Integrating Heterogeneous Tools for Physical Simulation of multi-Unmanned Aerial Vehicles

Fabio D'Urso, Corrado Santoro, Federico Fausto Santoro
University of Catania
Department of Mathematics and Informatics
Viale Andrea Doria, 6
95125 - Catania, ITALY
EMail: {durso,santoro}@dmi.unict.it, federico.santoro@unict.it

***Abstract*—This paper presents a multi-layer software architecture to simulate, in a accurate and realistic way, a set of unmanned aerial vehicles (UAVs) operating in a specific mission. A set of tools are employed, each one to simulate a specific part of the overall UAV hardware and software structure: a 3D visualization engine, a physical simulator, the flight stack and a network simulator to handle interactions among UAVs. A software architecture able to orchestrate and coordinate such tools is proposed, based on multiple layers of processes divided into two categories. The described approach is based on a protocol system for exchanging messages to synchronize the various simulation tools. The simulation of the unmanned aerial vehicles can therefore be performed on a single machine or distributed on several machines in order to create a distributed simulation and spread the workload. In this way, it is possible to simulate the behavior of the UAVs and also to reason about the problems due to network communications.**

I. Introduction

Recent research in the field of *ummanned aerial vehicles* (UAVs) shows a trend to investigate approaches and algorithms for *autonomous flights* in *flocks* [17], [19], [6], [8], [10], [14], [5], [9], [11], [16], [20], [7], [22], [21], [3]. Such a research area implies the integration of different technologies.

First of all, UAVs must be made able to fly more or less autonomously; to achieve this objective, a proper CPU-based *flight control system* must be employed, implementing all the control algorithms to perform UAV stabilisation, position control, path following, etc. In addition, since UAVs of a flock should be able to communicate to each other in order to perform coordination, a "inter-UAVs" wireless communication system is mandatory, together with an adequate protocol guaranteeing a communication which is reliable to a certain extent. Often also an interaction with a *base station* is required, in order to obtain telemetry data and let operators to have a form a mission control; this requires obviously a wireless communication channel that, however, could have different characteristics with respect to the inter-UAV system. Flocking not only implies the presence of communication channels but also—and this is indeed more important—a *flocking algorithm* able to drive UAVs in forming and maintaining the flock shape and, altogether, perform the flight mission; in this sense the literature reports many approaches that can be mainly subdivided in *centralised* and *distributed*. Centralised approaches adopt on a central entity that elaborates and sends flight or path commands to UAVs; while these solutions can guarantee optimality in UAVs distribution or area coverage, they have the drawback of presenting a single point of failure, that is the central entity itself. Distributed approaches are based on algorithms that let UAVs (by interacting) self-organise and self-maintain the flock, and usually exploit the classical *separation*, *alignment* and *cohesion* rules [17] that are proper of flocking [10], [9]; these approaches, that indeed are preferred to centralised ones, are surely more fault-tolerant but often are not able to achieve optimality, since each UAV performs its evaluation of flight path basing on the information, coming from other UAVs, that could not be complete or updated.

As a result, UAV flocking integrates concepts, techniques and solutions proper of *control systems*, *wireless communication*, *coordination and agreement* and *self-organisation*: the natural consequence of this statement is that implementing such kind of systems is quite far from a simple task. And to make things yet more complicated, *testing* UAV-based solutions is particularly hazardous: flying machines have the bad characteristic, when a fault occurs (and during testing this could be a common case), of falling into the ground, thus provoking crashes and the need to repair or rebuild the mechanical frame[1]. For this reason, the common approach is to make extensive *simulation campaigns* before testing solutions on real UAVs.

Simulators are thus key tools in this research field, but, in order to be as realistic as possible, they must integrate all the technology aspects cited above plus another not less important one: the *physical behaviour* of the UAVs.

Indeed, UAVs are weird mechanical system with a dynamics that tends to oscillation or instability, above all in presence of specific environmental conditions (e.g. wind or turbulence). Unfortunately, a simulator able to integrate all of this aspects does not exist, but there are many simulators implementing each one of the specific aspects described: an integration of all of them is the simplest way, but it is not straightforward and often requires hacks or instrumenting the code in order to achieve the objective.

