Dissertation Title

Luis Yallico Ylquimiche

***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.

While coordination and task allocation have been widely studied, recent surveys highlight bandwidth, latency and energy usage as key limitations to real-world swarm deployments [1][4]. These issues become more pronounced as swarm sizes scale, resulting in an increase in data volume being transmitted among swarm peers, often overwhelming individual agents' hardware capabilities.

Beyond the swarm's exchange 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less communication can sometimes enhance swarm performance, as trimming neighbourhood size helps populations forget outdated beliefs and re-adapt faster [7][4].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 profiles swarm communication on ESP32 hardware, evaluating performance under varied conditions such as agent density, locomotion, topology inference, imposed message budgets and stochastic transmission. With a focus on direct peer-to-peer communication we aim to (i) provide empirical data to validate the challenges flagged by literature, and (ii) inform tangible communication design rules for future swarm systems.

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. 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 remaines the same.

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

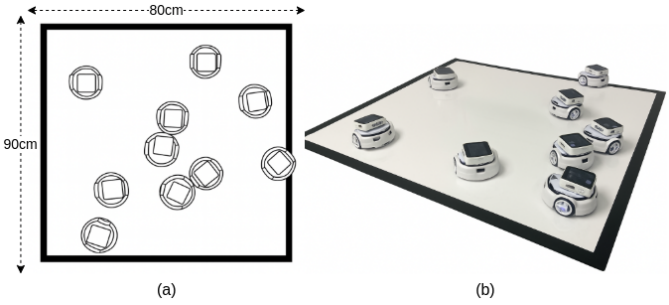

Fig. 1. Robot Arena: (a) dimensions and (b) example of locomotion experiment with 8 robots.

Across the study, we evaluated swarm performance under different communication and system configurations. Each experiment manipulates a specific independent variable while holding all other conditions constant. These include:

- **Swarm density**: The number of agents deployed simultaneously, ranging from 2 to 13.
- **Locomotion mode**: Robots were either stationary (*static*) or navigated using a *Brownian motion* gait (Section III-B1).

- **Topology inference**: Message transmission priority was governed by either a *STOCHASTIC* shuffle or a *COMM_AWARE* ranking strategy based on link quality metrics (Section IV-D1).
- **Message budget**: A token-bucket rate limiter controlled how frequently agents could transmit messages (Section IV-D2).
- **Transmission frequency**: Each message was optionally delayed by a random interval derived from the maximum observed peer latency to reduce potential network collisions (Section IV-D3).

A. Rastrigin Function

The Rastrigin function is defined as follows:

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

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 $x = [0, 0, ..., 0]$ for all dimensions [9]. For the purpose of this study we used $n = 10$ and a solution bounded at $-5.12 <= x_i <= 5.12$.

B. Robot Platform

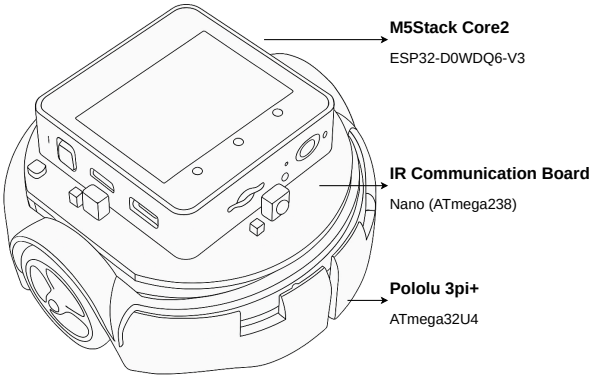

Fig. 2. The Swarm-B2 platform: M5Stack Core2, IR Communication board, and Pololu 3Pi+

Table I summarises the core hardware components of the Swarm-B2 platform used in this study [REF]. Coordination between devices is handled over a I^2C 100kHz bus. Although the IR board can transmit and receive 32-byte frames, we did not conduct experiments with this feature. The board therefore behaves as a I^2C bridge between the M5 and the Pololu 3Pi+. This guarantees that the communication results in this study stem only from a single data link.

The ESP32 was used for running dual-core FreeRTOS parallel tasks, logging data on the local SD card and handling communication via the ESP-NOW data link (2.4 GHz). The 8MB PSRAM and larger flash memory on the master device allowed for concurrent task execution without peripheral starvation.

