Available online at www.sciencedirect.com

SciVerse ScienceDirect

Procedia Computer Science 18 (2013) 2018 – 2027

Procedia Computer Science

International Conference on Computational Science, ICCS 2013

A Testbed for Investigating the UAV Swarm Command and Control Problem Using DDDAS

R. Purta[a], M. Dobski[a], A. Jaworski[a], G. Madey[a]

[a] *Department of Computer Science and Engineering University of Notre Dame 384 Fitzpatrick Hall, Notre Dame, Indiana 46556*

Abstract

Unmanned Aerial Vehicles (UAVs) may become the future of military aviation as technology advances, especially sensors and miniaturization techniques. Currently, however, UAVs are controlled individually and require many resources, including ground-based pilots, to function. In our project, we attempt to explore how to remedy this using a Dynamic Data-Driven Application System (DDDAS) to control a group, or swarm, of UAVs. DDDAS takes real data and injects it into a running simulation, as well as allowing the running simulation to influence what real data is gathered, and as such is an ideal system to control real UAVs. We describe here how we created a testbed system that allowed two simulations to communicate data to one another using DDDAS principles, as well as the beginnings of incorporating commercially-available UAVs into the system.

Keywords: UAVs, swarm, DDDAS, mission planning, command and control

1. Introduction

In the sections that follow, we explain why it is worth exploring Unmanned Aerial Vehicles (UAVs), especially in the form of a swarm. We shall also discuss why we believe a Dynamic Data-Driven Application System (DDDAS) is a viable approach to explore them.

1.1. Unmanned Aerial Vehicles (UAVs)

In 2001, Congress set a goal for the military to have "by 2010, one-third of the aircraft in the operational deep strike force aircraft fleet...unmanned" and "by 2015, one-third of the operational ground combat vehicles...unmanned" [1]. It is difficult to find whether or not the military is on schedule to achieve these goals, but it is definitely a work in progress. The Air Force, for example, stated that they planned to buy more unmanned aircraft than manned ones for the year 2011 [2]. Earlier in the decade, UAVs were used in Operation Enduring Freedom and later Operation Iraqi Freedom [2], and it was UAVs that aided the U.S. military to track and eventually find Bin Laden [3].

But why UAVs? Besides the vehicles' performance in recent military operations, there are a variety of reasons why UAVs are being favored over manned vehicles. One of these is the very fact that UAVs are unmanned, meaning they have no pilot physically in the vehicle, but are controlled from a distance. For this reason, UAVs are valuable for what has been termed the "dull, dirty, and dangerous" missions, so that military personnel can

Email addresses: rpurta@nd.edu (R. Purta), mdobski@nd.edu (M. Dobski), ajaworsk@nd.edu (A. Jaworski), gmadey@nd.edu (G. Madey)

Selection and peer review under responsibility of the organizers of the 2013 International Conference on Computational Science
doi:10.1016/j.procs.2013.05.371

focus on the safer, more complex missions [4]. UAVs are well-suited for these missions not only because they can prevent the loss of human lives, but because they are inexpensive compared to manned vehicles, especially the smaller ones. The vehicles used in the Navy's SWARM system, for example, cost about $2000 each [5]. UAVs and their sensors are also becoming smaller and cheaper to make as technology progresses, so smaller UAVs are becoming more capable [5].

Small UAVs and other mini UAVs are the least expensive, but because they are so small, they have limited use by themselves. This is the reason why swarming technology has become a major area of research.

1.2. Swarms

A swarm in computer simulation is a group of simply-behaving entities that together produce significant results or behavior [6]. In nature, there are a number of swarming organisms, though the swarms are often referred to by different names. A group of bees or ants, for example, can be called a swarm, but a group of birds is a flock. The most famous simulation of a swarm, created by Craig Reynolds, was actually of a flock of birds, or "boids" as he calls them [7], but other simulations of swarming entities have been created. The MASON library, discussed later, has a few of these swarm simulations, including one of ants bringing food back to the nest, Reynolds' boids, Icosystem's swarm game, and human crowds [8]. MASON's simulations are all agent-based, which is only one method of simulating a swarm. Others that we have found in our literature search include behavior-rules, which is very close to agent-based but often involves artificial-intelligence techniques, gradient-vector movement, graph theory, mathematically-determined patterns, and more. Many of these can incorporate hundreds, if not thousands, of entities. Now they are being applied to UAVs.

