ArduSim: Accurate and real-time multicopter simulation

Francisco Fabra[a,*], Carlos T. Calafate[a,**], Juan Carlos Cano[a,**], Pietro Manzoni[a,**]

[a]*Department of Computer Engineering (DISCA), Universitat Politècnica de València (UPV), Camino de Vera s/n, Valencia, 46022 Spain*

Abstract

As the popularity and the number of Unmanned Aerial Vehicles (UAVs) increases, new protocols are needed to coordinate UAVs when flying autonomously, and to avoid that these UAVs collide with each other. Directly testing such novel protocols on real UAVs is a complex procedure that requires investing much time, money and research effort. Hence, it becomes necessary to have the possibility to first test different solutions using simulation. Unfortunately, existing tools present significant limitations: some of them only simulate accurately the flight behavior of one UAV, while some other simulators can manage several UAVs simultaneously, but not in real-time, thus loosing accuracy regarding the mobility pattern of the UAV. In this work we address such problem by introducing ArduSim, a novel simulator that allows controlling in **soft real-time** the flight and communications of multiple UAVs, being the developed protocols directly portable to real devices. The contributions of this work include: (i) the ArduSim simulation platform, which allows realistic simulation and control of multiple UAVs simultaneously, offering functionalities not provided by existing alternatives; (ii) a model for the WiFi communications link between UAVs, based on real experiments, and that is integrated into ArduSim itself; and (iii)

*Principal corresponding author
**Corresponding author
Email addresses: `frafabco@cam.upv.es` (Francisco Fabra), `calafate@disca.upv.es` (Carlos T. Calafate), `jucano@disca.upv.es` (Juan Carlos Cano), `pmanzoni@disca.upv.es` (Pietro Manzoni)

a thorough study of the scalability performance of our simulator.

Keywords: unmanned aerial vehicles, multicopters, real-time simulation, MAVLink, performance evaluation

1. Introduction

Unmanned Aerial Vehicles (UAVs), colloquially known as drones, are flying devices able to perform programmed flights or being remotely controlled. During the past few years they have gained high relevance thanks to their capabilities in terms of performing a wide range of tasks. For instance, planned missions can be defined to supervise farmlands, deliver packages to remote locations, or contributing to create delay-tolerant networks in the scope of Smart Cities [1]. Moreover, by adopting adequate algorithms, it is possible to develop new routing protocols [2], control the flight of a group of UAVs acting as a swarm [3], or dynamically create an aerial network infrastructure in a dynamic, on-demand fashion [4].

These new applications demand for the establishment of communication protocols between UAVs to avoid collisions when they are in close proximity, and to coordinate them when performing complex tasks, such as those undertaken by UAV swarms.

Experimenting with UAV-based networking in order to develop and validate new protocols presents several restrictions including: (i) pilots should meet the regulation requirements of each country, (ii) weather conditions should be favorable, (iii) battery lifetime is quite limited, and (iv) certain applications require testing with a high number of UAVs simultaneously. For instance, Lee et al. [5] analyze a new routing protocol between UAVs and a Ground Control System (GCS) using up to six UAVs simultaneously, while Y. Chai et al. [6] test a UAV formation protocol with up to six virtual UAVs.

In general, the approach adopted by most researchers relies on simulation. However, simulations should be as realistic as possible, that is, they should account for the physical properties and flight behavior of the aerial vehicle, and they should also integrate a model for wireless communications between UAVs that resembles real-life behavior. In addition, it is important that the simulation environment is able to manage several UAVs simultaneously, and that the code developed is compatible with existing flight controllers, thereby simplifying the process of porting the developed protocols to real UAVs, and completing the development cycle.

In this paper we detail and validate our proposed ArduSim platform, which allows us to simultaneously simulate up to 256 multicopters in a realistic manner, while also allowing to develop and validate communication protocols for inter-UAV coordination in the scope of Flying ad-hoc networks (FANETs) without resorting to real UAVs. ArduSim allows saving time and money, and it also avoids the risks inherent to field tests.

In ArduSim, the communication with virtual UAVs relies on the MAVLink protocol [7], a *de facto* standard for communicating with the flight controller of a real UAV. This way we are able to simplify the process of bringing the implemented protocols to real devices. In fact, ArduSim has been designed to directly allow porting the developed protocols to real UAVs merely by modifying a runtime parameter, as detailed in section 3.4.

ArduSim includes an experimental model of the communications link between multicopters (drone-to-drone communcation, or D2D), without the intervention of a GCS, which has been obtained based on experimentation using real multicopters, and it accounts for packet losses occurring during transmission. The wireless communications technology chosen is WiFi, as it offers plenty of bandwidth, has an acceptable communications range, and is widely available and adopted. Specifically, the model proposed for the communications link is based on the IEEE 802.11a standard (5 GHz band), and it has been derived from experimental results using actual UAVs based on the strategy defined in [8]. Regarding this frequency band choice, it is based on the findings of our previous work [9]. This model has a computational cost that is significantly low compared to the one used in traditional network simulators by being simplified to a polynomial equation, thus requiring fewer computational resources than the Friis equation, or than models like Okumura and Nakagami [10], without sacrificing accuracy. Our simulation model also accounts for the medium occupancy (carrier sensing and collision detection) in the communication between UAVs, as detailed in section 3.3.

ArduSim is based on the Software In The Loop (SITL) simulator [11], which is able to simulate with great accuracy different types of UAVs. SITL is also open source, multiplatform, and it offers direct communications with the simulated flight controller using a TCP connection. In addition to the UAVs physical characteristics, it is also able to simulate the presence of wind. ArduSim was developed so that each SITL instance runs as an independent process. So, ArduSim created a high-level logic layer where as many SITL instances as required are launched for each experiment.

The contributions of this work are the following:

- ArduSim, an open platform offering realistic and near real-time simulation of UAVs through a direct MAVLink-based connection. In addition, a detailed graphical interface and extensive logging features have been developed.
- A realistic communications model based on experimental data, obtained with real UAVs using the IEEE 802.11a protocol.
- A thorough study of the scalability performance of our simulator.

The remainder of this paper is organized as follows: in the following section we present the most relevant works in the field of UAV swarm simulation. In section 3 we explain the internal structure of the simulator, we provide implementation details concerning the mechanism used to control virtual UAVs, we justify the wireless link model used in the simulator, and we explain how to use the simulator, and how to deploy the developed protocol in real UAVs. Then, section 4 includes a thorough validation of the simulator. Lastly, section 5 concludes the paper and refers to future works.

2. Related Work

Recently, UAV simulation has become a hot topic, especially when aiming at developing swarm-based solutions. In [12] we can find a list of existing flight simulators. Several of them mimic with considerable accuracy the characteristics and physical properties of the UAV, although (i) their code is proprietary, (ii) they are only compatible with a few platforms, and (iii) they only allow controlling a single UAV at a time, thereby failing to offer inter-UAV communications support. On the other hand, generic network simulators like ns-2, ns-3, and OMNeT++ [13, 14] allow simulating with great accuracy the communications link, but they are unable to simulate physical UAV properties, as well as the mobility between UAVs, in a realistic manner. Ben-Asher et al. [15] created IFAS, a network simulator aimed to ad-hoc networks, although it is only oriented to develop routing protocols, failing to provide real-time network simulation.

Other authors have addressed the simulation of multiple UAVs. An example is the work of Richard Garcia et al. [16] where they introduce a simulator based on X-Plane that is able to emulate up to 10 planes or helicopters; however, differently from ours, their solution is not oriented to multicopters. Moreover, their solution requires a PC for each simulated UAV, whereas our solution is able to simulate many more UAVs in a single machine. In [17], J. Holt et al. develop a symbiotic simulation architecture, although it is exclusively focused on the development and analysis of collision avoidance protocols.

UAVSim [18] is focused on securing the communications, and runs over OMNeT++. UAVSim works by extending OMNeT++, introducing a mobility model based on the properties defined for a specific UAV model which allows updating each UAV's route based on the interactions with other neighboring UAVs.

Simbeeotic [19] is able to simulate with great accuracy a swarm of UAVs using JBullet, but it relies on its own language to control these virtual UAVs, which difficults bringing these protocols to real-world devices. UB-ANC [20]

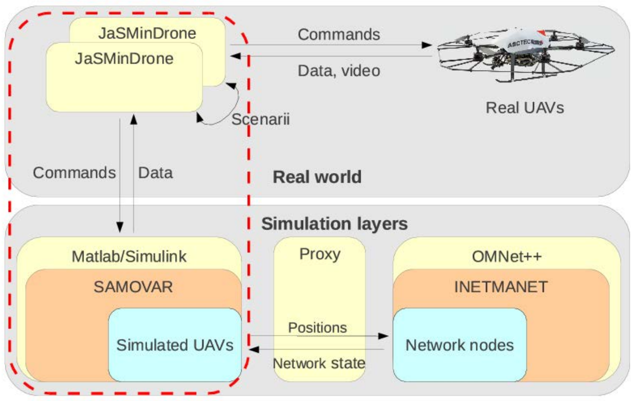


Figure 1: AEOTURNOS pilot interface, UAV, and communications network simulation tools coupling architecture.

also simulates a set of UAVs, but it fails to model the communications channel, and it does not include a graphical interface to allow analyzing UAV mobility, meaning that all the information analysis depends on interpreting log files and third-party applications.

We can also find approaches like AETOURNOS [21], based on MATLAB, that attempt to combine, on the long term, the simulation of a custom multicopter model in real time, or even real UAVs (see Figure 1), with simulated communications using OMNeT++. Its initial development is limited to the use of TCP, and authors fail to evaluate the temporal mismatch between real-time UAV simulation and simulation-time communications between UAVs.

In this work we present ArduSim, a solution where mobility and the communications between many UAVs are simulated in real time, and protocols are directly portable to real devices, thereby skipping the problems detected in the simulators described above.

3. ArduSim Design and Implementation

ArduSim[1] was developed in Java, and it has a modular structure, that is, the graphical interface, the communication with the virtual UAVs and between them, and the usability of the simulator itself, are all implemented on independent packages. This eases the implementation of new inter-UAV communication protocols, while avoiding having to learn all the details associated to communication with virtual UAVs using the MAVLink protocol.

3.1. Simulation Architecture

To simulate a great number of UAVs simultaneously, we have used the SITL application as a basic development module. SITL contains control code resembling a real UAV, simulating its physical and flying properties with great accuracy**. A SITL instance is executed for each virtual UAV, and it runs together with its physical engine on a single process**. The main limitation of SITL is that it only simulates a single UAV, being thus inadequate to develop communication protocols between UAVs.