TABLE I
SWARM-B2 HARDWARE STACK

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

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. A *Brownian-motion* gait was selected to maintain unbiased mobility across the arena and ensure constant movement.

The gait code exposes the Pololu 3pi+ 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 3pi+ driver handles continuous motion, while the ESP32 decides when to go for each experimental condition.

2) Local Data Storage

We implemented local data logging mechanism (Section IV-E) 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.

3) User Interface

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

This section outlines the software design and implementation of the swarm firmware. The Espressif IoT Development Framework (ESP-IDF) was used as it provides low-level hardware access, offering greater flexibility for ESP-

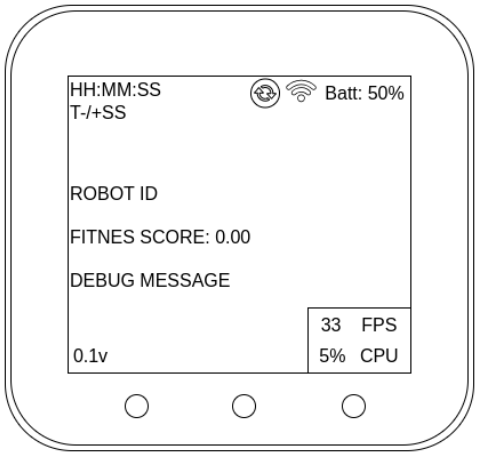

Fig. 3. M5 user interface showing the current fitness score and device information.

NOW communication (Section IV-D). Unlike the Arduino framework, ESP-IDF provides direct access to FreeRTOS, enabling fine-grained control over concurrent task creation and dual-core processing, such as isolating communication tasks to a specific core. It also supports advanced features like over-the-air (OTA) updates (Section IV-A1), unit testing, and custom debugging tools [6].

A. Software Development Environment

We employed a Continuous Integration and Continuous Deployment (CI/CD) pipeline via GitHub to automate OTA deployment, ensuring synchronized updates and easier debugging across the swarm (Fig. 4).

The CI/CD integration proved especially valuable during experiments, where consistent updates across multiple agents were necessary. It also facilitated easier rollbacks in the event of unexpected bugs. The automated build pipeline in GitHub ensured that only validated firmware versions were propagated to the swarm, catching any environment discrepancies early in the process.

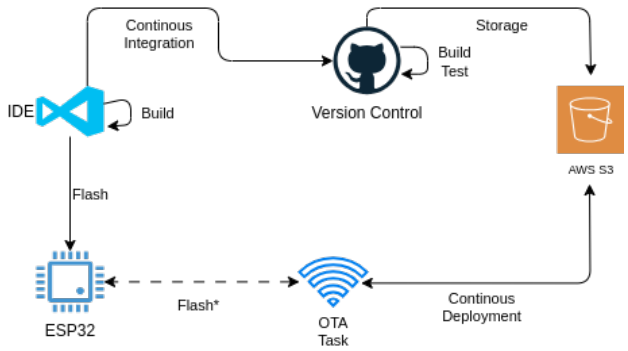

Fig. 4. CI/CD Architecture: Local development, GitHub repository, AWS S3 storage, and OTA updates.

Our software development framework for the swarm is as follows:

1) **Local environment development**: Application development takes place locally using VSCode, our local environment uses version 5.1.4 of ESP-IDF and Python 3.11 to build and flash the code in-situ.

2) **Version control**: We push updates of the codebase to a public repository on GitHub: https://github.com/yallico/robotics_dissertation, this allows for version control and triggers a custom build and test process. The ESP32 project is then compiled remotely and generates the binary file used for OTA.

3) **Cloud storage**: The OTA binary file is uploaded to an AWS S3 public bucket making it accesible to the swarm via HTTPS.

1) OTA Process

Upon initialization, each agent checks its local firmware version against the latest version stored in S3. If a mismatch is detected, the ESP32 downloads and installs the updated .bin file. While relying on a central server and exposing the swarm to the internet for updates may seem counterintuitive, a similar update mechanism could be implemented in a decentralized manner using consensus protocols. However, this would have introduced additional complexity beyond the scope of this study.