When literally hundreds of UAVs are working together in a swarm, the problem of how to control them all arises. Currently, it takes at least one human operator to fly some types of UAVs [6], [9], while others take at least two [10], [11]. The Office of the Secretary of Defense seeks to remedy this by proposing that UAVs be given more autonomy, and have outlined the plan to realize this goal in the document *Unmanned Aircraft Systems Roadmap*, as well as predicting that the technology for full autonomy will exist by 2030 [4]. Military policy, however, prevents UAVs from being fully autonomous, as there must be at least one person to approve the UAVs' actions [2], especially in the case of weapons [10].

Merely automating a UAV by itself is not enough, if one operator is still needed to aid in its decision-making process. A swarm of hundreds of UAVs would then require hundreds of operators, which is not practical. This is why much research, including this paper, has focused on controlling the swarm as a whole, rather than each UAV individually. Simulation and its various methods of creating an artificial swarm has proved useful for this area of research. To be realistic, however, a simulation needs realistic data with which to perform its calculations. The concept of DDDAS provides this much-needed realistic data, because it allows the simulation system to interact with the real world.

1.3. Dynamic Data-Driven Application Systems (DDDAS)

The concept of DDDAS was first proposed in the 1980s by Dr. Darema, then at the NSF and now at the U.S. Air Force, but since has been discussed in multiple conferences and workshops, including the recent ICCS 2012 [12]. It is defined as a distributed system that has "the ability to incorporate dynamically data into an executing application simulation, and in reverse, the ability of [the system] to dynamically steer measurement processes" [13]. As applied in our system, DDDAS allows our swarm simulation (MASON) to receive location and other information from either real-world UAVs or simulated UAVs, and in return allows our swarm simulation to steer the UAVs.

1.4. Prior Work

As mentioned, we have found in our literature search several simulation techniques that can be applied to UAVs: behavior-rules swarms can be found in [6], [14], [9], and [15], gradient-vectors in [16], graph theory in [17], mathematical patterns in [18], [19], and artificial "pheromones" in [11], and [20]. Note that [15] and [19] have tested and designed these swarm simulations for robots, though their simulation could be applied to UAVs. Works that have used agent-based simulation are either a mix of techniques or modify an existing model. The works [2] and [21] use Reynolds' boids, though [21] adds additional rules to the model. In [22], the boids model

is modified to add a genetic algorithm that will choose a good path for a mission. The Air Force and Icosystem collaboration in [23] analyzes several agent-based approaches to simulation, including "pheromone" tracking, random movement, movement within a global space, and straight-line movement.

Only a few of these works have actually used these simulation techniques to control swarms. In [14], a genetic algorithm was used to determine what behavior controller for a UAV would be good enough, given its input from sensors. Price did something similar in [9], but he defined rules that UAVs would obey and used the genetic algorithm to determine what the overall swarm behavior would be. The collaborated work with Icosystem in [23] uses the various strategies mentioned to control the swarm, running several simulations of each to see which control strategy works best on a search-and-destroy mission. Their results favor the pheromone strategy, and so in their second paper, [20], the group concentrates on this strategy and combine it with a genetic algorithm to evolve behaviors, or states, for the UAVs. In [11], they also used a form of pheromones, only they digitized them into signals that essentially formed a potential field, and had the UAVs follow it. The work in [16] actually did use potential fields, only to force the swarm into a formation.

1.5. Model Choice

The swarm model we have chosen to use is based on the Icosystem swarm game, partially described on their website, `http://www.icosystem.com/labsdemos/the-game/`, and investigated for predictability in [24]. The basic structure of the model has an agent choose two other agents (which we choose to call "helpers") to interact with, following one or all of several rules. The traditional rules from Icosystem are defender and aggressor, but the MASON implementation has five: defender, aggressor, stalker, avoider, and random. The defender and aggressor rules depend on both of the two "helpers", while stalker and avoider depend on only one, and random depends on none. All the rules besides random are depicted in Figure 1. Random is not depicted as it only has the agent move in a random direction.