Figure 2 shows the proposed simulation platform, which relies on a multi-agent simulation architecture that implements a high-level control logic above SITL itself. ArduSim allows configuring UAVs and starting experiments directly from its graphical interface (*GUIControl*). In addition, it includes the simulation of packet broadcasting between UAVs (*Simulated broadcast*), and the detection of possible collisions (**UAV Collision detector**). The later has been solved using a thread that periodically checks if the simulated UAVs are close enough to assert that a collision has happened, based on the information provided by the virtual UAVs.

Each virtual UAV is composed of an agent in charge of controlling the UAV behaviour, and the different threads required for the protocol being tested. The communication between UAVs requires a minimum of two threads, one for sending data packets (*Listener*), and another one for their reception (*Talker*).

[1]https://bitbucket.org/frafabco/ardusim

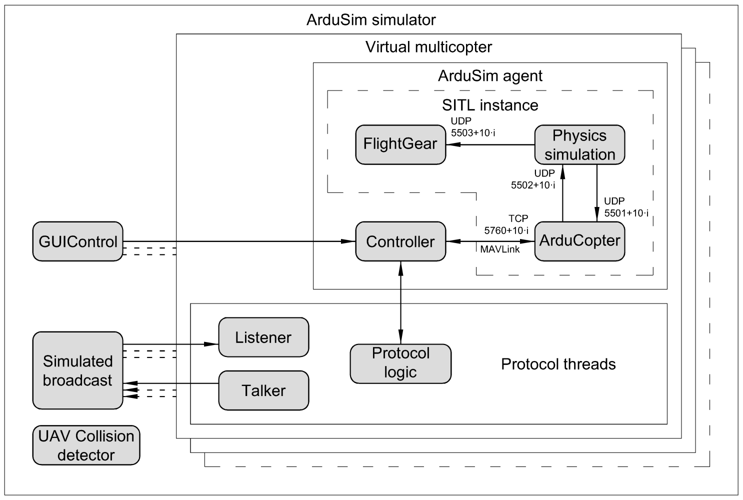


Figure 2: **ArduSim internal architecture.**

Moreover, it can have an additional thread (*Protocol logic*) to command the UAV taking into account the logic of the protocol under development, and the messages received by other surrounding UAVs.

An ArduSim agent includes a SITL instance, and a thread (*Controller*) in charge of sending commands to the multicopter, and of receiving the information that it generates. Such communications rely on the MAVLink protocol (see section 3.2). When running ArduSim on a real UAV it becomes a controller agent, as explained in detail in section 3.4.

SITL uses an 8-bit identifier for each UAV, meaning that the simulation is limited to a maximum of 256 UAVs.

3.2. Controlling multicopters

The *Controller* thread transmits control messages in the MAVLink format via TCP to a SITL instance. Simultaneously, it receives and processes the answers to the given instructions, and the information messages provided by the virtual flight controller. This information (e.g. current position, speed, etc.) can be used by the protocol being developed in order to achieve the desired functionality. In real UAVs, such communications rely on a serial port

Table 1: Control commands as shown in Figure 3.

Simple CMD commands

setParam	Modifies an UAV parameter.
getParam	Retrieves an UAV parameter value.
setMode	Switches to another UAV flight mode.
armEngines	Allows arming the engines before takeoff.
doTakeOff	Takes off until reaching a specific height (m).
setSpeed	Changes the flight speed (m/s).
setCurrentWP	Indicates the waypoint where to move to.
moveUAV	Moves the UAV to specific GPS coordinates.
clearMission	Eliminates the current mission.

Throttle on command

setThrottle	Stabilizes the UAV height before stopping it.

Send wp list command

sendWPList	Loads the specified mission on the UAV.

Get wp list command

getWPList	Retrieves details about the current mission.

connection towards the flight controller (see section 3.4).

3.2.1. ArduSim-to-UAV communications API

To simplify the implementation of UAV coordination protocols, we developed an API including the set of commands shown in table 1.

The commands are grouped into different categories according to the type of message that is being transmitted between the simulator and the virtual flight controller, as detailed in section 3.2.2.

Regarding the implemented set of commands, several clarifications are due. First, the flight mode used by the flight controller is implementation-dependant, meaning that the different flight modes used should be tested when porting any protocol to a real UAV. Second, if more than 15 seconds pass between the engine arming and the takeoff processes, the flight controller will disarm the UAV for security reasons. Third and last, to move an UAV to a specific set of coordinates, it must previously be in the *guided* flight mode, as required by the MAVLink protocol.

The set of commands defined make the communications with the UAV trans-

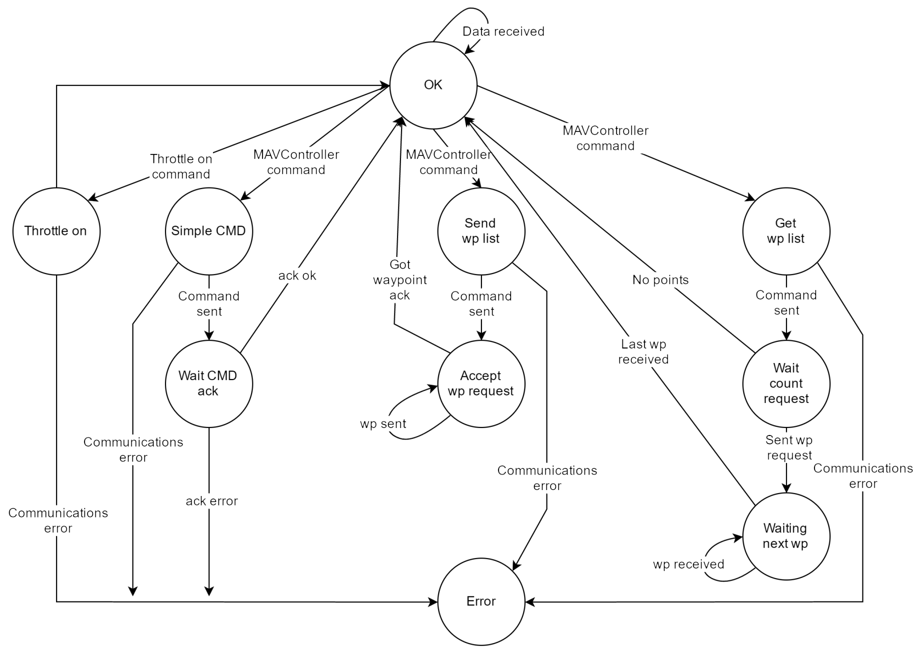


Figure 3: MAVLink communications finite state machine.

parent to the developer, and they return a boolean value to indicate whether execution was successful or not, thereby simplifying the handling of communication errors at a high level. At low level, they are in charge of complying with the communication protocols defined in the MAVLink standard.

3.2.2. MAVLink communications implementation

The finite state machine depicted in Figure 3 shows all the communications taking place between the *Controller* thread and the virtual flight controller. It is in fact a simplified version of the actual state machine, which has a total of 36 states, and that takes into account all the commands implemented.

Each time a data packet is received from the flight controller, the simulator checks its current state, and analyzes whether it should take any action. If the state is *OK*, then no command has to be executed. Otherwise, it means that a command was issued, or that some message sent from the flight controller requires a reply. In addition to the answers to the different commands, the simulator constantly receives a great amount of MAVLink messages with information about the actual situation of the UAV. Among others, it receives data regarding the position, speed, attitude, and flight mode.

Concerning the implemented functions (see table 1), there are four types of interaction between the simulator and the flight controller of a virtual UAV, as shown in Figure 3:

- *Simple CMD.* Adopted by the overwhelming majority of commands. A command is issued, and the flight controller must return an ACK. When this ACK is received, the interaction ends.

- *Throttle on.* Used to take control of the flight altitude during a flight. The interaction ends just after the command is issued, and no ACK is required. This command simulates the presence of a remote control when none is controlling the UAV, i.e., when the protocol under development does not require the intervention of a pilot. This command must be used when the UAV leaves the *auto* flight mode, as the flight controller considers that the communication with the remote control has been lost, causing the UAV to perform an emergency landing.

- *Send wp list.* Required to send a planned mission to the UAV. First we submit the total number of waypoints associated to the mission, and the controller reacts by requesting, one by one, the different waypoints; the thread will then submit them sequentially until the flight controller returns an ACK to confirm that all waypoints have been successfully received.

- *Get wp list.* Employed to recover a mission stored in the UAV. It starts by requesting the mission. The controller returns a message to indicate the number of waypoints conforming the mission. If the UAV has a mission stored, that is, if the number of waypoints is not null, the thread will request them sequentially until all are received; at that time, an ACK is sent back to the controller.

3.3. UAV-to-UAV communications

Currently, communications between real UAVs typically relies on broadcasting using UDP. Since a wireless link is created, the simulator should take into account the signal range in order to determine whether or not a packet arrives to the neighboring UAVs. This means that a realistic communications model should be adopted. Some existing simulators rely on a simple model, where a distance threshold is used to discriminate between received and discarded packets, while others rely on the Friis equation, or a theoretical model such as Nakagami or Okumura, all having significant computational costs.

ArduSim includes three different channel models depending on the desired degree of accuracy:

- *Unrestricted.* It uses an ideal medium where data packets always arrive to all possible destinations (basic model).
- *Fixed range.* Data packets arrive to another UAV only if the distance between them is lower than the defined threshold (simple model).
- *Realistic* 802.11*a with* 5*dBi antenna.* The probability that a data packet is received by another UAV depends on the distance between UAVs according to a model obtained from real experiments (realistic model).

Notice that the third model was obtained by studying the communications link properties between two real multicopters during flight (wireless-enabled UAV, model GRCQuad from Quaternium), and measuring the packet loss rate produced on a WiFi ad-hoc network link in the 5 GHz band (channel 36, 23 dBm transmission power), based on the strategy defined in [8].

Figure 4 shows the packet loss rate (y) obtained when varying the distance between UAVs (x). Beyond 1350 meters we consider that packet losses reach 100%, while for lower distances the following polynomial applies:

$$y = 5.335 \cdot 10^{-7} \cdot x^2 + 3.395 \cdot 10^{-5} \cdot x \tag{1}$$

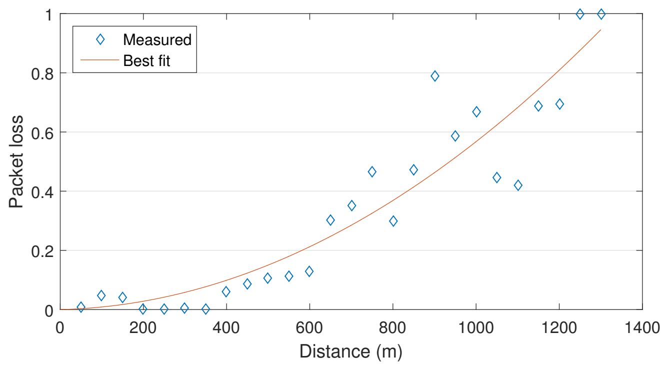


Figure 4: Packet loss vs. distance (IEEE 802.11a, 5 dBi antenna).

