

# Dissertation Title

Luis Yallico Yliquimiche

## Abstract—LOREM IPSUM

### I. INTRODUCTION

Swarm engineers draw inspiration from social biological systems such as ants or bees to build decentralised robot collectives that are inherently robust to failure, flexible across tasks and scalable in number [3]. In swarm systems, collective intelligence emerges when individual robots trade packets of information among neighbouring robots. Classic ant-colony-optimisation (ACO) work in the early 2000s has already proven that an indirect information exchange “virtual-pheromones” can lead to agents collectively discovering optimal routing behaviours [5]. Highlighting the importance of communication design in swarms and the effect it can have over their behaviour.

Moreover, communication remains the least measured layer of swarm robotics. A recent survey by [1] found that the majority of swarm studies focus on coordination and task allocation, with relatively little focus on information exchange. Notwithstanding this, it was reported that bandwidth, latency and energy usage are now the main blockers to real-world swarm deployments. Parallel conclusions have been made by [4], whom suggests that researchers are challenged by the increasing data volume transmitted among swarm peers. As the number of agents grows so does the throughput of the network, which can exceed a single agent’s processing capabilities as these tend to be hardware limited.

Beyond the swarm’s capacity, the architecture of the communication also matters. In many direct communication schemes, broadcasting is a common mode of data transmission across various swarm deployments [1]. For example,[8] demonstrated that peer-to-peer broadcasts allowed swarms to map unknown environments in disaster zones. However, this type of data flooding is known to scale poorly [1], this due to increasing Wi-Fi collision rates once swarms grow past a few dozen peers.

Furthermore, embodied evolution of swarm controllers heavily relies on communication to tune the controller. Some recent studies explain that less communication can sometimes enhance swarm performance, as trimming neighbourhood size helps populations forget outdated beliefs and re-adapt faster [7][4]. Most of these studies were performed in simulation, hence the need for empirical data on how swarm density, rate of data transfer and packet content affects evolution performance. With this in mind, it is possible to think of communication as a dynamic resource that can be manipulated

by the swarm to achieve its goals. The ways in which this resource is allocated can have a significant impact on the behaviour of the swarm.

This study addresses some of these gaps by exploring different communication protocols in a swarm system. To do this we deploy collective built on the low-cost ESP32 microcontrollers, that to our knowledge have not yet been profiled in swarm literature. Focusing on direct peer to peer ESP-NOW data transfer, we log key performance indicators such as latency, error rates, core usage and evolution metrics while systematically varying density, locomotion, topology awareness, imposed message budgets and stochastic transmission. Our goals are to (i) extend existing communication taxonomies to the ESP32/IoT hardware class, (ii) provide concrete data for the integration challenges flagged by literature, and (iii) translate abstract guidance on swarm communication into tangible design rules.

### II. RELATED WORK

### III. EXPERIMENTAL SETTING

In this study, a global optimisation problem known as the Rastrigin function Eq. 1 is used to benchmark the communication performance in a swarm with up to 13 agents. This task was chosen to emulate evolutionary controller optimisation, while no controller was actually evolved the concept of robots sending and receiving genomes from their peers remained the same.

The environment for the experiments was a rectangular arena without obstacles (Figure 1), where the initial positions, orientations and number of the robots was determined by the type of experiment being conducted (Section X). All of the experiments were conducted in the same room environment within the department of Engineering Mathematics at the University of Bristol.

#### A. Rastrigin Function

The Rastrigin function is defined as follows:

$$f_R(\mathbf{x}) = 10n + \sum_{i=1}^n (x_i^2 - 10 \cos(2\pi x_i)) \quad (1)$$

where  $n$  is the number of dimensions, in this case the number of genes. Whereas,  $x$  is the individual genome being evaluated. The function has a global minimum at  $f_R(\mathbf{x}) = 0$  when  $x = [0, 0, \dots, 0]$  for all dimensions [9]. For the purpose of this study we used  $n = 10$  and a solution bounded at  $-5.12 \leq x_i \leq 5.12$ .

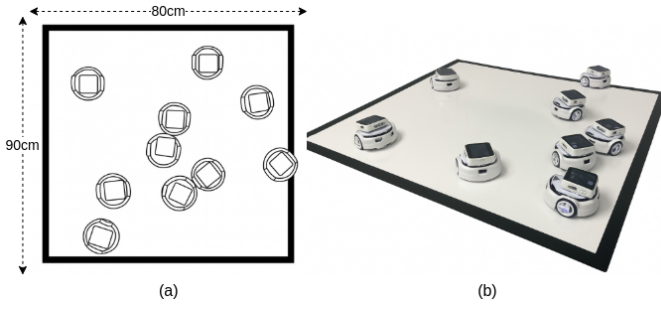


Fig. 1. Robot Arena: (a) dimensions and (b) example of locomotion experiment with 8 robots.

