2015 International Conference on Unmanned Aircraft Systems (ICUAS)
Denver Marriott Tech Center
Denver, Colorado, USA, June 9-12, 2015

Multi-UAV Software Systems and Simulation Architecture

Michael A. Day[1], Michael R. Clement[2], John D. Russo[1], Duane Davis[3], and Timothy H. Chung[1]
{maday,mrclemen,jdrusso1,dtdavi1,thchung}@nps.edu

Abstract— **As unmanned aerial systems (UAS) continue to increasingly require greater integration of sophisticated software systems, developing and utilizing best practices and principles of formal software systems engineering can enhance and ensure the safety, reliability, and performance of these systems. This paper highlights the detailed implementation of a number of such tools, including agile software development methods such as automated software testing, and enhanced simulation-in-the-loop testing for multi-UAS virtual and live-fly capabilities. Significant and tangible benefit to active field experimentation is demonstrated through description of these integrated approaches, impacting ongoing efforts in multi-UAS research, testing, and assessment practices.**

I. INTRODUCTION

Rapid development of unmanned aerial vehicle (UAV) technologies, including accelerating proliferation of low-cost systems, such as autopilots and hobby-grade systems, can benefit from elements of well-established software systems engineering principles. This paper defines several test and evaluation approaches best suited for autonomous system development, specifically UAVs in our case, where software is necessarily integrated with hardware and embedded systems for autonomous and coordinated, behaviors. We also describe alternative methods for setting up a simulation environment in which multiple flight simulators can operate in tandem as well as interact with physical planes in flight.

The context of this research and software development effort is an ongoing initiative by the Advanced Robotic Systems Engineering Laboratory (ARSENL) at the Naval Postgraduate School to field large (50-member) teams of UAVs in simultaneous, live-fly field experiments and competition [1]. This research is motivated by the benefits large cooperative teams of UAVs could have to search-and-rescue, disaster recovery, military, and other domains. Accelerating and enhancing the development and testing of *software* elements stands out as one of the key challenges in such an ambitious endeavor, and strongly motivates the investigation and integration of software systems engineering processes showcased in this paper.

In seeking to define test and evaluation approaches best suited for autonomous system development, specifically teams of UAVs in our case, where software is necessarily integrated with hardware and embedded systems for autonomous and coordinated behaviors, we found it beneficial (if not critical) to follow the software development principles of Agile Development and Continuous Integration (CI). In setting up a series of tests to support a CI approach to software and systems development, we also found it necessary to iterate through different approaches for inter-plane and inter-simulator communication.

Contributions of the research detailed in this paper include the integration of modern-day software engineering principles and tools, such as automated testing, to facilitate agile development of flight-ready software systems supporting autopilot, collaborative autonomy, and human-swarm interface components for a multi-UAV, live-fly field test capability. Additionally, this work extends and enhances previous single-plane simulation-in-the-loop (SITL) capabilities to enable a realistic virtual environment for multi-UAV testing and evaluation of, e.g., aircraft and swarm failsafes, coordination and communication algorithms, and operational concepts and procedures. These contributions have demonstrated to be critical in the accelerated de-

[1]Department of Systems Engineering, Naval Postgraduate School, Monterey, California, USA
[2]Department of Information Sciences, Naval Postgraduate School, Monterey, California, USA
[3]Cyber Academic Group, Naval Postgraduate School, Monterey, California, USA

velopment and deployment of live-fly capabilities for multi-UAV field experimentation. This paper details both the current framework actively used as well as the various iterations in methodology in arriving at the present solution.

Given this paper's contributions described above, the following section provides a primer on the software systems engineering principles embraced by this research initiative. Section III provides an overview of the overall multi-UAV system architecture, including physical and simulation system descriptions. Section IV details the implementation and integration of automated software testing methods employed to accelerate field test capabilities, and Section V describes the enhanced multi-UAV simulation-in-the-loop test configuration and associated design choices. Finally, the combined benefits found by integrating these core software systems engineering tools are summarized in Section VI, which also highlights several lessons learned and areas of future work.

II. Background

A large-scale, live-fly capability for UAVs, let alone the deployment and simultaneous employment of 50-UAV teams, imposes a high degree of cost and complexity. Evaluating the effectiveness of systems as they are developed is complicated by the multiplicity of systems. The challenges of fielding such a system of systems not only include its software and hardware development, but also spans the logistical effort required to field multiple planes during field experiments, the design and development of novel concepts of multi-UAV operations, and obstacles to managing such large teams of UAVs with one or few operators. This paper focuses on the software systems development process, given its central role in enabling collaborative autonomy as envisioned.

Software employed in research environments is often (if not necessarily) new, and therefore may be relatively untested. Though research teams may employ well known algorithms in new ways or implement new algorithms of their own design, such integration efforts, especially in new environments, can be prone to error. Known software engineering principles are available to increase the speed with which errors introduced by innovation are found; these include agile software development and Continuous Integration (CI).

