Dissertation Title

Luis Yallico Ylquimiche

Abstract—**LOREM IPSUM**

CONTENTS

Introduction Human speech likely emerged as vital tool for coordination and the formation of social bonds among early humans [Charles Darwin's Descent of Man]. The concept, that language evolved as an adaptive trait via natural selection is widely accepted by the scientific community.

Inspired by nature's evolutionary strategies, Swarm Engineers aspire to harness the collective intelligence mechanisms evident in social insects such as ants and bees [Ref]. This dissertation examines the communication protocols of a swarm of robots, with the aim to engineer a communication framework that enables robots to efficiently share information and collaborate on tasks, mirroring the sophisticated social interactions found in natural systems.

I. BACKGROUND

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Literature Review

II. PREVIOUS STUDIES

Nulla malesuada porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero. Vivamus viverra fermentum felis. Donec nonummy pellentesque ante. Phasellus adipiscing semper elit. Proin fermentum massa ac quam. Sed diam turpis, molestie vitae, placerat a, molestie nec, leo. Maecenas lacinia. Nam ipsum ligula, eleifend at, accumsan nec, suscipit a, ipsum. Morbi blandit ligula feugiat magna. Nunc eleifend consequat lorem. Sed lacinia nulla vitae enim. Pellentesque tincidunt purus vel magna. Integer non enim. Praesent euismod nunc eu purus. Donec bibendum quam in tellus. Nullam cursus pulvinar lectus. Donec et mi. Nam vulputate metus eu enim. Vestibulum pellentesque felis eu massa.

Quisque ullamcorper placerat ipsum. Cras nibh. Morbi vel justo vitae lacus tincidunt ultrices. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. In hac habitasse platea dictumst. Integer tempus convallis augue. Etiam facilisis. Nunc elementum fermentum wisi. Aenean placerat. Ut imperdiet, enim sed gravida sollicitudin, felis odio placerat quam, ac pulvinar elit purus eget enim. Nunc vitae tortor. Proin tempus nibh sit amet nisl. Vivamus quis tortor vitae risus porta vehicula.

Implementation

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.

III. System Architecture

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 V has more details on this.
- `espnow_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 `espnow_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:     $ga\_has\_run\_before$ ← true; $s\_last\_ga\_time$ ← 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.

IV. Reasons for choices

Why use ESP-IDF over Arduino IDE in this project? [1][2]

As we are using a ESP32 microcontroller to build our swarm, which left us with two options to program it: Arduino IDE or ESP-IDF. The reasons we chose the latter are because, first, it is the official development framework for the ESP32 microcontrollers, this means that ESP-IDF is native to ESP32 whereas Arduino is an API wrap around ESP-IDF. Making ESP-IDF more stable and enabling more advanced features, specially for communication data links such as Bluetooth and Wifi. Secondly, it is more powerful and flexible than Arduino IDE, because it allows the use of FreeRTOS which allows multi core development support (our M5 Stack has two cores) and is a pre-requisite for running microROS in the ESP32 microcontroller (at the time of writing this microROS does not support Arduino), hence making it more suitable for complex projects like this one. Thirdly, it is more efficient in terms of memory and speed (as it enables parallel processing) which is important for a project that requires real-time communication between multiple devices in a swarm. Finally, it is more professional and an industry standard, it allows dependency tracking, Over the Air (OTA) updates, unit testing, enhanced debugging and comprehensive documentation around it, which means it is more likely to be supported in the future and software is less likely to become deprecated over time.

How was the WIFI implementation done?

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.

How was the IR implementation achieved?

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 100kHtz 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.

How was the ESPNOW implementation achieved?

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.

V. OTA

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.

Our project integrates Continuous Integration and Deployment (CI/CD) to maintain dynamic software development for the swarm. This ensures that features and fixes are promptly integrated and tested, maintaining software quality and allowing the swarm to consistently operate with the latest releases. This approach is crucial for testing various communication parameters within our experiments, ensuring reliable and systematic updates.

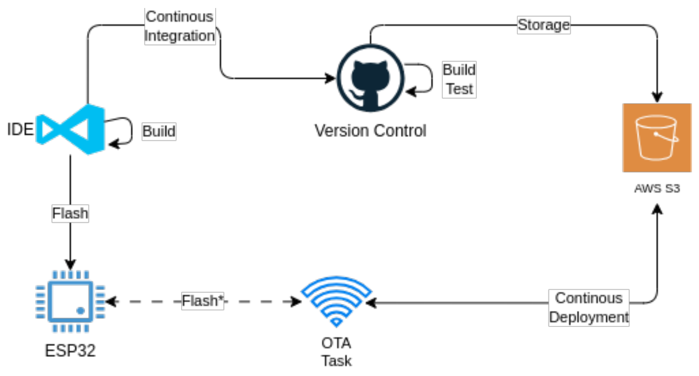

Fig. 1. System Architecture

Figure 1 shows the implementation of our system to enable over the air updates (OTA) and continuous integration & continuous deployment (CI/CD) framework over the swarm.