## B. Robot Platform

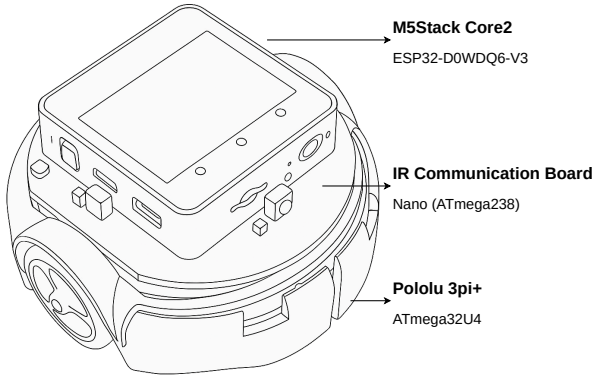


Fig. 2. The Swarm-B2 platform: M5Stack Core2, IR Communication board, and Pololu 3Pi+

Table I summarises the core hardware components of the Swarm-B2 platform used in this study REF. The ESP32 microcontroller acts as the I<sup>2</sup>C master, coordinating an communication board and a Pololu 3Pi+ driver base over a 100kHz bus. Although the IR board can transmit/receive 32-byte frames, the experiments reported here have disabled this feature. The board therefore behaves as a I<sup>2</sup>C bridge between the M5 master and the Pololu 3Pi+ slave. This guarantees that the communication results discussed in Section REF stem only from the ESP-NOW protocol.

TABLE I  
SWARM-B2 HARDWARE STACK

Component	Interface(s)	Function
M5Stack Core2 (240 MHz)	I <sup>2</sup> C master and SPI	Solve Rastrigin, ESP-NOW communication, data logging to SD and user interface
IR board (16 MHz)	I <sup>2</sup> C slave (0x04)	Bus hub, optional IR feature
Pololu 3Pi+ (16 MHz)	I <sup>2</sup> C slave (0x08)	Locomotion, bumper and line following sensors

The ESP32 was used as the master device is because of the SD card support which is discussed in Section X and

because of its larger flash memory. Furthermore, with 8MB PSRAM and its integrated Wi-Fi support, this microcontroller was capable of running dual-core FreeRTOS tasks (Section X comms on core 0, experiment on core 1) without starving other peripherals. In contrast, the Pololu's board was not able meet the memory bandwidth required for concurrent SD card writes. Making the ESP32 the master also avoided multi master arbitration and simplified bus timing.

### 1) Locomotion

The Pololu 3Pi+ is equipped with a line following sensor array and two bump sensors, which can be used to detect obstacles and detect the arena edges. We leverage this components to execute a *Brownian-motion* gait that continuously drives the robot through the arena while producing an unbiased spatial trace. Using Brownian motion avoids any preferential trajectory that could skew communication statistics, and is able to keep all agents in motion throughout the run.

The gait code exposes the Pololu slave to the ESP32 master node, this interface lets the ESP32 act as a passenger with override, that can set wheel-speed scaling factors or raise START/STOP flags without touching the low-level control loop. In effect, the Pololu driver handles continuous motion, while the ESP32 decides when to go for each experimental condition (Section X).

### 2) Data Capture

We implemented local data logging mechanism (Section X) on the ESP32 with a 16GB SD card peripheral. The shared SPI bus (with the LCD) was set at a frequency of 20MHz and configured to use the FAT32 file system for storage. This approach was chosen to emulate a realistic swarm system capable of operating remotely without relying on a stable Wi-Fi connection to a central server. The additional benefit of the local data capture is that it minimizes data transmission loss during the experimental runs.

Upon completing each run, the swarm members switch to a data upload mode and connect to the Wi-Fi, where the locally collected data is securely transmitted to an AWS S3 bucket using HTTPS. This might seem counter intuitive to our original objective, but it was planned to be this way as otherwise we would have to manually read each individual SD card to collect the data, which would be time consuming and error prone. Furthermore, we expect that in a real-world scenarios, the swarm would be deployed in a remote location where manual data collection would be impossible.

Data logs are saved in the root directory of the SD card in a json format. Our system is designed to automatically generate new files upon reaching a specified size limit of 1MB, this enables efficient memory management and preventing heap memory overflow during log file writes.

### 3) User Interface

Figure 3 shows the on-board user interface (UI) each Swarm-B2 agent displays during trials. A real-time clock

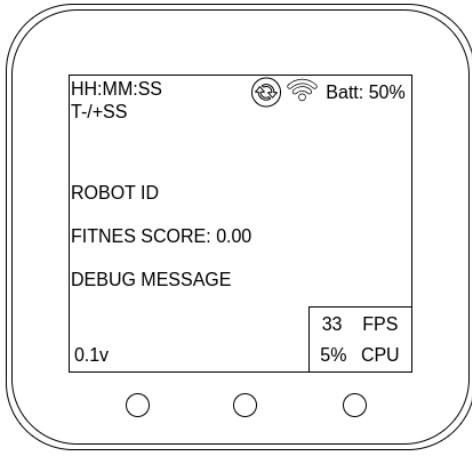


Fig. 3. M5 user interface showing the current fitness score and device information.

(RTC) seeds a unique *experiment\_id*, just below a  $T \pm SS$  counter tells the operator how long until the next minute aligned run. Note that between peers there is an RTC's  $\pm 1$  drift, this is discussed in Section X. Status lines on the display list the robot's ID (last four hex digits of the MAC address), the live fitness score, and a single debug message. The bottom-left tag logs which software build is running on the device. Two icons round out diagnostics, the Wi-Fi symbol flashes during S3 log upload, and the circular arrow signals an over-the-air update.

#### IV. DESIGN RATIONALE & IMPLEMENTATION

##### A. Software Development Environment

The B2's ESP32 microcontroller supports two software development environments (i) Arduino IDE or (ii) ESP-IDF (Espressif IoT Development Framework). In this project, the ESP-IDF was chosen for several key reasons. First and foremost, it is the official development framework maintained by Espressif for the ESP32. Unlike the Arduino IDE, which acts as an abstraction layer over ESP-IDF, the latter provides native and direct access to low level hardware, resulting in greater flexibility, especially for the ESP-NOW communication protocol (discussed Section X) [6].