TABLE II
SYSTEM PARTITION TABLE

Name	Type	Size	Description
`nvs`	data	16KB	Non-volatile storage
`otadata`	data	8KB	OTA metadata
`phy_init`	data	4KB	PHY layer calibration data
`factory`	app	4MB	Default application
`ota_0`	app	4MB	OTA slot 0
`ota_1`	app	4MB	OTA slot 1

Table II shows how each device partitions was configured with a dual-partition OTA scheme with two application slots: `ota_0` and `ota_1`. During an update, the new firmware is written to the inactive partition. Once the write and integrity checks pass, the bootloader switches to boot from the updated partition on the next reboot. This allows safe rollback in case of update failure. The update binaries ranged from 1 to 1.2 MB, note that update propagation and version control were managed manually through a quick check of the robot's LDC display before the experimental run.

B. Embodied Evolution

The swarm uses a distributed genetic algorithm (GA) to find the global minimum for Eq. 1.A visual representation of this is shown in Fig. 5. As the local population in each agent evolves, the swarm begins to communicate their local best

fitness and corresponding genes to their peers.

Our implementation employs an elitist migration strategy, this happens when the local GA reaches a patience threshold and the agent pushes its ”best” (lowest fitness score) genome to other swarm members via ESP-NOW. The incoming remote genes from another peer are integrated into the local population by replacing the worst performing 5% individuals, this value was chosen to preserve genomic high locally.

To avoid stagnation over 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 (Table 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.

C. Real Time Operating System

Each swarm member boots into a FreeRTOS runtime by calling `app_main()`, which performs the initialisation of the following components: non-volatile storage, I2C peripherals, RTC, SD card, and ESP-NOW.

Figure 6 illustrates the sequence of these and their relation to the tasks that are spawned during run time. These tasks include:

- `i2c_task`: Handles communication with the I^2C peripherals, including the AXP192 power supply, the IR board, the display and the Pololu 3pi+.
- `gui_task`: Manages the UI on the M5Stack display, used for real-time feedback and debugging.
- `pololu_heartbeat_task`: Handles the I^2C signal from the Pololu 3pi+, ensuring that the robot is operational and responsive.
- `ota_task`: Manages OTA updates if a new version is available in S3 (Section IV-A1).
- `espnow_task`: Manages ESP-NOW communication between swarm members, handles message sending and receiving.
- `ga_task`: Runs the local GA and coordinates with other tasks to log and transmit data.
- `write_task`: Handles SD card operations, including data logging, managing file storage and uploading experimental results.

Following initialization, each robot enters the experiment phase explained in Algorithm 1. Using a random seed the initial local population for the GA is generated. The `ga_task` then evolves the local population until 60 generations are reached without achieving a 0.01 decrease in population fitness. At which point the genome with the lowest fitness score is transmitted via ESP-NOW to remote peers. The `espnow_task` then dequeues and parses all messages from the buffer, updating statistics and evaluating remote genomes with local genomes.

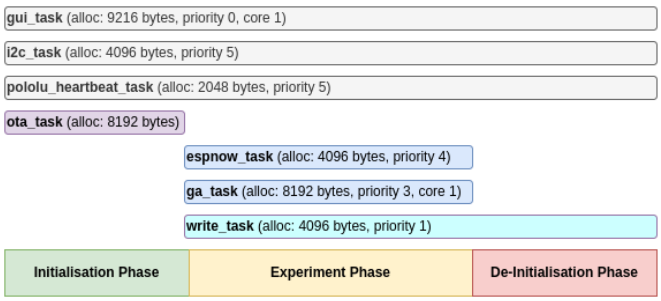

Fig. 6. Task schedule with memory allocation and priority.

Inter-task coordination is managed by event groups and queues. 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 the measurements.

Note that the local population undergoes evolution until the experiment phase is terminated. The experiment phase is terminated when one of the following criteria is met:

1) **Global solution**: Any robot attains the exact global minimum of the Rastrigin function (fitness = 0).
2) **Time limit**: A maximum duration of 60 seconds elapses without global convergence.

Upon reaching the de-initialisation phase, the `espnow_task` drains pending message queues, final fitness and communication metrics are uploaded to an Amazon S3 bucket via HTTPS via the `write_task`. Finally, the system peripherals are de-initialised to ensure a clean shutdown.