Agile software development is a branch of Extreme Programming [2], [3]. The methodology facilitates rapid feedback from a software system as it is being developed. Simply stated, agile methodologies seek to eliminate or limit long analysis of the requirements of an entire project, and instead break the project into very small sub-projects with many more deliverables [4]. For example, rather than spending a few months gathering requirements for a software project and then spending several months building software to those specifications, only a small part of the requirements are determined and a prototype of the final software is delivered quickly, e.g., within a week, and then the next (small) requirements set is worked on the following week such that the prototype is further iteratively developed. In a business setting, this tends to make software more resilient to changing customer demands; in a research setting, such practices can limit the negative impact of incorrect or unforeseen assumptions made at the early stages of a project. Indeed, one of the major tenets of agile methodologies is that development systems need to expect and respond to change, rather than stress predictability [5]. Certainly, this is also true of research in general.

The benefits of agile development over a traditional software development model, where software is developed in isolation in a laboratory setting for months, are clear for addressing the rapid pace of advances in UAV technologies. Especially in the context of live-fly field tests, such experiments often expose errors that are only realizable through physical systems integration and testing. However, field experiments for UAVs are costly: time must be obtained on an airfield, equipment and personnel must be transported to and from the location, and these logistics problems are exacerbated with multi-UAV experiments where the number of planes being tested in a field test continuously increases. In keeping with the frequent iterative approach (though perhaps not as frequently as a wholly software-based project), the research effort highlighted herein actively and aggressively conducts field experiments on about a four-to-six week cycle. Further, while simulation cannot supplant operational testing, it can greatly augment it, and as such, we employ the simulation environments discussed in this paper to perform testing against prototypes on a significantly more frequent basis.

It should be stated here that while many of the principles of agile development serve to accelerate research, not all of them may fit a given project, as is the case in our multi-UAV research efforts. Though too much requirements analysis can hinder a project, analyzing the problem to be solved before diving into the details of implementation is definitely beneficial so we do not wholly reject doing limited requirements analysis prior to a short development iteration. Further, given the uncertainty of innovating new technologies, we also do not adhere to the idea of self-organizing teams, but rather maintain a manager to oversee our project. We also do not use pair programming as a general practice, though we do employ it from time to time when attempting to debug a particularly complex software implementation or design problem. An in-depth analysis of the tradeoffs to consider when best to employ agile principles is available in the literature [6], [7]. However, many principles of agile development, especially its extension into Continuous Integration, have proven quite useful to our laboratory's efforts.

Continuous Integration (CI) extends the ideas of agile development by requiring central repositories to store software, against which developers can maintain their local code bases and keep them up-to-date (ideally updating daily), and discourages researchers from working in isolation from one another. Further, tests are continuously performed against the current state of the software in the repositories to ensure the system is developing as expected and to see that integration errors are quickly identified and remedied [8]. By introducing this level of automation to the process, the system state can be evaluated frequently, as often as multiple times per day. Further, when a team member wishes to introduce new functionality to the system, these automated tests can be run against proposed modifications before they are brought into the shared repositories. CI has become well accepted and many consider it a best practice in agile software development [9].

III. SYSTEM ARCHITECTURE

This section provides a brief overview of the physical and virtual configurations of the multi-UAV system of systems.

A. Physical UAV System Overview

The *ZephyrII* flying wing UAV leverages the Pixhawk autopilot, which runs the ArduPlane autopilot software [10]. The onboard computer, an ODroid U3, runs a series of Robot Operating System (ROS) [11] nodes which comprise the swarm autonomy software. This autonomy payload software communicates over a physical serial connection with the autopilot to issue navigation commands (e.g., waypoints) and receive updated information (e.g., UAV pose data). The payload software also provides a means of communication between UAVs in the swarm, and allows individual swam members to autonomously make navigation decisions based on the states of their neighbors. To date we have a basic leader-follower capbility implemeneted though other behaviors such as formation flight and Boids flocking are in development. Currently we ensure separation between swarm members using altitude separation. If interaction with an individual UAV becomes necessary, a redundant communication path is available via the 3DR serial modem radio. In such cases, the single-plane MAVProxy ground control station [12] can then connect directly to the autopilot, that is, circumventing the payload link, leveraging a standardized communication protocol called MAVLink [13].

B. Simulation Architecture Overview

Though UAVs comprise much more than software, simulators can facilitate the evaluation of UAVs' software components rapidly, and even identify areas where sensors or other hardware might best benefit from improvement. Indeed, the role of modeling and simulation for complex systems, such as multi-UAS field experimentation, is critical, if not necessary, to avoid costly and time-consuming development otherwise.