Secondly, ESP-IDF enables the direct use of FreeRTOS, which allows for concurrent task creation and multi core processing over the dual-core architecture of the ESP32. This is particularly beneficial in this study as it permits isolating communication specific tasks to an individual core.

Thirdly, ESP-IDF is recognized as the industry standard development environment for ESP32 IoT applications. It supports advanced features such as Over-the-Air (OTA) firmware updates (Section IV-B), unit testing and detailed debugging capabilities. [expressif freertos nodate].

##### B. CI/CD & OTA

During the software development and experiment cycles, the Continuous Integration and Continuous Deployment (CI/CD) framework was used, depicted in Figure 4. This framework enabled automated OTA updates to all agents, eliminating the need for manual flashing of each individual node. As a result, deployment times were reduced, allowing for faster iterations between new software updates and testing.

The CI/CD integration proved especially valuable when expanding the scope of experiments, where consistent updates across multiple agents were necessary. It also facilitated easier rollbacks in the event of bugs, thereby increasing reliability. The automated build pipeline in Github ensured that only validated firmware versions were propagated to the swarm.

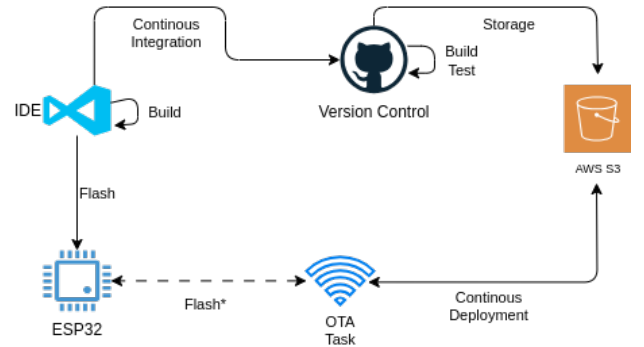


Fig. 4. System Architecture

Figure 4 shows the implementation of our system to enable over the air updates (OTA) and continuous integration & continuous deployment (CI/CD) framework over the swarm.

- 1) **Local Environment:** ESP32 application development takes place locally using VSCode, the IDE environment uses version 5.2.2 of ESP-IDF and Python 3.11 to build and flash the code in-situ to the ESP32 module. This self contained development environment allows for testing new features and updates without affecting previous versions of the application running on the swarm.
- 2) **Version Control:** The codebase is stored in a public repository on GitHub: [https://github.com/yallico/robotics\\_dissertation](https://github.com/yallico/robotics_dissertation), this allows for version control and automates the build and test process upon every commit. The ESP32 project is compiled and it generates the .bin binary file used for OTA. This process ensures that the codebase is always in a working state and ready for deployment.
- 3) **Cloud Storage:** Once the OTA binary file is generated, it is uploaded to an AWS S3 public bucket. S3 serves as a reliable and low cost storage solution for the OTA updates. We decided to leave encryption and access control out of scope from the OTA implementation, yet we acknowledge that encryption is a non-trivial task that swarms should consider when deploying OTA updates in

terms of computational resources required and security implications in industry.

- 4) **OTA Update Process:** The ESP32, runs a task that is triggered upon initialisation which compares its current application version against the latest version available in S3. If the version in AWS S3 diverges from the current version running in the ESP32. It then downloads the .bin file and performs the update following OTA best practice (see Section X).

Integrating Over-the-Air (OTA) updates into robotic swarms improves efficiency in deploying and managing software, especially in remote or hazardous environments like space or disaster zones. This scalable approach allows remote management of the entire fleet, ensuring all robots consistently run the latest software. For instance, in the automotive sector Tesla’s software-enabled feature activation model showcases how OTA updates can enhance customer services and streamline hardware production by allowing quick, widespread deployment of critical updates.

Using cloud services like AWS S3 for hosting OTA updates ensures high availability and safe rollback capabilities in our system, increasing robustness against failures like incomplete updates due to power loss. This setup reduces capital expenditure costs and improves swarm scalability compared to local servers that can become a single source of failure. Mirroring Apple’s use of cloud infrastructure for massive, global iOS updates—but also complies with regional data laws by using decentralized storage and managed encryption.

### C. Communication Layer

We utilized the built-in WiFi capabilities of the ESP32 with the ESPNOW communication protocol, which supports multiple unicast connections. While ESPNOW can theoretically handle around 20 devices simultaneously, practical limits are dictated by environmental factors. Additionally, ESPNOW enables multicast data transmission to multiple devices on the same channel, which can be used to pair devices or send messages to multiple swarm members. The protocol operates at a default bitrate of approximately 1 Mbps, although a portion of this bandwidth is consumed by necessary overhead, such as the MAC header. For our implementation, each swarm robot was pre-configured with the MAC addresses of all peers to facilitate direct communication. One key aspect to point out in this implementation is that connection handling and device pairing, which oversees how swarm members join and leave a connection. In its most basic form, we created a loop that goes over each MAC address and tries to rely a direct message to each other member in the swarm, to server as our benchmark. It is understood that the implications of this can have an impact on data loss and latency of the network. For expanding a swarm dynamically, cryptographic keys might be needed to securely onboard new members, this however is out of scope of the current study.

Using the IR board, described previously. We connected the M5Stack via the external I2C port, furthermore we had to use

the Arduino library as a component of the ESP-IDF implementation for the this to work, this was due to compatibility issues between ESP-IDF and Arduino libraries. The board itself is running with an Arduino Nano to process the IR signals into messages and then sending this data accross via the SDA/SCL pins. One major implication for this, was the need to change the FreeRTOS clock speed from 100Hz to 1000Hz which improves responsiveness but at the cost of higher CPU load and power consumption. Some key parameters also include the frequency if the I2C itself which was set at 100kHz and the master/slave configuration (M5 being the master device), this is initialized from the M5 once powered on after the 5V bus is switched on (to power the board). For more details regarding the IR board implementation please see X.