D. Communication Layer

We implemented ESP-NOW by pre-registering the MAC addresses for all peers in each agent, enabling direct unicast communication between swarm robots. Our design uses push only event-driven peer-to-peer messaging, where each device transmits data to peers without waiting for requests. Using this scheme avoids the complexity of pull-based communication protocols. Recall that once a local elite genome is found, the best solution is sent to all peers, if there are incoming messages these are queued for later processing, allowing task operations to continue uninterrupted. In this section we describe the specific communication independent variables that were manipulated during experiments.

1) Topology Inference

We investigate the impact of different communication schemes on the emergent swarm topology by introducing two distinct message transmission strategies: **STOCHASTIC** and **COMM_AWARE**. Implemented in the communication layer, these were designed to influence the order and priority with which each agent sends data to its peers. The aim is to understand how these strategies affect the connectivity and

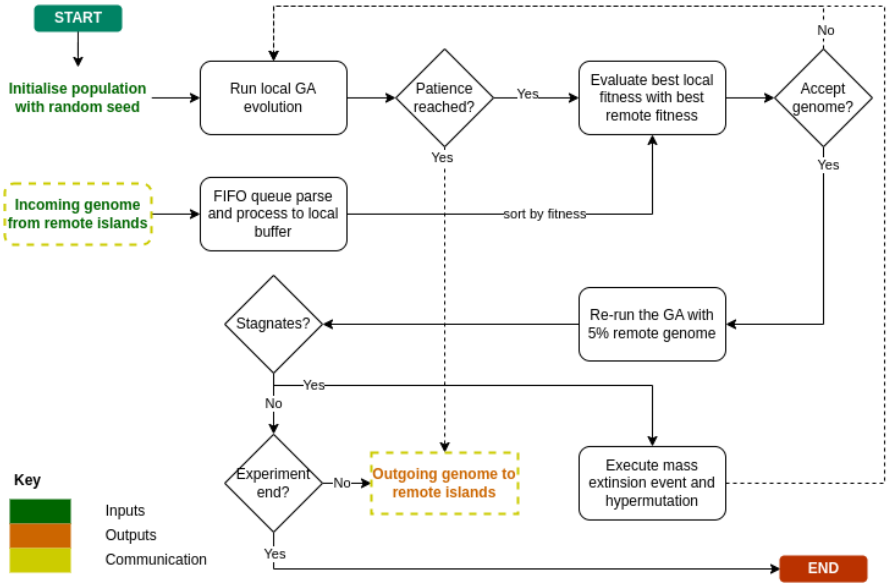

Fig. 5. Island Model Flowchart

Algorithm 1 Experiment Phase Main Loop

Generate random seed and initialise genetic algorithm population: INIT_GA(seed)
loop
    Check experimental phase: ESPNOW_COMPLETED_BIT ▷ If set, proceed to de-initialisation phase
    Run GA: EVOLVE
    Wait for GA completion flag `ga_ended` ▷ set in `ga_complete_callback`
    ESPNOW_PUSH_BEST_SOLUTION(local_genome)
    $ga\_has\_run\_before \leftarrow$ true; $s\_last\_ga\_time \leftarrow$ now_ts
    DRAIN_BUFFERED_MESSAGES ▷ Dequeue all ESP-NOW frames
    for all msg in `ga_buffer_queue` **do**
        PARSE_OUT_MESSAGE(msg) ▷ Parse raw messages
        GA_INTEGRATE_REMOTE_SOLUTION(msg.genes)
    end for
    CHECK_HYPER_MUTATION ▷ Apply if stagnation is detected
    Reset `ga_ended`
end loop

robustness of the swarm network.

These are implemented as follows:

- **STOCHASTIC**: Each agent calls a random seed to apply a Fisher-Yates shuffle over the list of peer MAC addresses to sort them before sending its message. This ensures that the order of communication is random for each transmission cycle, preventing biases.
- **COMM_AWARE**: Each agent ranks its peers based on the most recent measurements of communication quality, specifically the last known latency and Received Signal Strength Indicator (RSSI). Peers with unknown metrics (during initialisation) are prioritized first to ensure all peer links are measured. Then, peers are scored by normalizing both latency and RSSI, and those only in the worst half (highest latency, lowest RSSI) are prioritized for message transmission.