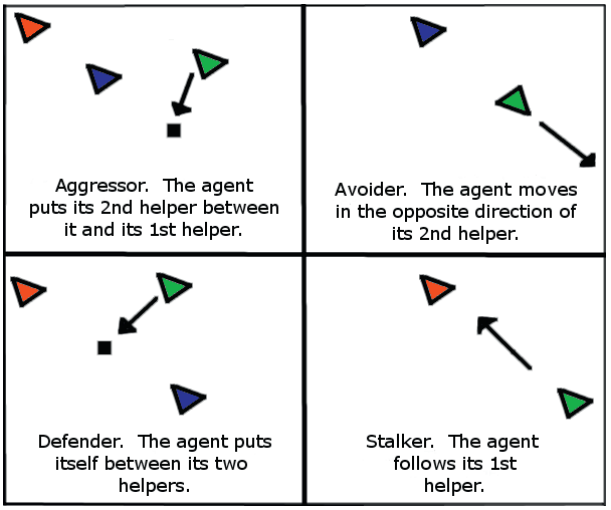

Fig. 1. Four of the rules of MASON's implementation of Icosystem's swarmgame. The agent is shown in green, while the first helper is red and the second is blue.

Our previous paper [25] used the boids model, but allowed each UAV to move independently so that individual UAVs could carry out different actions. This complicated the instructions that needed to be sent to the swarm. Since the Icosystem model, however, allows an individual agent to keep track of the movements of two other agents, with some model tuning we are able to make the agents move as a swarm with very simple instructions. This means that all the agents can move toward a particular target. We give movement instructions to the swarm using what is called the "leader-follower" method [6] or "leader-initiated" [9] method. In this method, all UAVs do not need to be told to move toward the target, only the leader, with the others behaving according to the rules of the model. We envision this method to have an advantage when implemented on actual UAVs, because only

the leader would have to receive movement instructions from the operator, while the others will align themselves according to their internal algorithm, as long as each UAV can receive location information from all other UAVs within a set radius. We are also aware of the disadvantage of the leader-follower method, that is, the loss of the leader. We have attempted to remedy this problem by implementing dynamic leader reassignment.

The Icosystem model also has the advantage of being conveniently simple, as its calculations are vector-based. It does not use computation-intensive methods, while behavior-rules, gradient-vectors, graph theory, and other mathematically-based methods tend to be. Artificial pheromones, while not computation-intensive, have only been applied to search applications for UAVs, though it may be possible to modify the method for other applications.

The DDDAS portion of our testbed allows our model, briefly described above and further discussed later, to interact with the commercially-available Parrot AR.Drone 2.0, though complete functionality is still a work in progress. We were successfully able, however, to send the AR.Drone some instructions and receive data, such as percentage of battery available and GPS coordinates, to be incorporated into our running simulation.

2. Project Overview

This section describes the testbed created for the project. It starts with an explanation of the system's functionalities and architecture. Then it goes to the implementation part, describing all used technologies.

2.1. Proof-of-Concept Testbed Overview

The first thing that needed to be done on the project was creating the proof-of-concept testbed. Following the DDDAS paradigm, it should allow sending orders to the environment as well as getting sensory data from it. Therefore, our system consisted of four main modules: the command module, the GUI module, the environment module, and the middle-ware module. The command module is responsible for controlling and getting data from the environment module, the GUI module is responsible for user interaction and environment choice, and the two are connected using the middle-ware layer. Created modules were made as independent as possible, so that each element could be replaced without much effort in the future if needed. Let us take a brief look at each of the mentioned modules, starting with the command module.

Command Module

The command module is the heart of whole system, as it is responsible for making decisions based on mission goals and knowledge received from the environment. The swarming intelligence is implemented here, using a Multi-Agent simulation. In Figure 2, this module is colored red, and consists only of the Multi-Agent simulation.

GUI Module

The GUI module allows user interaction, especially the choice of which environment to use, whether drones or the simulator. This module's main purpose is to make the choice of environment transparent to the Multi-Agent simulation, meaning the simulation implementation would not need to change when switching between the simulator and drones. Besides that, the Console provides a GUI for many other administrative activities, e.g. previewing the current UAVs' positions, generating new mission targets, or emergency landing of hardware drones. It also contains the log window, which informs what actions have been taken by the system and the results of those actions.