During typical flight operations, environmental status data required by the autopilot to determine control inputs are available via sensors on board

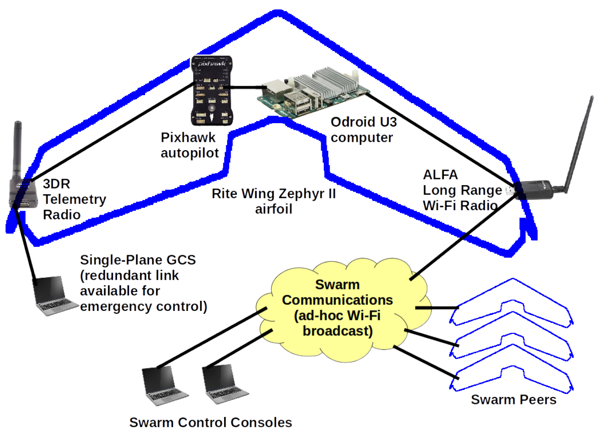

Pixhawk autopilot
Odroid U3 computer
3DR Telemetry Radio
Rite Wing Zephyr II airfoil
ALFA Long Range Wi-Fi Radio
Single-Plane GCS (redundant link available for emergency control)
Swarm Communications (ad-hoc Wi-Fi broadcast)
Swarm Peers
Swarm Control Consoles

Fig. 1. Illustration of a single UAV hardware configuration, including the flying wing airframe, autopilot, onboard computer, wireless radio and serial modem

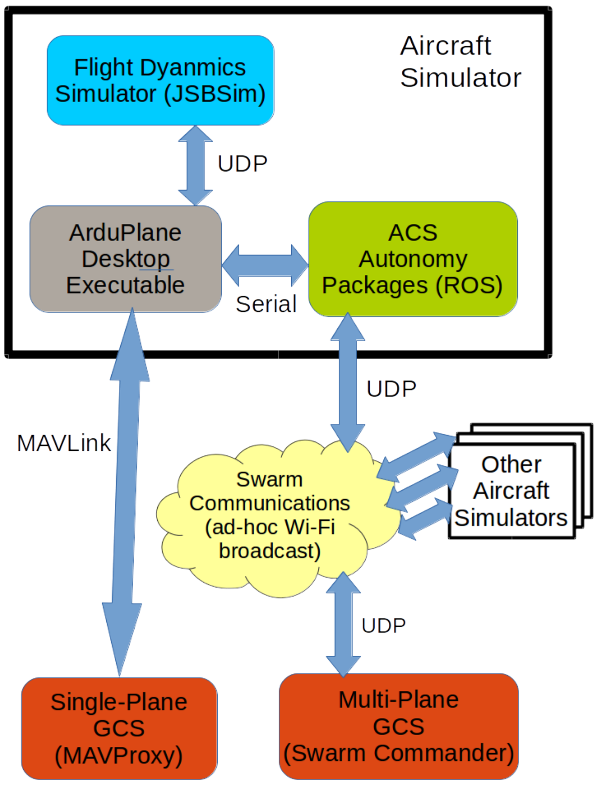

Aircraft Simulator
Flight Dyanmics Simulator (JSBSim)
UDP
ArduPlane Desktop Executable
Serial
ACS Autonomy Packages (ROS)
UDP
MAVLink
Swarm Communications (ad-hoc Wi-Fi broadcast)
Other Aircraft Simulators
UDP
Single-Plane GCS (MAVProxy)
Multi-Plane GCS (Swarm Commander)

Fig. 2. Illustration of the multi-UAV simulation and networked communications architecture

the autopilot that interact with the real world environment. To simulate a flight environment and provide sensor inputs during simulation, we make use of the link between a flight dynamics simulator, namely JSBSim [14], the ArduPlane software, and the MAVProxy ground control station interface, as provided by 3D Robotics in a simulation-in-the-loop (SITL) environment [15].

We have extended this system (both real-world and SITL) to allow communication between our autonomy package and the autopilot. We have also found that the SITL can be easily spawned in multiple instances, and have developed some scripts to facilitate UDP port deconfliction that allow, e.g., multiple SITL instances modeling multiple different planes to exist at once. We can regularly run more than ten simulators simultaneously on a reasonably powered laptop, scaling appropriately for more instances towards our objective of 50 UAVs in simultaneous flight operations. We can interact with these simulators via a custom graphical user interface specifically developed for swarm UAV operations, termed *Swarm Commander* (SC). A block diagram of the resulting multi-UAV simulation environment is depicted in Figure 2.

One important facet of communications between UAVs in our system is that all communication between the ROS autonomy payloads and the *Swarm Commander* interface is performed via broadcast across a TCP/IP subnet. This facilitates communication between aircraft which need to share position and other information for coordination, and also allows the swarm operator to have a simple interface by which to obtain status data about all aircraft. In the context of a single machine running multiple SITL instances simultaneously, broadcasting is not as straightforward, as network loopback interfaces typically do not support broadcasting. We specifically address these implementation aspects for enabling multi-SITL capabilities in Section V.

