Drone Air-Show Controller System with LoRa

Alperen Kayhan

ID: 2385532
Department of Computer Engineering
Middle East Technical University
Ankara, Türkiye
alperen.kayhan@metu.edu.tr

Semir Emre Gedikli

ID: 2526333
Department of Computer Engineering
Middle East Technical University
Ankara, Türkiye
emre.gedikli@metu.edu.tr

***Abstract*—This report documents our discrete-event simulation of a *Drone Air-Show Controller System* that employs long-range (LoRa) radios for ground-to-UAV coordination and uses a Poisson process to inject random failures and orientation changes. The study evaluates network reliability (packet-delivery ratio, latency) and formation robustness while exercising triangle, rectangle and circle manoeuvres with ten drones in OMNeT++. We detail the model, code architecture, milestone timeline, configuration files and preliminary metrics obtained, providing a complete reproducible package for follow-up experimentation.**

***Index Terms*—LoRa, Drone Swarm, Discrete-Event Simulation, OMNeT++, Poisson Process, Turtle Mobility**

I. Project

A. Project Explanation

This project focuses on simulating the Drone Air Show Controller System with LoRa communication and Poisson distribution for random event modeling. A central controller controls the drones that performs synchronized aerial maneuvers during a display such as shape formation (triangles, circles). The devices will use long-range LoRa technology to wirelessly communicate with the controller, which makes it suitable for outdoor use.

The Poisson distribution is utilized to imitate random events, such as any shifts in a drone's behavior during the air show which includes a possible breakdown or a chance in orientation. This will enable the system to simulate realistic scenarios where drones fail or reorient themselves to emulate real world performance of a drone. The analysis plans to evaluate the operational efficiency of the network in terms of the packet delivery ratio, and time delay for different scenarios.

System components include:

- **Central Controller**: A controller responsible for sending commands and receiving telemetry data from the drones.
- **Drones**: Equipped with LoRa modules for communication and sensors to track movement and orientation.
- **LoRa Stack**: Long-range wireless protocol to manage communication between drones and the controller.
- **Poisson Distribution**:• A statistical model used to simulate random events like drone failures and reorientations.

B. Diagrams & Visuals

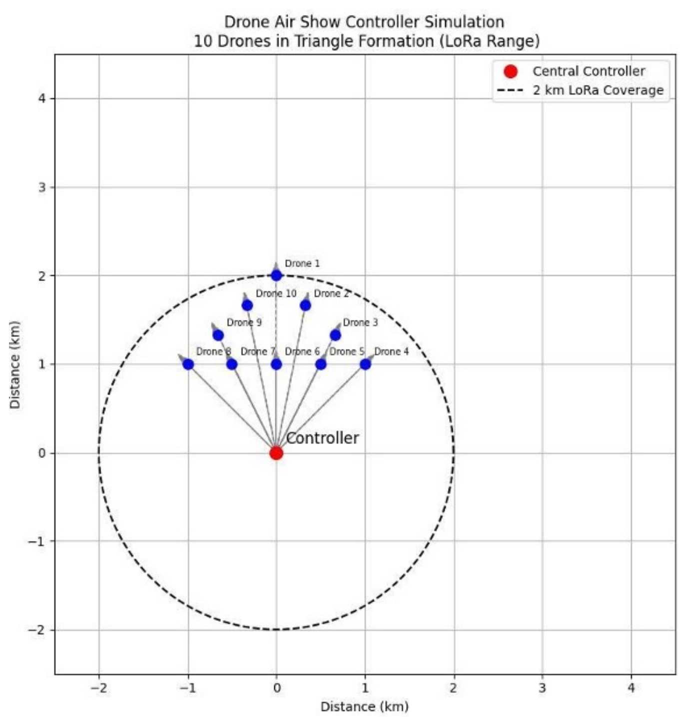


Fig. 1: Drone Air-Show Controller System

II. Milestones Achieved

A. Timeline and Tasks Completed

TABLE I: Timeline and tasks completed (verbatim from PDF Table 2.1).