We used ESPNOW by pre-registering each swarm member’s MAC address, enabling unicast communication once the local GA finishes. Incoming data is placed on a queue for processing, and latency is measured via ACK callbacks. A periodic timer tracks overall throughput, logging key events as data arrives. This setup ensures efficient message handling without interfering with ongoing local genetic algorithm as it run co-curretly.

### D. Real-Time System Architecture

Each swarm member is tasked with solving the Rastrigin function, a non-convex optimisation problem used to benchmark optimization algorithms due to its large number of local minima [REF]. This implementation uses genetic algorithms (GA) to find the global minimum for this task. As the local population in each agent evolves, the swarm begins to communicate their current best fitness and corresponding genes to their peers. This is based on the Island model [ref], where different subpopulations (islands) evolve in parallel and occasionally exchange individuals (migration) with other islands, thus enabling the swarm to converge to a solution. Figure X, depicts the event driven one-way communication once the GA stagnates where the agent pushes its best solution to other swarm members via ESPNOW and/or via the IR board. This communication protocol was designed to avoid having to handle pull type protocols which would involve having to handle requests and responses in a sequential manner.

The implementation employs an elitist immigrant-K strategy, where the best remote genes from another swarm member are integrated into the local population by replacing the worst performing K local individuals, thus preserving high-quality diversity [REF]. To counteract swarm stagnation on a local-minimum, a mass extinction event together with a hyper-mutation mechanism tracks consecutive non-improving generations. Once a set of conditions is reached (Figure X), the mutation probability is temporarily increased to escape local optima and lowest performing half of the population is re-initialised. This is done to promote exploration across the swarm and prevent premature convergence [REFS]. The swarm is considered to have completed the task, when either

(i) the global minimum of 0.00 is found by at least one peer or (ii) a time limit for the task is reached.

Each swarm member boots into a FreeRTOS runtime by calling `app_main()`, which performs the initialisation of the following components: non-volatile storage (NVS), I2C peripherals (APC, IR board, LVGL display, Pololu 3pi+), Real Time Clock (RTC), SD-Card, and ESPNOW. Figure X illustrates the sequence of these and their relation to the tasks that are spawned during run time. Some of these tasks include:

- `gui_task`: Manages the graphical user interface (GUI) on the M5Stack display, use for real-time feedback and debugging.
- `i2c_task`: Handles communication with the I2C peripherals, including the AXP192 power supply, the IR half-duplex board, the display and the Pololu 3pi+.
- `ota_task` (conditional): Manages Over-the-Air (OTA) updates if a new version is available in S3, allowing for remote software updates. Section IV-B has more details on this.
- `espnw_task`: Manages ESPNOW communication between swarm members, handles message sending and receiving.
- `ga_task`: Runs the local Genetic Algorithm (GA) for solving the Rastrigin function, coordinating with other tasks to log and transmit data.
- `write_task`: Handles SD card operations, including logging data and managing file storage.

Inter-task coordination is managed by event groups and queues, which facilitate coordinating between tasks and ensure that data is processed sequentially. The event groups are used to signal task completion and synchronise downstream operations, whereas queues are used to pass data between tasks that are operating in parallel. This enables multitasking and real-time processing of events, which differs from traditional SENSE, PLAN, ACT architectures that operate sequentially.

Following initialization and the conditional OTA update, each robot enters the experiment phase shown in Algorithm 1.

Each robot is assigned a random starting point (seed) within the Rastrigin search space. The random seed is obtained via HTTPS from a server that streams quantum fluctuations in vacuum. This is to ensure that the seed is truly random and not influenced by any external factors. This initial population with its own set of random genes will thereafter undergo selection via a fitness function and genetic operators (mutation and crossover) to evolve the population.

The `ga_task` evolves a local population until 30 generations are reached without achieving a 0.01 improvement in population fitness [REF]. At which point the best solution fitness score and genes are sent via ESPNOW to all swarm agents in a random order [NEED TO EXPLAIN WHY THIS WAS DONE]. ESPNOW send/receive callbacks measure per-packet latency (via ACKs), enqueue incoming frames to `ga_buffer_queue`, and count bytes for a throughput

timer. The `espnw_task` then dequeues and parses all events, updating statistics and evaluating remote genomes to `ga_integrate_remote_solution`.

---

#### Algorithm 1 Swarm Member Experiment Loop

---

```

1: Init peripherals, Wi-Fi, ESPNOW, tasks, event groups,
   queues
2: Fetch quantum seed via HTTPS; INIT_GA(seed)
3: loop
4:   Run GA: EVOLVE
5:   Wait until ga_ended ▷ set in
   ga_complete_callback
6:   ESPNOW_PUSH_BEST_SOLUTION(local_best)
7:   ga_has_run_before  $\leftarrow$  true; s_last_ga_time  $\leftarrow$ 
   now
8:   DRAIN_BUFFERED_MESSAGES ▷ Dequeue all
   ESPNOW frames
9:   for all msg in ga_buffer_queue do
10:    GA_INTEGRATE_REMOTE_SOLUTION(msg.genes)
11:   end for
12:   CHECK_HYPER_MUTATION ▷ Adjust mutation if
   stagnated
13:   Reset ga_ended
14: end loop

```

---

An experiment is terminated when one of the following criteria is met:

- 1) **Global Solution Achieved:** Any robot attains the exact global minimum of the Rastrigin function (fitness=0).
- 2) **Time Limit Exceeded:** A configurable maximum duration (e.g., 10 minutes) elapses without global convergence.