Since communication between simulators and operator interfaces occur over UDP, by employing multiple machines (we currently employ a cluster of Mac Minis running Ubuntu 14.04, e.g., see Fig 3), we can scale up to a virtual environment that provides more than 50 simulated UAVs running simultaneously, all with full autopilot and autonomy payload interfaces active.

As we work towards our goal of 50 planes fielded simultaneously in real world experiments, this virtual environment with scalable simulation instances allows us to work out unforeseen issues, explore challenging development concerns, and

Fig. 3. Cluster of 20 Mac Mini computers for serving networked, multi-UAV simulation-in-the-loop instances

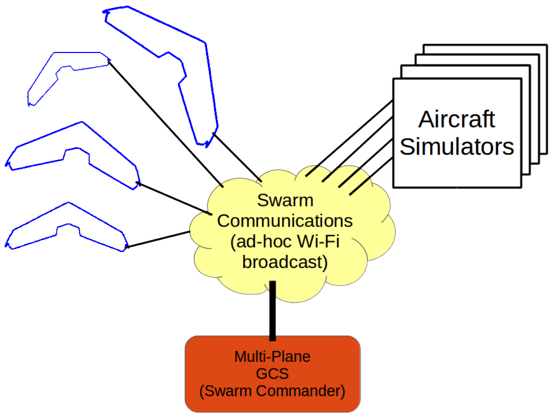

Aircraft Simulators
Swarm Communications (ad-hoc Wi-Fi broadcast)
Multi-Plane GCS (Swarm Commander)

Fig. 4. The software architecture enables experiments simultaneously using both physical and simulated UAVs, which greatly accelerates software development for large-scale tests while supporting logistical considerations for multi-UAV operations continue to be addressed through parallel research efforts.

test and evaluate new configurations.

C. Blending Physical and Simulated UAVs

The logistics of launching, simultaneously operating, and recovering 50 planes in live-fly experiments are undoubtedly challenging, and many aspects of relevant solutions are under active research and development, including improved launchers, automated preflight checks, and extending a single UAV's endurance. As a stopgap solution, we blended the multiplane simulator described in the previous section with real world aircraft, that is, since the virtual environment enables direct use of flight-ready software sub-systems, we are able to conduct experiments where both physical and simulated aircraft are active, communicating, and available for swarm mission testing. This hybrid ability enhances our testing abilities during field experimentation events. As the payload's autonomy software communicates over UDP in the same fashion on both simulated and physical payloads, both are able to coexist in the same Wi-Fi network (as long as the UDP ports are appropriately deconflicted). The resulting architecture from the point of view of the system, including the swarm operator interface, is depicted in Figure 4. Specific implementation details are further described in Section V.

IV. AUTOMATED SOFTWARE TESTING

Leveraging the above system architecture, a cornerstone to the Continuous Integration efforts is the use of automated software testing, described in this section, to accelerate development efforts through persistent and frequent iteration.

Prior to integrating the ability to perform automated software testing, our experience was that we encountered errors at the airfield more often than we would like. Errors identified in the field are expensive, in that they not only hinder or derail planned experiments, but also that such errors are often more difficult to debug and repair due to being away from the resources of the laboratory setting. Examples of such issues regrettably have included: a software bug introduced into the position estimate filter that resulted in an automated UAV landing much farther from the intended location; a bug wherein the ROS-based autonomy payload failed to stop sending altitude commands during landing; and a buffer overrun introduced when adding redundant compass sensors to the autopilot. Such errors were not easily identifiable in laboratory test conditions; such is the advantage (if not necessity) of frequent live-fly field tests using physical systems under operational settings.

We first describe our single-UAV configurations (Section IV-A), thereby ensuring that individual UAV systems are functional and highly reliable, followed by details on active development for multi-UAV automated software testing (Section IV-B). In particular, our research leverages automated software testing for single-plane SITL instances to ensure that the autopilot, ROS autonomy packages, and operator interfaces are functional in a single-plane context. As we have scaled up, these tests have remained relevant as individual plane components are modified as part of multi-plane operations; regression testing on all previously functional components of the system must be performed every time related components are modified. Such rigorous testing also supports software validation and/or certification purposes, to ensure that no new errors are introduced throughout the development process.

Our current Continuous Integration utilities include:

- All software repositories centralized and managed by the GitLab version control server [16].
- Automated builds and tests are facilitated by the Jenkins CI server [17], [18].
- A Python package, `nose`, that automates unit and integration test loading and running [19], [20].