This paper goes in this last direction: it presents a technique to integrate different simulation tools in a single large envi-

[1]When such crashes are not hazardous for people.

ronment able to simulate: *UAV physics*, *UAV control*, *wireless communication*, *flocking algorithms*. In particular, the tools employed are *Gazebo* [12] (for physics), *ArduPilot* [1], as flight control platform and *ns-3* [18], for wireless network simulation. All of them are integrated into a large system that coordinates them and the simulation. The adopted solution is also modular and distributed: while a 3D visualisation environment is provided (by Gazebo), it can also be disabled in order to perform batch simulations and gather specific numerical results; moreover, since in the presence of a good number of UAVs the CPU cost of simulation could become very high, the various parts can be also split over several interconnected computers.

The paper describes the software architecture we used in implementing such an integrated simulator and is organised as follows. Section II describes the single simulation tools used in the integration. Section III presents the integrated simulator. Section IV provides our conclusions.

II. Tools

A. Gazebo

Gazebo[2] [12] is a 3D robotic simulator with a physical engine capable of simulating, in efficient and accurate way, a large number of robot types. Simulations can be performed by simulating both an indoor or outdoor environment. Gazebo also provides the support to simulate various types of sensors, actuators and interfaces.

Robot models are designed by means of an XML file that defines the structure in terms of fixed parts, joints with their physical parameters (geometry and dynamics), driving characteristics, and specific sensors. Robots in Gazebo can be controlled externally by means of software *plugins*; the model is based on a *publisher-subscriber* paradigm that lets plugins obtain data from sensors and intervene on actuators by sending proper commands or set-points.

The software structure of Gazebo is designed to keep separated the 3D visualisation part (called *gzclient*) from the physical engine (called *gzserver*). A simulation can be executed without display (this is useful to run batch simulations) and both parts can run in two different computers in order to take advantage of a distributed environment (both parts are CPU-intensive tasks thus the execution in different servers can speed-up the simulation).

This simulator is widely used to test new algorithms in virtual environments and to analyse robots' behaviors. In addition, Gazebo takes advantage of multiple physical engines and provides an extensive library of ready-to-use robot models. Of course, the simulator also allows one to build customised models. Gazebo is also fully integrated with the ROS system [15] thus allowing users to use the same code tested on Gazebo directly on the physical robot.

A peculiar aspect of this type of simulations is that it is possible to run them in non real-time (i.e. either faster or slower than a wall-clock time source) without affecting accuracy of the simulated system.

[2]http://gazebosim.org/

B. ArduPilot

ArduPilot [1] is a control stack for UAVs (with the ArduCopter subproject) and UGVs. It is an open-source product with a wide community of developers and, together with PX4 [13], is one of the most widely used flight control stacks for drones. It runs upon a variety of hardware platforms and provides all the algorithms to control stability and flight of a UAV, including the autonomous flight over a path of GPS waypoints.

ArduPilot can be externally interfaced (via serial line or TCP connection) by means of an ad-hoc protocol called *MAVLink* [4] that is specifically designed to connect a Ground Control Station (for telemetry or set-up operations) or an external computer that implements high-level mission control. In addition, ArduPilot provides support for Software In The Loop (SITL) simulations through the Gazebo simulator; SITL is a very useful tool because it enables to test the high-level logic using the same interface a real UAV would offer (i.e. MAVLink).

An interesting additional tool that is often integrated with ArduPilot is *DroneKit* [2], a set of applications, libraries and APIs, provided for various programming languages and platforms, that implement the MAVLink protocol and allow developers to easily write high-level applications for drones that run the ArduPilot stack.

C. Network Simulator 3

Network Simulator 3 (ns-3) [18] is a network simulator capable of simulating several types of infrastructures and their network protocols.

