Software Update Propagation in Heterogeneous UAV Swarms: Hierarchical Protocols and Learned Coordination Policies

by

Lin Geng

A thesis submitted to the Graduate Program in Computing in conformity with the requirements for the Degree of Master of Science

Queen's University

Kingston, Ontario, Canada

September, 2026

Copyright © Lin Geng, 2026

Abstract

Heterogeneous unmanned aerial vehicle (UAV) swarms consist of dozens to hundreds of drones with different roles and varying hardware and software requirements collaborating towards a shared mission. While traditional approaches for synchronized software updates assume swarms to be unstructured and homogeneous, the heterogeneous nature of modern swarms and the emerging need of drones to update their deep learning (DL) perception models as missions unfold has made efficient software update propagation crucial for swarms to adapt to dynamic environments.

This thesis investigates software update propagation in heterogeneous UAV swarms through two complementary coordination philosophies: deterministic hierarchical protocols and learned decentralized policies. We introduce three contributions. SwarmSync is a hierarchical update synchronization strategy that distributes software updates to the correct subset of drones within a swarm using ACK-guaranteed delivery. SwarmModelPatch is a DL model patching method that reduces update payload size by selectively freezing model layers during fine-tuning, transmitting only the changed parameters. SwarmMARL is a Multi-Agent Proximal Policy Optimization (MAPPO) controller that trains agents to autonomously decide when to broadcast, request, or remain idle during dissemination, without pre-defined roles or explicit protocol rules.

We evaluate all three contributions through large-scale simulations in the ARGoS swarm simulator. SwarmSync achieves up to 78.3% faster convergence than SOUL and 47.7% faster than Gossip while maintaining reasonable overhead. SwarmModelPatch reduces update payload size by up to 99.6% across model architectures, with accuracy trade-offs of up to 15.9% depending on the degree of layer freezing. Swar-

mMARL achieves faster convergence than SwarmSync in most tested configurations through multi-source dissemination; SwarmSync becomes competitive or preferable primarily as swarm size grows, with SwarmMARL's advantage most pronounced at small-to-medium swarm sizes and SwarmSync's formal delivery guarantees and lower transmission overhead becoming the more attractive trade-off at the largest swarm size tested.

Both coordination approaches are additionally implemented and functionally validated on physical hardware in a controlled environment, confirming that the update workflow operate correctly beyond simulation; due to the highly controlled setting, the hardware experiments serve as a proof-of-concept validation rather than a reproduction of the simulation's quantitative performance comparisons.

Co-authorship and Statement of Collaboration

This work contains material that has been developed in collaboration with other researchers; the contributions of each researcher are listed below.

- **Allen (Lin) Geng:** Led the design, implementation, and analysis of the research.
- **Hao Li:** Contributed to the review of the research and provided guidance and feedback throughout the project.
- **Bram Adams:** Co-supervisor of this research, providing expertise in software engineering aspects.
- **Sidney Givigi:** Co-supervisor of this research, providing expertise in robotics and hardware, as well as access to necessary equipment.

Disclosure of Sources of Funding and Conflicts of Interest

This work is supported by the funding from Queen's University.

There are no conflicts of interest with respect to this research.

Statement Of Originality and Disclosure of Use of Artificial Intelligence

The work presented in this thesis is my own, and I certify that its intellectual content is the result of my independent research. All references and contributions of others have been appropriately cited.

In the preparation of this thesis, Artificial intelligence tools were used for grammar, writing clarity, and LaTeX formatting. During software development, AI-assisted tools were used for debugging and formatting only. The design of research, experiments, and analysis are by the author and not assisted with AI.

Acknowledgments

We extend our gratitude to Vivek Shankar Varadharajan for his invaluable support in providing the initial guidelines and source code for [112, 114] that contributed to this work.

Table of Contents

Abstract ii

Co-authorship and Statement of Collaboration iii

Disclosure of Sources of Funding and Conflicts of Interest iv

Statement Of Originality and Disclosure of Use of Artificial Intelligence v

Acknowledgments vi

Table of Contents vii

List of Figures xiii

List of Tables xvii

Chapter 1 Introduction 1

1.1 Introduction 1

1.2 Motivational Example 4

1.3 Thesis Objectives 8

1.4 Thesis Structure 12

Chapter 2 Background and Related Work **14**

2.1 Background 14

2.1.1 Heterogeneous UAV Swarm Systems 14

2.1.2 Software Update Dissemination 18

2.1.3 Deep Learning for UAV Perception 20

2.1.4 Model Compression and Adaptation Terminology 22

2.1.5 Multi-Agent Reinforcement Learning 24

2.1.6 Statistical Methods for Comparative Evaluation 29

2.1.7 Hardware and Middleware for Real-World Validation 31

2.2 Related Work 32

2.2.1 Software Update Strategies for Swarms 33

2.2.2 Deep Learning Model Updating 35

2.2.3 Multi-Agent Reinforcement Learning for Swarm Coordination 36

2.2.4 Distributed and Federated Learning for Edge and Swarm Systems 38

2.2.5 Incremental Learning Beyond Layer Freezing 40

2.2.6 Swarm Communication Protocols 41

2.2.7 Software Engineering for UAV Systems 42

Chapter 3 Problem Formalization and System Model **44**

3.1 System Model 45

3.2 Simulation Environment 46

3.2.1 Swarm Configuration 47

3.2.2 Communication Range and Transmission Speed 48

3.3 Problem Definition, Convergence, and Evaluation 49

3.3.1 Problem Definition 49
3.3.2 Update Convergence 50
3.3.3 Assumptions 51
3.3.4 Evaluation Metrics 52

Chapter 4 SwarmUpdate: Hierarchical Deterministic Coordination 55

4.1 SwarmSync 56
4.2 Formal Analysis of SwarmSync 65
4.2.1 Scalability of SwarmSync 65
4.2.2 Almost-sure Termination of SwarmSync 67
4.2.3 Failure Resilience 70
4.3 SwarmModelPatch 70
4.3.1 Formal Analysis: Layer Freezing 74
4.4 Illustrative Example: End-to-End SwarmUpdate Workflow 75
4.5 Empirical Evaluation 77
4.5.1 Baseline Strategies 78
4.5.2 Experimental Design 83
4.6 SwarmUpdate Results 85
4.6.1 Dissemination Efficiency of SwarmSync 86
4.6.2 Transmission Overhead of SwarmSync 91
4.6.3 Model Patch Size and Accuracy Trade-off 95

Chapter 5 SwarmMARL: Learned Decentralized Coordination 101

5.1 SwarmMARL Design 103

5.1.1 Observation Space 104
5.1.2 Action Space 108
5.1.3 Reward Function 109
5.1.4 Actor-Critic Architecture 114
5.2 ARGoS Integration 117
5.2.1 State-Action Protocol 118
5.2.2 Episode Boundaries 120
5.2.3 Timing and Synchronization 120
5.3 Training Procedure 121
5.3.1 Rollout Collection 121
5.3.2 Advantage Normalization 122
5.3.3 PPO Update 123
5.3.4 Checkpointing and Logging 123
5.4 Illustrative Example: SwarmMARL Coordination in the Wildfire Scenario 124
5.5 MARL Results 127
5.5.1 Convergence Time 127
5.5.2 Transmission Overhead 129
5.5.3 Training Convergence 132
5.5.4 Discussion of Standalone Results 133

Chapter 6 Unified Evaluation and Comparison 136

6.1 Comparison of SwarmSync and SwarmMARL 136
6.1.1 Convergence Time 136

6.1.2 Transmission Overhead 137
6.1.3 Statistical Analysis 138
6.1.4 Practical Deployment Considerations 142

Chapter 7 Real-World Validation **146**

7.1 Hardware and Software Setup 147
7.1.1 Hardware Platform 147
7.1.2 Software Stack 148
7.1.3 Topology Configuration 149
7.2 Real-World Experiment Objectives 150
7.3 SwarmSync Real-World Experiments 151
7.3.1 Experimental Protocol 151
7.3.2 Results 152
7.4 SwarmMARL Real-World Experiments 154
7.4.1 Adaptations for Physical Deployment 154
7.4.2 Experimental Protocol 155
7.4.3 Results 156
7.5 Hardware Constraints 157
7.6 Deployment Lessons 159

Chapter 8 Discussion and Limitations **161**

8.1 Deterministic vs. Learned Coordination 162
8.2 Implications of SwarmSync and SwarmModelPatch 164
8.3 Implications of SwarmMARL 167

8.4 Real-World Implementation and Validation 170
8.5 Cross-Cutting Implications 171
8.5.1 Energy Consumption 171
8.5.2 Environmental Obstacles 172
8.5.3 Generalization to Other Simulators 174
8.5.4 Dynamic Transfer Rates 174
8.5.5 Security Considerations for Heterogeneous Swarms 176
8.5.6 Timeout Policy Selection 177
8.6 Limitations and Scope 178
8.6.1 Simulation Fidelity 178
8.6.2 SwarmMARL Scope 179
8.6.3 SwarmSync and SwarmModelPatch Scope 182
8.6.4 Shared Constraints 184