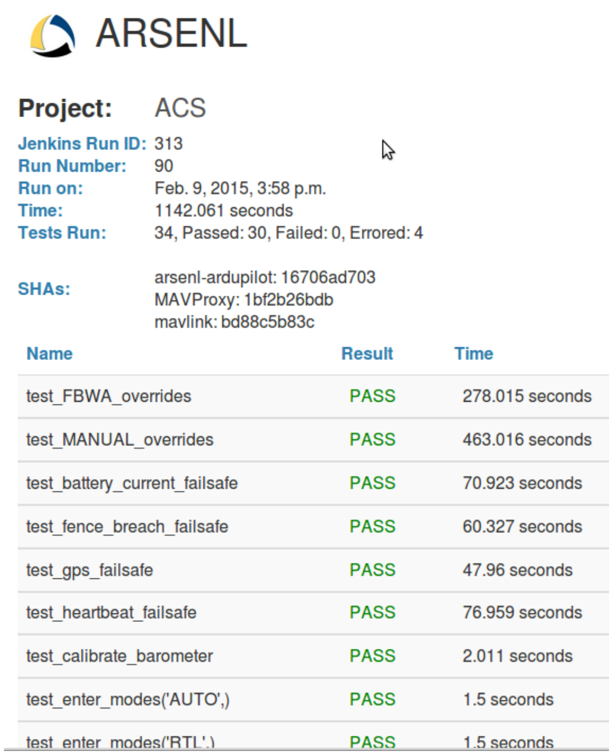
The integration of these tools in our software development efforts is detailed further below.

A. Continuous Integration for Single UAV Operations

Currently our tests focus on ensuring a single plane has a software configuration that is safe to fly, which critically requires that all failsafes and standard operations function as expected. Key examples of CI tests currently used in our research efforts include:

- Commands from the operator interface to the autopilot function as expected, e.g., a command to upload a new mission is successful.
- If the payload enters an unexpected state or fails to send heartbeats to the autopilot, the autopilot ceases to accept payload inputs.
- Minor failure conditions result in an immediate return to an emergency waypoint. Examples include: plane breaches a predefined geofence, plane loses communications with operator, plane low on battery.
- Major failure conditions result in a cut to throttle so that the aircraft remains contained inside the designated airspace. Examples include: loss of GPS, loss of control surface control.
- Autopilot commanded limits are obeyed, e.g., plane stays above minimum commanded air speed.
- Automated takeoff capabilities function as expected.
- Automated landing capabilities function as expected.
- Autopilot always obeys a safety pilot's command to override automatic inputs in favor of manual inputs (via radio control).

A human-readable, web-based report from the CI server covering these and other tests is available for any automated simulation test run, as illustrated in Figure 5.


ARSENL

Project: ACS

Jenkins Run ID:	313
Run Number:	90
Run on:	Feb. 9, 2015, 3:58 p.m.
Time:	1142.061 seconds
Tests Run:	34, Passed: 30, Failed: 0, Errored: 4
SHAs:	arsenl-ardupilot: 16706ad703 MAVProxy: 1bf2b26bdb mavlink: bd88c5b83c

Name	Result	Time
test_FBWA_overrides	PASS	278.015 seconds
test_MANUAL_overrides	PASS	463.016 seconds
test_battery_current_failsafe	PASS	70.923 seconds
test_fence_breach_failsafe	PASS	60.327 seconds
test_gps_failsafe	PASS	47.96 seconds
test_heartbeat_failsafe	PASS	76.959 seconds
test_calibrate_barometer	PASS	2.011 seconds
test_enter_modes('AUTO',)	PASS	1.5 seconds
test_enter_modes('RTL',)	PASS	1.5 seconds

Fig. 5. An example of a simulation test run report automatically generated for web-based review from the ARSENL Continuous Integration server.

Since introducing a Continuous Integration capability, examples of software errors identified *prior* to live-fly field testing have included: a bug in the GPS loss failsafe; and a bug in the failsafe against a loss of control surfaces. Both of these conditions occur very infrequently in the field and are also difficult to test otherwise using hardware configurations, yet are critical failures that could result in the loss of the aircraft. As another example, incorrect assumptions about the ability to hold heading during automated landing were discovered during automated tests. Field testing still proved invaluable to solve all automated landing problems, but some of these initial errors in assumptions about the autopilot behavior were found and corrected with the help of the SITL and CI server. The ability to test and retest at a high rate clearly accelerated the effort to arrive at reliable autonomous landings, now a core capability for our system, and further continues to assist in our ongoing research and software development.

B. *CI For Multi-UAV operations*

Multi-plane operations benefit from reliable processes ensuring that single-plane failure modes and standard operations occur as expected, but multi-plane operations also require tests specific to multi-plane scenarios. A partial yet growing list of examples includes:

- Multiple planes experiencing a minor failure mode condition simultaneously must have the emergency waypoints deconflicted in altitude and/or geographic coordinates.
- Landings and takeoffs of multiple aircraft must occur with spatial and/or temporal separation.
- A safe distance between aircraft must always be maintained.
- Any limits on the number of UAVs that a single human operator can safely command at a time can be studied in simulation.