Compared to the its previous version (ns-2), which was written in C++ but required simulations to be written in object-oriented Tcl (o-Tcl), the new version lets a programmer define a simulation directly in C++ or Python, whereas ns-2's mixture of o-Tcl and C++ was hard to debug and unfamiliar to most people. Ns-3 is designed to run "pure C++"-based models, for greater performance, and it also provides a Python-based scripting API that allows ns-3 to be integrated with other Python-based environments or programming models. Users of ns-3 can write their simulations as either C++ *main()* programs or Python programs.

The new version has been developed to be as modular as possible. Indeed, the structure of ns-3 makes it simple to develop new models of simulations or customise existing ones.

The simulator supports most wired, wireless and mobile network protocols as well as routing protocols. Furthermore, ns-3 supports interactions with the "real world", such as the creation of a simulated network where nodes can actually ping a server on the Internet.

III. The Integrated Simulation Environment

The tools briefly presented in Section II are the basic blocks to build a complete simulation environment for a flock of

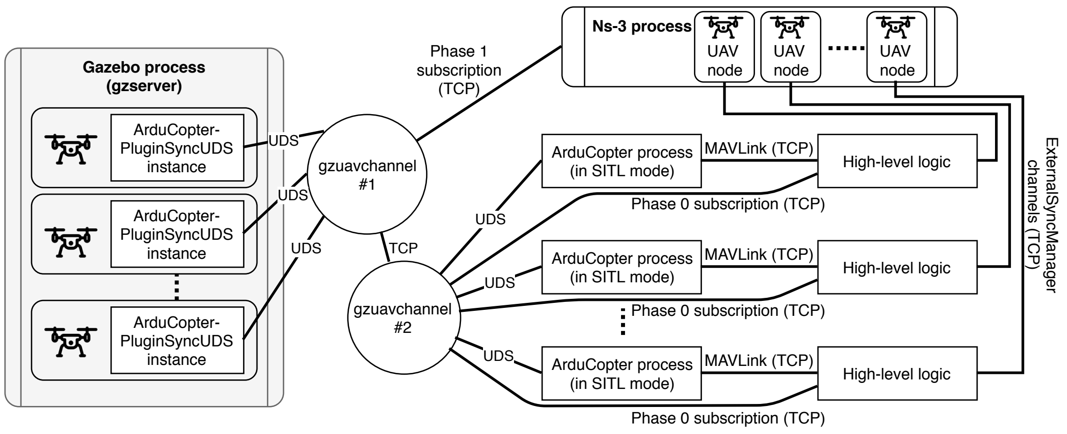

Ns-3 process
UAV node
UAV node
UAV node
Phase 1 subscription (TCP)
Gazebo process (gzserver)
ArduCopter-PluginSyncUDS instance
ArduCopter-PluginSyncUDS instance
ArduCopter-PluginSyncUDS instance
UDS
UDS
UDS
gzuavchannel #1
TCP
gzuavchannel #2
UDS
UDS
UDS
UDS
ArduCopter process (in SITL mode)
MAVLink (TCP)
High-level logic
Phase 0 subscription (TCP)
ArduCopter process (in SITL mode)
MAVLink (TCP)
High-level logic
Phase 0 subscription (TCP)
ArduCopter process (in SITL mode)
MAVLink (TCP)
High-level logic
Phase 0 subscription (TCP)
ExternalSyncManager channels (TCP)

Fig. 1: Software Architecture of the Integrated Simulator

UAVs interconnected through a wireless network; the main issue with such tools is that they are not designed to work together in an integrated manner. Other than the obvious problems regarding the way in which these tools should exchange data, there are important issues related to *clock synchronization*: indeed, since all the tools are designed to provide a simulation which is more realistic as possible, each of them includes its own notion of *time* that is *(i)* rather different than wall-clock time and *(ii)* tied to the events they simulate. However, the overall integrated environment must have a *common* notion of time, otherwise the simulation will not proceed realistically. The aim of our integrated environment is thus a set of modules and a proper protocol that lets the employed tools interact and proceed in a *strictly synchronised way*.