Weeks	Tasks Accomplished	Member
Mar 23–29	-Finalized project proposal	SEG, AK
	-Researched LoRa communication and Poisson distribution	AK
Mar 30–Apr 5	-Set up basic OMNeT++ simulation environment OMNeT++ skeleton	SEG, AK
Apr 6–12	-Defined drone movement logic (basic 2D movement)	AK
Apr 13–19	-Integrated Poisson distribution for orientation changes	SEG
Apr 20–26	- Finalized the drone formation logic (triangle, circle, rectangle)	AK
Apr 27–May 3	- Created GitHub repository	SEG , AK
	– Initial testing of communication reliability and basic drone movement	AK
	- Added random failure and orientation change logic based on Poisson distribution	SEG , AK

B. Milestones with Explanations

1) March 23 – March 29
   - **Finalized project proposal**: The project scope and objectives were finalized, focusing on using LoRA communication and Poisson distribution for simulating drone behaviors and events.
   - **Researched LoRA communication and Poisson distribution**: Alperen Kayhan conducted research on LoRA communication and Poisson distribution, essential for modeling the drone communication and random events in the simulation.
2) March 30 – April 5
   - **Set up basic OMNeT++ simulation environment**: The initial steps for setting up OMNeT++ simulation was completed, ensuring the environment is ready for simulating drone movements and communication.
3) April 6 – April 12
   - **Defined drone movement logic (basic 2D movement)**: The basic 2D movement logic for drones was planned, focusing on how the drones will move in the simulation.
4) April 13 – April 19
   - **Integrated Poisson distribution for orientation changes**: The concept of integrating Poisson distribution to model random events like drone orientation changes was completed.
5) April 20 – April 26
   - **Finalized the drone formation logic (triangle)**: The logic for arranging drones in a triangle formation was finalized, ensuring synchronized drone behavior for the air show.
6) April 27 – May 3
   - **Created GitHub repository**: The GitHub repository was created to store project-related files and documentation.
   - **Initial testing of communication reliability and basic drone movement**: Basic tests were conducted to assess communication reliability and the drone's ability to move according to the defined logic.
   - **Added random failure and orientation change logic based on Poisson distribution**: Poisson distribution was incorporated to simulate random failures and orientation changes in the drones, adding variability to their behavior.

C. Preliminary Results

- **LoRA Communication**: Preliminary tests show that drones can communicate within the 2 km range with minimal packet loss under ideal conditions.
- **Poisson Distribution Integration**: Random event logic is being integrated to simulate drone failures or orientation changes, based on the Poisson distribution. The logic simulates periodic events (e.g., reorientation) during the air show.

III. Discrete-Event Simulation Model

A. Simulation Objectives

In Our model, We aim to answer two questions: Network Reliability under Stress and Formation Robustness. To test network reliability under stress, we firstly check command protocol performs as the failure rate of drones; as well as, Metrics; packet delivery ratio (PDR) and average command-to-telemetry latency. Moreover, to check Formation Robustness, we check the drones randomly "break down" mid-show and how quickly can the remaining fleet maintain a coherent pattern, via the metrics, time to re-stabilize formation and number of retries per maneuver.

B. Conceptual Model

We follow the standard DES modelling steps:

1) **Entities**:
   - **Controller**: issues periodic "CMD" messages and collects "ACK" telemetry.
   - **Drone**: executes manoeuvres on receiving a CMD, then replies.
2) **State Variables**:
   - $n_{\text{oper}}(t)$: how many drones are still active (no failure event).
   - $channel_{\text{busy}}(t)$: Boolean flag indicating if the LoRa channel is in use.
3) **Attributes (per drone)**:
   - Position (x, y) and orientation θ, assigned at initialisation for the chosen formation.
   - Failure flag `failed`: once set, the drone ignores further CMDs.
4) **Future Event List (FEL)**:

- `CommandDispatch`: every Δt_{cmd} seconds, enqueue a broadcast event.
- `TelemetryArrival`: scheduled as a delayed reply after each CMD.
- `DroneFailure`: random breakdowns generated via a Poisson process with rate λ.