Chapter 9 Conclusion **186**

Bibliography 189

List of Figures

1.1 A possible situation where an OTA P2P software update is needed. An unusual activity is detected in the mountains, so a reporter UAV is sent to HQ to report, carrying information about the current situation. Later an Updater UAV is sent back to update the swarm with the newer model. Fire and smoke will be correctly classified as a result of this update. 6

3.1 18 drones surrounding the Updater, derived from a hexagonal packing within ARGoS's default communication range of $r = 3$ meters. 48

4.1 The SwarmUpdate pipeline. When an update is needed, SwarmModelPatch generates a minimal differential patch if a DL model update is required. SwarmSync then dispatches an Updater UAV to carry the patch to the swarm and propagate it to all compatible agents. 56

4.2 Sequence diagram of the SwarmSync process. 58

4.3 Structure of a typical CNN model. 71

4.4 The model has its earlier layers frozen (weights unchanged), its later convolutional layers unfrozen and updated during training, and its final classification layer reinitialized with random weights, as opposed to the unfrozen layers, which retain and build on their existing weights. . . . 72
4.5 Swarm UAV formations and roles for SwarmSync and the two baseline software update synchronization strategies. 78
4.6 The average time taken **per drone** for each synchronization strategy to converge the swarm. 86
4.7 The **total** time taken for each synchronization strategy until convergence of the swarm 87
4.8 The average overhead **per drone** for each synchronization strategy until convergence of the swarm 90
4.9 The **total** overhead for each synchronization strategy until convergence of the swarm 91
4.10 The average number of control steps for swarm convergence across varying patch sizes. 96
4.11 The average overhead for swarm convergence across varying patch sizes. 98
4.12 The model accuracy vs size reduction for the two datasets, the dashed line represents the model's performance before learning the new class. 98

5.1 SwarmMARL convergence time (control steps) across swarm sizes $s \in \{20, 100, 200, 500\}$ and failure rates $f \in \{0, 0.25, 0.50, 0.75\}$, compared against SwarmSync and Gossip. 128

5.2 SwarmMARL transmission overhead (MB) across swarm sizes $s \in \{20, 100, 200, 500\}$ and failure rates $f \in \{0, 0.25, 0.50, 0.75\}$, compared against SwarmSync and Gossip. 130

5.3 SwarmMARL training loss curves over 58,350 gradient steps. Faint lines show raw per-step values; solid lines show a rolling average. Value loss and total loss decline sharply in the first 10,000 steps as the critic learns to predict returns, then stabilise near zero. Entropy decays monotonically from 1.09 to near zero, consistent with the annealing schedule, indicating the policy transitions from near-uniform exploration to confident action selection. Actor loss remains close to zero throughout, reflecting PPO's clipping constraint preventing large policy updates. The stable plateau after approximately step 20,000 indicates policy convergence. 132

5.4 SwarmMARL convergence time over training episodes, showing the number of control steps required for all agents to reach the target software version within each episode. Episodes that timed out without full convergence are marked with red indicators. 133

7.1 Real-world system architecture. The ground server runs the RadioOrchestrator node for SwarmSync experiments and the MAPPO inference process for SwarmMARL experiments. CrazySwarm2 bridges ROS 2 topics to the Crazyradio 2.0 RF link. The Vicon motion capture system provides pose estimates used by the Crazyflie onboard EKF during hover sequences. Two drones (C26, C25) are configured with `flies=true` and execute a hover sequence as a physical convergence signal; the remaining four drones stay grounded throughout the update. 149

7.2 Real-world experiment. Four stages of a single update run on six Crazyflie 2.1 Brushless drones in a controlled laboratory environment. 153

8.1 ARGoS vs. Gazebo results for 20 UAVs under failure rates $f \in \{0, 0.25, 0.50, 0.75\}$. 175

List of Tables

4.1 Mann-Whitney U test results for SwarmSync vs SOUL and SwarmSync vs Gossip convergence speed across swarm sizes and failure rates. Effect size r is the rank-biserial correlation; negative values indicate the competing algorithm converges faster, positive values indicate SwarmSync converges faster. Non significant($p \geq alpha$) entries are highlighted in bold. 89

4.2 Mann-Whitney U test results for SwarmSync vs SOUL and SwarmSync vs Gossip across swarm sizes and failure rates. Effect size r is the rank-biserial correlation; negative values indicate the competing algorithm causes less overhead, positive values indicate SwarmSync causes less overhead. Non significant($p \geq alpha$) entries are highlighted in bold. . 94

5.1 SwarmMARL hyperparameters. 118

6.1 Mann-Whitney U test results for SwarmSync vs SwarmMARL across swarm sizes and failure rates. Effect size r is the rank-biserial correlation; negative values indicate SwarmMARL converges faster, positive values indicate SwarmSync converges faster. (entries where it is statistically non-significant is highlighted in bold) 140

6.2 Mann-Whitney U test results for SwarmSync vs SwarmMARL across swarm sizes and failure rates for communication overhead. Effect size r is the rank-biserial correlation; negative values indicate SwarmMARL has lower overhead, positive values indicate SwarmSync has lower overhead. 142

6.3 Summary comparison of SwarmSync and SwarmMARL across key evaluation dimensions. 144

6.4 Decision guide for choosing between **SwarmSync** and **SwarmMARL** based on deployment scenario. 145

8.1 Estimated energy consumption per UAV at 200 UAVs with failure rate $f = 0.5$ and a full update payload (≈3 MB). 172

9.1 Summary of thesis contributions, mapped to their corresponding objective and chapter. 190

Chapter 1

Introduction

1.1 Introduction

Distributed autonomous systems increasingly operate in environments where continuous software maintenance is not just desirable but operationally critical. As these systems grow in scale and complexity, the ability to propagate software updates reliably and efficiently across all active agents — without interrupting ongoing missions or relying on centralized infrastructure — has become a fundamental software engineering challenge [14, 4]. Autonomous unmanned aerial vehicle (UAV) swarms represent one of the most demanding instantiations of this challenge: they operate in remote, resource-constrained environments, communicate over unreliable peer-to-peer links, and must remain operational during the update process itself [104].

In the last decade, research has increasingly focused on heterogeneous UAV swarms that comprise multiple types of UAVs, each designed with distinct roles and capa-

bilities [84]. In practice, heterogeneous fleets typically comprise a small number of distinct agent types, commonly between two and five, each corresponding to a specific hardware capability or mission role (e.g., a vision-capable UAV for perception and a lightweight UAV for navigation or relay), rather than a large number of highly specialized types. Compared to homogeneous swarms (i.e., identical UAVs with uniform roles and update requirements), heterogeneous UAV swarms offer enhanced scalability and performance, capable of addressing complex and diverse task requirements [113, 115, 25, 48]. In mission-critical situations, such as search and rescue, hazard detection, or remote surveillance, timely software updates are essential. Delays in disseminating new objectives or perception models can lead to costly time losses and, in extreme cases, endanger human safety.

Beyond task-level updates, UAV swarms increasingly rely on deep learning (DL)-based perception to interpret their environment. These perception models support core functions such as object detection (e.g., identifying survivors, vehicles, or hazards) and scene segmentation (e.g., mapping fire spread or flooded areas). As missions unfold, these models often need to be adapted to new data distributions or unforeseen scenarios, such as smoke in wildfires or debris in disaster zones [96, 36]. Without rapid DL model updates, UAVs risk reduced accuracy and poor decision-making in the field. Because retraining and redistributing an entire model is often prohibitively expensive over constrained swarm links, this thesis considers patching as transmitting/receiving the set of payloads, rather than transmitting a full updated model. The set of parameter-wise weight differences produced during fine-tuning, is sent. Each receiving UAV applies this payload to its existing model through element-wise

addition, updating the model's weights in place and producing a model that behaves as though it had been fully retrained, without requiring the full set of parameters to be transmitted.

Over-the-air (OTA) software updates [112, 4] provide a natural solution for keeping both task logic and perception models current in the field, as they are designed to function under unreliable peer-to-peer communication and are more efficient than manual update methods. Prior work on OTA programming techniques [14, 112] has shown that updates can be disseminated across homogeneous swarms within seconds by rebroadcasting once a single UAV receives the update. However, existing OTA strategies were designed for homogeneous swarms and do not transfer cleanly to heterogeneous settings. Three challenges in particular remain unaddressed:

- **Update relevance divergence:** Not all agents require the same update (e.g., only vision-capable UAVs need model reconfiguration under weather shifts), which breaks the uniform dissemination assumption of homogeneous protocols.
- **Coordination complexity:** Updates may involve multiple interdependent payloads (e.g., a DL model for vision-capable drones and a task logic update for navigation drones) that must be consistently applied across roles to maintain system coherence.
- **Behavioral adaptability:** Fixed, rule-based dissemination protocols cannot adapt their coordination behavior to dynamic conditions such as battery depletion, partial connectivity, or in-progress transfers, potentially wasting bandwidth on misdirected or redundant transmissions.

These challenges call for update propagation strategies that can handle role-based payload targeting, guarantee reliable delivery under packet loss, minimize transmission overhead, and reduce model update payload size, all without requiring centralized infrastructure or interrupting the swarm's mission. Two fundamentally different philosophies exist for meeting these requirements. The first is a *deterministic, protocol-based* approach: define explicit roles, a structured communication hierarchy, and guaranteed delivery semantics so that convergence is provably bounded regardless of individual agent behavior. The second is a *learned, policy-based* approach: train agents to autonomously decide when to broadcast, request, or remain idle based on their local context, allowing coordination behavior to emerge from experience rather than explicit programming. Each philosophy carries distinct trade-offs in terms of reliability, adaptability, and deployability that are not well understood in the context of heterogeneous UAV swarms.