Overall, the simulator determines whether a data packet transmitted by an UAV is received by each of the neighboring UAVs according to the model used, and the inter-UAV distance.

IEEE 802.11-based networks rely on the CSMA/CA algorithm for medium access arbitration. Thus, to make communications more realistic in the scope of our simulator, we have implemented the carrier sensing functionality. Regarding the collision avoidance mechanism used in 802.11, it involves very short waiting times (DIFS) before transmitting a data packet, which is not possible to implement in real-time simulation without performing active waiting. This occurs because the time slice that the system grants to each thread is larger than this value, and so there are no guarantees that the packet will be transmitted after that time if a passive wait is made. On the other hand, the solution is not scalable if active waiting is performed, because each thread tries to use a CPU core completely, preventing the simulation of more than 2 or 3 simultaneous UAVs on standard PCs. Such limitations forced us to implement a mechanism to detect collisions on the wireless channel, while discarding some of the collision avoidance features of CSMA/CA. We consider that our solution offers an adequate trade-off between channel behaviour accuracy and performance, meeting real-time constraints despite CPU limitations.

The carrier sensing, the collision detection (physical level), and the reception buffers, have been simulated together by means of two functions, the first one for

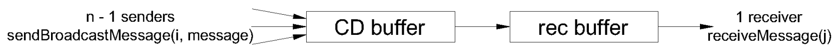


Figure 5: Simulated broadcast model.

the packet transmission, and the second one for the packet reception process; the data structures shared between both these functions act as reception buffers. This way, carrier sensing is simulated when a message is sent, while collision detection is done when it is received.

Figure 5 shows the model used to simulate packet transmission via broadcast. The reception buffers ($rec\ buffer$) are FIFO, block the thread until a data packet is received, and have a configurable size (163840 bytes by default). In addition, each reception buffer is preceded by another buffer ($CD\ buffer$) used to detect collisions on the channel.

When simulating n UAVs, each of them can simultaneously receive messages from a maximum of $n-1$ UAVs (thread $Talker$, see Figure 2, using the *sendBroadcastMessage* function), which are then inserted in the $CDbuffer$ and ordered according to the instant when transmission starts. If carrier sensing is activated, the transmission does not start until the medium is available.

When a protocol being developed requests a message (thread $Listener$, using the *receiveMessage* function), it first checks if there is any message in the reception buffer. Otherwise, collision detection is applied to the messages available in the $CD\ buffer$, eliminating those messages that have collided, and moving the rest to the reception buffer for its own use. This solution allows us to detect collisions only when there is no data in the reception buffer, and not whenever a new message is requested, thereby reducing the computational cost considerably.

The intermediate buffer, in charge of simulating the wireless medium, allows us to detect collisions, and it could grow indefinitely if the receiver does not request any message. Although this approach would be the ideal solution from the collision detection mechanism perspective, it is not viable since RAM memory is a limited resource. For this reason, the size of the $CD\ buffer$ is limited to twice the size of the reception buffer ($rec\ buffer$). It is worth mentioning that, if collision detection is not required, the intermediate buffer is deemed unnecessary. In this case, the threads insert the messages directly into the $rec\ buffer$.

In order to determine if the medium is busy (carrier sensing), and whether two received messages have collided (collision detection), it is necessary to determine the start and end times of a message transmission taking into account the transmission speed and the length of the frame. Regarding the transmission speed, the communications model uses the 5 GHz band, and the transmission is made via broadcasting, meaning that the transmission rate is 6 Mbps. The end of the transmission is determined by also taking into account the size of the frame, including the preamble, according to the specifications of the 802.11 protocol. In addition to the start and end times for message transmission, it is also necessary to store the value of two variables (isChecked, isOverlapped) for each message in order to detect collisions, as explained below.

Algorithm 1 details the message transmission process. If the communications protocol has been deployed in real UAVs, the transmission is done directly over UDP; otherwise, broadcast transmission is simulated. Once the transmission of the last message has been completed, it checks if no other UAV within range of the transmitting UAV has began a new transmission (carrier sensing). We determine whether an UAV is within the range of the transmitter (function $isInRange$) by relying on any of the communication models described at the beginning of this section, which can be selected by the user. The transmission consists of storing a copy of the message on the $CD\ buffer$ of each UAV within range. If the collision detection is not activated, the message is directly stored in the reception buffer ($rec\ buffer$). If any of the two buffers is full, the message is discarded, meaning that it is not received at that particular destination.

Every time an UAV sends a message, it stores a copy ($prevSentMessage$). The instant of completion of a transmission is saved along with the message ($prevSentMessage.end$), thus allowing to determine if the transmission has finished, and if the medium is available (carrier sensing).

Algorithm 2 details the process of receiving a message. If there are no

Algorithm 1 sendBroadcastMessage(n, message)

```
Require: $n \in [0, number\ of\ UAVs] \wedge message \neq \emptyset$
  if $is\ a\ real\ UAV$ then
    $send\ message\ through\ UDP\ broadcast$
  else
    if $n.prevSentMessage \neq \emptyset$ then
      while $n.prevSentMessage.end > now$ do
        $sleep\ 1ms$
      end while
    end if
    if $carrier\ sensing\ is\ enabled$ then
      while $\exists i \neq n \wedge i.prevSentMessage \neq \emptyset \wedge i.prevSentMessage.end > now \wedge i.isInRange$ do
        $sleep\ 1ms$
      end while
    end if
    $i = 0$
    while $i < number\ of\ UAVs$ do
      if $i \neq n \wedge i.isInRange \wedge (i.prevSentMessage = \emptyset \vee i.prevSentMessage.end < now)$ then
        if $collision\ detection\ is\ enabled$ then
          if $\neg i.virtualQueue.isFull$ then
            $add\ message\ to\ i.virtualQueue$
          end if
        else
          if $\neg i.queue.isFull$ then
            $add\ message\ to\ i.queue$
          end if
        end if
      end if
      $i{+}{+}$
    end while
    $n.prevSentMessage \leftarrow message$
  end if
```

messages in the reception buffer, algorithm 3 is executed to discard the messages that have collided, moving to the *rec buffer* the remaining messages.

Algorithm 2 receiveMessage(n)

```
Require: $n \in [0, number\ of\ UAVs]$
Ensure: $message \neq \emptyset$
  if $is\ a\ real\ UAV$ then
    $message \leftarrow receive\ through\ UDP$
  else
    if $collision\ detection\ is\ enabled$ then
      while $message = \emptyset$ do
        if $n.queue.isEmpty$ then
          if $n.virtualQueue.isEmpty$ then
            $sleep\ 1ms$
          else
            $process\ algorithm\ 3$
          end if
        else
          $message \leftarrow n.queue.poll()$
          while $m.end > now$ do
            $sleep\ 1ms$
          end while
        end if
      end while
    else
      while $n.queue.isEmpty$ do
        $sleep\ 1ms$
      end while
      $message \leftarrow n.queue.poll$
    end if
  end if
  return message
```

If the protocol has been deployed in real UAVs, message reception is done via UDP; otherwise, the transmission medium is simulated.

If collision detection is not enabled, it waits until there is some message available in the reception buffer whose transmission has been completed, and it is delivered. Otherwise, if there are no messages available, the buffer that simulates the medium is analyzed. If this buffer does not contain messages either, it is necessary to wait for a message to arrive; otherwise, if it contains some message(s), the collision detection algorithm is executed.

Algorithm 3 packetCollisionDetection(n)

```
Require: $n \in [0, number\ of\ UAVs]$
  $iterator \leftarrow n.virtualQueue$
  $previous \leftarrow iterator.next$
  $previous.isChecked \leftarrow true$
  while $previous.end > now$ do
    $sleep$ $1ms$
  end while
  if $previous.end > endMax$ then
    $endMax \leftarrow previous.end$
  end if
  while $iterator.hasNext$ do
    $following \leftarrow iterator.next$
    $following.isChecked \leftarrow true$
    if $following.start < endMax$ then
      $previous.isOverlapped$
      $following.isOverlapped$
    end if
    if $following.end > now$ then
      $following.isChecked \leftarrow false$
      if $following.isOverlapped$ then
        $previous.isChecked \leftarrow false$
      end if
      $break$
    else
      if $following.end > endMax$ then
        $endMax \leftarrow following.end$
      end if
      $previous \leftarrow following$
    end if
  end while
  $iterator \leftarrow n.virtualQueue$
  while $iterator.hasNext$ do
    $message \leftarrow iterator.next$
    if $\neg message.isChecked$ then
      $break$
    else
      $iterator.remove\ message$
      if $\neg message.isOverlapped \wedge \neg n.queue.isFull$ then
        $add\ message\ to\ n.queue$
      end if
    end if
  end while
```

The collision detection process has a computational cost $\mathcal{O}(2n)$ on the number of received messages, $\mathcal{O}(2n^2)$ on the number of UAVs, and it consists of two steps. In the first one, the messages, already sorted according to the transmission start time ($start$), are marked as analyzed ($isChecked$), and among them, those that have collided with other messages are also identified ($isOverlapped$). A second step is used to eliminate overlapping messages, and to transfer the remaining marked ones to the reception buffer, discarding the message in case this buffer is already full.

The first step of the analysis process is stopped when the last message is analyzed, or when a message is found whose transmission has not yet been completed. The second run stops after the last message, or when an unchecked message is found ($isChecked = false$), that is, a message not found during the first run. This solution takes into account the possibility of inserting a message among existing ones that have already been analyzed during the first run, since insertions are made concurrently. If the last message analyzed has collided with the second-last one, both are preserved for the next analysis to account for those cases when a message that collides with one of them arrives later on.

3.4. Protocol Deployment on Real UAVs

The ArduSim simulator has been designed to facilitate the deployment of the protocols implemented in real UAVs. The application was developed using Java, and it communicates with virtual UAVs via TCP, simulating the communication among UAVs through buffers that are shared by different threads (Figure 2). However, when the application is executed in a real UAV (Figure 6), the graphical interface is not shown, the communication with the virtual UAV is replaced by a serial port connection, and the wireless communication between UAVs relies on the broadcasting of UDP datagrams. All the simulation-dependent software elements are disabled merely by changing an execution parameter, which makes the deployment of a newly developed protocol somewhat trivial.