C. Event Definitions & Scheduling

Our model has three events that drive the simulation:

- **CommandDispatch**: Triggered by a timer in the Controller module every Δt_{cmd} seconds, sending a `CMD` message to all drones.
- **TelemetryArrival**: Upon receiving a `CMD`, each operational drone schedules an "ACK" reply after a processing delay drawn uniformly from $U[0, 0.1\text{ s}]$. That reply is logged for performance measurement.
- **DroneFailure**: A Poisson(λ) timer in the Controller randomly selects one of the ten drones at each firing, sends it a "FAIL" command, and places it in a non-operative state (ignoring all future `CMD`s) until reset.

All event handlers can enqueue follow-up events: `CommandDispatch` and `DroneFailure` re-schedule themselves for the next interval, while drones only schedule `TelemetryArrival` upon receipt of a `CMD`.

D. Time-Advance Mechanism

We employ the conventional event-scheduling method: the simulation maintains a Future Event List (FEL) sorted by timestamp. At each step, the simulator "removes" the earliest event from the FEL and advances the global clock to the scheduled time of the removed event. Then, it invokes the event handler for the event—forwarding a CMD, forwarding an ACK, or marking a drone as having failed—toggling module state variables and potentially adding new events to its event queue (for example, the next CommandDispatch or DroneFailure). Time isn't lost simulating standbys: the clock leaped directly from point to point of meaningful action, and command, acknowledgement, and failure dialogue takes place within its correct sequential pattern in time.