A. Software Architecture

The overall software architecture of the simulator is depicted in Figure 1 and described below. Each UAV is represented by the following software modules:

1) *Gazebo model*
2) *ArduCopterPluginSyncUDS*
3) *ArduCopter process*
4) *High-level Logic*
5) *UAV node*

For each simulated UAV, an instance of all the software modules above is created; each instance runs within its specific environment (e.g. Gazebo or ns-3) or as a stand-alone process. All modules are then coordinated by some GZUAVCHANNEL processes according to a schema and protocol which is described in Section III-B.

The first two models are related to the graphical and physical simulation and run inside Gazebo. The *Gazebo model* represents the definition of the frame and inertia of the UAV, which, in our experiments, is a quadri-rotor VTOL aerial vehicle. The *ArduCopterPluginSyncUDS* is a Gazebo plugin that interacts with the simulated model by directly driving the motors of the quadri-rotor and reading its pose (absolute position, euler angles, angular rates, etc.).

The *ArduCopter process* is a stand-alone process running an instance of the ArduPilot flight stack which is specifically compiled to support the "software-in-the-loop" (SITL) mode. In a similar way, the *High-level Logic (HLL)* is a process implementing the mission control for the single UAV; in the case of flocking, the HLL implements the specific flocking algorithm and path planning, as well as the functionalities related to UAV interaction and messaging, which are then simulated by means of ns-3. The HLL interacts with the flight stack through the MAVLink channel that, in the simulated environment, is made of a TCP connection (this interface is supported by means of the DroneKit library).

Finally, the *UAV node* represents the communication endpoint (i.e. the wireless interface) of the UAV; it is an object that runs within ns-3, which enables the simulation of the wireless communication channel.

B. Interaction and Synchronization

All the software modules cited so far are responsible to simulate or implement specific parts of the overall behaviour of each UAV; however they must be coordinated in such a way as to let them proceed in a strictly synchronized way and with a common notion of simulation time.

In our environment, this task is performed by GZUAVCHANNEL, a software module which mainly serves as a clock synchroniser in case of multiple UAVs. GZUAVCHANNEL also defines a protocol for external processes to connect, be notified of changes in the position of UAVs, and synchronise their own clocks as well. Each process intending to take part in the simulation performs a *subscription* to the GZUAVCHANNEL, specifying its *type*; in particular, we identified two classes of external processes:

- **Type 0 subscribers**: processes that logically run inside the simulated environment, e.g. a DroneKit-based process that connects to ArduPilot via MAVLink and implements the high-level logic of an autonomous UAV;
- **Type 1 subscribers**: processes that implement part of the simulation environment, e.g. the ns-3 network simulator.

Depending on its type, during the simulation each process

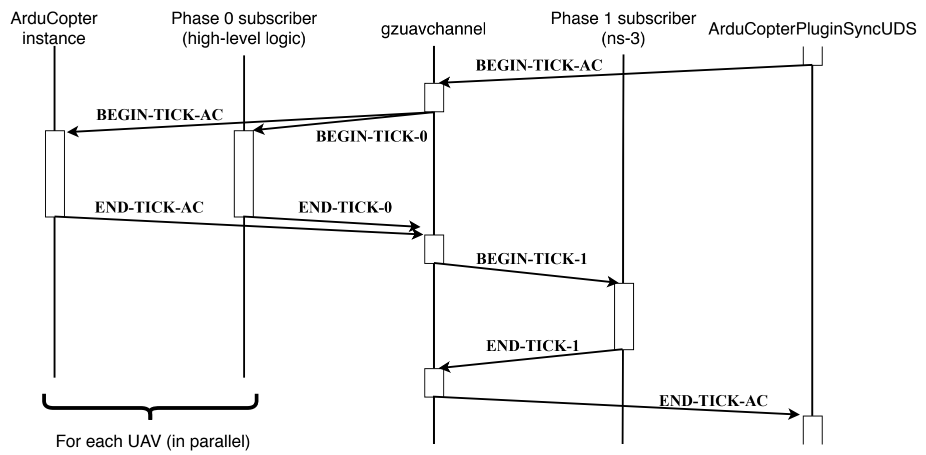


Fig. 2: Sequence diagram showing messages exchanged among processes for each simulation step