We are currently studying those problems as part of the presented research initiatives. Though these challenges remain open research questions and areas of active investigation, we further see the need for and benefit of expanding the SITL and CI capabilities into a multi-aircraft domain. To date, we have most extensively used our CI automated software testing on single-plane SITL contexts, and actively and continually expanding such tests into a multi-plane context. For example, in formation flight, a typical measure of performance requires monitoring of lateral distances and/or altitude separations between multiple aircraft. Using both "ground truth" values as represented by the multiple flight dynamics simulation instances and local estimates (i.e., as measured by each individual UAV) of inter-plane distances, we can not only assess the performance of the formation-keeping algorithms, but also evaluate other systems-level characteristics, e.g., the impact of latency of payload messages on safety of flight (see Figure 6).

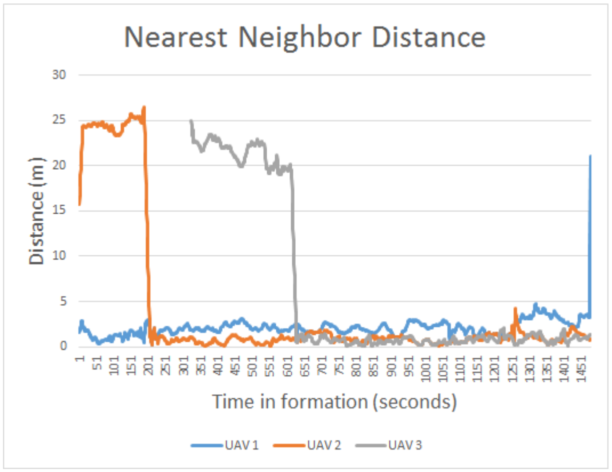


Fig. 6. A simple example of data that an automated test on the CI server can monitor in a multi-SITL environment. In this three-UAV example, if the distances to the nearest neighbor fall outside defined limits, then such infractions or failures can be noted in the automatically generated report of a simulated test run.

V. MULTI-UAV SITL INTERFACES

As briefly described previously, we augment our software development efforts, including Continuous Integration methods, with the ability to test various collaborative autonomy modes or swarm behaviors by running multiple virtual aircraft instances – both autopilot and autonomy payload software – on one or across many machines. Note that simulating multiple ArduPlane and JSBSim instances for autopilot testing is already supported by the community. The key obstacle overcome and described in this paper was how to run multiple autonomy payload instances on a single machine that could both communicate with each other and with other payload instances on other distributed machines.

Because aircraft and ground stations communicate primarily via UDP broadcast, a broadcast capability, either simulated or actual, is required between (a) SITL instances on a single machine; (b) between SITL instances on separate machines; and (c) between SITL instances and physical aircraft. We have explored several methods for providing these capabilities, each with advantages and disadvantages, which we discuss in detail below.

A. *Virtual Machines*

Our initial approach creates a virtual machine (VM) for each SITL instance, and uses a combination of virtual and physical networking to provide the broadcast medium across SITL instances. We employ Oracle VirtualBox VM software [21] as our VM implementation.

This method allows multiple broadcasting SITL instances to communicate with one another on a single machine and across multiple machines. Assuming the machines on which the multiple SITL instances are running possess Wi-Fi hardware interfaces, then the separate simulations are also able to communicate with physical aircraft. Also, since each SITL instance resides within a distinct VM, it is possible to emulate network topology effects (such as latency, packet loss, etc.) on a plane-by-plane basis without requiring modification to SITL software. However, each VM consumes significant system resources to run a separate operating system kernel and emulate hardware, which reduces the number of SITLs that can be run on a given computer.

B. *Local Broadcast*

Seeking an approach that would not be as resource-intensive as using separate virtual machines for each simulated UAV, we designed the option to assign distinct UDP port numbers to each autonomy payload instance. This solution requires a repeater utility (written in Python) that receives traffic from each port and resends it out across all other ports to mimic broadcast behavior (refer to Figure 7). We later extended the repeater utility to listen for and transmit broadcast traffic to support SITL instances across multiple machines. However, the repeater utility added unnecessary complexity and overhead by having a userspace application receive and resend each UDP datagram multiple times.

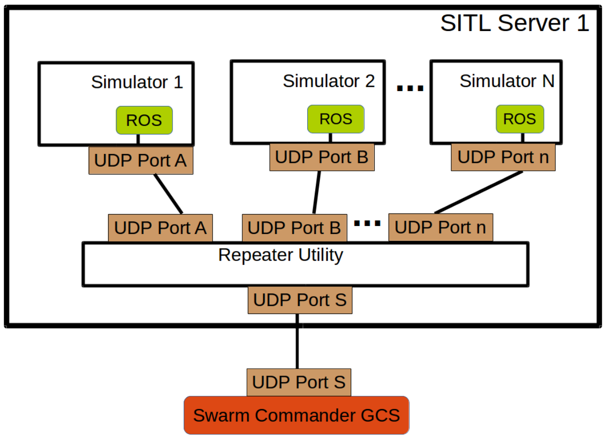


Fig. 7. Initial implementation for allowing multiple SITL simulation on a single machine requiring a repeater utility

Ultimately, we eliminated the repeater utility upon discovering an under-documented feature of Linux UDP sockets. TCP/IP implementations typically do not allow a given address and port pairing to be reused across multiple sockets at once. Some operating systems, such as BSD variants, provide a "port reuse" option (e.g., `SO_REUSEPORT`) that overrides this behavior, but Linux (until very recent developments) only provides a `SO_REUSEADDR` option, which sounds similar but as documented has different semantics that only apply to TCP.