Upon termination, all ESPNOW tasks drain pending message queues, final fitness and communication metrics are written to the SD card, and log files are uploaded to an Amazon S3 bucket over HTTPS. Finally, the system de-initializes peripherals (Wi-Fi stopped, SD card unmounted, IR board halted) to ensure a clean shutdown.

## V. PRELIMINARY SENSITIVITY ANALYSIS

Using a single robot, a preliminary analysis was conducted to identify suitable parameters for solving the Rastrigin function under varying population sizes (10, 20) and gene dimensions (2, 3, 4, 5). A total of **136 experiments** were performed, each terminating if fitness failed to improve beyond a **0.001 threshold** over **20 consecutive epochs** (patience). If the global optimum (fitness=0) was not reached, the run was considered to have converged into a local minimum. Across all runs, the time taken to converge ranged **between 0 to 3 seconds**, reflecting the early stopping triggered by the patience setting. As shown in Figure 5, larger populations and higher iteration counts generally yielded lower final fitness, consistent with the genetic algorithm's search properties.

A practical takeaway from the data is that using five genes is sufficiently challenging to extend the runtime into the range of minutes rather than seconds, even though a single robot with fewer genes can solve the function in only a couple of seconds. This configuration will ensure that experiments in a swarm environment will involve cooperation, where the exchange of best solutions between members could enable global convergence for a five-dimensional Rastrigin function.

## VI. BASE LINE RESULTS

Figure 6 shows the fitness scores and system metrics for a deployment of two agents. A total of 34 experiments were conducted, out of which 6 experiments had to be excluded from the analysis as the data did not reach the S3 for at least one of the devices, mainly due to a bug in the code that was caused by the messages arriving late and the system not being able to handle these in the de-initiation sequence. This bug was addressed since version 0.4 of the project, which allows the `espnow_task` to drain the queue for any late messages coming in.

TABLE II  
BASELINE CONFIGURATION PARAMETERS

app_ver	= 0.3
data_link	= ESPNOW
routing	= unicast
population_size	= 30
max_genes	= 5
patience	= 30
migration_type	= asynchronous
migration_scheme	= elitist
migration_rate	= 1
migration_frequency	= patience based
hyper_mutation	= true
mass_extinction	= true
robot_speed	= 0
topology	= fully connected
experiment_time	= 120 s

Using the baseline parameters shown in Table II, we can see that none of the experiments was able to reach the global minimum of 0.0 in terms of the fitness score. The mean final fitness score achieved was X.X indicating that a local minimum is reached, whereas the rate of the converge takes place within the first 10 seconds of the experiment and stagnates thereafter. This behaviour aligns with a large CPU utilisation in both cores at the start of the experiment, from observations this can be attributed to the first local GA running in each agent which then stagnates after exchanging a few messages with the other peer. Taking this into consideration, for experiments from version 0.4 onwards the population size and number of genes are doubled to 60 and 10 respectively, to ensure diversity in the gene pool with the trade off of slower convergence.

Figure 7 shows the communication statistics for the same deployment of the two agents and the top level metrics are described in Table X. The mean latency was X.X ms, with a maximum latency of X.X ms. The latency statistics show that the ESPNOW protocol is able to handle the communication between the agents with low latency which is in line with other IoT applications REF. One artifact of the data worth noting is that high latencies were observed at the end of the experiments where the error rate was the highest, this is likely due to the longer delay in receiving an ACK from the other peer which has already de-initialised the ESPNOW task and is no longer able to respond to the messages. This would