1.2 Motivational Example

To illustrate the potential impact and importance of this thesis, consider the following real-world scenario described in Fig. 1.1. A common task for a surveillance UAV swarm might involve patrolling remote mountainous areas [46, 7]. A common constraint imposed by the geographical environment is that remote UAVs are usually unable to communicate directly with the headquarters [133]. Instead, the swarm communicates with the headquarters at set intervals by dispatching a lightweight "reporter" drone to relay status updates such as altitude, remaining battery life, and

environmental readings.

Now consider such a swarm during a routine patrol when an anomaly is detected. UAV sensors record unusual thermal signatures near the edge of the patrol sector. The swarm relays the anomaly data to headquarters via a reporter drone. Upon analysis, the headquarters identifies a nascent forest fire igniting in the main flight corridor of the UAVs. Early intervention is critical to contain the damage and ensure safety. In this scenario, the UAV swarm could potentially assist in real-time fire mapping and early suppression efforts, but only if its on-board models can be updated for the new conditions and monitoring task. The headquarters in this case decides to adapt the swarm's mission to the tasks of monitoring and cordoning off the area, a common procedure to follow [95].

Why is an immediate task and model update so important? The emergence of thick smoke plumes and intense heat fundamentally changes the environment for which the drones were trained. The current deep learning models of the swarm, originally tuned for ordinary patrol conditions, may misinterpret or misclassify these novel inputs. Researchers note that small smoke plumes are easily confused with clouds or haze due to similar colors and shapes in aerial imagery, and that standard vision models struggle to detect smoke without specialized training [124]. This increases the risk of a UAV inadvertently entering a dangerous area, potentially causing severe damage or even catastrophic failure. Likewise, navigation models that do not account for turbulence and updrafts caused by fire could struggle to maintain stable flight. Essentially, the wildfire introduces a significant concept drift [36]: the data distribution no longer matches the swarm's previous training. If not addressed, this

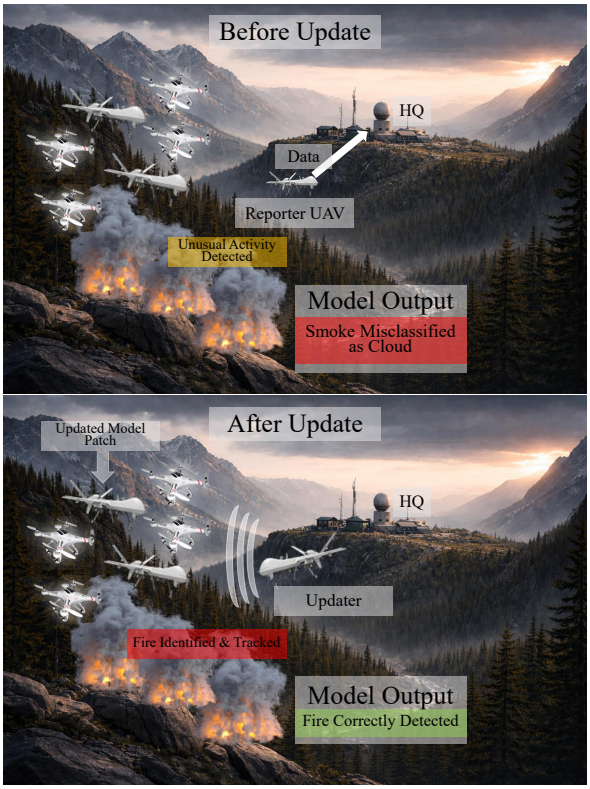


Figure 1.1: A possible situation where an OTA P2P software update is needed. An unusual activity is detected in the mountains, so a reporter UAV is sent to HQ to report, carrying information about the current situation. Later an Updater UAV is sent back to update the swarm with the newer model. Fire and smoke will be correctly classified as a result of this update.

concept drift can severely degrade model accuracy. Furthermore, swarm detection and path-planning algorithms must be updated to prioritize and investigate emerging fires to guarantee comprehensive coverage of the affected area [111].

From a technical feasibility standpoint, modern UAV platforms are increasingly capable of sustained field operation while running computationally demanding workloads such as DL-based perception models. Advances in energy-dense batteries and lightweight embedded GPUs (e.g., NVIDIA Jetson and Qualcomm Flight) confirm

that long-duration deployment with on-board deep learning is not only possible but increasingly practical, making over-the-air model updates a realistic operational requirement for heterogeneous UAV swarms.

One alternative would be to recall every drone to the base for a full firmware or model update, but doing so would waste precious time during a critical window. Traditional update workflows often assume that the system can be taken offline or that all units are within communication range of a base station. However, in an evolving crisis like a wildfire, these assumptions break down. Varadharajan et al. [112] point out that conventional swarm update methods requiring a central node and close-range broadcast have "critical disadvantages" because they need pre-deployed infrastructure or require the swarm to regroup within range, conditions that are "difficult and sometimes even impossible to achieve in real-world missions such as [. . .] off-grid emergency response."

This scenario illustrates the two complementary research directions this thesis pursues. The first is how to *structure* update dissemination reliably across a heterogeneous swarm: SwarmSync ensures every compatible drone receives the full update through hierarchical, ACK-guaranteed delivery, while SwarmModelPatch ensures the payload it carries is as small as possible by transmitting only the model parameters that changed. The second is how to *learn* effective coordination behavior that adapts to the swarm's dynamic state during dissemination: SwarmMARL agents autonomously decide when to broadcast or request the update based on local context, suppressing unnecessary transmissions from drones with critically low battery or already-completed transfers. Together, these approaches address the full spectrum

of design choices available to a practitioner deploying software update propagation in heterogeneous UAV swarms, from formal protocol guarantees to adaptive learned behavior, and from large-scale simulation to physical validation on real hardware.

1.3 Thesis Objectives

This thesis investigates software update propagation in heterogeneous UAV swarms through the lens of both coordination philosophies. Our overarching hypothesis is that neither deterministic nor learned coordination dominates the other across all operating conditions: each is preferable under identifiable swarm sizes, failure rates, and resource constraints, and both are capable of transferring from simulation to physical deployment. Concretely, we pursue **four** objectives, the first three of which establish and characterize each coordination philosophy individually, and the fourth of which tests this hypothesis directly through head-to-head comparison and real-world validation:

Objective 1: Efficient and reliable dissemination. Design and evaluate a deterministic hierarchical update protocol for heterogeneous UAV swarms that targets updates to compatible agents, guarantees delivery under packet loss, and scales efficiently with swarm size. We seek to establish whether role-based sub-swarm organization can outperform existing gossip-based and auction-based baselines in terms of convergence time across a range of swarm sizes and failure rates, shown in Chapter 4.

Objective 2: Minimal model update payload. Design and evaluate a DL model

patching method that reduces the size of an update payload by transmitting only the parameters that changed during fine-tuning, and characterize the controllable trade-off between patch size, convergence time, and model accuracy that this introduces for swarm operators, shown in Chapter 4.

Objective 3: Learned coordination behavior. Design and evaluate a multi-agent reinforcement learning controller that trains UAV agents to make context-sensitive broadcast and request decisions during update dissemination, without relying on pre-defined roles or explicit protocol rules. We seek to establish whether a learned policy can reduce redundant transmissions and adapt to resource constraints such as low battery that a fixed protocol cannot avoid, shown in Chapter 5.

Objective 4: Comparative understanding and real-world validation. Directly compare the deterministic and learned coordination approaches across shared experimental conditions to identify the conditions under which each is preferable, and validate both approaches on physical UAV hardware in a controlled laboratory environment to assess the gap between simulation and deployment, shown in Chapter 6.

Proposed Approaches

To pursue these objectives, this thesis proposes, evaluates and compares two complementary systems.

SwarmUpdate addresses Objectives 1 and 2 through two components, which

consists of **SwarmSync** and **SwarmModelPatch**. **SwarmSync** is a hierarchical software update synchronization strategy that organizes UAVs into role-based sub-swarms with designated leaders, using ACK-based reliable delivery to guarantee that every compatible UAV receives every update packet. **SwarmModelPatch** is a DL model patching method that selectively freezes early network layers during fine-tuning and transmits only the changed parameters, producing a differential patch that is a fraction of the full model size.

SwarmMARL addresses Objective 3 through a Multi-Agent Proximal Policy Optimization (MAPPO) [98] controller trained across all agents simultaneously. Each agent observes its local state — including update status, battery level, neighborhood composition, and transfer progress — and selects a coordination action: broadcast, request, or idle. SwarmMARL requires no pre-defined roles or static hierarchies, making it a flexible alternative to SwarmSync whose behavior adapts to swarm conditions at runtime.

Both systems are evaluated in simulation using the ARGoS robotics simulator [87] across swarm sizes, failure rates, and swarm compositions. Critically, both are also validated on physical UAV hardware in a controlled laboratory setting, providing evidence that the approaches transfer beyond simulation. The real-world implementation uses a small fleet of three to five drones; due to onboard computational constraints, SwarmMARL policy inference is executed on a ground server that communicates with the drones in real time, reflecting a realistic edge-offloading deployment pattern that is common in resource-constrained swarm systems [104].