receives a specific notification from the GZUAVCHANNEL; to this aim, GZUAVCHANNEL coordinates the overall simulation execution so that all **Type 0** subscribers can proceed in parallel while **Type 1** subscribers are blocked and, vice versa, when all **Type 1** subscribers run in parallel, **Type 0** subscribers are blocked, in an alternating fashion (Figure 2). **Type 1** subscribers also receive the position of all UAVs at the beginning of each simulation step. Communication between GZUAVCHANNEL and **Type 0/1** subscribers uses a TCP channel. In addition, the internal architecture lets several GZUAVCHANNELs interact (also via TCP); this is useful when, to reduce the computational load, a distributed system is used to perform simulation. In this case, the various processes taking part to the simulation can be split over different machines of a distributed system and can be coordinated by the various GZUAVCHANNELs running upon such machines.

Synchronisation of the activities of a simulation is initiated by the physical engine of Gazebo (*gzserver*) which, since it manages the physics of the agents, is the entity in charge of governing the simulation clock. At the beginning of each time step, the various instances of ArduCopterPluginSyncUDS within Gazebo send a **BEGIN-TICK-AC** message to the GZUAVCHANNEL that starts coordinating activities of the simulation according to the protocol depicted in the sequence diagram in Figure 2. The **BEGIN-TICK-AC** is first sent to the relevant ArduCopter instance which performs a step of its control activities[3]; in parallel, GZUAVCHANNEL announces the start of **Phase 0** by sending a **BEGIN-TICK-0** message to all subscribed **Type 0** processes. **Phase 0** is particularly important since it is the moment in which all Type-0 processes can execute their simulation step: it is in this phase that HLL processes (for example) execute one step of their flocking (or other) algorithm. This phase can also include an interaction with the flight stack or the network; indeed, the algorithm surely would include the computation of next speed or position set-points for the UAV, as well as the transmission or reception of messages; these operations are performed during this phase by means of DroneKit, for the set-points or any other of interaction with ArduCopter, and by interacting with ns-3 to simulate message exchange through the network. In particular, in this phase, HLL processes can send data over the network by providing ns-3 a packet through the **ENQUEUE-NS3** message, as detailed in Figure 3a; here the message is not really processed but only placed in a queue: the simulated transmission and delivery are not performed by ns-3 until Phase 1.

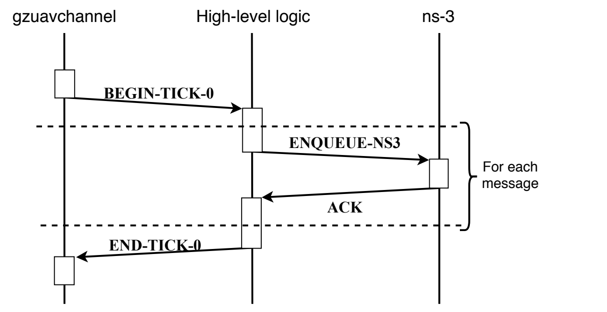
When the HLL process completed its step, it replies to GZUAVCHANNEL with an **END-TICK-0** message. In the same way, when ArduCopter processes have completed their simulation steps, they reply to GZUAVCHANNEL with an **END-TICK-AC** message. When all such replies have been gathered, the Phase 0 is completed.

After that, the GZUAVCHANNEL starts the **Phase 1** by sending **BEGIN-TICK-1** to all subscribed **Type 1** processes and waiting for **END-TICK-1**. As Figure 3b shows, during Phase 1, ns-3 is asked to do its simulation step by processing enqueued messages and delivering them (if necessary) to HLL processes. Only when all **END-TICK-1** messages have been received, GZUAVCHANNEL delivers the **END-TICK-AC** messages to Gazebo, allowing it to proceed to the next simulation step.

