Dissertation Title

Luis Yallico Ylquimiche

June 18, 2025

Abstract

LOREM IPSUM

Contents

Chapter 1

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.

1.1 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.

Chapter 2

Literature Review

2.1 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.

Chapter 3

Implementation

3.1 System Architecture

Each swarm member boots into a FreeRTOS-based runtime by calling `app_main()`, which performs staged initialization of hardware and software subsystems: non-volatile storage (NVS), I2C peripherals (IR board, LVGL display), Arduino core, RTC, SD-card manager, Wi-Fi station mode, and ESPNOW. During init, the main task spawns several FreeRTOS tasks—`gui_task`, `i2c_lvgl_task`, `espnow_task`, `ga_task`, `write_task` (logging), and an optional `ota_task`. Inter-task coordination is managed by event groups (`s_wifi_event_group`, `s_espnow_event_group`, `ota_event_group`, `ga_event_group`) and queues (`s_example_espnow_queue`, `ga_buffer_queue`, `LogQueue`).

Once initialized, each robot enters the experiment loop shown in Algorithm 1. The GA task evolves a local population until `ga_ended` is set by the completion callback, at which point the best solution is packaged and sent via ESPNOW unicast (`espnow_push_best_solution`). 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 passing remote genomes to `ga_integrate_remote_solution`.

Each swarm member is tasked with solving the Rastrigin function [ref], a non-convex optimization problem that is commonly used to benchmark optimization algorithms due to its large number of local minima. In our implementation, we use genetic algorithms to solve the function and find the global minimum. At initialisation each robot is assigned a random starting point (seed) within the 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 (the Rastrigin function) and genetic operators (mutation and crossover) to evolve the population.

Algorithm 1 Swarm Member Experiment Loop

Init peripherals, Wi-Fi, ESPNOW, tasks, event groups, queues
Fetch quantum seed via HTTPS; INIT_GA(seed)
loop
    Run GA: EVOLVE
    Wait until `ga_ended` ▷ set in `ga_complete_callback`
    ESPNOW_PUSH_BEST_SOLUTION(local_best)
    $ga\_has\_run\_before \leftarrow$ true; $s\_last\_ga\_time \leftarrow$ now
    DRAIN_BUFFERED_MESSAGES ▷ Dequeue all ESPNOW frames
    for all msg in `ga_buffer_queue` **do**
        GA_INTEGRATE_REMOTE_SOLUTION(msg.genes)
    end for
    CHECK_HYPER_MUTATION ▷ Adjust mutation if stagnated
    Reset `ga_ended`
end loop

Once the robots execute the experiment task, the swarm begins to communicate their current best fitness and corresponding genes found to their peers. This is akin to the "Island Model" [ref], in which each subpopulation (islands) evolve in parallel and occasionally exchange individuals (migration) with other islands, thus enabling the swarm to converge on the optimal solution. In our implementation we employ an immigrant-K strategy: after each GA epoch the top K solutions dequeued from `ga_buffer_queue` are integrated into the local population by replacing the worst-performing K individuals, thus preserving high-quality diversity. To counteract stagnation, a hyper-mutation mechanism tracks consecutive non-improving generations—once a threshold is reached, the mutation probability is temporarily increased to escape local optima and promote renewed exploration across the swarm.

The swarm is considered to have successfully solved the problem when they either (i) find the global minimum by at least one robot or (ii) converge (e.g., all robots reaching similar fitness levels) to a local minimum as a group.

3.1.1 Termination Conditions

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.

3.2 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.

3.3 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.

Figure 3.1 shows the implementation of our system to enable over the air

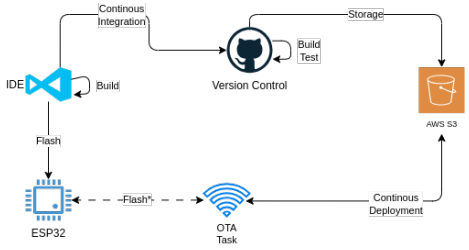

Figure 3.1: System Architecture

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).

3.4 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.

3.5 Preliminary Sensitivity Analysis

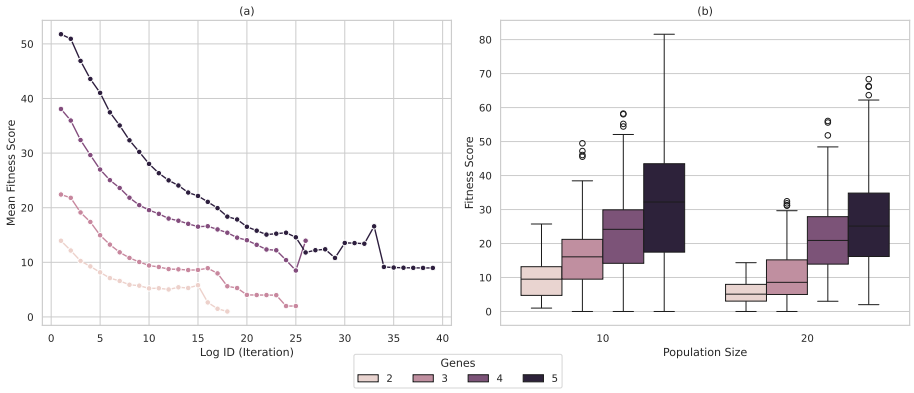

Figure 3.2: a: Number of iterations for algorithm to converge to either a global or local minimum. b: Fitness distribution by population size and genes.

Using a single robot, a preliminary analysis was conducted to identify suitable parameters for solving the Rastrigin function under varying population sizes and gene dimensions. A total of **136 experiments** were performed, each terminating if fitness failed to improve beyond a **0.001 threshold** over **20 consecutive epochs** (patience). Only iterations exceeding this improvement threshold were logged. If the global optimum (fitness=0) was not reached, the run was considered converged to a local minimum. Across all runs, convergence times ranged **from 0 to 3 seconds**, reflecting the early stopping triggered by the patience setting. As shown in Figure 3.2, 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 permit convergence for a five-dimensional Rastrigin function.

Bibliography

[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).