collision avoidance algorithms. Thus it would be very difficult to replicate experimental results. Proper behavior includes both performance and reliability issues. Concerning performance, the system must be to handle potentially large amounts of data processing while still producing accurate, informative results for the user. Concerning reliability, the GCS must respond the way the user thinks it should respond. This does not simply mean that the GCS should have few or no bugs. It also means there should be no ambiguous behavior. For example, a function such as set sim speed could either be interpreted as a fast forward function or as a function that merely increases the air speed of simulated planes. In a mature GCS-system, all these behaviors must be well-defined and meet their definitions.

Overview of Software Testing Strategies

In order to inspect the performance and reliability of our system, we have made use traditional software testing models at the level classes, modules, and the entire system. The ROS framework contains built-in functionality for minimally- intrusive testing at most of these levels. Tests are mostly written using the Google Test framework, which provide detailed summaries based on assertion statements. Additionally, ROS provides the Rostest package through which test nodes may be built. These test nodes behave the same way as ordinary nodes except that they are only launched when the testing system is invoked. When used together with Google Tests, Rostest allows for testing of runtime interactions that would otherwise be difficult to test without more intrusive code. Thus, ROSs architecture allows tests to remain mostly distinct from the main source code. We will briefly discuss how we have conceived our unit tests, integrations test, and system tests. At the unit level, testing can be usefully modeled around two test patterns the functional and the state-based. Functional or black box patterns are aimed at classes with specific input and output specifications through which a test can interface. A test oracle then is programmed to know in advance how inputs should produce outputs. Classes such as Coordinator and CollisionAvoidance will be modeled this way, since they are primarily occupied with transfer of data for other objects. In state-based patterns, a class is modeled as if it were a State Machine, and the test seeks to cover all the transitions and paths that this machine can take [4]. For many of the individual GCS classes, State-based models serve as a better pattern for unit testing. Instances of PlaneObject and SimPlaneObject can both enter significantly different states depending on the state of their waypoint queues. For instance, if a PlaneObject does not hold any avoidance waypoints, it should only create new Commands when a new waypoint is reached. However, if there are any avoidance waypoints at all, the PlaneObject should create new Commands constantly. There are many other such state-based behaviors related to PlaneObject as well. The PlaneObject can then be thought of as a state machine whose transitions are the changes in its waypoint queues. The goal of State-based testing here would be to cover all the paths of the PlaneObject. Thus, units here must extend beyond simple input and output. They must trace the effects of multiple inputs. Integration testing will be concerned with the node and module levels of the GCS. For testing

purposes, entire system can be divided into the five runtime executables: Coordinator, GuiInterfacer, Simulator, XBeeIn, and XBeeOut. The integration tests will consist of one or more of these executables along with a driver portion. The drivers can be implemented as Rostest nodes, and they serve to model the input and output behavior of the nodes not being tested. The test for the Coordinator demonstrates many possible features for integration tests. The test is encapsulated inside a test node called CoordinatorTester. It aims to simulate clients for the Coordinator's services and services for the Coordinator's clients. It also publishes artificial Telemetry data that the Coordinator subscribes to while subscribing to the Commands that Coordinator sends back. Unlike bare unit testing, the testing for the Coordinator extends beyond the simple Coordinator class. The Coordinator as a node is currently tightly coupled to the plane-related classes that it helps encapsulate, including PlaneBuilder, SimPlaneObject, PlaneObject and collisionAvoidance. Thus, all these classes are tested as a larger module by the CoordinatorTester. On the system level, testing will focus on stress testing of the GCSs internal communications and data storage. At any given moment, the system must synchronize data on planes, simplanes, waypoints, Commands, Telemetry, and various internal messages. Any final GCS should be aware of its own safe operating limits and enforce them on the user. There are many dangers of overloading the system. Obviously real planes could potentially crash, but even without this threat, simulations and their data could become inaccurate or corrupted. As will be discussed later, the GCS should be able to recognize stress failures so that they are not misinterpreted as errors in the planes, GUIs, or the avoidance algorithms.