- **Local Development 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.
- **Version Control System & CI/CD**: The codebase is stored in a public repository on GitHub: 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.
- **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.
- **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).

VI. Data Capture

We implemented local data logging on the M5Stack by integrating a 16GB SD card peripheral, using the SPI bus (shared by the LCD) set at a frequency of 20MHz and utilizing 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. Local logging minimizes data transmission loss, supporting the swarm's objective of avoiding any single point of failure.

Upon completing the experimental tasks, the swarm members switch to a data upload mode, where the locally collected data is securely transmitted to an S3 bucket using HTTPS. This might seem counter intuitive to our original decentralized 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 scenario, 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, this enables efficient memory management and preventing heap memory overflow during log file writes.

VII. 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 2, 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.

VIII. Base Line Results

Figure 3 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 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 I
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 I, 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 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 is doubled to 60 to ensure diversity in the gene pool, with the trade off of slower convergence.

Figure 4 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 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

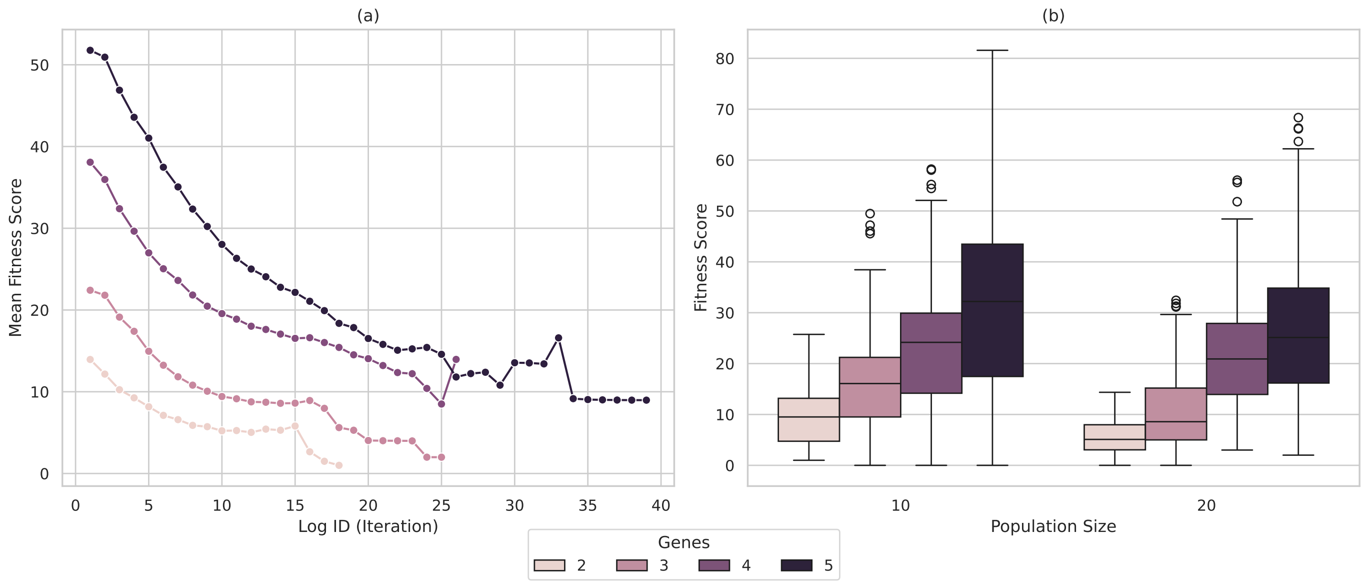

Fig. 2. a: Number of iterations for algorithm to converge to either a global or local minimum. b: Fitness distribution by population size and genes.