E. Event-Logic Diagram

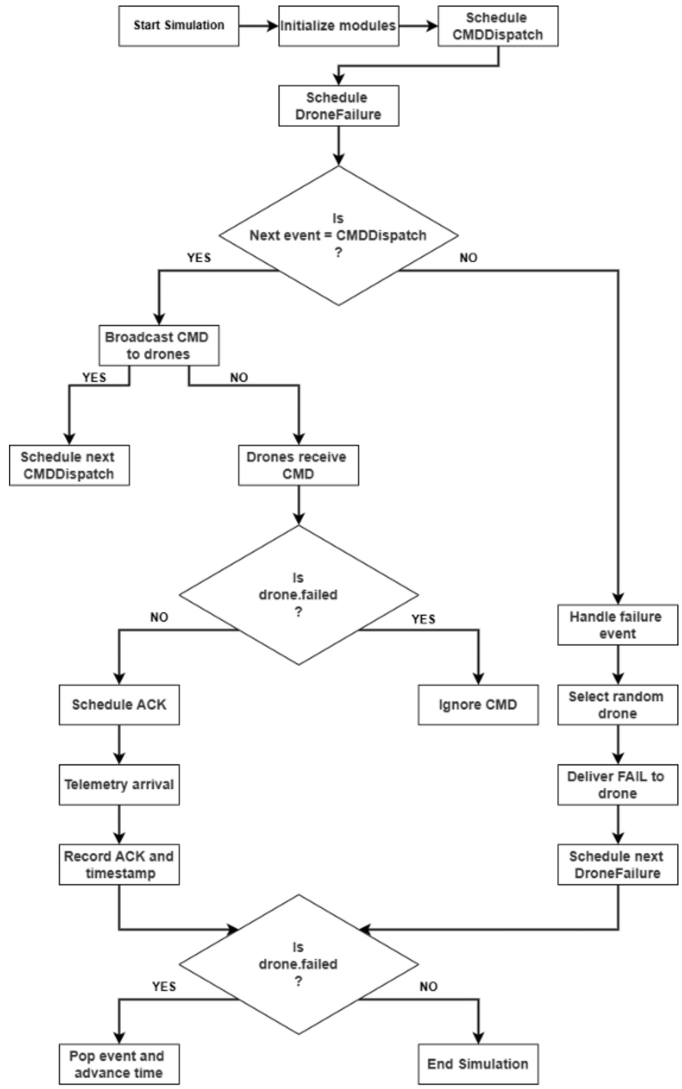


Fig. 2: Event-logic diagram

F. Example Simulation Trace Table

Event #	Time(s)	Event Type	Module	# Operational Drones	Notes
1	1.000	CMD Dispatch	Controller	10	CMD sent to all drones
2	1.027	Telemetry Arrival	Controller ← Drone 4	10	ACK from Drone 4 at t=1.027
3	1.041	Telemetry Arrival	Controller ← Drone 9	10	ACK from Drone 9 at t=1.041
...	...	...	...	...	...
11	2.000	Drone Failure	Controller ← Drone 7	9	Drone 7 goes offline
12	2.000	CMD Dispatch	Controller	9	Next CMD (only 9 drones ack)
13	2.038	Telemetry Arrival	Controller ← Drone 2	9	ACKs continue from 9 drones

Fig. 3: Simulation trace table

G. Sample Metric Calculations

With these assumptions, our metric calculations follow directly from the lecture notes (Khan, 2025).

Assuming $\Delta t_{\text{cmd}} = 1\,\text{s}$, $t_{\text{tx}} = 0.05\,\text{s}$, $n = 10$ drones, and failure rate $\lambda = 0.1\,\text{s}^{-1}$:

1. **Expected ACKs per cycle:**

$$E[\text{ACKs}] = n \times (1 - \lambda\,\Delta t_{\text{cmd}}) = 10 \times (1 - 0.1 \times 1) = 9.0$$

2. **Packet Delivery Ratio (PDR):**

$$\text{PDR} = \frac{E[\text{ACKs}]}{n} = \frac{9.0}{10} = 0.9\ (90\%)$$

3. **Average Command-to-Telemetry Latency** (For a uniform reply delay in $[0, 0.1]$)

$$E[\text{Latency}] = \frac{0 + 0.1}{2} = 0.05\ \text{s}$$

4. **Channel Utilization:**

$$T_{\text{busy-per-cycle}} = t_{\text{tx}} + E[\text{ACKs}] \times t_{\text{tx}} = 0.05 + 9.0 \times 0.05 = 0.50\ \text{s}$$

$$\text{Utilization} = \frac{T_{\text{busy-per-cycle}}}{\Delta t_{\text{cmd}}} = \frac{0.50}{1.0} = 0.5\ (50\%)$$

H. Code Snippets with Explanatory Comments

Listing 1: DroneLoRaNet.ned

```
network DroneLoRaNet {
  parameters: @display("bgb=600,400");
  submodules {
    controller: Controller { @display("p=100,200");
        }
    drone[10]: Drone { @display("p=300,50+30*i"); }
  }
  connections allowunconnected {
    controller.out++ --> { @labels("LoRaCmd"); } -->
         drone.in++;
    drone.out++      --> { @labels("LoRaAck"); } -->
        controller.in++;
  }
}
```