The following pseudocode outlines the logic for each scheme:

Algorithm 2 Randomized Peer Selection

Input: List of peer MAC addresses
Fisher-Yates shuffle using a random seed
for each peer in shuffled list **do**
    if peer is not self **then**
        Send message to peer
    end if
end for

Algorithm 3 Communication-Aware Peer Ranking

Input: List of peer MAC addresses, last known RSSI and latency for each peer
for each peer **do**
  if RSSI or latency is null **then**
    Assign highest priority
  else
    Normalize RSSI and latency across all peers
    Compute score: $score = norm\_latency + norm\_rssi$
  end if
end for
Sort peers: null metrics first, then by descending score (worst first), only **50% scope of network**
for each peer in sorted list **do**
  if peer is not self **then**
    Send message to peer
  end if
end for

Note that these schemes are specifically designed for **unicast** communication, where messages are sent directly to individual peers and round-trip latency can be measured via acknowledgements (ACKS). Having said that, we can think of the **STOCHASTIC** algorithm as a pseuso-broadcast communication scheme as the message is sent to all peers with negligible delays between transmissions.

2) Limited-Rate Communication

Inspired by the "less-is-more" effects reported by [2] using infrared links, we implemented a token-bucket limiter to the ESP-NOW layer and treat this quota as an independent variable. Each agent is given a small budget of 1 message in a sliding window of length 8 seconds. 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 solely spacing individual transmissions.

Algorithm 4 Token-Bucket Throttled Send

Require: B: message budget per window, W: window length (ms), t_{last}: window start time, $tokens$: remaining sends
procedure MAYBESEND($payload$)
  $now \leftarrow$ CURRENTTIMEMS
  if $now - t_{\text{last}} \geq W$ **then** ▷ Window refresh
    $tokens \leftarrow B$
    $t_{\text{last}} \leftarrow now$
  end if
  if $tokens = 0$ **then**
    return ▷ Bucket empty → no send
  end if
  if IMPROVEDFITNESS **then**
    $tokens \leftarrow tokens - 1$
    ESP_NOW_SEND($payload$)
  end if
end procedure

3) Transmission Frequency

To further explore the communication behaviour of the swarm under flooding conditions, we modulate the transmission frequency. In this communication mode, each message transmission is delayed by a random interval, otherwise no delay is explicitly applied. This random delay, drawn from a range determined by the maximum observed latency among peers, is described by Algorithm 5.

Algorithm 5 Stochastic Transmission Frequency

Input: List of peer MAC addresses, maximum latency observed (max_rand)
for each peer in the target list **do**
  if peer is not self **then**
    Compute a random delay: $delay \leftarrow rand(0, max\_rand)$
    Wait for $delay$ milliseconds
    Send message to peer
  end if
end for

In the firmware, this is implemented by checking if `DEFAULT_MIGRATION_FREQUENCY` is set to `FREQUENCY_RANDOM` and, if so, randomly delaying each call to `esp_now_send` by a value within the range of $[0, max\_rand]$, where max_rand is derived from the maximum measured latency among peers. The intention of this approach is to control the flooding of messages in a stochastic manner that can help reduce unintended collisions in the swarm network.

E. Data Logging

Reliable and precise data logging is a pre-requisite for evaluating the communication performance and evolution dynamics of the swarm. To achieve this, our firmware implements several mechanisms to capture and record key

metrics such as latency, message exchanges, internal state changes, and experiment metadata. These measurements are logged using well-defined data structures and are incrementally written to an SD card.

1) Messaging Structure

Table III summarizes the `out_message_t` structure used for transmitting messages between swarm peers via ESP-NOW. Note that any floating point values in the message content are rounded to $3d.p.$ to ensure compact representation. The total size of the struct is kept within the raw payload limits (250 bytes) imposed by ESP-NOW to guarantee reliable transmission.

TABLE III
MESSAGE STRUCTURE (`OUT_MESSAGE_T`) FOR ESP-NOW DATA TRANSFER

Field	Type	Description
`log_id`	uint32_t	Unique internal identifier of the event.
`robot_id[5]`	char	MAC identifier for the sender robot plus a null terminator.
`created_datetime`	time_t	Timestamp based on the internal RTC.
`message[128]`	char	Content of the message including the fitness score and genome delimited by "|".