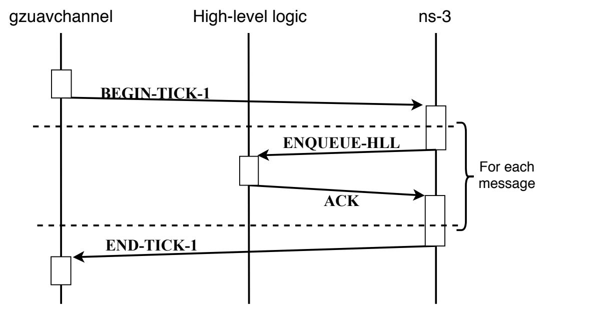
C. *Manging Simulations in a Distributed Environment*

The way in which the GZUAVCHANNEL is designed allows a user to distribute the simulation over different interconnected servers, in order to take advantage of a multi-node environment and reduce CPU load. In this way, the computational workload can be spread over a network, by e.g. partitioning the set of UAV into a number of groups, each controlled by a different GZUAVCHANNEL instance on a dedicated computational node. As it is reported in Figure 4, in the most general form, a tree can be created, in which the nodes are GZUAVCHANNEL instances and the edges are TCP connections; in such a tree, the root is represented by

[3]See http://ardupilot.org/dev/docs/apmcopter-programming-attitude-control-2.html for the details about the control loops of ArduCopter.



(a) Messages sent by UAVs are enqueued in ns-3 during Phase 0



(b) Ns-3 runs the network simulation, including the delivery of messages to UAVs, during Phase 1

Fig. 3: Interactions between high-level logic UAV processes and ns-3

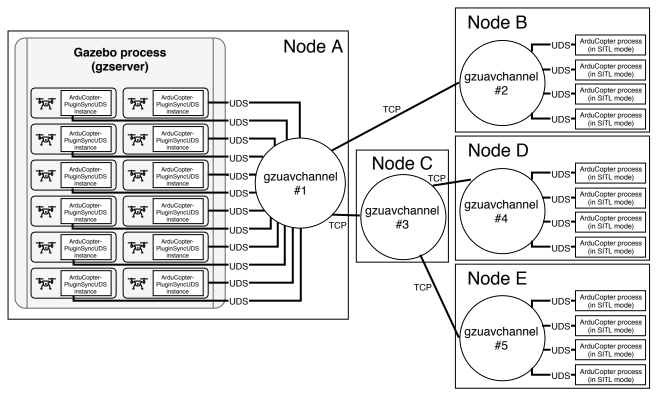


Fig. 4: Architecture of the Integrated Simulator in a Distributed Environment

the instance directly connected to Gazebo, while the leaves are connected to ArduCopter processes.

In a distributed environment, the dynamics of interaction are quite similar to the centralised case; coordination is performed by using the messages **BEGIN-TICK-AC** and **END-TICK-AC** that are exchanged by GZUAVCHANNELs. In detail, when the first **BEGIN-TICK-AC** is received from the root GZUAVCHANNEL, it is forwarded to children nodes, down to the leaves of the tree. Each GZUAVCHANNEL thus behaves according to the protocol in Figure 2, coordinating the processes which it is has the responsibility for. Finally, when Phases 0 are completed, the **END-TICK-AC** message is generated by leaf nodes and forwarded along the tree until the root is reached, thus signalling Gazebo the end of the simulation tick.

D. Implementation Issues

The framework user can either directly interact with ArduPilot, employing its built-in capability to take-off and follow preprogrammed paths of GPS waypoints, or use an external system (usually a companion computer) to send position/speed targets in real-time according to a custom high-level logic. The latter scenario offers greater flexibility and it is the only way to program UAVs with complex behaviors. However, it is non-trivial to integrate such external systems into the simulation, because it is necessary to synchronise their clocks as well (so that the timing of commands such as “take-off, then pause for 5 seconds, then go to a given GPS point” stays coherent to the simulated environment). We developed a Python module that subscribes to **Phase 0** and *overrides* sleep() and time() with versions that use the simulation clock. The Python language was chosen because of the availability of the DroneKit library, which greatly simplifies the tasks of connecting to ArduPilot, sending and receiving MAVLink messages (offering the same interface both for a real and a simulated UAV).

IV. Conclusions

The ability to manipulate and test algorithms in a simulated physical environment greatly facilitates software development and validation phases of autonomous UAVs. ArduPilot and Gazebo are two software solutions that, if used together, offer a solid and accurate UAV simulation toolkit.