Contributions

The contributions of this thesis are as follows:

- **SwarmSync:** A hierarchical, ACK-guaranteed software update synchronization strategy for heterogeneous UAV swarms, establishing that role-based subswarm organization achieves logarithmic dissemination time in swarm size and outperforms gossip-based and auction-based baselines across a wide range of swarm sizes and failure rates.

- **SwarmModelPatch:** The first DL model patching method designed for UAV swarms, demonstrating that selective layer freezing during fine-tuning produces a controllable and predictable trade-off between patch size, convergence time, and model accuracy. Using SwarmModelPatch enables mission-aware tuning of update payload instead of retraining from scratch.

- **SwarmMARL:** The first application of multi-agent reinforcement learning to software update coordination in heterogeneous UAV swarms, demonstrating that a MAPPO-trained shared policy learns context-sensitive broadcast and request behavior that reduces redundant transmissions and adapts to battery and connectivity constraints without requiring pre-defined roles or explicit protocol design.

- **Comparative evaluation:** A direct empirical comparison of deterministic and learned coordination philosophies for swarm update propagation across shared experimental conditions, identifying when each approach is preferable and providing empirical guidance for practitioners choosing between the two.

- **Real-world validation:** A physical implementation and evaluation of both SwarmUpdate and SwarmMARL on UAV hardware in a controlled laboratory environment, demonstrating that both approaches transfer from simulation to deployment and characterizing the practical adjustments required under real hardware constraints.

- A replication package containing simulation environment setups, scripts, data, and DL models at `https://github.com/SAILResearch/replication-25-allen-update-swarms`.

Although our domain involves UAVs, the core challenges we address are broadly applicable to any distributed autonomous system requiring coordinated software maintenance under resource and connectivity constraints. The deterministic and learned coordination mechanisms that we propose constitute contributions to both software engineering and multi-agent systems research that extend beyond the UAV domain.

While some of these methods have been researched in other domains, our work is the first to apply them to a heterogeneous swarm setting, addressing the unique coordination and resource-constraint challenges that heterogeneity introduces.

1.4 Thesis Structure

This thesis is structured as follows.

Chapter 1 introduces the concepts of software update in UAV swarm and the importance of the problem, the motivational example illustrates the real-world urgency of in-field software updates for heterogeneous UAV swarms.

Chapter 2 surveys related work on swarm update strategies, DL model updating, multi-agent reinforcement learning for swarms, and swarm communication protocols.

Chapter 3 formalizes the system model, update convergence definition, and evaluation metrics used across both approaches.

Chapter 4 presents SwarmUpdate, including the SwarmSync synchronization protocol, its formal analysis, SwarmModelPatch, and empirical evaluation against baseline strategies.

Chapter 5 presents SwarmMARL, including the observation and reward design, the MAPPO architecture, its simulator integration, and its standalone empirical evaluation.

Chapter 6 presents a unified comparative evaluation of SwarmSync and SwarmMARL across shared experimental conditions.

Chapter 7 presents the real-world validation of both approaches on physical Crazyflie 2.1 Brushless hardware in a controlled laboratory environment.

Chapter 8 discusses implications, limitations, and practical deployment considerations for both approaches.

Chapter 9 concludes the thesis and outlines directions for future work.

Chapter 2

Background and Related Work

2.1 Background

This section introduces the foundational concepts and technical background required to understand the contributions of this thesis. Readers familiar with UAV swarm systems, software update dissemination, deep learning for perception, multi-agent reinforcement learning, and the statistical and hardware terminology used in the evaluation chapters may skip to Section 2.2, where we discuss related work that leverages these foundational concepts.

2.1.1 Heterogeneous UAV Swarm Systems

A UAV swarm is a collection of autonomous aerial agents that coordinate to accomplish a shared mission without relying on a centralized controller [104]. In a homogeneous swarm, all agents are identical in hardware capability, software configuration,

and role. In a heterogeneous swarm, agents differ along one or more of these dimensions: some may carry cameras and run deep learning models for visual perception, while others execute rule-based navigation logic without any sensing payload [84, 25]. This distinction is fundamental to this thesis because it determines which agents need which updates and how coordination must be structured to respect those differences.

Communication in UAV Swarms

UAV agents communicate through wireless peer-to-peer links within a bounded communication range r. Because no agent has a global view of the swarm, information propagates through multi-hop relay: an agent that receives a message can forward it to neighbors that are within range, who can forward it further. This epidemic-style propagation is the basis of gossip protocols and is also the mechanism by which SwarmSync propagates update signals before structured delivery begins.

Communication links in UAV swarms are unreliable. Packet loss can occur due to distance, interference, or physical obstacles, and is modeled in this thesis using a failure rate $f \in [0, 1)$, where each transmitted packet is independently dropped with probability f. A failure rate of $f = 0$ represents ideal conditions with no loss, while $f = 0.75$ represents a highly degraded channel. Under any $f < 1$, a packet eventually reaches its destination given sufficient retransmissions, which is the property that SwarmSync's ACK-based delivery exploits.

Beyond the simplified peer-to-peer model used in this thesis, UAV swarms are more broadly studied as mobile ad-hoc networks (MANETs) and, when composed of aerial nodes specifically, flying ad-hoc networks (FANETs) [12]. In these networks,

topology is dynamic: links appear and disappear as drones move, and no node has a global view of network state. Standard routing protocols have been adapted for this setting, including the Ad-hoc On-Demand Distance Vector (AODV) protocol, which discovers a route to a destination only when needed by broadcasting route request messages and waiting for a reply, and the Optimized Link State Routing (OLSR) protocol, which instead proactively maintains routing tables at every node through periodic exchange of link-state information, trading higher background overhead for faster route availability. Both families of protocol are designed for lightweight control messages rather than large software payloads, which limits their direct applicability to the update dissemination problem this thesis addresses.

Simulation with ARGoS

ARGoS [87] is a widely used open-source robotics simulator designed specifically for large-scale swarm simulation. It supports multi-threaded execution, making it practical for simulating hundreds of UAVs simultaneously. ARGoS models time in discrete control steps, each corresponding to a fixed real-time duration of 100 ms in the default configuration. At each control step, every agent executes exactly one iteration of its controller, performing movement, signal transmission, and signal reception atomically. A key constraint of ARGoS is that an agent can only read signals transmitted during the previous control step, introducing a one-step communication delay that affects protocol timing. This constraint is discussed further in the limitations in Chapter 8.

For generalization purposes (Section 8.5.3), we also validate our results in Gazebo Harmonic, a general-purpose robotics simulator that models realistic rigid-body physics

and sensor noise and is widely used outside of swarm-specific research, integrated with ROS 2 (Robot Operating System 2), a widely used open-source middleware framework that handles communication between a robot's sensors, controllers, and actuators as a graph of message-passing processes. Unlike ARGoS, Gazebo lacks a global time step and therefore models communication asynchronously. Webots is a similar general-purpose simulator occasionally used as a point of comparison in the swarm robotics literature, though it is not used for experiments in this thesis. ARGoS's emphasis on multi-threaded execution makes it, unlike Gazebo or Webots, particularly well suited to swarms with hundreds of agents, which is why it is used as the primary simulation environment throughout this thesis.

Agent Roles in This Thesis

Following established practice in heterogeneous swarm research [26], this thesis models the swarm using two agent types available in ARGoS. Eyebots are vision-capable agents that run convolutional neural network models for perception tasks and require DL model updates when their operating environment changes. Footbots are navigation-oriented agents that execute rule-based task logic and require task-level software updates. A dedicated Updater agent carries the new software version and initiates the dissemination process. These three roles capture the core sensor-actuator dichotomy of heterogeneous swarms while keeping the experimental design focused on the update coordination problem rather than task planning variables.

Two agent types are sufficient to constitute a heterogeneous swarm in this sense: the swarm robotics literature defines heterogeneity by the presence of robots with di-

vergent capabilities and roles rather than by a minimum number of types, and recent work continues to treat two-role swarms, such as worker/guide [113] or capability-differentiated pairs [73], as heterogeneous in contrast to single-type homogeneous baselines. The Eyebot/Footbot pairing reflects this same minimal but genuine heterogeneity: each type has a distinct capability (perception vs. navigation) and a correspondingly distinct update requirement, which is the property this thesis's update-propagation problem is addressing.

2.1.2 Software Update Dissemination

Software update dissemination is the process of propagating a new software version from a source to all agents that require it. In centralized systems, this is straightforward: a server pushes the update to all clients over stable connections. In distributed autonomous systems such as UAV swarms, the problem is substantially harder because there is no persistent central server, communication links are unreliable and intermittent, and agents may be geographically dispersed with no guarantee of direct connectivity.

Over-the-Air Updates

Over-the-air (OTA) programming refers to the delivery of software updates over wireless links without physical access to the device [112]. In UAV swarms, OTA updates allow the swarm to receive new software versions while deployed in the field, avoiding the need to recall agents to a base station. The update payload is segmented into fixed-size packets, each labeled with an index. The receiver tracks which packets have

arrived and requests retransmission of any that are missing. This packet-based model is used throughout this thesis, with a packet size of 12.5 kB and a transmission rate of 10 packets per second, corresponding to an effective throughput of 1 Mbps consistent with the Bitcraze Crazyradio 2.0 module used in our real-world experiments.