2) Data Processing

The data logging pipeline is designed to ensure that all key experimental metrics are captured and preserved locally by each agent. Log entries are posted to two FreeRTOS queues: `LogQueue` for internal logging events and `LogBodyQueue` for detailed message logs. A `QueueSet` allows the dedicated `write_task` to efficiently monitor and process both queues in real time.

The core logging framework relies on three primary data structures:

- `experiment_metadata_t`: Stores overall experiment parameters, including experiment and robot IDs, random seed, GA parameters, migration settings, and application version.
- `event_log_t`: Used for logging system events such as latency measurements, RSSI, CPU usage, and state changes.
- `event_log_message_t`: Holds the parsed version of the `out_message_t` structure and links it to the internal event ID.

When an entry is retrieved from either queue, it is serialized into a JSON-formatted string (via `serialize_log_to_json()`) to ensure structured and consistent downstream analysis. The serialized data is then written incrementally to SD card files, with each file capped at 1MB to prevent memory overflow and ensure robust storage. Figure 7 illustrates the internal data logging pipeline, from event generation to SD card storage.

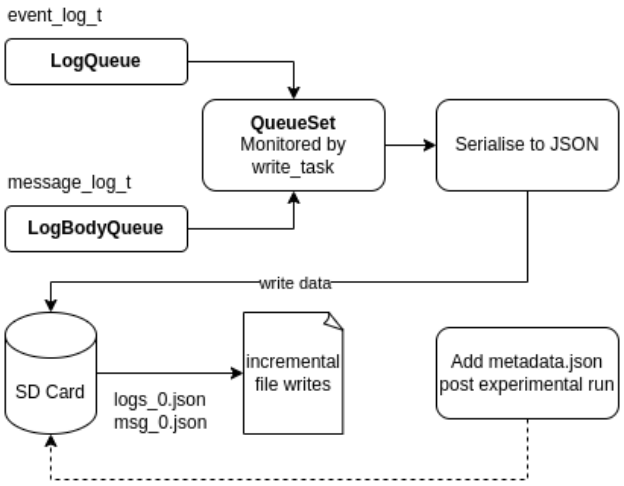

Fig. 7. Internal data logging pipeline

F. Post-experimental Processing

After the experimental phase concludes (Fig. 6), the system enters a data upload mode. In this mode, the files stored on each agent's SD cards are uploaded to an AWS S3 bucket via HTTPS. This was done by design as we did not want to stream the data from the peers whilst they were running the experiment, as this could have introduced additional collisions potentially impacting the measurements.

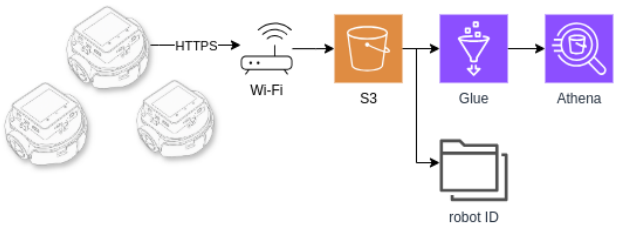

Fig. 8. Data Upload Architecture

As depicted by Figure 8, the S3 bucket is structured to store each agent's data in a separate folder, named by the robot's ID. Each file is named with a timestamp and the experiment ID, ensuring that all data is uniquely identifiable and traceable. The data is then processed using AWS Glue to prepare it for analysis in Amazon Athena. This process allowed us to automate the storage and evaluation of data logs from over 500 experiment runs.

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). Over 100 experimental runs were performed, each terminating if fitness failed to improve beyond a 0.001 threshold over 20 consecutive epochs. Across