To be able to deploy new protocols, it becomes necessary to port the Java application to a Linux or Windows®-based device, with Java 7 pre-installed, along

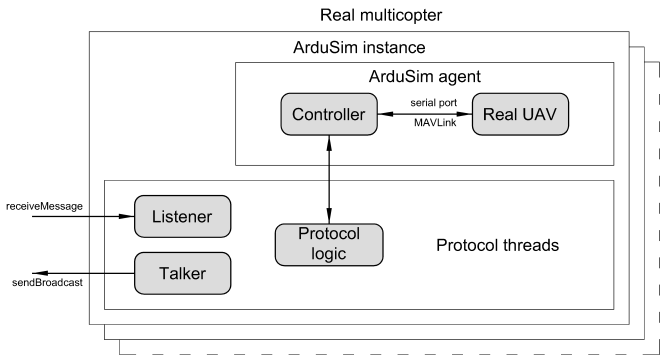


Figure 6: ArduSim architecture on real UAVs.

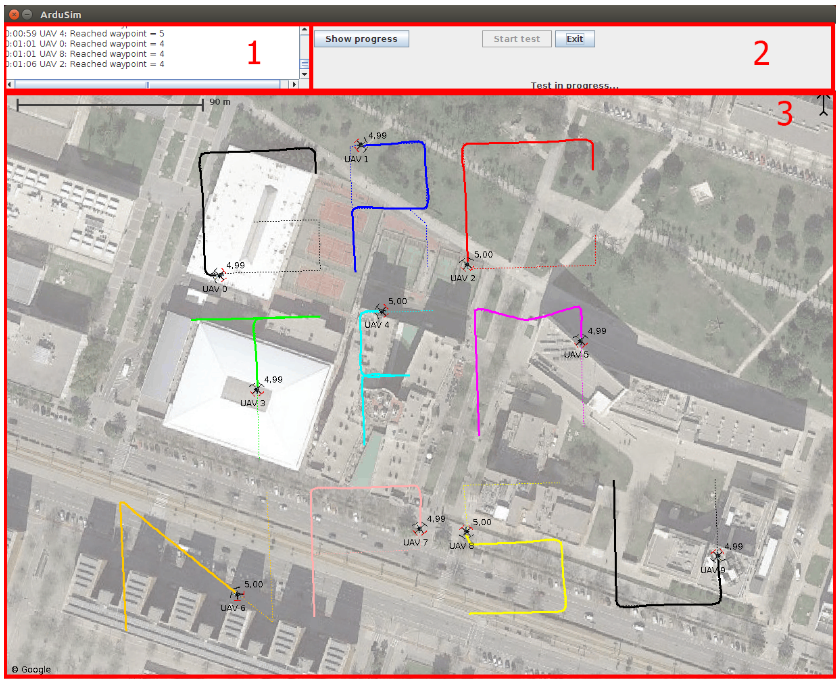
with the RXTX native library [22] for accessing the serial port, and a physical serial port connection with one of the telemetry ports of the flight controller, in addition to following the instructions provided with ArduSim. Preliminary tests have been performed with a Raspberry Pi 3 having its $ttyAMA0$ serial port connected to the $Telem2$ telemetry port of the Pixhawk controller embedded in our GRCQuad multicopter, allowing us to check the proper deployment of a test flight coordination protocol.

3.5. ArduSim Graphical User Interface

ArduSim is oriented to the development of protocols applicable to UAVs performing planned missions, or conforming an UAV swarm. As an example, Figure 7 shows ten UAVs performing a mission, represented as letters 'GRCTFM-NPSU'.

On the upper left corner of the window (1) we can find the application log. It details the functioning of the simulator, as well as the results of the commands sent to the UAVs.

On the right (2) we have the controls that allow the user to manage the application, to start the test, or to close ArduSim. While running a swarm experiment, it also shows an additional button to perform the setup step. In



```
0:00:59 UAV 4: Reached waypoint = 5
0:01:01 UAV 0: Reached waypoint = 4
0:01:01 UAV 8: Reached waypoint = 4
0:01:06 UAV 2: Reached waypoint = 4
```

Test in progress...

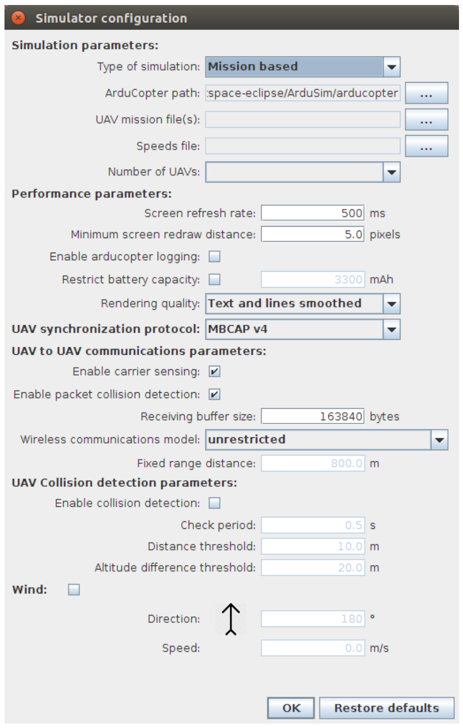
Figure 7: ArduSim main window: experiment in progress.

addition, it provides general information about the simulator itself.

Most of the window space (3) is used to visualize UAV flights during tests. On the upper right corner we show the wind direction (if defined for the test). The discontinuous lines represent the mission assigned to each UAV. On each UAV, we indicate its identifier and its altitude. Before starting, each UAV loads the mission to be completed, and simulated wind is also applied. A thick stroke represents the real path followed by each UAV. If the UAVs collision detection feature is enabled, a red circle centered on each UAV is drawn. When an UAV invades that circle, we consider that a collision between UAVs has occurred.

In addition to the main window, an additional dialog window is also opened to show the position, the speed, and the flight mode according to the MAVLink protocol, as well as state information relative to the collision-avoidance protocol during the experiment, if needed.

The dialog box of Figure 8 is shown when ArduSim is started. It allows


Simulator configuration

Simulation parameters:

Type of simulation:	Mission based	
ArduCopter path:	space-eclipse/ArduSim/arducopter	...
UAV mission file(s):		...
Speeds file:		...
Number of UAVs:		

Performance parameters:

Screen refresh rate:	500	ms
Minimum screen redraw distance:	5.0	pixels
Enable arducopter logging:	☐	
Restrict battery capacity:	☐ 3300	mAh
Rendering quality:	Text and lines smoothed	

UAV synchronization protocol: MBCAP v4

UAV to UAV communications parameters:

Enable carrier sensing:	✓	
Enable packet collision detection:	✓	
Receiving buffer size:	163840	bytes
Wireless communications model:	unrestricted	
Fixed range distance:	800.0	m

UAV Collision detection parameters:

Enable collision detection:	☐	
Check period:	0.5	s
Distance threshold:	10.0	m
Altitude difference threshold:	20.0	m

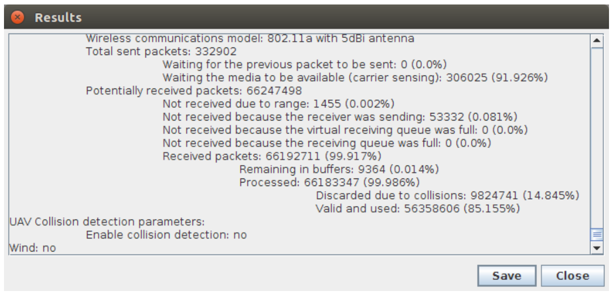
Wind: ☐

Direction:	180	°
Speed:	0.0	m/s

Figure 8: Initial configuration dialog.

the user to specify several simulation parameters, including the mission to be followed by the UAV, the flight speed, some performance parameters, the synchronization protocol to be tested, the wireless model to be used and some of its properties, whether or not to detect when collisions happen, in addition to simulated wind speed and orientation.

When an experiment ends, the user decides whether to save the results obtained or not. A dialog is shown (see Figure 9) with the configuration and general results of the experiment, which includes detailed statistics of the communications among the virtual UAVs, such as the total number of data packets sent, how many had to wait for the media to become available (*carrier sensing*), or were discarded due to collisions, among others. Several independent files per UAV are also saved with additional information, such as the actual path followed by each UAV during the experiment.



```
                Wireless communications model: 802.11a with 5dBi antenna
                Total sent packets: 332902
                                Waiting for the previous packet to be sent: 0 (0.0%)
                                Waiting the media to be available (carrier sensing): 306025 (91.926%)
                Potentially received packets: 66247498
                                Not received due to range: 1455 (0.002%)
                                Not received because the receiver was sending: 53332 (0.081%)
                                Not received because the virtual receiving queue was full: 0 (0.0%)
                                Not received because the receiving queue was full: 0 (0.0%)
                                Received packets: 66192711 (99.917%)
                                                Remaining in buffers: 9364 (0.014%)
                                                Processed: 66183347 (99.986%)
                                                                Discarded due to collisions: 9824741 (14.845%)
                                                                Valid and used: 56358606 (85.155%)
UAV Collision detection parameters:
                Enable collision detection: no
Wind: no
```

Figure 9: Results dialog.

Table 2: Hardware used for experiments.

	i7PC	*i5PC*
Processor	Intel Core i7-7700	Intel Core i5-2500K
Speed (GHz)	3.6 (max 4.2)	3.3 (max 3.7)
Cores	4	4
Hyper-Threading	yes	no
Cache L1-2-3	4x64KB-256KB-8MB	4x64KB-256KB-6MB
RAM	32GB DDR4 2133MHz	8GB DDR3 1333 MHz
HHDD	480GB SSD	2TB 7200 rpm
GPU	NVIDIA GeForce 8400	HD Intel 3000
Monitor	1920x1080 & 1280x1024	1280x1024
OS	Ubuntu 16.10	Ubuntu 16.10
Java RE	SE 8	SE 8

4. ArduSim Validation

Once the ArduSim platform has been introduced, we now proceed to validate its correctness and scalability. To this end, we performed a wide set of experiments by having a variable number of UAVs (i.e., from 1 to 256 UAVs) following a straight path from origin to destination during 5 minutes, being that all UAVs are overlapped, and collision detection is disabled. The flight altitude was set to 5 meters, the speed was of 10 m/s, and the default values of the simulator parameters were used. Experiments were made on two different computers (see table 2) to evaluate the influence of the hardware used on the ArduSim performance.

The target metrics were RAM, hard disk, and CPU usage, as well as the time lag between the UAVs of each experiment with respect to a reference UAV in a single-UAV experiment. This last measurement allows us to evaluate how an increase in the resource consumption levels affects the real-time performance of our tests, and therefore the scalability level supported. Notice that, when resource consumption is very high, execution is delayed with respect to the situation when there are sufficient resources, and threads do not have to wait for the scheduler to let them access CPU resources.