Reliable Delivery and ACK-Based Flow Control

The Transmission Control Protocol (TCP) [16] provides reliable, ordered delivery over unreliable networks through acknowledgement (ACK) signals and retransmission. When a sender transmits a packet, it waits for an ACK from the receiver before proceeding to the next packet. If no ACK arrives within a timeout window, the sender retransmits the packet. This mechanism guarantees eventual delivery as long as the channel failure rate satisfies $f < 1$. SwarmSync adopts TCP-style ACK-based flow control as its delivery primitive, adapting it to the group communication setting where a single sender must collect ACKs from multiple receivers before advancing. Where confidentiality or authenticity is required, this delivery primitive can be layered with standard cryptographic protections such as TLS (Transport Layer Security, for TCP-style connections) or its datagram variant DTLS, both of which establish an encrypted, authenticated channel between two parties before any application data is exchanged.

Convergence

A swarm update is said to have converged when every agent that requires the update has received and applied all update packets. Convergence is the primary termina-

tion condition for both SwarmSync and SwarmMARL, mirroring how other robotic swarm OTA systems similarly require the swarm to reach a consensus on running the same code version before proceeding [112], in contrast to per-device IoT and automotive OTA update protocols, which typically treat each device's update as complete independently without a swarm-wide convergence barrier [1, 102].

In SwarmSync, convergence is explicitly signaled: followers notify their leader upon completion, and leaders notify the Updater, producing a confirmed global convergence event. In SwarmMARL, convergence is implicit: the environment transitions to a terminal state when all compatible agents reach the target software version. The time to convergence, measured in ARGoS control steps, is the primary efficiency metric used throughout this thesis.

2.1.3 Deep Learning for UAV Perception

Many UAV missions require on-board interpretation of sensor data. Deep learning models, particularly convolutional neural networks (CNNs), have become the dominant approach for visual perception tasks such as object detection, scene classification, and anomaly detection [57, 63]. Understanding the structure of CNNs and the challenges of maintaining them in deployment is essential background for SwarmModelPatch.

Convolutional Neural Networks

A CNN processes input images through a sequence of layers. Early convolutional layers learn general, low-level features such as edges and textures that are broadly useful

across many visual tasks. Later convolutional layers learn increasingly task-specific features. The final classification layer maps these features to output class probabilities. This hierarchical structure has an important practical consequence: the early layers of a model trained on one visual domain often transfer well to related domains, while only the later layers need to be adapted to a new task [125]. SwarmModelPatch exploits this property by freezing early layers during fine-tuning, limiting the parameters that change and therefore limiting the size of the update patch that must be transmitted.

Concept Drift

Concept drift occurs when the statistical distribution of the data an agent encounters in deployment diverges from the distribution it was trained on, causing model accuracy to degrade over time [36]. In UAV swarms, concept drift can arise from environmental changes such as shifting weather conditions, seasonal variation, or the emergence of new hazards such as wildfire smoke. When concept drift is detected, the deployed model must be updated to restore accuracy. This update involves fine-tuning the model on new data and transmitting the updated parameters to all agents running the affected model, precisely the scenario that SwarmModelPatch and the update dissemination protocols address.

Catastrophic Forgetting and Replay Buffers

A known challenge in incremental model training is catastrophic forgetting: when a model is fine-tuned on new data, its performance on previously learned classes can

degrade sharply [66]. A common mitigation is to include a replay buffer: a small set of representative samples from previous training data, in the fine-tuning dataset alongside the new data. This forces the model to retain its prior knowledge while adapting to the new distribution. SwarmModelPatch uses a replay buffer containing 40% old training data during fine-tuning to balance adaptation to new classes against retention of prior classification ability.

2.1.4 Model Compression and Adaptation Terminology

This thesis references several standard techniques for reducing model size or adapting models to new data, most of which are discussed only briefly in Chapter 2 as points of comparison to SwarmModelPatch. This section defines these terms for readers unfamiliar with the model compression and continual learning literature.

Pruning. Pruning reduces model size by removing individual weights, neurons, or entire channels that contribute little to the model's output, typically identified by their magnitude or their measured effect on accuracy when removed [39]. The result is a smaller model with fewer parameters to store and compute, though the removed structure is permanently discarded rather than recoverable.

Quantization. Quantization reduces model size by lowering the numerical precision used to represent each weight, for example, storing weights as 8-bit integers instead of 32-bit floating-point numbers [93]. Because every weight is represented with fewer bits, the overall model becomes smaller and faster to compute, typically at a modest cost to accuracy. Quantization changes *how precisely* each weight is stored, whereas pruning changes *how many* weights exist.

Knowledge distillation. Knowledge distillation trains a smaller "student" model to reproduce the behavior of a larger, already-trained "teacher" model, rather than training the student directly on the original labeled data [59]. The student learns from the teacher's output probabilities, which often convey more information than hard labels alone, allowing the student to approximate the teacher's accuracy at a fraction of the size.

Transfer learning and layer freezing. As introduced in Section 2.1.3, transfer learning adapts a model trained on one task to a related task by reusing its learned features rather than training from scratch. Layer freezing is the specific mechanism used to do this: during fine-tuning, the weights of selected layers are held fixed (frozen) and excluded from gradient updates, while the remaining layers continue to be trained normally [125]. Only the unfrozen layers' weights change during fine-tuning, which is the property SwarmModelPatch relies on to produce a small differential patch.

Model merging and weight averaging. Model merging combines multiple separately trained or separately adapted models into a single model by directly combining their weights, most commonly by averaging corresponding weights across models, rather than by further training [121]. This is distinct from ensembling, which keeps multiple models separate and combines only their predictions at inference time. Task arithmetic is a related technique that treats the weight difference between a fine-tuned model and its original base model as a reusable "task vector" that can be added to or subtracted from other models' weights to transfer or remove a learned behavior [44].

Federated learning. Federated learning (FL) is a distributed training approach in which multiple participants each train a model on their own local data and period-

ically share only their model updates, typically averaged weights or gradients, with a central server, rather than sharing the raw data itself [68]. FL addresses *how a model is collaboratively trained* without centralizing data; it is a distinct problem from *distributing an already-trained model update* to devices that need it, which is the problem this thesis addresses (see Section 2.2 for further discussion of this distinction).

2.1.5 Multi-Agent Reinforcement Learning

Multi-agent reinforcement learning (MARL) is a framework for training multiple autonomous agents to make decisions that maximize a shared or individual reward signal through interaction with an environment [78, 129, 108]. This section introduces the key concepts and algorithms that underpin SwarmMARL.

Markov Decision Processes

A single-agent reinforcement learning problem is formalized as a Markov Decision Process (MDP), defined by a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \gamma)$, where $\mathcal{S}$ is the state space, $\mathcal{A}$ is the action space, $\mathcal{T} : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$ is the transition function, $\mathcal{R} : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is the reward function, and $\gamma \in [0, 1)$ is a discount factor that controls the relative importance of immediate versus future rewards. An agent learns a policy $\pi : \mathcal{S} \to \Delta(\mathcal{A})$, a possibly stochastic mapping from states to a probability distribution over actions, so as to maximize the expected discounted cumulative reward $\mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t r_t\right]$.

In the multi-agent setting, the MDP is extended to a Decentralized Partially Observable MDP (Dec-POMDP) [78], where each agent i receives a local observation $o_i \in \mathcal{O}_i$ rather than the full global state, and selects actions from its own action

space $\mathcal{A}_i$. Agents share a joint reward signal in fully cooperative settings, meaning all agents benefit when any agent makes progress toward the shared goal. Update dissemination in UAV swarms is a fully cooperative task: every agent benefits when the swarm converges, and no agent has an incentive to defect.

Policy Gradient Methods and PPO

The policy π defined above is, in practice, represented as a parameterized function $\pi_\theta(a|s)$, typically a neural network with weights θ that outputs a probability distribution over actions given a state. Policy gradient methods directly optimize these parameters θ to maximize the expected discounted cumulative reward, expressed as an explicit function of θ:

$$J(\theta) = \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{\infty} \gamma^t r_t\right]$$

Since $J(\theta)$ cannot be computed or differentiated directly (the transition and reward dynamics are unknown to the agent), policy gradient methods instead estimate its gradient from sampled trajectories, using the following identity [107]:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(a|s) \cdot A(s,a)\right]$$

where $A(s,a)$ is the advantage function, which measures how much better action a is compared to the average action in state s. In practice, this expectation is approximated by an empirical average over a batch of timesteps collected by running the current policy π_θ in the environment, with $A(s,a)$ estimated from the resulting trajectories (e.g., via Generalized Advantage Estimation [97]) rather than computed

analytically. For clarity, we present the single-agent case here using the MDP notation (s, a) introduced above; the extension to the Dec-POMDP setting, in which each agent optimizes a policy $\pi_{\theta_i}(a_i|o_i)$ over its own local observations, is described in Section 2.1.5 when we introduce MAPPO.