Phenomenon of Message Lag

Among system-wide stress issues for the GCS, Message Lag is perhaps the most dangerous. During normal operation, Simulator and XBee are publishing Telemetry updates while Coordinator is handling every incoming update in order. Thus, when Coordinator calls CollisionAvoidance, CollisionAvoidance is running its calculations with the most up to date plane locations. This is what should happen in the ideal scenario where a Subscriber always keeps up with the Publisher. However, if a Subscriber cannot keep up with a Publisher, the ROS Subscriber model does not mandate what will happen to excess messages. In ROS, the user can decide exactly how many old messages they wish to handle. They do this by specifying the Subscriber Queue Size. If the Queue Size is set to 10, then a maximum of 10 old messages can be saved for later processing at any given time. If an eleventh unprocessed message were to arrive, then the oldest message would be dropped and the eleventh message would be added to the Queue. We will call the phenomenon of processing outdated messages Message Lag while the phenomenon of dropping messages will be called Message Loss. In the case of the GCS, there is currently a greater danger for Message Lag than Message Loss. Of course, it should be acknowledge that both Message Loss and Message Lag could be problems. If, for any given a plane, a brand new Telemetry update has just been sent, then ideally we want to ignore all the old updates from that plane. Obviously, if CollisionAvoidance is making decisions based on outdated data, then the Commands it generates will not reflect reality. On the other hand, if every old Telemetry message is dropped (or if its published too infrequently), then the planes whose Telemetry was dropped could often be skipped over and its information ignored by CollisionAvoidances decisions. However, in the legacy GCS, the Telemetry Callback Queues were all given a size of 1000. Thus, Message Lag is more likely to occur in normal operation than Message Loss. Message Lag was never seriously considered in the Legacy GCS. This is at least partially understandable. After all, if the system is using a reliable algorithm and running around the usual simulation speeds, Message Lag almost never occurs. CollisionAvoidance would handle all of its duties well before the next Telemetry message arrived. However, in our experience, GCS users often want to accelerate their simulations and then review the results. This is especially useful when large amounts of test conditions or data needs to be processed. Also, there is no guarantee that future algorithms will operate as efficiently as the RIPNA algorithm that was built into the legacy GCS. During our research, many of our colleagues worked on so-called pro-active algorithms that seek to plan complete avoidance paths rather than send back a single avoidance waypoint. Especially in the case of pro-active algorithms, CollisionAvoidance might need more time but could potentially generate better results. Thus, there are many situations outside of normal conditions where Message Lag could become a problem. Our goals have been to track this problem and figure out appropriate responses to it.

Test Methods and Results for Message Lag

In order to test Message Lag in the system, we have setup a tracking system between Simulator and Coordinator. Ideally, we would simply keep track of the Queue Size for the Coordinators Telemetry Callback function. However, since this information is hidden from us within the ROS internals, we instead created a system based on message confirmation. In the system, Sinulator keeps track of how many messages it has sent. Then, every time Coordinator handles a new Telemetry message, it publishes the number of messages it has handled. The Simulator then Subscribes to this confirmation messaging, and whenever it gets a new confirmation, it subtracts Sent minus Confirmed to get the current Queue Size. As Simulator calculates this statistic, it may then handle it however the user specifies. As we have discovered through trial and error, Message Lag seems to behave as a function of several variables in the GCS architecture. First and most obviously, there is a greater chance of Message Lag if there are either many planes or if the Simulator is being run at a high frequency For our purposes, frequency here acts as a multiplier for plane speed and telemetry frequency where a frequency of 1.0 means real-life (and so 5.0 would mean five times real life). In addition to these measures, we also found that there will be much more Message Lag when there are many waypoints given per plane. Thus, we have included Waypoints Per Plane as a third testing parameter. Our intention here was only to demonstrate the existence of Message Lag and to give rough guidelines as to when it occurs. Since this data may change quickly with modifications to the software, we did not aim to gather rigorous statistics. The results of our data collection can be seen in figures 3-6.

Frequency	20 Planes	25 Planes	30 Planes
30	11	overflow	overflow
20	3	24	132
10	0	0	0