However, we have discovered that most modern Linux implementations allow UDP sockets to reuse address-port pairs; `SO_REUSEADDR` then functions very much like the BSD `SO_REUSEPORT` [22], [23]. Adding this option to all our payload sockets allows us to directly broadcast from multiple payload instances, removing the need for the repeater utility, and resulted in a solution that allows communications across multiple SITL servers as in Figure 8.

This approach allows single-machine SITL and multi-machine SITL implementations which are lighter weight than the VM solution, making more SITL instances per machine possible. SITL instances can also communicate with physical aircraft, provided the machines they reside on are equipped with physical Wi-Fi interfaces. However, in this approach, there is not a straightforward way to emulate the effects of latency or other network characteristics of a Wi-Fi network in simulation on an individual basis without modifying the SITL

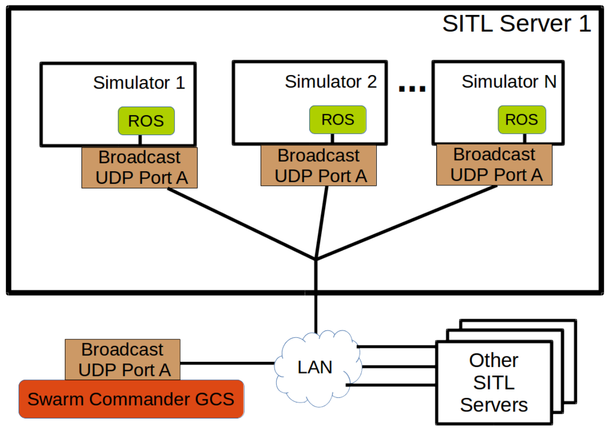


Fig. 8. Implementation of multiple simulation instances using SO_REUSEADDR, which removes the need for a repeater utility and facilitates inter-server communication

software itself.

C. Control Groups

Linux provides a lighter weight alternative to VMs with control groups [24]. Control groups make it possible to create multiple "namespaces" within a single operating system instance. Using network namespaces, we are able to run multiple payload instances on a single Linux system. Each simulated payload runs in its own namespace with a virtual network adapter connecting that namespace back to the "primary" namespace and optionally to a physical network.

All ArduPlane and JSBSim instances run in the primary namespace, as there already exists a way to run multiple instances of each using deconflicted TCP and UDP ports for each. Using Linux control groups provides all the advantages of the approach outline in Section V-B, but also more easily allows for emulation of Wi-Fi network latencies in each namespace separately without modification to SITL software.

Given the different configurations and their respective pros and cons for executing a distributed network of simulation-in-the-loop instances for autopilot and autonomy software, the above options allow for tailoring of the multi-UAV virtual environment depending on the research focus areas of interest. These new developments represent a contribution to multi-UAV research efforts by enabling tighter integration between simulation and physical experimentation, and further enable greater collaborative engagement with different research communities (e.g., wireless networking modeling and communications protocol design, human-swarm interface testing, accelerated collaborative autonomy algorithm development).

VI. CONCLUSIONS AND FUTURE WORK

As software systems become ever more increasingly complex and critical for autonomous systems research, the cross-over between software systems and robotic systems engineering continues to accelerate the development of significant advances, increasingly so for unmanned aerial vehicle technologies. Presented in this paper are two developments in our active research efforts to field and demonstrate simultaneous flight operations of 50-UAV teams in live-fly field experiments that leverage such software systems engineering principles. We designed and implemented various elements of agile software development processes, including continuous integration with automated software testing, to improve the rate and reliability of single- and multi-UAV software subsystems, that has demonstrated their significant benefits over the course of multiple development cycles including integrated field tests. Further, in keeping with the need for rapid prototyping and transition to operational testbed systems, we investigated and developed several test configurations for instantiating multiple simulated UAV entities, comprising both autopilot and autonomy payload software, in a networked, potentially distributed, virtual environment. Both of these developments, highlighting process improvements in software development practices, have demonstrably enhanced and accelerated our overall research program.

In addition to continued systems integration efforts for live-fly field experimentation of the software-enabled swarm UAV capabilities, several key avenues for further research can readily be identified. As the set of automated tests increases in both number and complexity (e.g., testing collaborative autonomy algorithms), the current limitation of conducting simulation-in-the-loop in real-time presents a constraint on the agility of the continuous integration process. Enabling faster-than-real-time simulation, including appropriate acceleration and/or modification of flight dynamics, sensor updates, algorithm execution, and network characteristics, will greatly enhance the ability to leverage the automated test suites. Aligned with this capability would be the use of parallel execution of automated tests on distributed servers to enable concurrent evaluation of the software.