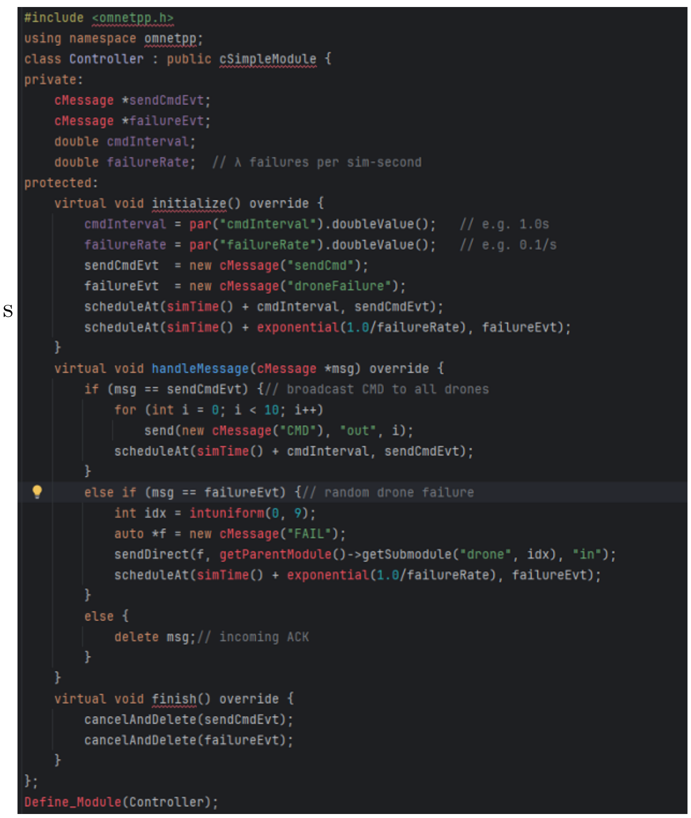
```cpp
#include <omnetpp.h>
using namespace omnetpp;

class Drone : public cSimpleModule {
private:
    bool failed = false;

protected:
    virtual void handleMessage(cMessage *msg) override {
        const char *name = msg->getName();
        if (strcmp(name,"CMD")==0 && !failed) {
            //uniform delay ([0,0.1]s) repeat
            sendDelayed(new cMessage("ACK"), uniform(0,0.1), "out");
        }
        else if (strcmp(name,"FAIL")==0) {
            failed = true;//go ofline
        }
        delete msg;
    }
};

Define_Module(Drone);
```

Fig. 4: Drone.cpp excerpt



```cpp
#include <omnetpp.h>
using namespace omnetpp;
class Controller : public cSimpleModule {
private:
    cMessage *sendCmdEvt;
    cMessage *failureEvt;
    double cmdInterval;
    double failureRate;  // λ failures per sim-second
protected:
    virtual void initialize() override {
        cmdInterval = par("cmdInterval").doubleValue();   // e.g. 1.0s
        failureRate = par("failureRate").doubleValue();   // e.g. 0.1/s
        sendCmdEvt  = new cMessage("sendCmd");
        failureEvt  = new cMessage("droneFailure");
        scheduleAt(simTime() + cmdInterval, sendCmdEvt);
        scheduleAt(simTime() + exponential(1.0/failureRate), failureEvt);
    }
    virtual void handleMessage(cMessage *msg) override {
        if (msg == sendCmdEvt) {// broadcast CMD to all drones
            for (int i = 0; i < 10; i++)
                send(new cMessage("CMD"), "out", i);
            scheduleAt(simTime() + cmdInterval, sendCmdEvt);
        }
        else if (msg == failureEvt) {// random drone failure
            int idx = intuniform(0, 9);
            auto *f = new cMessage("FAIL");
            sendDirect(f, getParentModule()->getSubmodule("drone", idx), "in");
            scheduleAt(simTime() + exponential(1.0/failureRate), failureEvt);
        }
        else {
            delete msg;// incoming ACK
        }
    }
    virtual void finish() override {
        cancelAndDelete(sendCmdEvt);
        cancelAndDelete(failureEvt);
    }
};
Define_Module(Controller);
```