Fig. 3. Message Lag for 5 waypoints

Frequency	10 Planes	15 Planes	20 Planes
30	0	37	overflow
20	0	3	130
10	0	0	0

Fig. 4. Message Lag for 50 waypoints

Frequency	20 Planes	22 Planes	24 Planes
30	199	44	overflow
25	3	70	44
20	1	0	3

Fig. 5. Message Lag for 5 waypoints

Frequency	15 Planes	17 Planes	19 Planes
30	1	19	21
20	0	0	1
10	0	0	0

Fig. 6. Message Lag for 50 waypoints

Possible Response to Message Lag

Since many factors can affect Message Lag (even beyond the ones we tested), users should not rely on simple guidelines for running their algorithms safely. Rather, the current tracking system can easily be extended to generate appropriate responses. Currently, we have considered two possible responses to Message Lag. In the first approach, we simply notify the user that Message Lag has occurred and pause the simulation. This will allow the user to make their own decision about how fast the simulation should run, how many planes or waypoints they should use, or whether to modify their algorithm. In the second approach, the system will be allowed to slow the simulation down but only to a certain point if Message Lag begins to occur. We can imagine that a user may have decided to save time by running their algorithms many times faster than normal real-life constraints. If their algorithms fail to keep up, this does not imply that their algorithms cannot meet their demands. It may simply mean that the simulation was too unrealistic. Under this scenario, the system would automatically slow down the simulation until Message Lag is no longer a problem, but not to a point that the algorithms are given an unfair amount of time. Naturally, the simulation minimum speed limit would have to be configured by the user. Beyond simply responding to Message Lag problems when they occur, more deeper changes could also be made to regulate the problem in adance. In terms of Collision Avoidance, the old system gave users too much lee-way. In the old workflow, the design of the GCS implied that users would create their own Collision Avoidance nodes. In addition to specifying their own algorithm logic, users were free to write their own timers, scheduling logic, and callback functions for Telemetry messages. In other words, users could control the data flow and strain that their algorithms had to handle. One group even took this freedom to the extreme and wrote two separate Collision Avoidance nodes: one they claimed was optimized for simulations and one they claimed was optimized for real planes. For their simulation CA node, the Telemetry callback would not call the avoidance algorithm for every Telemetry update. Instead, it would only call the algorithm once a complete cycle of updates had been received for all the planes. Meanwhile, for the real plane node, the callback would call Collision Avoidance every time a Telemetry message was received. By allowing this, the old GCS allowed for the possibility of unfair comparisons between algorithms. If an algorithm is handling updates less frequently than it can or handling old updates rather than new ones, it will not perform to its true potential. However, while the new GCS enforces predictable use, it does not yet allow enough flexiblility to meet the needs of many styles of algorithms. Users should be restricted as to how often their algorithms handle updates. If anything, they should be given greater freedom. When they do schedule updates and responses though, their scheduling should be explicit and lead to predictable behaviors in the system.

Other Know Issues

In addition to measuring Message Loss, our testing efforts have attempted to find and eliminate other bugs in the system. Many bugs have been found and corrected, but there are still some bugs that are more systemic in nature and will require deeper changes to fix. One such bug is the ambiguity of SimSpeed and SimFrequency. Increasing the SimSpeed alone causes planes to take jumps around the map instead of transitioning smoothly from point to point. As a result, collisions and waypoint-arrivals that would have occurred in the gaps are completely ignored. Increasing the Frequency alone doesnt cause any unexpected behavior. Planes will merely have more opportunities to respond. However, decreasing the Frequency causes planes to move a further distance before reacting to either planes or waypoints. The result is similar to the jumping behavior of increasing SimSpeed alone. Beyond a certain point, planes will never even arrive at their waypoints. Instead, they will shoot past a waypoint, process an update that says they havent arrived, and then fly back at the same waypoint for a similar results.

REFERENCES

[1] Aleti, A.; Buhnova, B.; Grunske, L.; Koziolek, A.; Meedeniya, I., "Software Architecture Optimization Methods: A Systematic Literature Review," Software Engineering, IEEE Transactions on , vol.39, no.5, pp.658,683, May 2013

[2] Babar, M.A.; Liming Zhu; Jeffery, Ross, "A framework for classifying and comparing software architecture evaluation methods," Software Engineering Conference, 2004. Proceedings. 2004 Australian , vol., no., pp.309,318, 2004

[3] Bertolino, Antonia, "Software Testing Research: Achievements, Challenges, Dreams," Future of Software Engineering, 2007. FOSE '07 , vol., no., pp.85,103, 23-25 May 2007

[4] Documentation. ROS.org, [online] 2013, http://www.ros.org/wiki/ (Accessed: June 2013)

[5] Howden, W.E., ”Software Test Selection Patterns and Elusive Bugs,” Computer Software and Applications Conference, 2005. COMPSAC 2005. 29th Annual International , vol.1, no., pp.25,32, 26-28 July 2005

[6] Jiang Guo, ”An approach for modeling and designing software architecture,” Engineering of Computer-Based Systems, 2003. Proceedings. 10th IEEE International Conference and Workshop on the , vol., no., pp.89,97, 7-10 April 2003

[7] Lyu, Michael R. ”Software reliability engineering: A roadmap.” In 2007 Future of Software Engineering, pp. 153-170. IEEE Computer Society, 2007.

[8] Mens, T.; Tourwe, T., ”A survey of software refactoring,” Software Engineering, IEEE Transactions on , vol.30, no.2, pp.126,139, Feb 2004

[9] Musa, J.D., ”Software reliability-engineered testing,” Computer , vol.29, no.11, pp.61,68, Nov 1996

[10] Pietrantuono, R.; Russo, S.; Trivedi, K.S., ”Software Reliability and Testing Time Allocation: An Architecture-Based Approach,” Software Engineering, IEEE Transactions on , vol.36, no.3, pp.323,337, May-June 2010

[11] Vlissides, John, R. Helm, R. Johnson, and E. Gamma. ”Design patterns: Elements of reusable object-oriented software.” Reading: Addison-Wesley 49 (1995): 120.

[12] Yao Chen; Xiaoqing Li; Lingyun Yi; Dayong Liu; Liu Tang; Hongli Yang, ”A ten-year survey of software architecture,” Software Engineering and Service Sciences (ICSESS), 2010 IEEE International Conference on , vol., no., pp.729,733, 16-18 July 2010

[13] https://github.com/ericwestman/sim/blob/master/src/collisionAvoidance.cpp