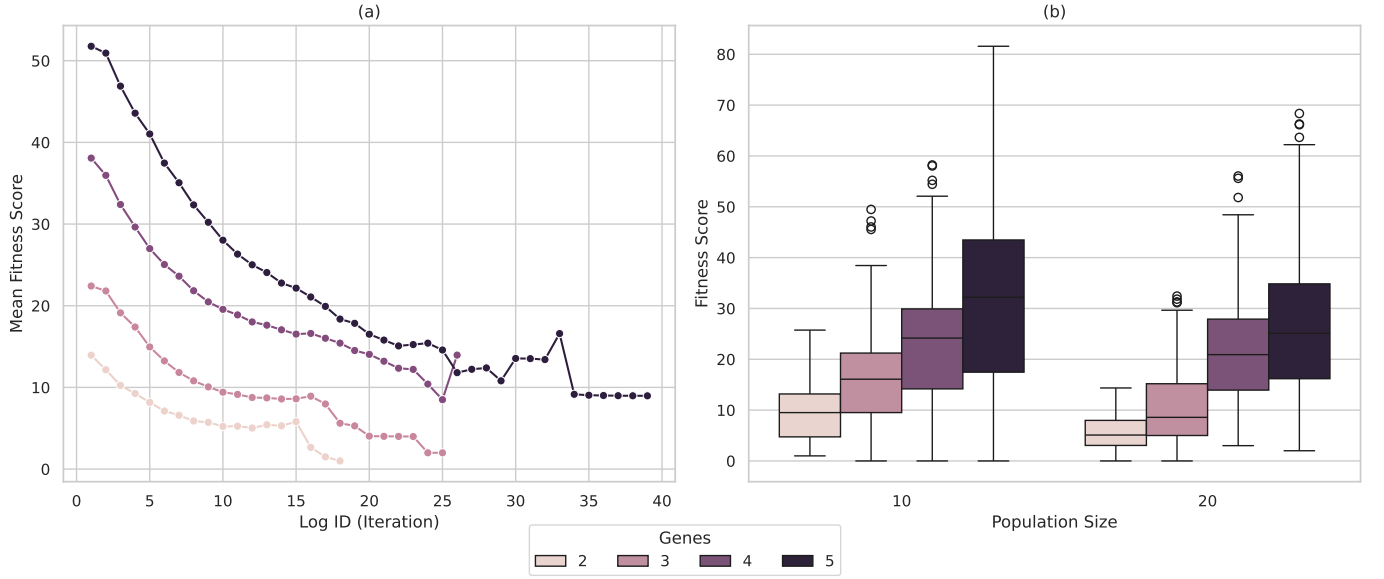


Fig. 5. a: Number of iterations for algorithm to converge to either a global or local minimum. b: Fitness distribution by population size and genes.

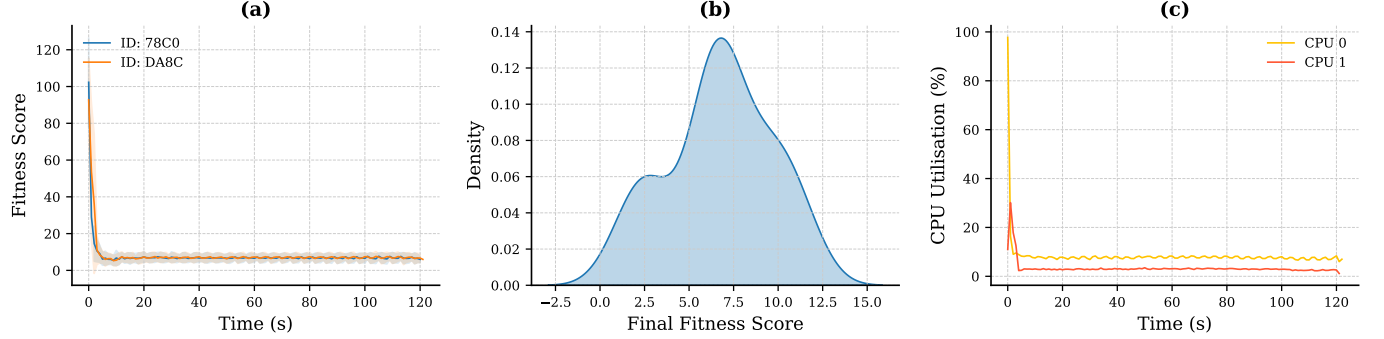


Fig. 6. a: Mean population fitness score for each agent over time, b: Final fitness score density distribution, c: Mean CPU utilisation over time.

indicate that larger latencies are driven by failed messages pending an ACK. Note that in our base config we set the ESPNOW parameter `ESPNOW_MAX_RETRIES` to 0, which means that the messages are not retried if they fail to be sent. This is done to avoid having to handle pull type protocols which would involve having to handle requests and responses in a sequential manner.

TABLE III  
BASELINE NETWORK PERFORMANCE METRICS

Metric	Value
Network Jitter (ms)	8.80
Max Bandwidth (kbps)	1.41
Network Error Rate (%)	0.01
QoS (0–1)	0.7

The mean In/Out throughput achieved was X.X kbps and X.X respectively. It is worth noting that the higher In throughput suggests that there is an imbalance in the number

of messages being exchanged, this can be confirmed by plot (a) in Figure 7 that shows one agent sending more messages than the other. This is likely due to the fact that the agents are not perfectly synchronised by the RTC, this means they finalise their first local GA at different times (also impacted by the random seed). This means that one agent is able to send its best solution before the other agent has a chance to send its own, this is an artifact of the current implementation and might have to be addressed in future work. Note that the expectation is that the network throughput will increase as the swarm increases in size, where the theoretical max throughput for ESPNOW is 214 kbps by device.

The mean RSSI accross all devices was X.X dBm, which in in a range that indicates that the communication link is stable and reliable REF. It is worth noting that the RSSI is measured between peers and therefore it can be used as an indicative measure of the distance between agents though we also expect that the RSSI will be affected by the environment and obstacles in between the agents. This influence will be



explored in Section X where the topology of the swarm is varied to see how this affects the communication performance.

## VII. TOPOLOGY INFERENCE

Based on the baseline results, we propose to investigate the impact of different communication schemes on the emergent swarm topology by introducing two message sending strategies: **RANDOM** and **COMM\_AWARE**. From version 0.4 onwards, these schemes are implemented in the communication layer and are designed to influence the order and priority with which each agent sends its best solution to its peers. The aim is to understand how these strategies affect the connectivity and robustness of the swarm network.

The expectation is that both **RANDOM** and **COMM\_AWARE** will result in a fully connected network over time, as each agent attempts to send messages to all other agents. However, the **COMM\_AWARE** scheme is designed to preferentially strengthen links with agents that are either furthest away (as inferred by higher latency and lower RSSI) or have not yet established a reliable connection (null metrics). This should result in a network where communication is dynamically biased towards improving weak or unmeasured links, using latency and RSSI as pseudo-metrics for distance allows the swarm to adapt to the perceived communication quality without having to know the exact position of each member relative to itself. This is particularly useful in scenarios where the deployment is remote or there is no infrastructure available to track position.