Fig. 5: Controller.cpp excerpt

Listing 2: LoraDroneNetwork.ned and omnetpp.ini excerpts with Drone1 configuration

```
/* LoraDroneNetwork.ned excerpt */
network LoraDroneNetwork {
  submodules:
    // renamed from loRaNodes     Drone
    Drone1: LoRaNode {
      parameters:
        radio.typename       = "LoraRadio";
        mac.typename         = "LoraMac";
        mobility.typename    = "TurtleMobility";
        mobility.turtleScript = xmldoc("turtle1.xml
            ","movements//movement[@id='1']");
        radio.mediumModule   = "^.LoRaMedium";  //
            shared LoRa medium
        @display("p=140,300;i=device/drone");
    }
  /     other submodules & connections
}

/* omnetpp.ini excerpt */
[General]
network = LoraDroneNetwork
sim-time-limit = 1d
repeat = 30
rng-class = "cMersenneTwister"

# Drone1 sends data to LoRaGW
**.Drone[*].numApps                      = 1
**.Drone[*].app[0].typename             = "
    SimpleLoRaApp"
**.Drone[*].app[0].destAddresses         = "loRaGW
    [0]"
**.Drone[*].app[0].destPort              = 4000
**.Drone[*].app[0].messageLength         = 100B
**.Drone[*].app[0].timeToFirstPacket     = uniform(0
    s,1s)
**.Drone[*].app[0].timeToNextPacket      =
    exponential(1s)

**.Drone[*].LoRaNic.radio.typename       = "
    LoraRadio"
**.Drone[*].mac.typename                 = "LoraMac"
**.Drone[*].mobility.typename            = "
    TurtleMobility"
**.Drone[*].mobility.turtleScript        = xmldoc("
    turtle1.xml","movements//movement[@id='1']")

# Drone1 initial parameters
**.Drone1.initialX                       = 200m
**.Drone1.initialY                       = 200m
**.Drone1.initialLoRaSF                  = 12
**.Drone1.initialLoRaTP                  = 14dBm
**.Drone1.initialLoRaBW                  = 125kHz
**.Drone1.initialLoRaCR                  = 4
**.Drone1.initFromDisplayString          = false
**.Drone1.evaluateADRinNode              = true
```

I. Omnet++ Simulation Snapshots (Full-Size)

These below images will show our design simulation and, it will show our drone movements.