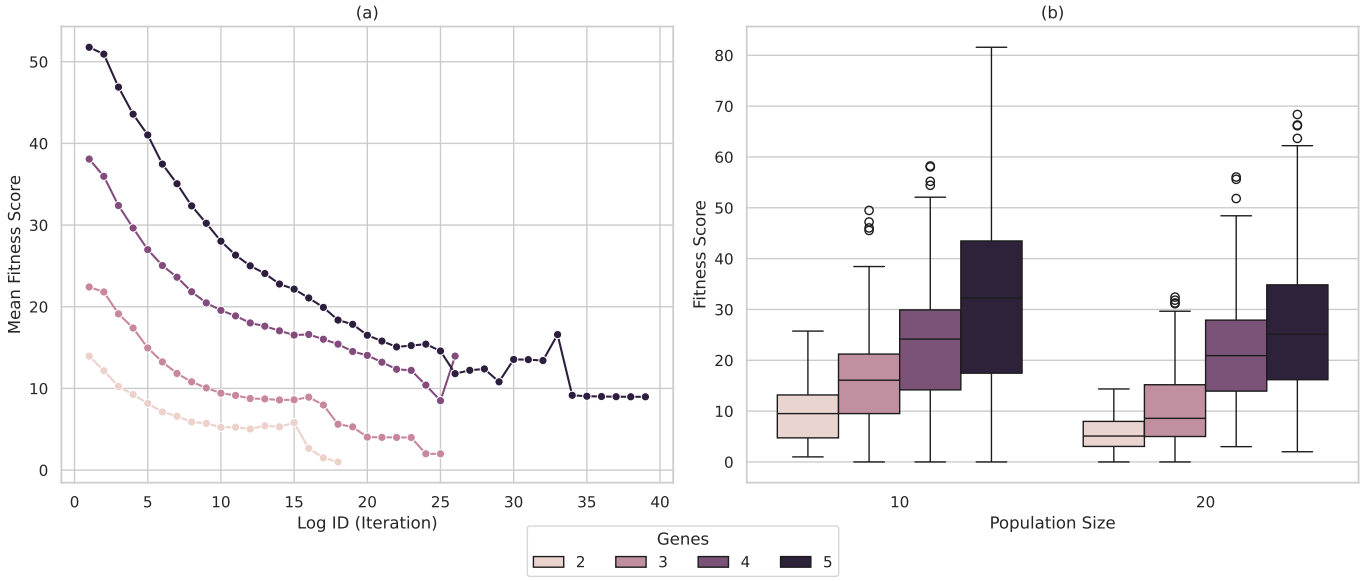

Fig. 9. a: Number of iterations for algorithm to converge to either a global or local minimum. b: Fitness distribution by population size and genes.

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 9, larger populations yielded lower final fitness.

The data gathered from these early experiments suggested that using five genes was not sufficiently challenging for a single agent as convergence was achieved too quickly. Due to this we decide to expand the Rastrigin search dimentionality to 10 genes and extend the local population and threshold epochs to 60. Hypermutation parameters are also set at this stage and detertime that the most practical experiment run time of 60 seconds is appropiate for this configuration.

VI. Base Line Results

Figure 10 shows the fitness scores and system metrics for a deployment of two agents. A total of 34 experimens 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-initiasation 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.

Using the baseline parameters shown in Table IV, we can see that none of the experiments was able to reach the global minimun 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 is aligns with a large CPU utilisation in both cores at the start of the

TABLE IV
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

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 11 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 recieving an ACK from the

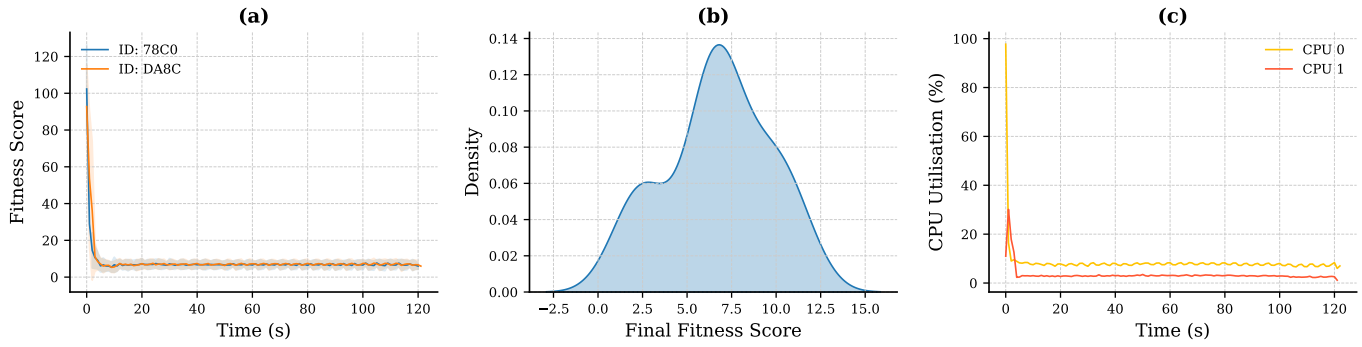

Fig. 10. a: Mean population fitness score for each agent over time, b: Final fitness score density distribution, c: Mean CPU utilisation over time.

other peer which has already de-initialised the ESPNOW task and is no longer able to respond to the messages. This would 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 V
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 11 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. HYPOTHESIS

The initial expectation is that both **STOCHASTIC** 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).