Environment Module

The command module needs to have control over some specific environment. In our testbed we have created two independent environments: the simulator and drones. The former is entirely software-based, while the latter constitutes of commercially available, remote controlled quadrocopters - Parrot's AR.Drone 2.0.

Middleware Module

Both of the mentioned modules need a way to communicate with each other. Therefore, the middle-ware layer was implemented. To achieve the highest level of module independence, it was written in concordance with the Service Oriented Architecture (SOA) paradigm using RESTful Web services.

Each of the mentioned modules is described more thoroughly in section 2.3.

2.2. Testbed Architecture

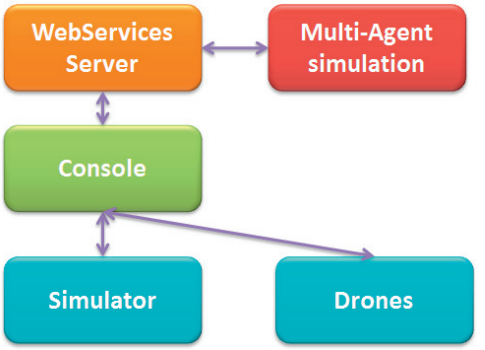

Fig. 2. Architecture of Testbed environment

The architecture of the system is depicted in Figure 2. As mentioned above, it consists of four main modules: the Multi-Agent simulation (red), the GUI (green), the controlled environment (blue) which can be either software (the Simulator) or real-world (Drones), and the middle-ware WebServices server (orange).

2.3. Proof-of-Concept Testbed (Simulation to Simulation) implementation

Multi-Agent Simulation (MASON)

The MASON multi-agent simulation side of the testbed needed work throughout the duration of the project, as slight adjustments needed to be made while the middleware and console modules were in progress. There were, however, core parts to the simulation that did not change once developed. As mentioned in section 1.5, the main swarm model used was based on Icosystem's swarm game, as implemented by Luke in the MASON project [8]. This implementation uses summed vectors to determine the direction a particular agent will move. Vectors are calculated according to the rules shown in Figure 1, as well as an additional avoidance rule that allows an agent to avoid all other agents whose vectors intersect theirs, as well as avoid the walls of the simulation. This avoidance rule is calculated differently than the one shown in Figure 1, as it avoids the vector of the other agent, not the agent itself, as well as avoiding all agents, not just helper agents. It is based on the avoidance algorithm in Luke's Flockers simulation, also found in the MASON package. The calculation to avoid the walls is similar, except the walls are given a shorter vector.

Another added feature to the swarm model was the addition of the leader-follower technique, so that the swarm could follow a planned route. The leader was designed to visit all the indicated points and discovered targets (more of which will be discussed soon), so consequently, it does not obey the Icosystem rules. It does, however, have the same avoidance algorithm as the other agents. In order to make the other agents follow the leader, however, the helper system was customized so that the leader would be the first helper (the red one in Figure 1) of all the other agents. This guarantees that the other agents will move in the same direction as the leader in almost any rule configuration, as long as the avoider rule is minimized.

As mentioned in section 1.5, we have also implemented dynamic leader reassignment to overcome the technique's disadvantage of losing the leader. What is required to do this is basically helper reassignment. When the leader's fuel is low or it lands, another functioning agent is randomly chosen to be the leader, and it is assigned as the first helper of all the other functioning agents. Path instructions are then sent to the new leader instead of the old one, and the other agents will follow.

The MASON simulation gathers information from both the console module and the environment module, whether the environment is another simulation or real-world drones. The environment sends fuel and updated position information, while the console generates the specified number of targets, which can be set by the MASON simulation. As the simulation progresses, the leader moves to the next waypoint in the list, unless a target is discovered, in which case the leader goes to visit the discovered target in order to confirm it. A target can be discovered by any agent as long as that agent passes close enough to "see" it. Discovered targets take precedence

over waypoints, so only when there are no more discovered targets does the leader continue its waypoint path. When there are neither discovered targets nor waypoints left, all the agents go back to base and land one at a time.

UAV Simulator

We wished to develop a simulator environment to work with the multi-agent simulation first, before making the drones communicate with the multi-agent simulation. This is because a simulator is more flexible, allowing us to change parameters and see how they affect the overall system. After work on the drones was completed, we wanted to switch between the drones and simulator without altering any code of the multi-agent simulation.