Regarding functionality, the application has shown to be fully stable after 1500 executions. The only problems detected occurred when more than 150 simultaneous UAVs where tested in the *i*5*PC*, which is an issue related to the excessive resource usage, as detailed below.

Hard disk I/O operations slightly affect CPU usage when simulating a high number of UAVs (more than 100); this is due to log maintenance tasks performed by the simulation environment managed by SITL. ArduSim has been designed with this issue in mind, and thus provides two non-exclusive options. First, you can deactivate the SITL log, so disk usage loses relevance, or it can be kept active while running the simulator in root/administrator mode. In the latter case, I/O operations are performed on a virtual disk instead, which is faster than in the *i*7*PC*'s SSD hard drive, and certainly much faster than in the *i*5*PC*'s mechanical hard drive. In order to compare the results obtained with both PCs, all experiments were performed with the log turned off, and in root mode.

The SITL executable requires a very small amount of RAM, since it is a compilation of a controller firmware, and so it is designed to be executed with very few resources. For this reason, the RAM usage of the simulator is very small, even when simulating 256 UAVs simultaneously. The only circumstance where the simulator consumes a significant amount of RAM is when it is run as root, and a RAM drive is used to store the SITL logs. In this case, a maximum of 50 MiB per UAV is used in the RAM drive, which represents a global memory usage of 12800 MiB, and a minimum amount of recommended system memory of 16 GiB to simulate up to 256 UAVs simultaneously. However, with the SITL log disabled, and even when using the RAM drive, its size is reduced to 1280

MiB, and so the recommended amount of memory to run ArduSim is reduced to only 6 GiB. If the execution is not performed as root, and therefore a RAM drive is not used, the memory consumption is further reduced to 1400 MiB both for the SITL instances and the simulator itself, and so a minimum of 4 GiB of RAM suffices.

4.1. CPU utilization

The CPU load is a critical factor affecting the simulator's scalability since the execution of each virtual UAV will be slowed down if the CPU cannot manage, in real time, all the necessary calculations. We have carried out experiments for different numbers of UAVs while varying the CPU usage associated to the graphical environment and the computer used, and introducing as a synthetic load the transmission of coordination information between the UAVs. Each experiment, lasting 5 minutes, was repeated three times measuring CPU usage once a second, and we then took the average value.

4.1.1. Rendering quality overhead

Figure 10 shows the CPU usage with different numbers of UAVs, and when using four different rendering qualities in the *i7PC* used for testing:

- *RQ* 1. Lowest level with maximum performance.
- *RQ* 2. Fonts smoothed. Font antialiasing enabled.
- *RQ* 3. Lines smoothed. Font antialiasing enabled, and lines with sub-pixel accuracy rendering.
- *RQ* 4. Same as *RQ*3, and alpha blending optimized for quality.

We can see that only the use of line rendering with sub-pixel accuracy (i.e., RQ3 and RQ4) is significant in terms of performance. In addition, in the *RQ*1 and *RQ*2 levels, we find that the processor's energy saving features apparently increase the CPU usage when the load is low, that is, with less than 200 UAVs, moving away from the theoretical line that would connect CPU usage with a

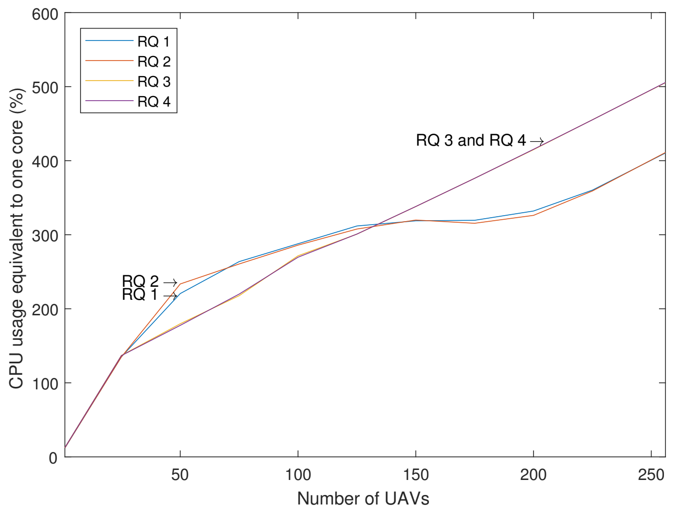
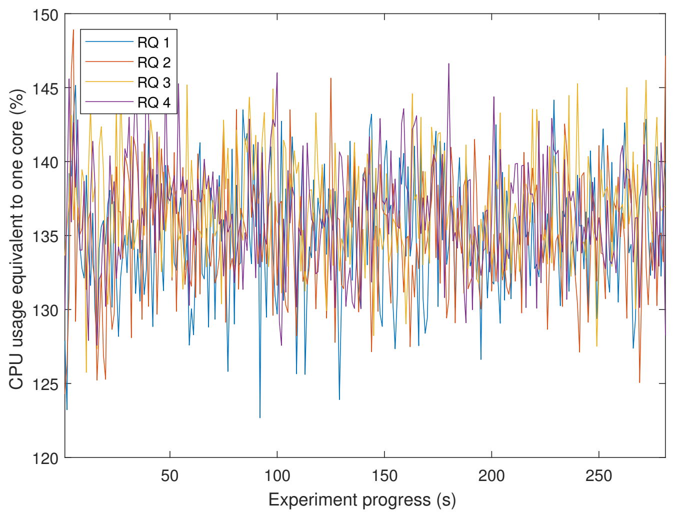


Figure 10: Rendering quality overhead ($i7PC$).

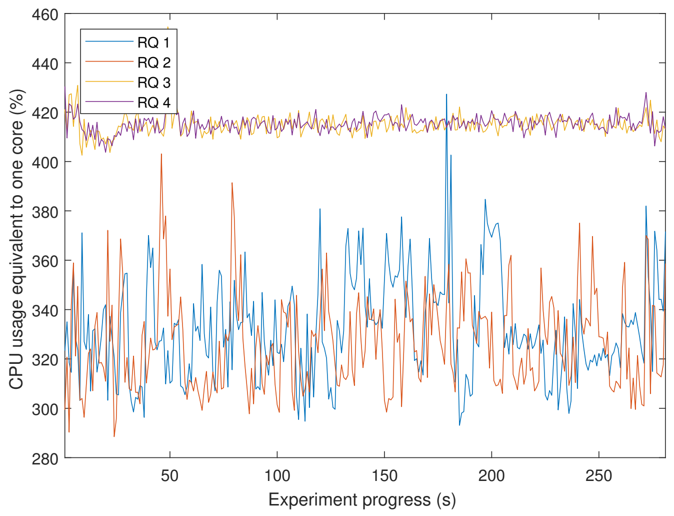
single UAV and with 256 UAVs. This effect takes place since the processor goes into inactivity for short periods of time, and the execution of different threads overlaps in time. This effect is also observed in Figure 13, when additional CPU load is added by enabling inter-UAV communications. The maximum load with 256 UAVs is approximately 500%, when the maximum possible value is 800% (4 cores with Hyper-Threading can run 8 threads simultaneously).

Figure 11 shows the evolution of CPU usage during the experiment with 25 and 200 UAVs. In Figure 11a, the values oscillate significantly and randomly, since the processor is far from saturated, and there are even time intervals during which the CPU is inactive. However, in Figure 11b, it is observed that, when line drawing with sub-pixel accuracy is activated, the CPU load increases and causes threads to start having to wait for execution, a situation that causes CPU usage to become more uniform.

Figure 12 shows the CPU usage with the 4 rendering quality levels, in this case for the $i5PC$. It confirms that only two sets of rendering quality combinations are significant, differentiated by the use of line drawing with subpixel accuracy (RQ1/RQ2 and RQ3/RQ4), although the difference between both levels is very small. In this case, the system is not capable of simulating more



(a) Results running 25 UAVs.



(b) Results running 200 UAVs.

Figure 11: CPU utilization when varying the rendering quality overhead ($i7PC$).

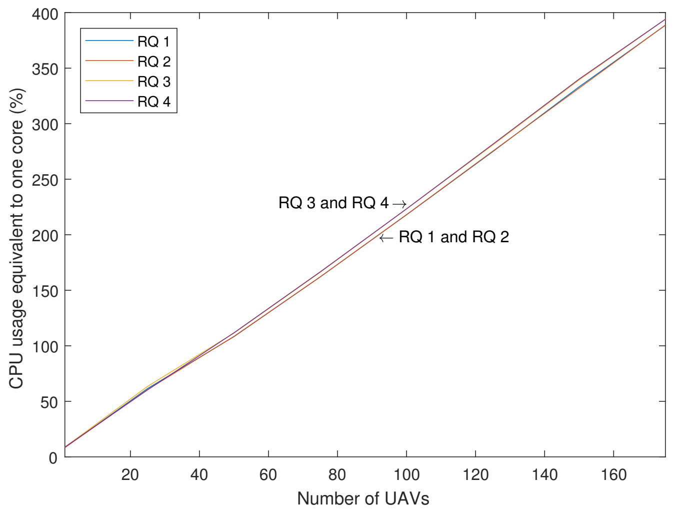


Figure 12: Rendering quality overhead ($i5PC$).

than 175 UAVs without destabilizing, since the CPU usage reaches 400 %, the maximum possible one supported (4 cores can run 4 threads simultaneously). In addition, the time lag introduced when simulating 175 UAVs is excessive, which is why later on, in Figure 17, only its magnitude is analyzed with up to 150 UAVs. Notice that, since the CPU is less powerful, it shows a linear resource consumption increase with the number of UAVs, and it does not show the same effects detected in the i7 platform, associated to energy saving mechanisms.