Proximal Policy Optimization (PPO) [98] stabilizes policy gradient training by clipping the ratio between the new and old policy probabilities, preventing destructively large updates:

$$\mathcal{L}^{\text{CLIP}}(\theta) = \mathbb{E}_t \left[\min \left(r_t(\theta) A_t,\ \text{clip}(r_t(\theta),\ 1-\epsilon,\ 1+\epsilon) A_t \right) \right]$$

where $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$ is the probability ratio and ϵ is the clipping threshold. As above, $\mathbb{E}_t[\cdot]$ denotes an empirical average over a batch of timesteps sampled by running the policy $\pi_{\theta_{\text{old}}}$ before the update, rather than a closed-form expectation. This clipped objective ensures that the policy does not move too far from its previous version in any single update step, which substantially improves training stability.

An important auxiliary quantity in policy gradient training is the entropy of the policy's action distribution, $\mathcal{H}(\pi_\theta(\cdot|s)) = -\sum_a \pi_\theta(a|s) \log \pi_\theta(a|s)$, which measures how random or exploratory the policy's behavior is in a given state. High entropy corresponds to a near-uniform distribution over actions, indicating exploratory, largely random behavior; entropy near zero indicates the policy has converged to a confident, near-deterministic choice of action. Many PPO implementations, including SwarmMARL's, add an entropy bonus to the training objective early in training and anneal it over time, encouraging broad exploration before the policy commits to a specific strategy. Monitoring entropy over training, as in Chapter 5, is a standard way to

verify that a policy is transitioning from exploration to confident, learned behavior rather than remaining stuck at random.

Generalized Advantage Estimation

The advantage function $A(s, a)$ is estimated using Generalized Advantage Estimation (GAE) [97], which trades off bias and variance in the advantage estimate through a parameter $\lambda \in [0, 1]$:

$$\hat{A}_t^{\mathrm{GAE}(\gamma,\lambda)} = \sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_{t+l}$$

where $\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$ is the temporal difference (TD) error, i.e., the discrepancy between the value predicted for state s_t and the value implied by the observed reward and the next state's estimated value, and $V(s)$ is the value function estimating expected future reward from state s. When $\lambda = 0$, GAE reduces to the one-step TD error (low variance, high bias); when $\lambda = 1$, it reduces to the full Monte Carlo return (high variance, low bias). SwarmMARL uses $\lambda = 0.95$, a standard value that balances these trade-offs.

Actor-Critic Architecture

PPO uses an actor-critic architecture, in which a shared neural network produces both a policy (the actor) and a value estimate (the critic). The actor outputs a probability distribution over actions given an observation, from which an action is sampled. The critic outputs a scalar value estimate used to compute the advantage. Sharing early network layers between actor and critic allows the network to learn a common representation of the observation that is useful for both action selection and

value estimation. SwarmMARL uses a two-layer shared network with LayerNorm stabilization, a normalization technique that rescales the activations within a layer to have a consistent mean and variance before they are passed to the next layer; this helps keep gradients well-behaved during training and generally makes deep networks faster and more stable to train, followed by separate policy and value heads.

Centralized Training with Decentralized Execution

The centralized training with decentralized execution (CTDE) framework [62] addresses a fundamental challenge in MARL: in the Dec-POMDP setting introduced above, each agent must ultimately act on its own local observation o_i rather than the full global state, yet training a policy from only this limited view can be slow and unstable. CTDE resolves this tension by allowing agents to share information during training to learn better joint policies, while still requiring that, at execution time, each agent act using only its own local observation, matching the Dec-POMDP constraint. In CTDE, a centralized critic has access to the global state or the observations and actions of all agents during training, providing a richer learning signal. At execution time, each agent uses only its own local observation to select actions.

MAPPO [126] applies the PPO objective within the CTDE framework, using a centralized value function during training and a decentralized policy at execution. Its clipped objective mirrors the single-agent PPO loss from Section 2.1.5, applied per agent and averaged across the batch of collected multi-agent trajectories:

$$\mathcal{L}^{\text{CLIP}}(\theta) = \mathbb{E}_t \left[\frac{1}{N} \sum_{i=1}^{N} \min \left(r_{i,t}(\theta) A_{i,t},\ \text{clip}(r_{i,t}(\theta),\ 1-\epsilon,\ 1+\epsilon) A_{i,t} \right) \right]$$

where N is the number of agents, $r_{i,t}(\theta) = \frac{\pi_\theta(a_{i,t}|o_{i,t})}{\pi_{\theta_{\text{old}}}(a_{i,t}|o_{i,t})}$ is agent i's probability ratio at timestep t, and $A_{i,t}$ is the advantage estimate for agent i, computed from the centralized critic using GAE as described above.

SwarmMARL adopts this approach: all agents share a single policy network trained centrally in simulation, which is then deployed on each agent independently during execution. In the real-world experiments described in Chapter 7, policy inference is offloaded to a ground server due to onboard computational constraints, which is a common edge-offloading pattern in resource-constrained swarm deployments.

Chapter 2 compares MAPPO against two other prominent cooperative MARL algorithms. QMIX [91] learns a factored value function that decomposes an overall joint action-value into per-agent contributions constrained to combine monotonically, allowing decentralized execution while still optimizing a shared team objective. MADDPG [62] instead extends deterministic policy gradient methods to the multi-agent setting using a centralized critic per agent that observes the actions of all agents during training. Both are CTDE-style algorithms, like MAPPO, but differ in how they represent and optimize the learning objective.

2.1.6 Statistical Methods for Comparative Evaluation

Chapter 6 compares SwarmSync and SwarmMARL using statistical hypothesis testing rather than relying on raw averages alone, to establish whether observed differences in performance are unlikely to have arisen by chance. This section introduces the statistical concepts used there.

Mann-Whitney U test. The Mann-Whitney U test is a non-parametric statis-

tical test that determines whether two independent samples are likely to have been drawn from the same underlying distribution, without assuming that the data is normally distributed [67]. It does so by ranking all observations from both samples together and comparing how the ranks are distributed between the two groups; if one group's values are consistently ranked higher than the other's, the test yields a small p-value, indicating that the two samples likely differ. This test is well suited to the convergence time and overhead measurements used in this thesis, which are not guaranteed to follow a normal distribution.

p-value and statistical significance. The p-value produced by a hypothesis test is the probability of observing a difference between two samples at least as large as the one measured, if in reality there were no true difference between the underlying distributions being compared. A small p-value (conventionally below a chosen significance threshold α) indicates that the observed difference is unlikely to be due to chance alone, and is reported as statistically significant.

Effect size and rank-biserial correlation. A statistically significant result indicates that a difference likely exists, but not how large or practically meaningful that difference is; effect size measures the latter. This thesis uses the rank-biserial correlation r, the standard effect size measure associated with the Mann-Whitney U test, which ranges from -1 to 1. A value of $r = 0$ indicates the two groups' values are indistinguishable in rank; $|r| = 1$ indicates complete separation, where every value in one group ranks higher than every value in the other. Following common interpretation guidelines [116], this thesis treats $|r| < 0.11$ as a negligible effect, $0.11 \leq |r| < 0.28$ as small, $0.28 \leq |r| < 0.43$ as medium, and $|r| \geq 0.43$ as a large

effect.

Bonferroni correction. When many statistical tests are performed at once, for example, once for every combination of swarm size and failure rate in Chapter 6, the chance that at least one test appears significant purely by random chance increases with the number of tests performed. The Bonferroni correction compensates for this by dividing the standard significance threshold (conventionally $\alpha = 0.05$) by the number of tests being performed, yielding a stricter, corrected threshold that a result must meet to be considered significant. This thesis applies this correction whenever multiple related hypothesis tests are reported together.

2.1.7 Hardware and Middleware for Real-World Validation

Chapter 7 validates SwarmSync and SwarmMARL on physical Crazyflie 2.1 Brushless drones. This section introduces the supporting hardware and software concepts used in that validation.

Extended Kalman Filter (EKF). An Extended Kalman Filter is a standard sensor-fusion algorithm that combines multiple noisy sensor readings over time to produce a single, more accurate estimate of a system's state, in this case, a drone's position, velocity, and orientation. The Crazyflie's onboard EKF fuses data from its inertial measurement unit and external positioning references to maintain a stable estimate of its own pose during flight.

Vicon motion capture. Vicon is an infrared motion-capture system that tracks the position of small reflective markers attached to a physical object, in this case, each drone, using an array of ceiling-mounted cameras. It provides precise, low-latency

ground-truth position data that the Crazyflie's onboard EKF uses as an external reference during the hover sequences described in Chapter 7.

ROS 2. As introduced in Section 8.5.3 for simulation, Robot Operating System 2 (ROS 2) is a widely used open-source middleware framework for robotics software. It structures an application as a graph of communicating processes (nodes) that exchange messages over named channels (topics), allowing sensor drivers, control logic, and coordination software to be developed and run somewhat independently while still communicating in real time. In the real-world experiments of Chapter 7, ROS 2 nodes bridge communication between the ground server and the physical drones.

Bluetooth Low Energy (BLE). Bluetooth Low Energy is a short-range wireless communication standard designed for low power consumption relative to classic Bluetooth or Wi-Fi, commonly used in battery-constrained embedded and IoT devices. The energy model in Section 8.5 adopts per-bit transmission and reception energy costs representative of BLE, as BLE-class radios are widely used in UAV communication systems.