The main part of getting both drone and simulator environments to work in the same manner was to set the same steering mechanism. We considered two levels of abstraction: the higher one, where the UAV would receive a point in the world to go to, and lower one, where the drone could be rotated to a specified heading and fly with the specified speed. In the simulator we decided to use the lower level of abstraction, since it allows more flexibility. As described in the preceding section, however, the higher abstraction level was used in the MASON simulation, so that the UAVs could follow a planned route if needed.

To make the simulated UAVs act similarly to real-world drones, we had to implement inertia in their actions, so that each change of speed or heading is not performed immediately, but with a specified delay. Each parameter, including acceleration, deceleration, translation and rotation speed, can be set in the code to make the simulation resemble the real-life scenario. Although quadrocopters can fly following a curved path, we decided not to use it to make collision avoidance easier. Therefore, when we chose to change the drone's heading for more than a specified threshold (15-20 degrees), we implemented the simulated UAVs so that they would stop, rotate in place, and then continue moving.

The simulator environment has been written from scratch in Java. It is easily extensible and integrates well with the rest of the system.

Middleware

To make the Multi-Agent simulation communicate with the environment, we had to create a middle-ware layer. We decided to use a Web services approach to conduct whole communication because of its scalability and independence to the clients' architectures.

In the proof-of-concept testbed, we used RESTful Web services to give MASON simulation access to the environment, which is either the simulator or the drones. Using these services, the Multi-Agent simulation can get each UAV's position, heading, and state. It can also fiddle with the environment parameters, e.g. create a specified number of targets, generate a specified number of simulated UAVs, or start and stop the simulator environment. To comply with DDDAS paradigm, services to control the fleet have also been implemented - each UAV can be ordered to set its heading and speed to specified value.

Console

In addition to giving access to the environment from outside, Web services are an interface to the main control module, which is the Console. This module was added to the middle-ware layer to conduct all requests to their proper receivers and make operating the system easier for end users. It runs a server socket, which Web services can use to communicate with the underlying system. Messages exchanged by REST and the Console indicate what action to perform and pass parameter values.

The Console provides the GUI (see Figure 3) for controlling the system. Its main purpose is choosing the environment to use - the simulator or drones - which makes it completely transparent for Web services, and hence the Multi-Agent simulation. Besides that, the Console allows the performance of every action that can be accessed using Web services. The last part of the Console module is the supervision window, which shows the current position of the UAVs from the simulator or drones environment.

2.4. Real Data Input to Testbed: GPS and Parrot Drones

Nowadays there is a broad variety of drones available on the market. These devices can be either electro-gliders or hovercrafts. These drones, however, do not originate from the UAVs used by the military, but instead from the remote-control (RC) hobby that has become increasingly popular today.

Fig. 3. Left: the console GUI. Right: The supervision window, showing current UAVs' positions

Multi-rotor copters were designed because it was found that their predecessor, the RC helicopter, had a high risk of failing and falling into inhabited areas. For maximum safety and stability, multi-rotor copters consist of 4 to 8 rotors equi-spaced around the center of construction. With this design, a drone with 8 rotors can afford to lose up to two nonadjacent rotors before it loses stability. For the purpose of our proof of concept testbed, Parrot's AR.Drone 2.0[26] drones were chosen (see Figure 4). These drones are quadrocopters, meaning they have 4 rotors equi-spaced around the center.

Fig. 4. A Parrot AR.Drone 2.0, viewed from above

Wi-Fi Networking

Connections with the drones are done using a low-latency 802.11n Wi-Fi network [26]. Out of the box, the devices are configured to act as an access point, but this link is insecure. The only security available is pairing the drone with a selected Media Access Control (MAC) address. Since we need multiple vehicles in a single network, we created a solution to the security flaws by preparing the drones for on-demand connection to the testbed's defined common Access Point (AP). The network the drones connected to was provided by our control station laptop.

Navigation and Stabilization System

As a consumer product, the drones come embedded with an effective inner stabilization loop which allows them to hover in a single spot without drifting, even in windy conditions. This system is then exposed to a high-level control commands API. These commands include: take off, land, hover, tilt, rotate and ascent/descent.