4.1.2. Communications overhead

ArduSim has been designed to develop and validate communication protocols between UAVs. Thus, the CPU usage analysis must also take into account the load that communications between UAVs introduces. For this purpose, two experiments have been designed on the $i7PC$ by varying the network load. In both cases, all the simulated UAVs transmit data packets at a constant rate, and follow an overlapping trajectory, meaning that nearly all the packets reach all the UAVs, a situation that we consider the most unfavorable for our analysis, since it supposes a higher CPU load. The first experiment was done with a sending rate of 5 packets per second, while the second one was done with a rate of 10 packets

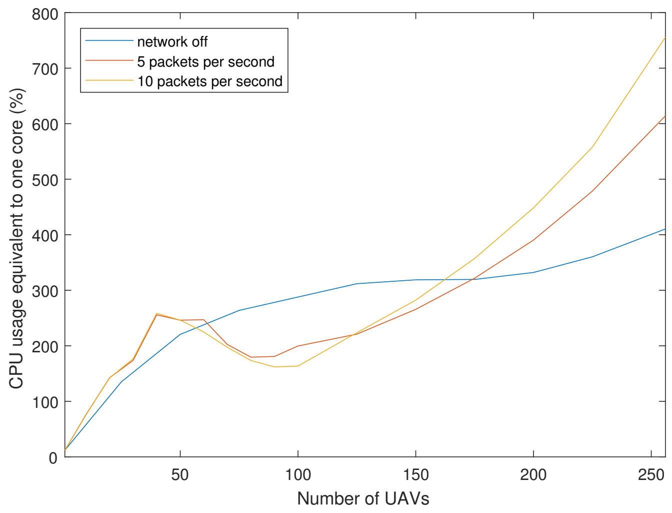


Figure 13: UAV-to-UAV communications overhead ($i7PC$).

per second. In both cases, the transmitted packet size is 705 bytes. Figure 13 shows that CPU usage grows faster with up to 40 UAVs. With more UAVs, the load causes the energy saving mechanisms of the i7 PC to lose significance, and the curves adjust better to the theoretical line connecting the results for 1 and 256 UAVs. However, the load introduced by the communications prevents the graph from being straight, which is consistent with the computational cost of sending and receiving data packets for the synthetic load used ($\mathcal{O}(2n^2)$, being n the number of UAVs).

4.2. Real-time constraints evaluation

Merely checking that the simulator is stable, and that the processor does not become saturated, is not enough to state that ArduSim correctly emulates the UAV flight in real time. In fact, a very high or irregular CPU usage could introduce a global delay in the execution of the emulated UAVs, or even a differential delay between them, thereby affecting the scalability of the simulator. Therefore, the maximum number of UAVs that can run simultaneously on a given computer will depend on these factors.

To analyze the scalability of ArduSim, experiments were carried out on both computers, $i7PC$ and $i5PC$, with the rendering quality set to $RQ1$, with a different number of UAVs, and measuring the time lag for each of the simulated UAVs regarding the simulation of a single UAV used as reference. In other words, considering the real-time performance of a single-UAV simulation, we analyze if the UAVs suffer any kind of lag among them and in regard to that single-UAV simulation. Each experiment was repeated 7 times, measuring the lag error every 5 meters throughout the followed path, and we assessed the individual results of each repetition, or the overall set of results as a single group, depending on the analysis carried out. All the experiments were performed with the UAVs following a straight path trajectory from origin to destination for 5 minutes. All trajectories are overlapped, and UAV collision detection is deactivated. The flight altitude was set to 5 meters, and their speed was of 10 m/s; the default values of the simulator were used for the remaining parameters.

Regarding the single-UAV experiment used as reference, it was selected among the 7 repetitions as the one located nearest to the median value.

4.2.1. Scalability analysis under the $i7PC$

Figure 14 shows a box-whisker plot with the lag time obtained when varying the number of UAVs. In addition to the median lag and the distribution of the lag values for all the simulated UAVs obtained along 7 experiments, it includes the mean lag value. Simulating 100 or less UAVs, the measurements shown are really close to the mean value; however, with a higher number of UAVs, data is more scattered, and the time lag increases. Similarly to previous results (see Figure 10), the processor's energy-saving mechanisms introduce a significant time lag for a number of UAVs between 150 and 225. The worst case is detected with 200 UAVs, with an average lag value of 0.45 seconds, and a maximum lag of 1.4 seconds. On the other hand, the dispersion of values with up to 100 UAVs is significantly lower than the one obtained with a higher number of UAVs.

Figure 15 shows the evolution of the mean value for the time lag corresponding to the 7 experiments performed with each number of UAVs. The time lag remains significantly constant along time for up to 100 UAVs (a mere increase

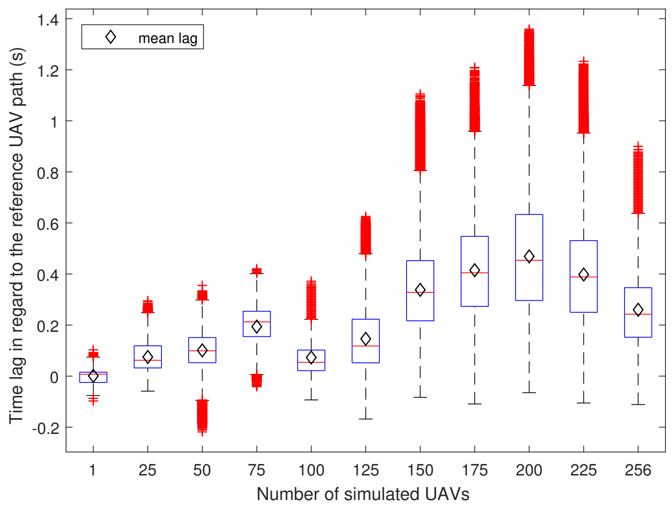


Figure 14: Time lag values of all the experiments ($i7PC$).

of 0.5 seconds per simulated hour), and it increases linearly with more UAVs. As shown in Figure 14, the processor's energy-saving mechanisms introduce a significant time lag for a number of UAVs between 150 and 225, which makes the slope for 200 UAVs to be actually higher than the slope for 256 UAVs. We can conclude that the computer $i7PC$ allows us to simulate up to 100 UAVs while meeting **soft** real-time constraints.

Now, a detailed analysis of the experiment that produces the bigger time lag in regard to the reference UAV (i.e., with 200 UAVs) is presented. Figure 16a shows the set of time lag values obtained in the 7 experiments performed. From that figure we find that the worst case is test number 3, with a maximum time lag of 1.4 seconds, and a mean lag value of 0.62. Second, Figure 16b shows the evolution of the time lag of each UAV in that test. We can see that, after an initial warm-up period (first 700 meters), the time lag between UAVs stabilizes. In addition, this lag is always greater than zero and it increases throughout time, evidencing that UAVs suffer a delay with respect to the reference UAV. Figure 16c shows the average, minimum, and maximum lag for each UAV in that same test. Notice that UAV number 103 is the one with the highest time lag. In addition, the average lag for each UAV varies in a very small range of 0.6

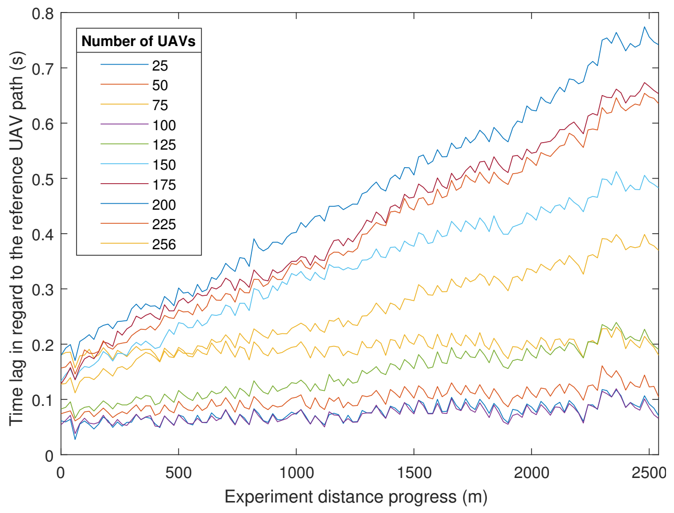


Figure 15: Time lag over experiment progress ($i7PC$).

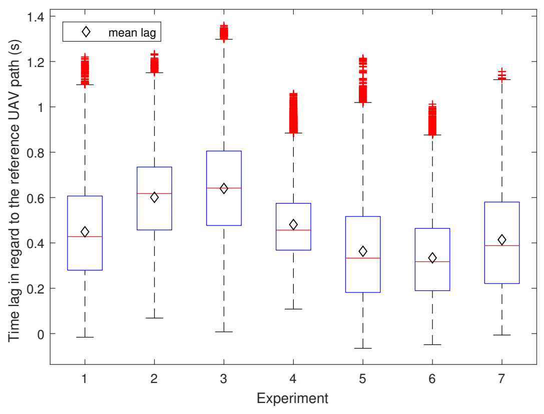
seconds. Thus, we can state that, although the simulation does not meet strict real-time constraints, it can be considered to be correct as long as the absolute simulation time is not relevant for the protocol under development since the simulation delay offset associated to the different UAVs remains similar.

4.2.2. Scalability analysis under the $i5PC$

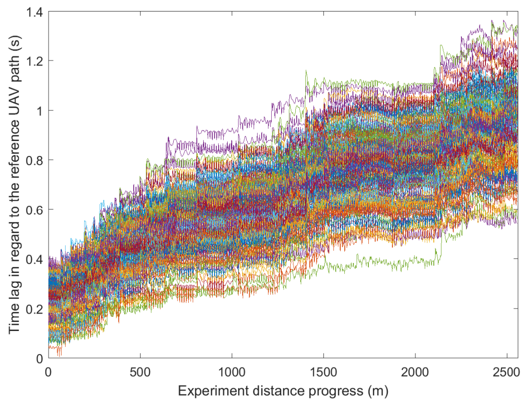
The previous tests have been performed on a high-end desktop computer with very high performance. Thus, we consider adequate to complement our analysis by also checking the time lag associated to PCs with a lower performance. In particular, this section details the results obtained with the $i5PC$ used for testing.

The results in Figure 17 are limited to 150 UAVs, since with more than 175 UAVs the simulation becomes unstable, and with 175 the time lag (14 seconds) is too high, something consistent with the results in Figure 12, where the CPU became saturated with 175 UAVs.

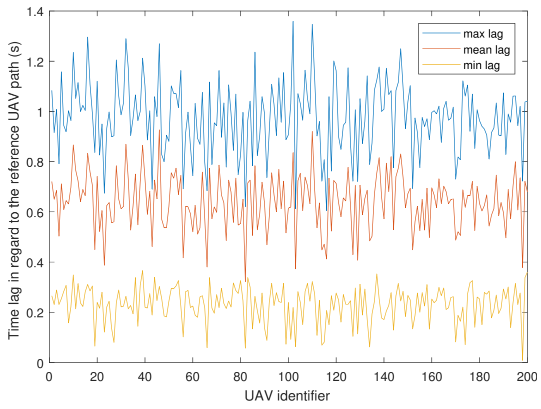
In the $i5PC$ computer the time lag remains stable over time for up to 100 UAVs (see Figure 18). There is also a temporary reduction in CPU usage towards the end of the experiment, when the UAVs approach the last waypoint



(a) Time lag values of each experiment.



(b) Experiment 3 (worst case). Time lag of all the UAVs.



(c) Experiment 3 (worst case). Minimum, mean, and maximum time lag for each UAV.