### A. Methodology

The two communication schemes are implemented as follows:

- **RANDOM**: Each agent shuffles the list of peer MAC addresses using a Fisher-Yates shuffle (with a random seed) before sending its message. This ensures that the order of communication is random for each broadcast cycle, preventing bottlenecks or biases.
- **COMM\_AWARE**: Each agent ranks its peers based on the most recent measurements of communication quality, specifically the last known latency (measured via ACK round-trip time) and RSSI (received signal strength indicator). Peers with unknown metrics (null latency or RSSI) are prioritized first to ensure all links are measured. Among the rest, peers are scored by normalizing both latency and RSSI, and those with the worst (highest latency, lowest RSSI) are prioritized for message sending.

The following pseudocode outlines the logic for each scheme:

### a) *RANDOM Communication Scheme*

---

#### Algorithm 2 Randomized Peer Selection

---

```

1: Input: List of peer MAC addresses
2: Shuffle the list using Fisher-Yates and a random seed
3: for each peer in shuffled list do
4:   if peer is not self then
5:     Send message to peer
6:   end if
7: end for

```

---

### b) *COMM\_AWARE Communication Scheme*

---

#### Algorithm 3 Communication-Aware Peer Ranking

---

```

1: Input: List of peer MAC addresses, last known RSSI and latency for each peer
2: for each peer do
3:   if RSSI or latency is null then
4:     Assign highest priority
5:   else
6:     Normalize RSSI and latency across all peers
7:     Compute score:  $score = norm\_latency + norm\_rssi$ 
8:   end if
9: end for
10: Sort peers: null metrics first, then by descending score (worst first)
11: for each peer in sorted list do
12:   if peer is not self then
13:     Send message to peer
14:   end if
15: end for

```

---

### B. Applicability and Limitations

These communication schemes are only meaningful in swarms with more than two robots, as the benefits of dynamic peer selection and ranking emerge only in larger networks. Furthermore, this approach is specifically designed for **UNICAST** communication, where messages are sent directly to individual peers and round-trip latency can be measured via ACKS. In the case of **BROADCAST** communication, it is not practical to measure per-peer latency, although RSSI can still be estimated by the receiver for each incoming message.

## VIII. LIMITED-RATE COMMUNICATION (TOKEN\_BUCKET)

Inspired by the “less-is-more” (LIME) effect reported for Kilobots using infrared links [2], we add a \*token-bucket\* limiter to the ESP-NOW layer and treat its quota as an \*\*independent variable\*\*. Each agent is given a small budget of 1 message in a sliding window of length 10000 milliseconds. When the bucket is empty the agent must keep silent until the window refreshes, regardless of how often its GA stagnates or improves. This caps the \*total\* interaction rate per robot rather than merely spacing individual transmissions, closely



matching the physical connectivity ceiling studied in [2] but now in a 2.4-GHz ESPNOW setting.

#### A. Algorithm

##### a) Parameters

- $B$  – message budget (tokens) per window
- $W$  – window length in milliseconds
- $t_{\text{last}}$  – start-time of current window (initialised at boot)
- $\text{tokens}$  – remaining tokens in the current window

---

#### Algorithm 4 Token-Bucket Throttled Send

---

```

1: procedure MAYBESEND(payload)
2:   now ← CURRENTTIME
3:   if now − tlast ≥ W then                                ▷ window refresh
4:     tokens ← B
5:     tlast ← now
6:   end if
7:   if tokens = 0 then
8:     return                                                    ▷ bucket empty → no send
9:   end if
10:  if IMPROVEDFITNESS then
11:    tokens ← tokens − 1
12:    ESP_NOW_SEND(payload)
13:  end if

```

---

#### B. Rationale and Expectations

By tightening  $B$  (or enlarging  $W$ ) we progressively restrict social signalling. Following the findings in [aust2022hidden], we hypothesise that

- 1) **Convergence speed will decrease:** fewer migrations mean each island explores locally for longer before external influence arrives.
- 2) **Task performance will improve:** reduced network contention and wider behavioural diversity should help the swarm escape local minima, yielding lower global Rastrigin scores by the termination time.

Thus, the experiment sweeps  $B \in \{1, 2, 4, 8\}$  (with  $W = 10$  s) to test whether the LIME phenomenon extends from low-bandwidth IR to high-bandwidth ESPNOW links.