Upon the embedded navigation system we have developed an even higher level navigation system. It runs continuously in a dedicated Java thread for each drone connected to the testbed on the control station. It allows us to set the drone's speed, heading and altitude. Once preset, the navigation system continues to send proportionally calculated low-level commands to the drone to maintain the desired flight parameters. This navigation loop is also a fail-safe mechanism. If it doesn't receive any updated parameters for more than a modifiable time threshold, it assumes that the MASON simulation, which sends such parameters, has failed or disconnected and forces the drones to hover.

This newly-developed layer made the hardware drones transparent to the rest of our testbed and therefore hot-swappable with the simulation environment. It also allowed running hybrid – hardware and software based – tests.

API and Fleet Manager

For maximum flexibility and portability we decided not to use the native C API provided by Parrot and instead integrated a pure Java project called Javadrone into the testbed [27]. This library was however oriented on working with a single drone and required some external wrapper code to work with a fleet of our quadrocopters. We developed what we named a "FleetManager" entity, responsible for delegating commands from the Console module or any other calling class to a uniquely identified drone. It also allows dynamic, runtime- varying fleet control, basically adding and subtracting drones in the fleet.

Sensors in the DDDAS Environment

In order to comply with the DDDAS paradigm, the drones have to report data back from their sensors into the testbed. Out of the box they come equipped with a variety of sensors, such as a magnetometer, accelerometer, gyroscope, altimeter, front facing HD camera, ground facing camera, and battery level sensor [26].

The video stream is unavailable at this stage of the project as Parrot changed the compression protocol [28] in version 2.0 of the drone and the JavaDrone code still hasn't been updated to the new specification.

Sensor Data User Datagram Protocol (UDP) port

The design made by Parrot allows only one drone to connect to a single computer. This limitation lies in the way the drone reports sensor data back. It connects to an UDP port on the control station and sends it data. It exclusively locks the port, preventing other devices from connecting. To overcome this problem, we have chosen to add a prerouting rule to the IP tables running on the drone's Linux environment. This rule rerouted the outgoing UDP packets from the UDP port hardcoded by Parrot to a user configurable one. With this fix each drone could then report the sensors' data to a single control station without having networking conflicts with other vehicles.

Linux Drivers and GPS Daemon

In order to enable the GPS dongles on the ARM powered drones we had to enhance the Linux running on them. The required work included cross–compiling kernel modules using the Codesourcery cross–compilation toolchain [29].

Acquired location data had to be forwarded from the drone to the base-station. For this purpose we cross-compiled a stripped – service oriented core only – build of the GPS Daemon [30]. It runs as a server on the drone and allows multiple clients to connect to it over the Wi-Fi network. The clients can then receive GPS data from the dongle attached to the drone. For the client side, the GPSD4Java project was chosen as it is a pure Java implementation of the gpsd protocol [31].

2.5. Results

We were able to complete the proof-of-concept DDDAS testbed, though the communication with our Parrot AR Drone 2.0 needs further testing to improve their performance. As the definition of DDDAS describes, we have a simulation that communicates data to an environment, whether that environment is simulated or real, and the environment communicates requested data back to the simulation. When the environment is the simulation, the MASON simulation gives movement instructions to the agents on the screen of the other simulation, and the other simulation sends fuel levels and discovered targets. When the environment is the real-world drones,

then the drones send their current GPS location and the MASON simulation tells them where to go next. Metric translations and communication are done through the JavaDrone package and the middleware module, and the human-in-the-loop interacts through the console display.

Currently, however, the data sent from the Parrot AR Drone 2.0 is not as accurate as we would like it to be. As mentioned, the accuracy of the GPS dongle is within 30 feet, which is not enough to avoid obstacles or collides with other drones. Furthermore, the GPS dongle has an update frequency of 1Hz, and therefore is too slow for the precise control needed for the drones. A possible solution to this has already been mentioned, namely to experiment with the high-quality chips that have been purchased so that they have a grounded USB connector. Even without a highly-accurate GPS, however, significant progress has been made.

3. Future Work

Our testbed has strong potential for future projects. Because of the flexibility of our design, as long as the appropriate calls are made to the RESTful web services, any simulation created with the MASON library can be used on the simulation side. This means that we can experiment with many different swarm models and UAV mission plans with relative ease. Also, since we have added a control GUI for the user, in the future we hope to be able to select and run multiple different MASON simulations simultaneously, chosen through the GUI. Some of these simulations could also enable look-ahead prediction, meaning it can extrapolate on the current data to show what would happen if a particular choice is made.