Further, the addition of realistic network and communication constraints in the simulated virtual environment would better reflect the true physical system, given the RF and link characteristics of dynamic flying aerial networks. Integration of network simulation software, such as `ns-3` [25], in the multi-SITL configuration to represent air-to-air and air-to-ground wireless communications would strengthen the fidelity of the simulation as well as provide more transparent transition to the live-fly testbed. Additional enhancements include further integration of mesh network technologies best suited for flying ad hoc networks (FANETs), and their incorporation into the virtual environment.

REFERENCES

[1] T. H. Chung, K. D. Jones, M. A. Day, M. Jones, and M. R. Clement, "50 VS. 50 by 2015: Swarm Vs. Swarm UAV Live-Fly Competition at the Naval Postgraduate School," in *AUVSI North America*, Washington, D.C., 2013.

[2] K. Beck, "Embracing change with extreme programming," *Computer*, vol. 32, no. 10, pp. 70–77, Oct. 1999.

[3] R. C. Martin, *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall, 2003.

[4] N. D. Fogelstrom, T. Gorschek, M. Svahnberg, and P. Olsson, "The impact of agile principles on market-driven software product development," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 22, no. 1, pp. 53–80, Jan. 2010.

[5] D. Cohen, M. Lindvall, and P. Costa, "An Introduction to Agile Methods," *Advances in Computers*, vol. 62, pp. 1–66, 2004.

[6] H.-C. Estler, M. Nordio, C. A. Furia, B. Meyer, and J. Schneider, "Agile vs. structured distributed software development: A case study," *Empirical Software Engineering*, vol. 19, no. 5, pp. 1197–1224, 2014.

[7] B. Meyer, *Agile!: The Good, the Hype and the Ugly*. Springer Science & Business Media, 2014.

[8] G. de Souza Pereira Moreira, R. P. Mellado, D. A. Montini, L. A. V. Dias, and A. M. da Cunha, "Software Product Measurement and Analysis in a Continuous Integration Environment," in *Seventh International Conference on Information Technology: New Generations*, Las Vegas, NV, USA, Apr. 2010, pp. 1177–1182.

[9] G. G. Claps, R. B. Svensson, and A. Aurum, "On the Journey to Continuous Deployment: Technical and social challenges along the way," *Information and Software Technology*, vol. 57, pp. 21–31, Jan. 2015.

[10] "APM:Plane," http://plane.ardupilot.com/, Jan. 2015.

[11] M. Quigley, B. Gerkey, K. Conley, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler, and A. Ng, "ROS: an open-source Robot Operating System," in *ICRA Workshop on Open Source Software*, 2009.

[12] "MAVProxy: A UAV ground station software package for MAVLink based systems," http://tridge.github.io/MAVProxy, Jan. 2015.

[13] "MAVLink Protocol," qgroundcontrol.org/mavlink/start/, Sept. 2012.

[14] Sourceforge.net, "JSBSim Open Source Flight Dynamics Model," http://jsbsim.sourceforge.net/, Sept. 2012.

[15] "Setting up SITL on Linux," http://dev.ardupilot.com/wiki/setting-up-sitl-on-linux/, Jan. 2015.

[16] "GitLab: Version Control on your Server," https://about.gitlab.com/, Feb. 2015.

[17] A. Berg, *Jenkins Continuous Integration Cookbook*. Packt Publishing Ltd, 2012.

[18] "Jenkins: An extensible open source continuous integration server," http://jenkins-ci.org/, Feb. 2015.

[19] D. Arbuckle, *Python Testing: Beginner's Guide*. Packt Publishing Ltd, 2010.

[20] "nose 1.3.4: nose extends unitest to make testing easier," https://pypi.python.org/pypi/nose/1.3.4, Feb. 2015.

[21] V. Oracle, "Virtualbox, user manual, 2011," 2012.

[22] "Socket options SO_REUSEADDR and SO_REUSEPORT, how do they differ?" http://stackoverflow.com/questions/14388706/, Feb. 2015.

[23] "FreeBSD Man Pages: GETSOCKOPT(2)," http://www.freebsd.org/cgi/man.cgi?query=setsockopt, Feb. 2015.

[24] R. Inam, J. Slatman, M. Behnam, M. Sjodin, and T. Nolte, "Towards implementing multi-resource server on multi-core linux platform," in *Emerging Technologies & Factory Automation (ETFA), 2013 IEEE 18th Conference on*. IEEE, 2013, pp. 1–4.

[25] G. F. Riley and T. R. Henderson, "The ns-3 network simulator," in *Modeling and Tools for Network Simulation*. Springer, 2010, pp. 15–34.