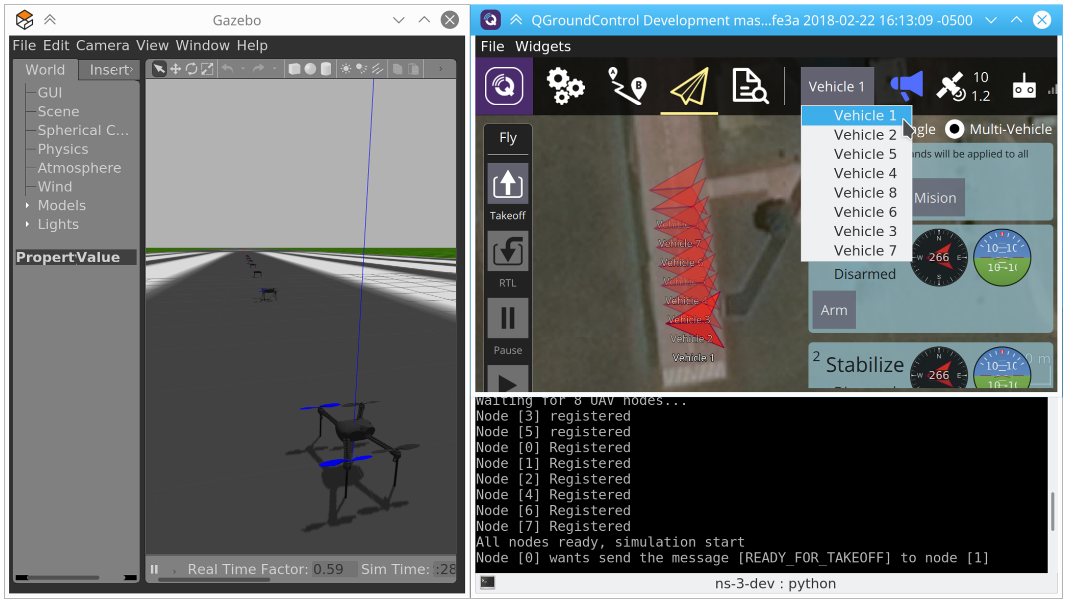
In the context of UAV flocks, each UAV also needs to be able communicate (either with a base station or a among UAVs themselves), using wireless networking hardware and protocols. However, wireless communications are often far from ideal, and the need for accurate network modelling and simulation tools arises.

The proposed multi-layered software structure combines Gazebo, ArduPilot and the ns-3 network simulator, resulting in a complete tool that enables not only to physically simulate the flight of UAVs, but also to integrate, in a realistic way, several wireless networking technologies that real UAVs are usually equipped with.

The described architecture has been implemented and validated (see Figure 5). The next steps will be the implementation of a full-fledged flocking and area coverage algorithm (e.g. [9]), in order to test the solution in a complex and realistic scenario, and the comparison of numerical results to those of a real flock of UAVs.

V. Acknowledgments

This work is supported by project CLARA SCN_00451 funded by the Italian MIUR.



```
Waiting for 8 UAV nodes...
Node [3] registered
Node [5] registered
Node [0] Registered
Node [1] Registered
Node [2] Registered
Node [4] Registered
Node [6] Registered
Node [7] Registered
All nodes ready, simulation start
Node [0] wants send the message [READY_FOR_TAKEOFF] to node [1]
```

Fig. 5: Screenshot of a run of Gazebo, ArduCopter and ns-3 with several UAVs

REFERENCES

[1] Ardupilot open source autopilot. [Online]. Available: http://ardupilot.org/

[2] Dronekit: Developer tools for drones. [Online]. Available: http://python.dronekit.io

[3] "Locust: Autonomous, swarming uavs fly into the future," http://www.onr.navy.mil/Media-Center/Press-Releases/2015/LOCUST-low-cost-UAV-swarm-ONR.aspx.