Our current MASON simulation could be improved as well, in order to discover more efficient and flexible ways to command and control UAVs. Our current MASON simulation can be classified as a target-search problem, but is rather simple in its approach. Using the GUI, we could have the user pick what points in the world to send the UAVs, instead of the automatic path currently developed. Also, currently the leader explores discovered targets in the order they were found, and we discovered that this can sometimes be inefficient if two consecutive targets found are far apart. This could be solved with a local traveling salesman problem solution, such as sorting targets as they are discovered, according to which is closest to the leader.

Other areas of the testbed have research potential as well. Research is being done to see what wireless protocols would be best to allow a swarm of UAVs to communicate effectively (see [6] for example), but often lack the ability to test the research in a real-world system. Though our drones are not military-grade UAVs, because of their Wi-Fi ability, they can still be used to demonstrate such research. Also, since the Parrot AR.Drone 2.0 has a high-definition camera, we can later incorporate vision techniques into our research. This could give real-world target data to our current simulation, instead of the simulated ones we have created.

4. Summary

Using the paradigm of the Dynamic Data-Driven Application System (DDDAS), we have created a testbed that can incorporate simulated or real-world data into a running simulation of a swarm of UAVs, and in reverse, send instructions to the simulated or real-world agents. Our real-world agent is the Parrot AR.Drone 2.0, which receives instructions through Wi-Fi and knows its location through a GPS dongle attached to it. Our swarm simulation uses the agent-based MASON library, particularly a modified, leader-follower version of Icosystem's swarmGame. Communication between the MASON simulation and the simulated drones is done with RESTful Web Services and a GUI controller, while interaction with the real-world drones is done using these and the additional JavaDrone library to translate instructions to the drones.

With our testbed, we can send instructions to a swarm of UAVs rather than only one, which is an important challenge in UAV swarm research. Currently, each military-grade UAV requires at least one operator [6], [9]. As UAVs are getting smaller and cheaper to make, and thus purchase, this requirement is not feasible for controlling a swarm of UAVs, which may one day contain hundreds of agents. We believe we are working toward solving this problem using DDDAS.

Acknowledgment

This research was supported in part under grants from the Air Force Office of Scientific Research, award No. FA9550-11-1-0351, and the National Science Foundation, award No. 1063084. Support also provided by the University of Notre Dame Center for Research Computing (CRC).

The authors would like to thank Christian Poellabauer, Brian Blake, Ryan McCune, Yi "David" Wei, Alexander Madey, and Radek Nabrzyski for their various contributions to this project.