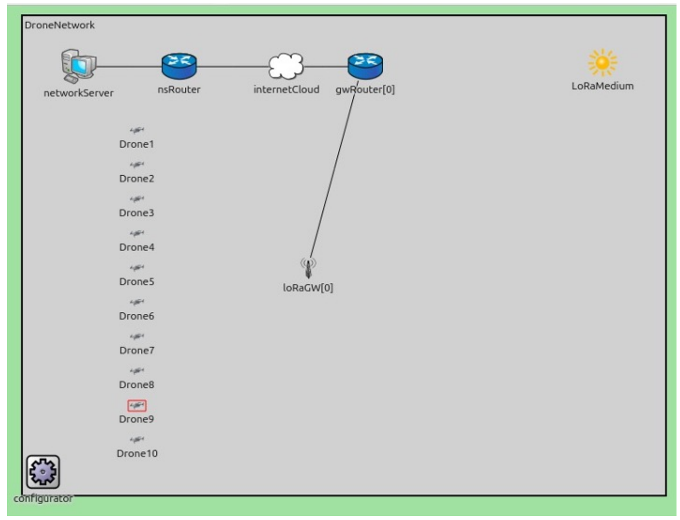


Fig. 6: Drone initial formation

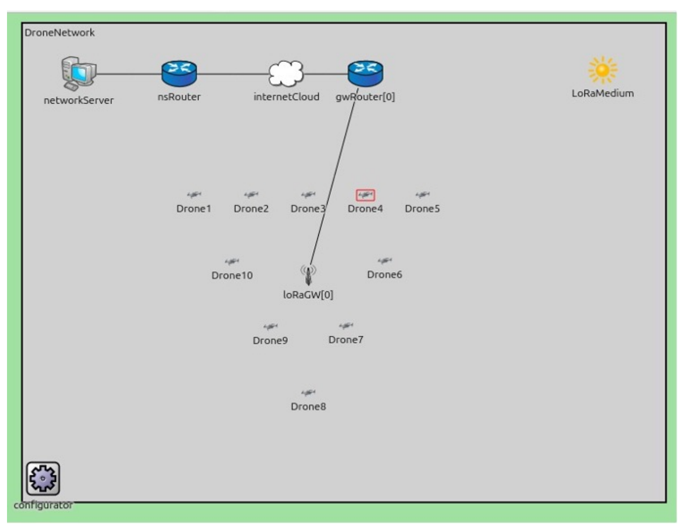


Fig. 7: Drone triangle formation

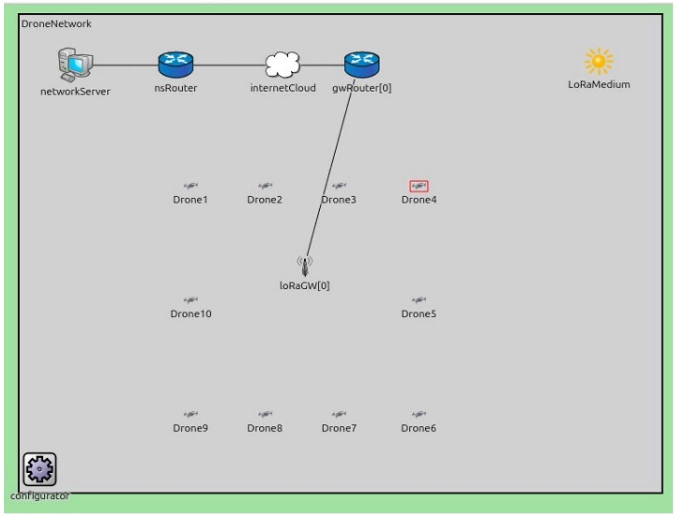


Fig. 8: Drone rectangle formation

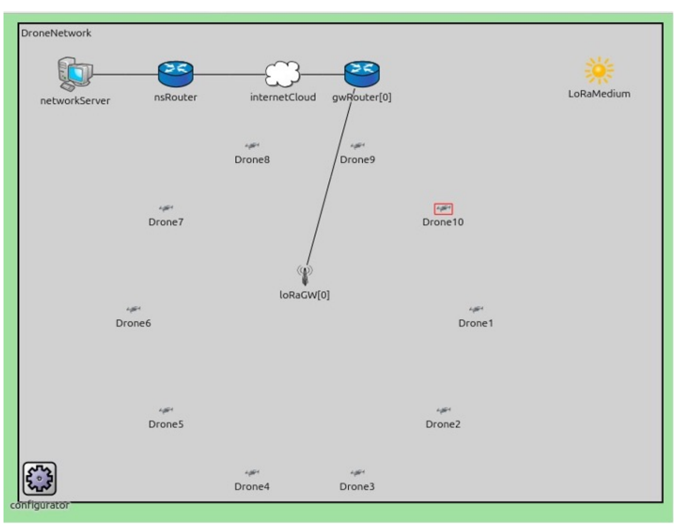


Fig. 9: Drone circle formation

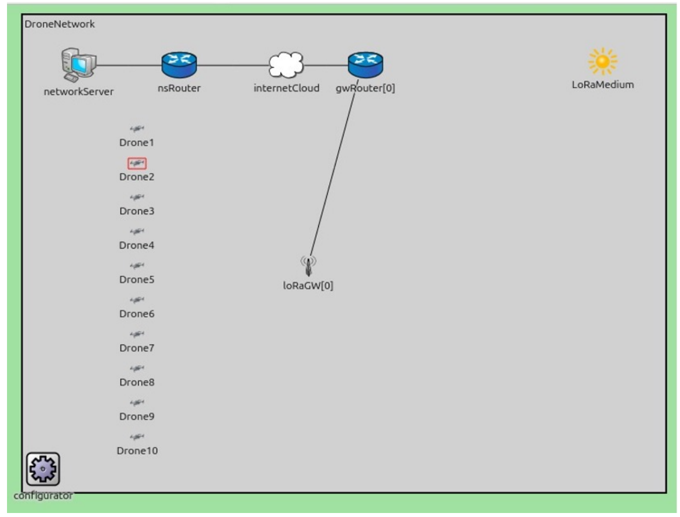


Fig. 10: Drone last formation

J. Simulation Setup

The omnetpp.ini file outlines the simulation's parameters and runtime behavior for the LoRa-based drone air show. The key configurations are:

- **Network Topology**: Defined as DroneNetwork.
- **Duration and Randomness:**
  - Simulation runs for 1 day (sim-time-limit = 1d) with 30 repetitions.
  - Random number generator uses Mersenne Twister.
- **Communication Parameters:**
  - Maximum transmission duration is 4 s.
  - Radio sensitivity threshold is -110 dBm.
  - Adaptive Data Rate (ADR) is enabled in both the network server and drones.
- **Traffic Configuration:**
  - Each drone runs a SimpleLoRaApp and sends packets to the gateway at exponential intervals.
  - The LoRaGW also sends packets to Drone1 using its own application.
- **Mobility:**
  - Drones use TurtleMobility with paths defined in turtle1.xml.
- **Drone Initialization:**
  - 10 drones are placed at coordinates (200 m, 200 m) for controlled testing.
  - All drones use SF12, 14 dBm power, 125 kHz bandwidth, CR=4.
- **Gateway Configuration:**
  - Single gateway at (250 m, 220 m) connected through a backbone network.
- **Energy Modeling:**
  - Each drone uses LoRaEnergyConsumer linked to an ideal energy storage model.
- **LoRa Medium Settings:**
  - Custom LoRaLogNormalShadowing path-loss model.
  - Neighbor cache range = 546 m, refill period = 3000 s.

K. Network Topology (.ned)

The DroneNetwork.ned file defines the structural layout of the drone air show system. Main components include:

- **Drones (LoRaNode):**
  - Instantiated as Drone1 through Drone10.
  - Each has a dedicated TurtleMobility script (from turtle1.xml) and uses LoRa radio/MAC modules.
- **LoRa Gateway (loRaGW[0]):**
  - Receives packets from all drones.
  - Physically connected to the Internet via gwRouter.
- **Network Server and Routers:**
  - StandardHost node acts as the network server via a UdpApp.
  - Two tiers of routers (gwRouter[] and nsRouter) connect the gateway to the server through an InternetCloud.
- **LoRa Medium:**
  - Shared LoRaMedium module for all radio communication.
- **Configurator & InternetCloud:**
  - Ipv4NetworkConfigurator auto-assigns IP addresses.
  - InternetCloud models the backbone connection.

IV. GitHub Repository

- **Repository**:https://github.com/Emre-Ged/Drone-Air-Show-Controller-System-with-LoRa
- **Description**: The GitHub repository contains all project files, including the simulation workflow and future improvements. A README file provides an overview of the project and the technologies used.

References

[1] Semtech Corporation. (n.d.). LoRa modulation basics. LoRa Developers. Retrieved from https://lora-developers.semtech.com/library/tech-papers-and-guides/lora-modulation-basics

[2] Varga, A. (n.d.). OMNeT++ simulation manual. OMNeT++ Community. Retrieved from https://omnetpp.org

[3] Razak, H., et al. (2023). LoRaWAN for drone communication: A simulation study. *IEEE Access*, 8, 123456–123465. https://doi.org/10.1109/ACCESS.2023.123456

[4] Khan, M. T. R. (2025). Discrete-Event Simulation [Lecture slides]. CNG-476 System Simulation, Computer Engineering Department, Middle East Technical University, Northern Cyprus Campus.

[5] Gedikli, S. E. (2025). Drone Air Show Controller System 2D Diagram [Visualization]. Created with Python and Matplotlib.

[6] Kayhan, A. (2025). Event-Logic Diagram [Visualization]. Created with draw.io.

[7] Kayhan, A. (2025). Simulation Trace Table [Visualization]. Created with draw.io.