By enlarging W we progressively restrict social signalling, the following hypothesise are put forward:

should in theory alleviate message collisions that can occur during the default pseudo-broadcasting mode of the swarm. By applying this stochastic delay on a per-peer basis, the system emulates a flooding mechanism with randomised inter-transmission intervals.

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 experiments tests whether the ”less is more” phenomenon extends from low-bandwidth IR to high-bandwidth ESPNOW links.

A. Experiment Schedule

A minimum of 16 re-runs per condition was conducted to ensure statistical significance of the results.

VIII. RESULTS

IX. DISCUSSION

COMM_AWARE scheme should result in a network where communication is dynamically biased towards improving weak or unmeasured links, using latency and RSSI as

psuedo-metrics for distance allows the swarm to adapt to the percieved communication quality without having to know the exact posititon 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. This 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.

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.

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.espboards.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).

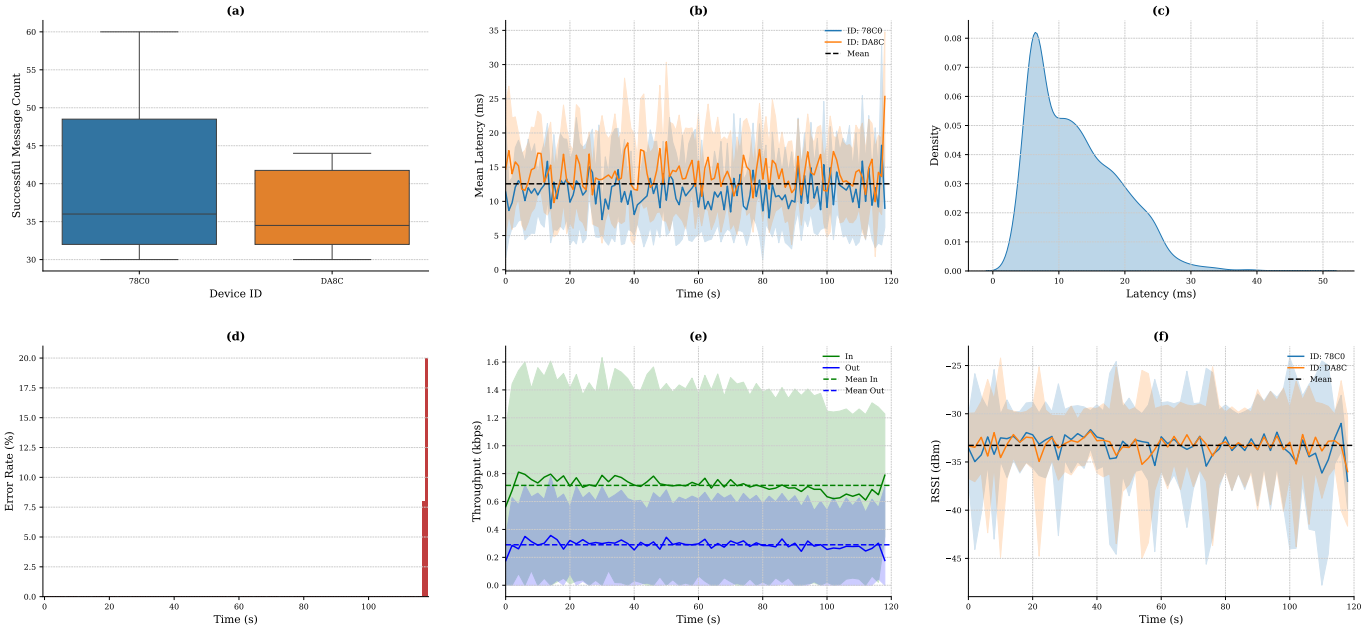

Fig. 11. a: Message sent distribution by each device, b: Mean latency by device over time, c: Latency distribution accross all experiments, d: Mean error rate from failed messages over time, e: Mean throughput In/Out over time, f: Mean RSSI by device over time.