References

[1] U.S. House 106th Congress, 1st Session, Public law 106-398, national defense authorization, fiscal year 2001, http://www.gpo.gov/fdsys/pkg/PLAW-106publ398/pdf/PLAW-106publ398.pdf.
[2] G. T. Bugajski, Architectural Considerations for Single Operator Management of Multiple Unmanned Aerial Vehicles, Master's thesis, Air Force Institute of Technology (2010).
[3] T. Ha, The UAV Continuous Coverage Problem, Master's thesis, Air Force Institute of Technology (2010).
[4] Office of the Secretary of Defense, Unmanned Aircraft Systems Roadmap: 2005 - 2030, Tech. rep., Department of Defense (2005).
[5] J. M. Abatti, Small Power: The Role of Micro and Small UAVs in the Future, Master's thesis, Air University (2005).
[6] R. L. Lidowski, A Novel Communications Protocol Using Geographic Routing for Swarming UAVs Performing a Search Mission, Master's thesis, Air Force Institute of Technology (2008).
[7] C. W. Reynolds, Flocks, Herds, and Schools: A Distributed Behavioral Model, in: Computer Graphics: SIGGRAPH '87, Vol. 21, ACM, 1987, pp. 25–34.
[8] S. Luke, Multiagent Simulation and the MASON Library, George Mason University, 1st Edition (2011).
[9] I. C. Price, Evolving Self-Organized Behavior for Homogeneous and Heterogeneous UAV or UCAV Swarms, Master's thesis, Air Force Institute of Technology (2006).
[10] D. Sebalj, Single Operator Control of Multiple Unmanned Air Vehicles: Situational Awareness Requirement, Master's thesis, Naval Postgraduate School (2008).
[11] H. V. D. Parunak, M. Purcell, R. O'Connell, Digital Pheromones for Autonomous Coordination of Swarming UAVs, Tech. rep., American Institute of Aeronautics and Astronautics (2002).
[12] F. Darema, Dynamic Data Driven Applications Systems: A New Paradigm for Application Simulations and Measurements, in: M. Bubak (Ed.), ICCS 2004, 2004, pp. 662–669, online. Available from http://www.springerlink.com.
[13] F. Darema, Dynamic Data Driven Applications Systems: New Capabilities for Application Simulations and Measurements, in: V. S. Sunderam (Ed.), ICCS 2005, 2005, pp. 610–615, online. Available from http://www.dddas.org/iccs2005/papers/darema.pdf.
[14] K. M. Milam, Evolution of Control Programs for a Swarm of Autonomous Unmanned Aerial Vehicles, Master's thesis, Air Force Institute of Technology (2004).
[15] D. J. Pack, B. E. Mullins, Toward Finding an Universal Search Algorithm for Swarm Robots, in: Proceedings of the 2003 IEEE/RSJ International Conference on Intelligence Robots and Systems, 2003, pp. 1945 – 1950.
[16] L. Barnes, M. Fields, K. Valavanis, Unmanned Ground Vehicle Swarm Formation Control Using Potential Fields, in: Mediterranean Conference on Control and Automation, 2007.
[17] B. Liu, et al, Controllability of a Leader Follower Dynamic Network With Switching Topology, in: IEEE Transactions on Automatic Control, Vol. 53, 2008, pp. 1009 – 1013.
[18] P. Vincent, I. Rubin, A Swarm-Assisted Integrated Communication and Sensing Network, in: R. Suresh (Ed.), Battlespace Digitization and Network-Centric Systems IV, Vol. 5441, SPIE, SPIE Digital Library, 2004, pp. 48 – 60.
[19] X. WangBao, C. XueBo, Artificial Moment Method for Swarm Robot Formation Control, Science in ChinaOnline. Available from http://www.springerlink.com.
[20] P. Gaudiano, E. Bonabeau, B. Shargel, Evolving Behaviors for a Swarm of Unmanned Air Vehicles, in: Proceedings of IEEE 2005, IEEE, 2005, online. Available from IEEEXplore.
[21] K. E. N. Karl Altenburg, Joseph Schlecht, An Agent-based Simulation for Modeling Intelligent Munitions, online. Available from http://www.cs.ndsu.nodak.edu/ nygard/research/munitions.pdf.
[22] J. N. Slear, AFIT UAV Swarm Mission Planning and Simulation System, Master's thesis, Air Force Institute of Technology (June 2006).
[23] P. Gaudiano, et al, Swarm Intelligence: a New C2 Paradigm with an Application to Control Swarms of UAVs (October 2003).
[24] I. A. Gravagne, R. J. Marks, Emergent Behaviors of Protector, Refugee, and Aggressor Swarms, in: IEEE Transactions on Systems, Man, and Cybernetics. Part B: Cybernetics, Vol. 37, IEEE, 2007, pp. 471 – 476.
[25] G. R. Madey, et al, Applying DDDAS Principles to Command, Control and Mission Planning for UAV Swarms, in: Procedia Computer Science, Vol. 9, Elsevier Ltd., 2012, pp. 1177 – 1186, online. Available from www.sciencedirect.com.
[26] Parrot, AR.Drone 2.0, http://ardrone2.parrot.com (2012).
[27] Codeminders, javadrone - AR.Drone Java API, http://code.google.com/p/javadrone/ (2011).
[28] Parrot, AR.Drone 2.0 SDK, https://projects.ardrone.org/attachments/download/434/ARDrone_SDK_2_0.tar.gz (2012).
[29] Mentor Graphics - ex Codesourcery, ARM cross-compile environment, http://www.mentor.com/embedded-software/codesourcery.
[30] E. S. Raymond, gpsd library, http://www.catb.org/gpsd/ (2004).
[31] Thorsten Hoeger, Taimos GmbH, gpsd4java, http://www.ohloh.net/p/gpsd4java (2011).