2.2 Related Work

This chapter surveys the research landscape that this thesis builds upon and contributes to. We organize the discussion around five areas: software update strategies for swarms, deep learning model updating techniques, multi-agent reinforcement learning for swarm coordination, distributed and federated learning for edge systems,

and software engineering for UAV systems. Together, these areas define the gap that this thesis addresses: no prior work has studied software update propagation in heterogeneous UAV swarms through both deterministic protocol-based and learned policy-based coordination, nor validated such approaches on physical UAV hardware.

2.2.1 Software Update Strategies for Swarms

Existing research on software update strategies focuses mainly on homogeneous swarms [112, 4]. Varadharajan et al. [112] propose an update protocol for homogeneous robotic swarms that incorporates retransmission and request-based recovery, demonstrating that gossip-style rebroadcasting can achieve rapid dissemination when all agents share the same update requirements. Al Blooshi and Han [4] extend this line of work by focusing on security threats during swarm updates. However, neither study addresses the complexities inherent in heterogeneous swarms, where different UAVs have distinct roles, hardware capabilities, and update requirements. The uniform dissemination assumption that underlies these protocols breaks down when only a subset of agents requires a given update, introducing both wasted transmissions and coordination ambiguity.

Since egalitarian, homogeneous swarms can be modelled as Peer-to-Peer (P2P) systems, swarm researchers have drawn on P2P software update strategies. Gossip-based protocols [15] are the most widely adopted, using epidemic spreading to propagate updates with minimal coordination overhead. However, gossip-based protocols suffer from redundant messaging, inconsistencies under concurrent updates, and poor performance in sparse or intermittent networks [86, 112]. While newer dissemination

strategies exist, including tree-based dissemination, hybrid dissemination, and coded gossip [32, 22, 58], these approaches are optimized for lightweight message passing where messages are short and transmission is near-instantaneous. Software updates in UAV swarms involve transferring relatively large payloads, where transmission duration and packet loss become the dominant challenges, making direct application of these protocols impractical.

Auction-based approaches such as SOUL [114] reduce redundancy by allowing UAVs to bid for update data, yet still treat the swarm as a monolithic group and do not account for role-based update targeting. Recent work by Varadharajan et al. [113] recognizes the value of hierarchical coordination, showing that hierarchical structures improve scalability in heterogeneous robotic swarms. This thesis builds on that insight through SwarmSync, which organizes UAVs into role-based sub-swarms with designated leaders and ACK-guaranteed delivery. Critically, however, no prior protocol-based approach has been compared against a learned coordination policy for the same update dissemination task, leaving open the question of when explicit protocol design is preferable to emergent learned behavior.

In addition to update-specific protocols, optimization-based approaches have been applied to UAV swarms to improve operational efficiency. Abid et al. [2] explore variants of the travelling salesman problem for UAV data collection using metaheuristic algorithms including ant colony optimization, artificial bee colony, and genetic algorithms, demonstrating reductions in flight distance that can indirectly reduce communication overhead. Shanmugam et al. [99] introduce a split control design for UAV swarms combining genetic algorithms with A$*$ trajectory planning and specialized

FANET communication protocols. Although these works optimize flight planning and communication, they do not address update payload distribution in heterogeneous swarms.

2.2.2 Deep Learning Model Updating

Updating deep learning (DL) models deployed on resource-constrained hardware is a critical challenge, as DL models are typically large and bandwidth-intensive. Researchers have proposed various techniques to mitigate the need for full retraining, including transfer learning [120] and lifelong learning [77, 79]. The idea of lifelong learning traces back to McCloskey and Cohen [66], who proposed training a model incrementally on new data to avoid catastrophic forgetting. Olewicki et al. [77] have more recently explored cost-efficient lifelong learning techniques that use a replay buffer of past training data to stabilize incremental adaptation. Transfer learning leverages pre-trained models for adaptation to new but related tasks [120], reducing computational cost. However, these studies focus primarily on improving accuracy or minimizing training resources [94, 29, 45], rather than reducing the size of the update transmitted across the network.

Updating a DL model in response to environmental shifts constitutes a runtime behavioral modification of the system, analogous to applying a software patch to fix or adapt functionality. This perspective is supported by recent work in MLOps, self-adaptive systems, and ML patching [65, 9, 8, 30]. Compression techniques such as pruning [39] and quantization [29, 132, 93] reduce the overall model size and computational cost, but do not specifically target the differential between model versions to

minimize patch size. Pruning removes less important neurons post-training, whereas quantization reduces weight precision; neither explicitly minimizes the size of the update transmitted between versions. SwarmModelPatch addresses this gap directly by freezing early layers during fine-tuning and transmitting only the parameters that changed, producing a differential patch that is a fraction of the full model size.

Layer freezing is an established transfer learning technique [125], but to the best of our knowledge, no prior work has purposed layer freezing, an established transfer learning technique. specifically as a mechanism for minimizing the size of the transmitted update between model versions for deployment on resource-constrained UAV swarms, nor formalized the resulting patch-size trade-off as a controllable parameter.

In parallel, UAV perception research has emphasized lightweight and real-time inference. He et al. [38] propose Quick-UAV OCSORT, a multi-object tracking-by-detection algorithm with a lightweight network and efficient matching strategies achieving strong performance on UAV benchmarks. While such work highlights efficiency gains in inference pipelines, it does not consider incremental DL model updates or the communication cost of patch distribution across a swarm.

2.2.3 Multi-Agent Reinforcement Learning for Swarm Coordination

Multi-agent reinforcement learning (MARL) has emerged as a powerful paradigm for learning decentralized coordination policies in systems where multiple agents must act based on local observations to achieve a shared goal [78, 129, 108]. The centralized training with decentralized execution (CTDE) framework has become the dominant

approach: agents share a centralized critic during training, which has access to global state information, but execute using only local observations at runtime [62]. This makes CTDE-based approaches practical for deployment on autonomous systems with limited communication, such as UAV swarms.

Proximal Policy Optimization (PPO) [98] is a widely used single-agent policy gradient algorithm that stabilizes training through a clipped surrogate objective, preventing large policy updates that can destabilize learning. Its multi-agent extension, MAPPO [126], applies the same clipped objective within the CTDE framework and has demonstrated strong performance across a range of cooperative multi-agent benchmarks, often matching or outperforming more complex algorithms such as QMIX [91] and MADDPG [62] in fully cooperative settings. The simplicity and stability of MAPPO make it a natural choice for coordination tasks in swarms where agents share a common objective, as is the case in update dissemination.

MARL has been applied to several swarm coordination tasks. Hüttenrauch et al. [42] demonstrate that deep MARL agents can learn formation control and coverage behaviors in robotic swarms without explicit programming of the desired formation. Preiss et al. [89] show that learned controllers can be deployed on physical multi-rotor platforms, though most MARL work in robotics remains in simulation. In the context of communication-constrained systems, Sukhbaatar et al. [106] propose a differentiable communication protocol that allows agents to learn when and what to communicate, showing that learned communication can outperform fixed protocols in cooperative tasks.

Despite these advances, MARL has not been applied to the software update co-

ordination problem in UAV swarms. Existing MARL work on swarms focuses on navigation, coverage, and formation tasks where the agents' physical positions are the primary decision variable. Update dissemination presents a different structure: agents must coordinate logical state transitions (from outdated to updated) using communication actions (broadcast or request) under resource constraints (battery, bandwidth) and without global knowledge of which agents have received the update. This thesis addresses this gap through SwarmMARL, which applies MAPPO to the update coordination problem and compares the resulting learned policy against the deterministic SwarmSync protocol under shared experimental conditions.

2.2.4 Distributed and Federated Learning for Edge and Swarm Systems

Federated Learning (FL) has emerged as a promising paradigm for collaborative perception in UAV swarms, enabling privacy-preserving model training without raw data exchange [11]. The canonical FL framework, FedAvg [68], assumes a central orchestrator and synchronous communication rounds, assumptions that are often violated in dynamic UAV swarms operating in remote environments with intermittent connectivity and heterogeneous hardware. Recent efforts have sought to adapt FL to aerial networks. Zhong et al. [130] propose a hierarchical FL framework where UAVs act as mobile aggregators, improving convergence under mobility but still relying on stable uplinks.

It is important to distinguish FL from the update dissemination problem this thesis addresses. FL focuses on *collaboratively training* a model from distributed

data without centralizing that data. The problem studied in this thesis is *distributing a pre-trained model update* to agents that need it, once that update has already been produced. These are complementary problems: FL could be used to produce the updated model at headquarters, and SwarmSync or SwarmMARL could then be used to disseminate it to the swarm. Existing FL approaches do not address the distribution of the resulting model update under the communication constraints of a heterogeneous UAV swarm, which is the gap this thesis targets.

Separately from FL, a growing body of work has explored *model merging*: combining multiple independently trained or independently adapted models into a single model, typically through weight averaging, rather than through joint training. Model soups [121] show that averaging the weights of multiple fine-tuned models, each initialized from the same pre-trained model, can match or exceed the accuracy of the best individual model without any additional training or inference cost. Related techniques such as task arithmetic [44] treat the weight difference between a fine-tuned and base model as an editable "task vector" that can be added to or combined with other task vectors to compose new model behaviors. These approaches are conceptually related to SwarmModelPatch's differential patch, which is also a weight-space difference applied via addition, but they address a different problem: reconciling multiple independently adapted models into one, rather than minimizing the size of a single differential update produced by one fine-tuning process. Extending SwarmModelPatch to a setting where sub-swarms independently adapt and must later merge their models, rather than applying one shared patch, is a natural direction that model merging techniques could inform.