Figure 16: Worst case analysis with 200 UAVs ($i7PC$).

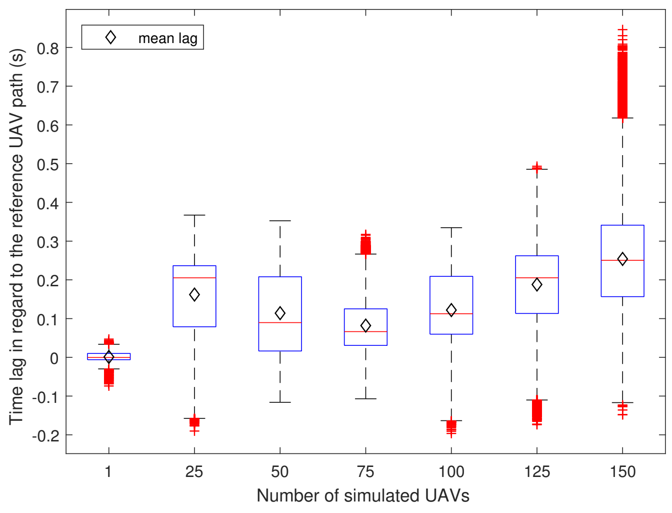


Figure 17: Time lag values of all the experiments ($i5PC$).

of the mission, just before landing. The initial lag has a bias deviation between 0 and 0.2 seconds because the UAVs start their flight at different instants on each experiment. Notice that this issue does not affect the real-time execution, and is due to control loops included in the implementation.

Similarly to the experiments made with the $i7PC$, we include a study of the maximum temporal lag achieved (see Figure 19). First, in Figure 17, we can observe that the maximum time lag is of 0.85 seconds, and it corresponds to experiments with 150 UAVs. More specifically, it corresponds to the maximum lag of the second experiment (see Figure 19a). Figure 19b shows that the time lag between UAVs remains uniform throughout the test, similarly to Figure 16b. Finally, in this case, the maximum time lag detected is associated to UAV 36 in the experiment (see Figure 19c). There is greater variability in the average lag of each UAV due to a greater dispersion in the time lag of each UAV. This occurs because the processor is closer to saturation compared to the situation where the $i7PC$ is used.

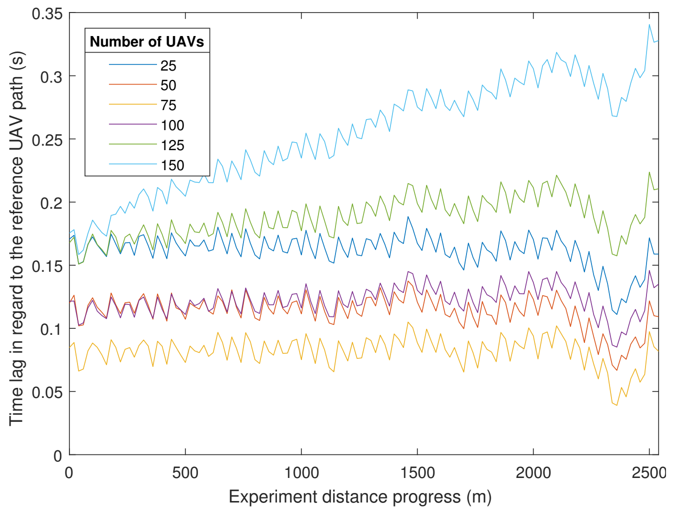


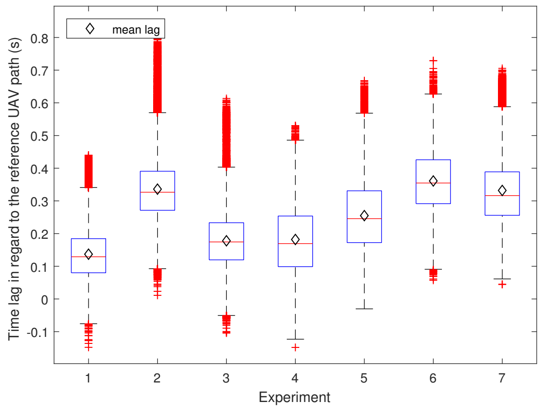
Figure 18: Time lag over experiment progress ($i5PC$).

4.2.3. Communications overhead analysis

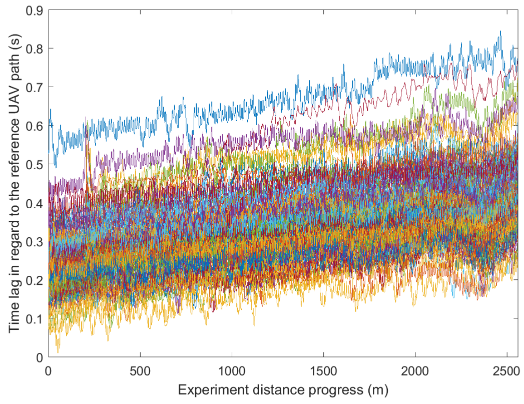
This set of experiments was carried out with a load of 5 packets per second per UAV. This means that, with 200 UAVs, 1000 packets per second are sent, meaning that potentially 199,000 messages reception events per second can take place when broadcasting these packets. In such case, the associated CPU usage becomes high, as shown in Figure 13, which can negatively affect the scalability of the simulator, as the temporal offset of each UAV with respect to the reference may increase.

Figure 20 shows that the measured time lag with a number of UAVs between 150 and 225 is lower than the one depicted in Figure 14. This occurs because the processor is working at full capacity, without activating the energy saving features referred to earlier. However, with 256 UAVs, the lag is greater; in this case CPUs operate close to their saturation point (see Figure 13).

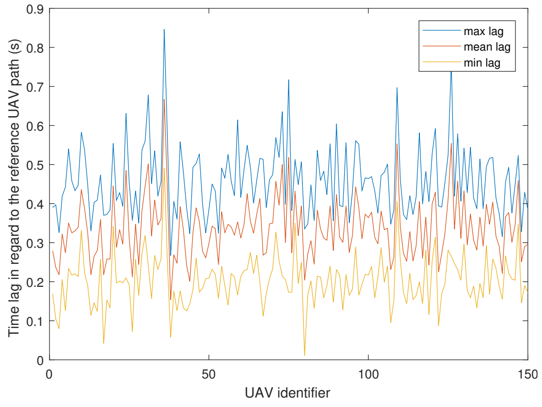
Similarly to previous cases, the evolution of the time lag throughout the experiments has been studied. Figure 21 shows a peak of CPU usage towards the end of the test, when the UAVs reach the last waypoint and reduce their speed. It is also observed that the time lag remains stable over time with up to 175 UAVs, increasing linearly for higher number of UAVs. These results are



(a) Time lag values of each experiment.



(b) Experiment 2 (worst case). Time lag of all the UAVs.



(c) Experiment 2 (worst case). Minimum, mean, and maximum time lag for each UAV.

Figure 19: Worst case analysis with 150 UAVs ($i5PC$).

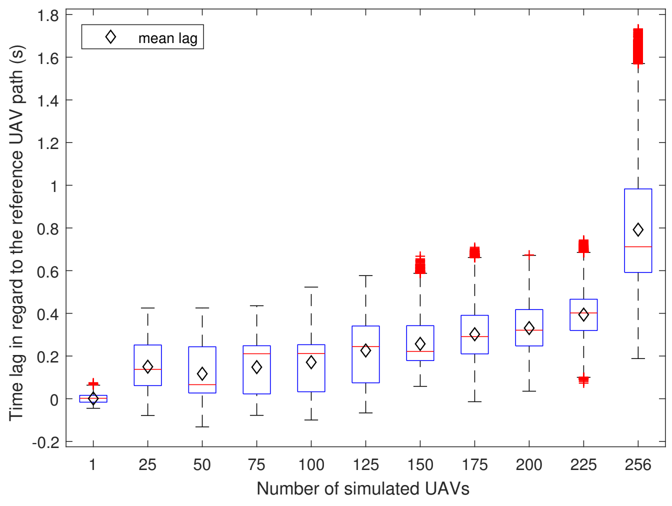


Figure 20: Time lag values of all the experiments at a sending ratio of 5 pps (*i7PC*).

better than those obtained before activating communications between UAVs, and occur because the CPUs are already working at their full capacity when reaching 90 UAVs (see Figure 13).

Another issue to be discussed is the accuracy with which the simulation of communications is able to detect the usage of the wireless medium (carrier sense), as well as the collision between data packets. Table 3 shows the results obtained when varying the number of UAVs while setting the transmission load to the value defined above (5 packets per second per UAV). It is observed that the results are somehow optimistic since, with 200 UAVs, 1000 packets sized 705 bytes are being transmitted at a speed of 6 Mbps (transmission time of $\approx$ 1 ms), which should saturate the medium. Although nearly all packets have to wait for the medium to become available, the collision rate remains lower than expected. To explain this phenomenon, we must take into account that the CPU of the *i7PC* can only run 8 threads at a time, while with 200 UAVs there are 800 threads / processes (one SITL process and three threads, *Listener*, *Talker*, and *Controller*, per UAV) that are competing for CPU time, which implies that there are 100 threads/processes that compete for being executed on a same core. On the other hand, the time slices provided to each thread by the Linux/Ubuntu

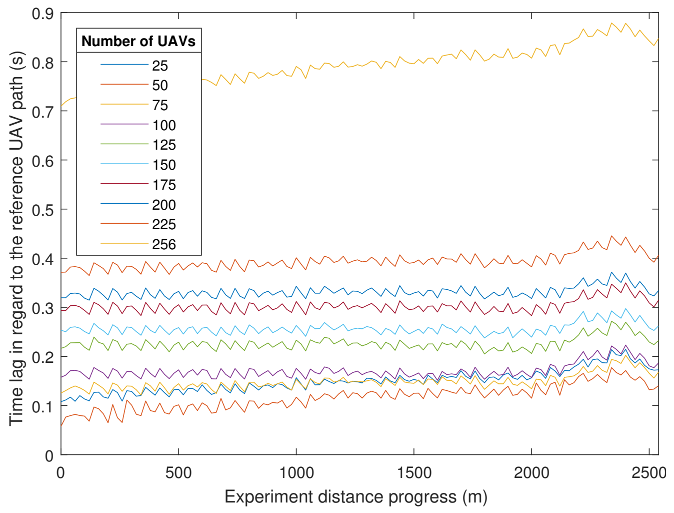


Figure 21: Time lag over experiment progress ($i7PC$, 5 pps).

Table 3: Percentage of packets that waited (carrier sense) and collided (collision detection).