[4] Mavlink micro air vehicle communication protocol. [Online]. Available: http://mavlink.org/

[5] M. D. Benedetti, F. D'Urso, F. Messina, G. Pappalardo, and C. Santoro, "Uav-based aerial monitoring: A performance evaluation of a self-organising flocking algorithm," in *2015 10th International Conference on P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC)*, Nov 2015, pp. 248–255.

[6] N. Bouraqadi and A. Doniec, "Flocking-based multi-robot exploration," in *Proceedings of the* 4^{th} *National Conference on Control Architectures of Robots*, Toulouse, France, 2009.

[7] G. Cai, J. Dias, and L. Seneviratne, "A survey of small-scale unmanned aerial vehicles: Recent advances and future development trends," *Unmanned Systems*, vol. 2, no. 02, pp. 175–199, 2014.

[8] G. Chmaj and H. Selvaraj, "Distributed processing applications for uav/drones: a survey," in *Progress in Systems Engineering*. Springer, 2015, pp. 449–454.

[9] M. De Benedetti, F. D'Urso, G. Fortino, F. Messina, G. Pappalardo, and C. Santoro, "A fault-tolerant self-organizing flocking approach for uav aerial survey," *J. Netw. Comput. Appl.*, vol. 96, no. C, pp. 14–30, Oct. 2017. [Online]. Available: https://doi.org/10.1016/j.jnca.2017.08.004

[10] M. De Benedetti, F. D'Urso, F. Messina, G. Pappalardo, and C. Santoro, "3d simulation of unmanned aerial vehicles," in *XVIII Workshop "Dagli Oggetti agli Agenti"*. CEUR-WS, 2017.

[11] J. Gonçalves and R. Henriques, "Uav photogrammetry for topographic monitoring of coastal areas," *ISPRS Journal of Photogrammetry and Remote Sensing*, vol. 104, pp. 101–111, 2015.

[12] N. Koenig and A. Howard, "Design and use paradigms for Gazebo, an open-source multi-robot simulator," in *International Conference on Intelligent Robots and Systems*, Sendai, Japan, 2004, pp. 2149–2154.

[13] L. Meier, D. Honegger, and M. Pollefeys, "PX4: A node-based multithreaded open source robotics framework for deeply embedded platforms," in *Robotics and Automation (ICRA), 2015 IEEE International Conference on*, may 2015.

[14] P. Pace, G. Aloi, G. Caliciuri, and G. Fortino, "A mission-oriented coordination framework for teams of mobile aerial and terrestrial smart objects," *Mobile Networks and Applications*, vol. 21, no. 4, pp. 708–725, 2016.

[15] M. Quigley, K. Conley, B. P Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y Ng, "Ros: an open-source robot operating system," 01 2009.

[16] S. A. Quintero, G. E. Collins, and J. P. Hespanha, "Flocking with fixed-wing uavs for distributed sensing: A stochastic optimal control approach," in *American Control Conference (ACC), 2013*. IEEE, 2013, pp. 2025–2031.

[17] C. Reynolds, "Flocks, herds and schools: A distributed behavioral model," in *Proceedings of the* 14^{th} *Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH '87. New York, NY, USA: ACM, 1987, pp. 25–34.

[18] G. F. Riley and T. R. Henderson, *The ns-3 Network Simulator*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 15–34.

[19] G. Vásárhelyi *et al.*, "Outdoor flocking and formation flight with autonomous aerial robots," in *IEEE/RSJ International Conference on Intelligent Robots and Systems*, ser. IROS '14, Chicago, IL, USA, 2014, pp. 3866–3873.

[20] C. Virágh *et al.*, "Flocking algorithm for autonomous flying robots," *Bioinspiration & biomimetics*, vol. 9, no. 2, p. 025012, 2014.

[21] Y. Wei, M. B. Blake, and G. R. Madey, "An operation-time simulation framework for uav swarm configuration and mission planning," *Procedia Computer Science*, vol. 18, pp. 1949–1958, 2013.

[22] L. Weng, Q. Liu, M. Xia, and Y. Song, "Immune network-based swarm intelligence and its application to unmanned aerial vehicle (uav) swarm coordination," *Neurocomputing*, vol. 125, pp. 134–141, 2014.