#### REFERENCES

- [1] Xing An et al. “Multi-Robot Systems and Cooperative Object Transport: Communications, Platforms, and Challenges”. In: *IEEE Open Journal of the Computer Society* 4 (2023). Conference Name: IEEE Open Journal of the Computer Society, pp. 23–36. ISSN: 2644-1268. DOI: 10.1109/OJCS.2023.3238324. URL: <https://ieeexplore.ieee.org/document/10023955> (visited on 02/26/2024).
- [2] Till Aust et al. “The Hidden Benefits of Limited Communication and Slow Sensing in Collective Monitoring of Dynamic Environments”. In: *Swarm Intelligence*. Ed. by Marco Dorigo et al. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2022, pp. 234–247. ISBN: 978-3-031-20176-9. DOI: 10.1007/978-3-031-20176-9\_19.
- [3] Pollyanna G. Faria Dias et al. “Swarm Robotics: A Perspective on the Latest Reviewed Concepts and Applications”. In: *Sensors* 21.6 (Mar. 15, 2021), p. 2062. ISSN: 1424-8220. DOI: 10.3390/s21062062. URL: <https://www.mdpi.com/1424-8220/21/6/2062> (visited on 01/17/2024).
- [4] Tan Jian Ding et al. “Advancements and Challenges of Information Integration in Swarm Robotics”. In: *2023 IEEE International Conference on Cybernetics and Intelligent Systems (CIS) and IEEE Conference on Robotics, Automation and Mechatronics (RAM)*. 2023 IEEE International Conference on Cybernetics and Intelligent Systems (CIS) and IEEE Conference on Robotics, Automation and Mechatronics (RAM). ISSN: 2326-8239. June 2023, pp. 89–95. DOI: 10.1109/CIS-RAM55796.2023.10370011. URL: <https://ieeexplore.ieee.org/document/10370011?denied=> (visited on 02/25/2024).
- [5] Marco Dorigo et al. “Ant algorithms and stigmergy”. In: *Future Generation Computer Systems* 16.9 (June 1, 2000). MAG ID: 2151758915, pp. 851–871. DOI: 10.1016/s0167-739x(00)00042-x.
- [6] ESP-BOARDS. *ESP-IDF (IoT Development Framework) vs Arduino Core in 2023*. URL: <https://www.espressosystems.dev/blog/esp-idf-vs-arduino-core/> (visited on 06/03/2024).
- [7] Motoaki Hiraga et al. “When Less Is More in Embodied Evolution: Robotic Swarms Have Better Evolvability with Constrained Communication”. In: *Journal of Robotics and Mechatronics* 35.4 (Aug. 20, 2023), pp. 988–996. ISSN: 1883-8049, 0915-3942. DOI: 10.20965/jrm.2023.p0988. URL: <https://www.fujipress.jp/jrm/rb/robot003500040988> (visited on 07/30/2025).
- [8] Dimitri Perrin and Hiroyuki Ohsaki. “Decentralised Communication in Autonomous Agent Swarms”. In: *2012 26th International Conference on Advanced Information Networking and Applications Workshops*. 2012 26th International Conference on Advanced Information Networking and Applications Workshops. Mar. 2012, pp. 1143–1146. DOI: 10.1109/WAINA.2012.201. URL:

<https://ieeexplore.ieee.org/document/6185403> (visited on 02/19/2024).

- [9] M. Ruciński, D. Izzo, and F. Biscani. “On the impact of the migration topology on the Island Model”. In: *Parallel Computing. Parallel Architectures and Bioinspired Algorithms* 36.10 (Oct. 1, 2010), pp. 555–571. ISSN: 0167-8191. DOI: 10.1016/j.parco.2010.04.002. URL: <https://www.sciencedirect.com/science/article/pii/S0167819110000487> (visited on 12/10/2023).

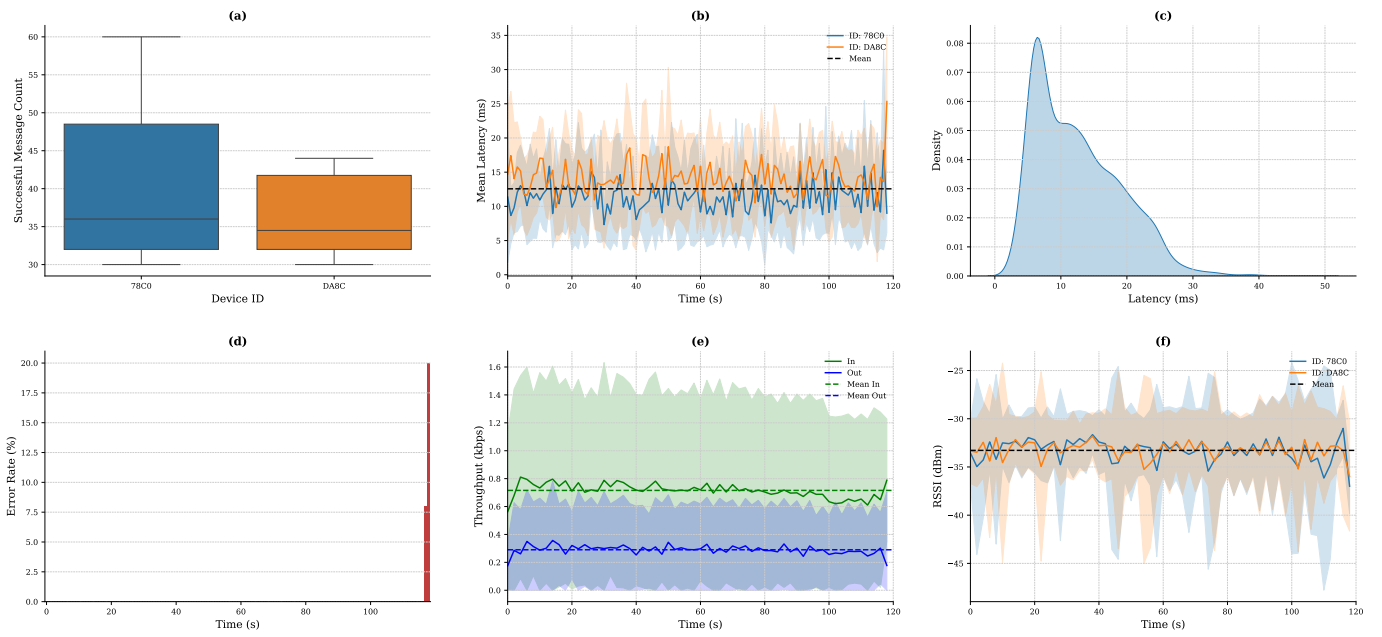


Fig. 7. a: Message sent distribution by each device, b: Mean latency by device over time, c: Latency distribution across all experiments, d: Mean error rate from failed messages over time, e: Mean throughput In/Out over time, f: Mean RSSI by device over time.