	25	50	75	100	125	150	175	200	225	256
CS	0.84	3.40	4.02	4.25	21.30	60.01	79.29	92.07	94.92	94.82
CD	0.08	0.06	0.10	0.13	0.56	3.20	8.69	15.09	17.52	20.29

operating system varies between 0.75 ms ($sysctl\_sched\_min\_granularity$) and 6 ms ($sysctl\_sched\_latency$), meaning that each thread can run about 13.3 times per second during 0.75 ms, or, what is the same, each $Talker$ thread can send a packet once every 75 ms. Considering that, in the experiment performed, the transmission time of each packet is approximately 1 ms, we find that, in a worst-case scenario, it can only collide with packets sent by other $Talker$ threads that are running when these have just left the processor, or will run in the next round. This behavior avoids that data packets collide with each other as often as expected, producing a packet collision rate lower than what would actually occur. Thus, this problem is inherent to the system itself, and can only be improved by using dedicated servers with a very high number of cores.

5. Conclusions and Future Work

The widespread adoption of UAVs is making developers face novel challenges, among which the communication between UAVs emerges as a striking requirement when attempting to avoid collisions, or when flying UAV swarms, among others.

In this paper we introduced ArduSim, a realistic simulator that allows operating with multiple UAVs simultaneously, when performing planned missions or when flying as a swarm. To date, no similar solution has been developed that offers similar characteristics, including the possibility to model inter-UAV communications using different channel models, as well as the way UAVs use the exchanged information to interact between them, paving the way for introducing a wide range of novel protocols. Through simulation, it becomes possible to analyze packet dissemination in Delay-Tolerant Networks (DTN), to develop new routing algorithms, or to propose new flight coordination mechanisms. Among the many benefits, our simulator allows validating the proposed solution beforehand, while porting that solution to real devices becomes straightforward, as the set of commands used is the same.

In addition to the simulator itself, we also modeled the WiFi communications link between UAVs based on real experiments performed in the 5 GHz frequency band. In particular, we focused on the relationship between packet losses and distance when broadcasting data. The model derived was then integrated into the simulator as one of the wireless channel models available. To further improve the degree of realism of our experiments, we also modelled the wireless channel occupancy through a carrier sensing mechanism, and included the possibility of detecting collisions of data packets.

The stability of ArduSim has been correctly validated, with 1500 different successful executions, having the maximum allowed number of UAVs (256) in the $i7PC$, and up to 175 UAVs in the $i5PC$. We found that any mid-range or high-end computer is capable of simultaneously simulating a high number of UAVs (approximately 100) in **near real-time**, even when considering the overload introduced by the communications between UAVs.

Regarding scalability, we have verified that the simulation **can be performed with up to 100 UAVs while meeting soft real-time constraints**, and that the delay offset between them is uniform. In addition, ArduSim is able to run at least 150 UAVs when hard real-time is not required in a computer similar to the $i5PC$, or even 225 with a high-end desktop computer ($i7PC$). We have also analyzed the influence of the rendering quality on the system load. We found that only the tracing of lines with sub-pixel quality has a significant effect on performance. Communications have a quadratic computational cost, so they also affect the system performance significantly. When configuring each UAV to transmit at a rate of 5 packets per second, the load affects real-time performance when having more than 225 UAVs in the $i7PC$. Also notice that, depending on the complexity of the UAV coordination protocol being developed, its impact on performance can also become non-negligible.

As future work we plan to extend ArduSim by integrating new communication models based on different types of antennas and wireless technologies. We will also enable flight traces obtained with ArduSim to be exported to OMNeT++ to analyze in more detail the performance of wireless communications in these environments, and thus develop a more realistic communications model. Finally, we will perform tests on real UAVs, comparing the results with the one obtained through simulation to assess in greater depth the degree of accuracy achieved by ArduSim.

Acknowledgment

This work was supported by the Ministerio de Economía y Competitividad for the "Programa Estatal de Investigación, Desarrollo e Innovación Orientada a Retos de la Sociedad, Proyecto I+D+I SMART@CARPHONE: Integración del smartphone y el vehículo para conectar conductores, sensores y entorno a través de una arquitectura de servicios funcionales" [grant number TEC2014-52690-R], and the Universitat Politècnica de València (UPV) under program

"Contratos Pre-doctorales para la Formación de Personal Investigador (FPI)" [grant number 0060100000].

References

[1] C. Giannini, A. A. Shaaban, C. Buratti, R. Verdone, Delay Tolerant Networking for smart city through drones, in: Proceedings of the International Symposium on Wireless Communication Systems, Vol. 2016-Octob, Poznan, Poland, 2016, pp. 603–607. doi:10.1109/ISWCS.2016.7600975.

[2] L. Ghouti, Mobility prediction in mobile ad hoc networks using neural learning machines, Simulation Modelling Practice and Theory 66 (2016) 104 – 121. doi:https://doi.org/10.1016/j.simpat.2016.03.001. URL http://www.sciencedirect.com/science/article/pii/S1569190X1600040X

[3] A. Ray, D. De, An energy efficient sensor movement approach using multi-parameter reverse glowworm swarm optimization algorithm in mobile wireless sensor network, Simulation Modelling Practice and Theory 62 (2016) 117 – 136. doi:https://doi.org/10.1016/j.simpat.2016.01.007. URL http://www.sciencedirect.com/science/article/pii/S1569190X16000149

[4] S. Park, H. Kim, K. Kim, H. Kim, Drone Formation Algorithm on 3D Space for a Drone-based Network Infrastructure, in: IEEE 27th Annual International Symposium on Personal, Indoor, and Mobile Radio Communications (PIMRC), Valencia, Spain, 2016, pp. 1–6.

[5] J. Lee, K. Kim, S. Yoo, A. Y. Chung, J. Y. Lee, S. J. Park, H. Kim, Constructing a reliable and fast recoverable network for drones, in: 2016 IEEE International Conference on Communications, ICC 2016, Kuala Lumpur, Malaysia, 2016, pp. 0–5. doi:10.1109/ICC.2016.7511317.

[6] Y. Chai, K. c. Cao, Distributed UAV formation control with two-hop relay protocol, in: 2017 36th Chinese Control Conference (CCC), 2017, pp. 8707–8712. doi:10.23919/ChiCC.2017.8028739.

[7] L. Meier, QGroundControl, MAVLink Micro Air Vehicle Communication Protocol, http://qgroundcontrol.org/mavlink/start, accessed 15/06/2017.

[8] F. Fabra, C. T. Calafate, J. C. Cano, P. Manzoni, A methodology for measuring UAV-to-UAV communications performance, in: 2017 14th IEEE Annual Consumer Communications Networking Conference (CCNC), 2017, pp. 280–286. doi:10.1109/CCNC.2017.7983120.

[9] F. Fabra, C. T. Calafate, J. C. Cano, P. Manzoni, On the impact of inter-UAV communications interference in the 2.4 GHz band, in: 2017 13th International Wireless Communications and Mobile Computing Conference (IWCMC), 2017, pp. 945–950. doi:10.1109/IWCMC.2017.7986413.

[10] C. Chen, Q. Zhu, C. Wang, Rejection Methods for Nakagami-m Fading Simulation, in: International Conference on Internet Technology and Applications- iTAP'11, Wuhan, China, 2011, pp. 1–4.

[11] A. D. Team, SITL Simulator (Software in the Loop), http://ardupilot.org/dev/docs/sitl-simulator-software-in-the-loop.html, accessed 15/06/2017 (2016).

[12] Dronethusiast, Drone Flight Simulator – Analysis & Comparison, http://www.dronethusiast.com/drone-flight-simulator/, accessed 15/06/2017.

[13] M. A. Khan, H. Hasbullah, B. Nazir, Recent open source wireless sensor network supporting simulators: A performance comparison, in: 2014 International Conference on Computer, Communications, and Control Technology (I4CT), 2014, pp. 324–328. doi:10.1109/I4CT.2014.6914198.

[14] S. Kang, M. Aldwairi, K.-I. Kim, A survey on network simulators in three-dimensional wireless ad hoc and sensor networks, International Journal of Distributed Sensor Networks 12 (9) (2016) 1550147716664740. arXiv:https://doi.org/10.1177/1550147716664740, doi:10.1177/1550147716664740.
URL https://doi.org/10.1177/1550147716664740

[15] Y. Ben-Asher, M. Feldman, S. Feldman, P. Gurfil, IFAS: Interactive flexible ad hoc simulator, Simulation Modelling Practice and Theory 15 (7) (2007) 817 – 830. doi:https://doi.org/10.1016/j.simpat.2007.04.004.
URL http://www.sciencedirect.com/science/article/pii/S1569190X07000524

[16] R. Garcia, L. Barnes, Multi-UAV Simulator Utilizing X-Plane, Journal of Intelligent and Robotic Systems 57 (1) (2009) 393. doi:10.1007/s10846-009-9372-4.
URL https://doi.org/10.1007/s10846-009-9372-4

[17] J. Holt, S. Biaz, L. Yilmaz, C. A. Aji, A symbiotic simulation architecture for evaluating UAVs collision avoidance techniques, Journal of Simulation 8 (1) (2014) 64–75. doi:10.1057/jos.2013.5.
URL https://doi.org/10.1057/jos.2013.5

[18] A. Y. Javaid, W. Sun, M. Alam, UAVSim: A simulation testbed for unmanned aerial vehicle network cyber security analysis, in: 2013 IEEE Globecom Workshops (GC Wkshps), 2013, pp. 1432–1436. doi:10.1109/GLOCOMW.2013.6825196.

[19] B. Kate, J. Waterman, K. Dantu, M. Welsh, Simbeeotic: A simulator and testbed for micro-aerial vehicle swarm experiments, in: Proceedings of the 11th international conference on Information Processing in Sensor Networks - IPSN'12, Beijing, China, 2012, pp. 49–60. doi:10.1145/2185677.2185685.
URL http://dl.acm.org/citation.cfm?id=2185685

[20] J. Modares, N. Mastronarde, K. Dantu, UB-ANC Emulator: An Emulation Framework for Multi-Agent Drone Networks, in: IEEE International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR), San Francisco, USA, 2016, pp. 252–258.

[21] L. Ciarletta, A. Guenard, Y. Presse, V. Galtier, Y. Q. Song, J. C. Ponsart, S. Aberkane, D. Theilliol, Simulation and platform tools to develop safe flock of UAVs: a CPS application-driven research, in: 2014 International Conference on Unmanned Aircraft Systems (ICUAS), 2014, pp. 95–102. doi:10.1109/ICUAS.2014.6842244.

[22] T. Jarvi, RXTX. Serial port library for Java SDK, http://rxtx.qbang.org/wiki/index.php/Main_Page, accessed 26/02/2018.