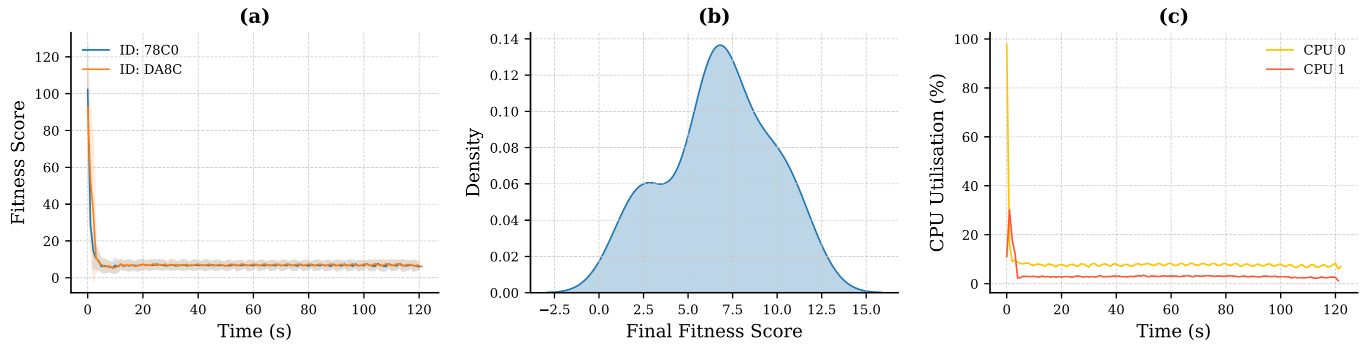

Fig. 3. a: Mean population fitness score for each agent over time, b: Final fitness score density distribution, c: Mean CPU utilisation over time.

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 II
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 4 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.

IX. 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**. 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 psuedo-metrics for distance allows the swarm to adapt to the percieved communication quality without having to know the exact positition 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.

References

[1] 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).

[2] Expressif Expressif. *FreeRTOS Overview - ESP32 - — ESP-IDF Programming Guide latest documentation*. URL: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/system/freertos.html (visited on 03/04/2024).

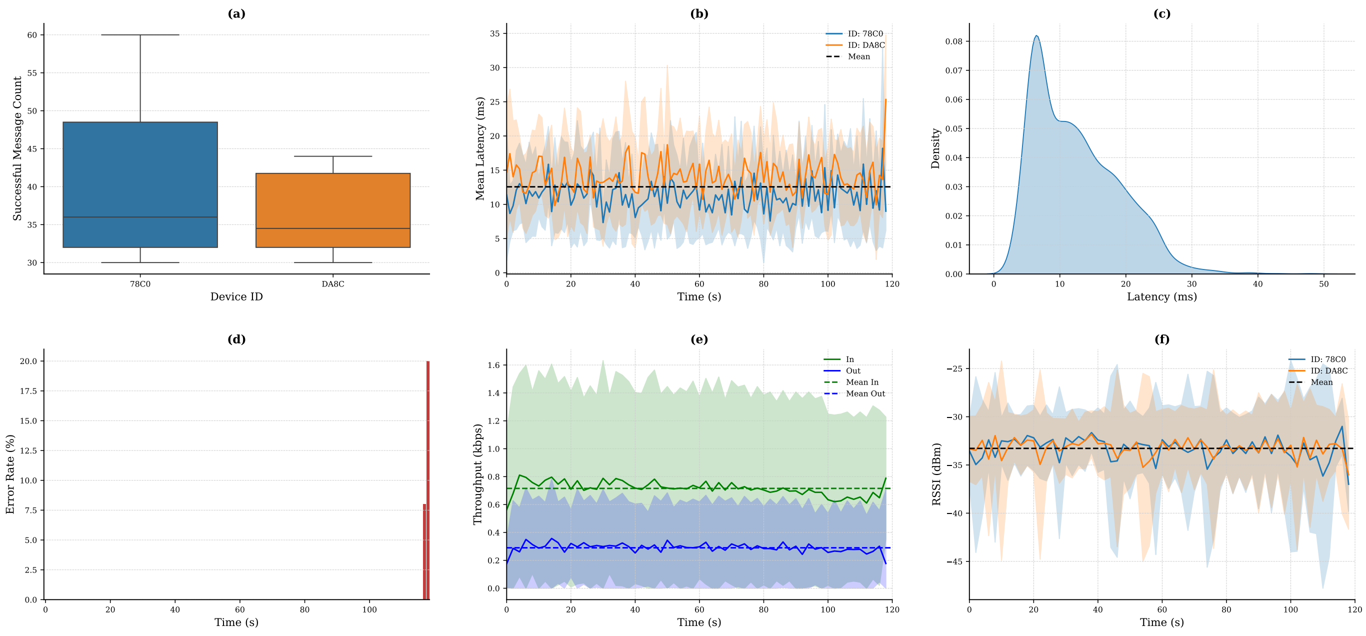

Fig. 4. a: Message sent distribution by each device, b: Mean latency by device over time, c: Latency distribution accross all experiments, d: Mean error rate from failed messages over time, e: Mean throughput In/Out over time, f: Mean RSSI by device over time.