2.2.5 Incremental Learning Beyond Layer Freezing

While transfer learning via layer freezing is widely used [125], modern incremental learning techniques offer more sophisticated approaches to mitigating catastrophic forgetting. Elastic Weight Consolidation (EWC) [52] adds a regularization term that penalizes changes to weights that were important for previous tasks. Learning without Forgetting (LwF) [59] uses knowledge distillation to preserve old task performance while adapting to new data. iCaRL [92] maintains a bounded set of exemplars from previous classes to anchor the decision boundary during incremental updates. Olewicki et al. [77] recently evaluated such strategies in software analytics, showing trade-offs between computational cost and stability across tasks.

However, these methods typically require storing past samples, maintaining auxiliary models, or computing additional loss terms - overhead that is incompatible with the memory and compute budgets of UAV onboard systems. SwarmModelPatch adopts a pragmatic middle ground: it freezes early layers to preserve general feature extractors, reinitializes only the classifier to accommodate new classes, and uses a small replay buffer to stabilize adaptation. This design deliberately prioritizes deployability on resource-constrained hardware over maximum accuracy retention, a trade-off that is made explicit and evaluated empirically in this thesis.

Layer freezing itself is a well-established transfer learning technique, typically used to reduce training cost or mitigate overfitting on small target datasets while preserving general feature extractors learned on a source task [125]. SwarmModelPatch does not propose freezing as a new training technique; rather, it repurposes freezing for a different objective: controlling the size of the resulting weight-space differential that

must be transmitted to update a deployed model. It then formalizes the resulting trade-off between the number of frozen layers, patch size, convergence time, and accuracy as an explicit, tunable design parameter for swarm operators, an objective and formalization that, to our knowledge, has not been previously characterized.

2.2.6 Swarm Communication Protocols

UAV swarm communication has been studied extensively in the context of mobile ad-hoc networks (MANETs) and flying ad-hoc networks (FANETs) [12]. The fundamental challenge in these networks is that topology is dynamic: links appear and disappear as drones move, and no node has a global view of network state. Standard routing protocols such as AODV and OLSR have been adapted for FANETs, but they assume lightweight control messages rather than large software payloads, limiting their direct applicability to update dissemination.

As discussed in Section 2.1, SwarmSync adopts TCP-style ACK-based flow control as its delivery primitive to guarantee reliability. However, this per-connection overhead is non-trivial in large swarms, which motivates investigating whether learned coordination can achieve comparable reliability with lower overhead by suppressing unnecessary transmissions before they occur rather than recovering from them after the fact. This tension between protocol-guaranteed reliability and policy-learned efficiency is a central theme of this thesis and has not been studied in the context of swarm software updates.

2.2.7 Software Engineering for UAV Systems

Researchers in software engineering have studied various aspects of UAV systems including dynamic updates, robustness, and vulnerability detection. Nahabedian et al. [75] introduce an automated dynamic controller synthesis approach to facilitate efficient updates of UAV controllers, validated in UAV surveillance systems. Wang et al. [119] present a method to detect anomalies in UAV logs to enhance system robustness, while Han et al. [35] implement a learning-guided search framework to detect vulnerabilities in UAV configuration modules. Jung et al. [47] propose a dedicated debugging system to identify and resolve configuration-related bugs in UAV swarms.

Ataiefard et al. [10] propose a hybrid deep neural network method to infer internal states of UAV autopilots by analyzing input-output signal data, supporting interpretability and safety analysis. Liang et al. [60] empirically evaluate the usage of bounding functions within open-source UAV software frameworks to understand safety-critical concerns. Vierhauser et al. [118] introduce a framework for interlocking Safety Assurance Cases to improve operational accountability in multi-UAV deployments.

Han et al. [33] present ICSEARCHER, a system that applies metaheuristic search guided by machine learning predictors to uncover range specification bugs in UAV configuration parameters, achieving 94% accuracy in identifying unstable configurations. Han et al. [34] propose Nyctea, a runtime rectification framework that monitors UAV state transitions and iteratively reconfigures parameters to eliminate instability caused by misconfigurations. Both works underscore the importance of configuration correctness in UAV systems but focus on misconfiguration handling rather than the

efficient distribution of software and DL model updates across heterogeneous swarms.

Across all of these lines of research, two gaps remain. First, no prior work addresses the unique combination of challenges that heterogeneous UAV swarms face when updating DL models in the field: role-based payload targeting, reliable delivery under packet loss, minimal transmission overhead, and reduced model patch size under resource constraints. Second, no prior work has compared deterministic protocol-based coordination against learned policy-based coordination for the software update dissemination problem, leaving open fundamental questions about when each philosophy is preferable. This thesis addresses both gaps.

Chapter 3

Problem Formalization and System Model

Overview of Proposed Approaches

Before introducing the formal model, we briefly summarize, at a high level, the two coordination approaches this thesis compares; each is described in full in Chapter 4 and Chapter 5 respectively.

SwarmSync is a deterministic, hierarchical update protocol: UAVs are organized into role-based sub-swarms, each with a designated *leader* that relays the update to its *followers* and tracks delivery via acknowledgement (ACK) signals, guaranteeing that every follower eventually receives every packet. **SwarmModelPatch** reduces the size of DL model updates sent under this protocol by transmitting only the parameters that changed during fine-tuning, rather than the full model.

SwarmMARL instead trains a shared policy, using multi-agent reinforcement

learning, that lets each UAV decide for itself, based on its local conditions (e.g., battery level, update status, neighborhood composition), whether to broadcast the update, request it from a neighbor, or remain idle, without any predefined leader/follower roles.

The system model below is common to both approaches, the specific parameters used differ between our simulation environment (Section 3.2) and our real-world hardware evaluation (Chapter 7).

3.1 System Model

We model the UAV swarm as a connected graph $G = (U, E)$, where each node $u \in U$ represents a UAV and each edge $(u_1, u_2) \in E$ represents a bidirectional communication link between two UAVs within mutual communication range. Time progresses in discrete control steps, formally indexed by $t \in \mathbb{N}$, each corresponding to a fixed real-time duration Δ. At each step t, every UAV $u \in U$ executes exactly one iteration of its controller, performing its complete set of actions atomically, including movement, signal transmission, and signal reception. Causality is enforced by restricting each UAV to receive only signals transmitted during step $t - 1$.

Each transmission succeeds with probability $1 - f$, where $f \in [0, 1)$ denotes the packet failure rate. Packet loss is modeled as an all-or-nothing event: packets are either delivered intact or completely dropped. While real wireless channels may exhibit partial corruption or bit errors, modeling such nuanced behavior falls outside the current scope. We incorporate HMAC-based integrity verification for each transmit-

ted packet as a lightweight safeguard against undetected corruption. Although this mechanism yielded no measurable benefit under ARGoS's binary failure semantics (Section 3.2), it represents a minimal-overhead step toward higher robustness should the system be extended to environments with more complex channel behavior.

This abstract model is instantiated concretely by our simulation environment, described in Section 3.2, and by our real-world hardware setup, described in Chapter 7.

3.2 Simulation Environment

The empirical evaluation in this thesis instantiates the system model above using ARGoS [87] as the simulation environment, consistent with prior studies in swarm update and communications [112, 114, 115].

This section describes the simulation-specific parameters used to produce the results in Chapter 4 and Chapter 5; the equivalent real-world instantiation, using physical Crazyflie hardware, is described separately in Chapter 7.

ARGoS is a widely used C++ simulator designed specifically for large-scale swarm simulation, supporting multi-threaded execution that makes it practical for simulating hundreds of UAVs simultaneously. Although Gazebo [54] and Webots [69] are also popular robotics simulators, ARGoS's emphasis on multi-threaded execution makes it particularly suitable for scenarios involving swarms with hundreds of UAVs. Furthermore, ARGoS was the simulator used to evaluate SOUL and Gossip, the baseline strategies against which SwarmSync is compared in Chapter 4.

In ARGoS, the abstract control step defined in Section 3.1 corresponds to a fixed

real-time duration of $\Delta = 100$ ms; this one-step communication delay is a property of the simulation environment and is discussed further as a limitation in Chapter 8. Each simulation is run 20 times and results are averaged to ensure statistical significance.

3.2.1 Swarm Configuration

Similar to earlier research on heterogeneous swarms in ARGoS [25, 26], we simulate a swarm consisting of three types of UAV, each expecting different types of software update: Updater, Eyebots, and Footbots. The Updater manages and initiates the update process, relaying it to the swarm. The Eyebots require a DL model update due to sudden concept drift, such as weather changes from sunny to snowy conditions. The Footbots require a task-level software update. These agent types capture the core sensor-actuator dichotomy of heterogeneous swarms while keeping the experimental design focused on the update coordination problem rather than task planning variables.

In this study, heterogeneity refers to both hardware capabilities and update requirements. The swarm consists of Eyebots equipped with vision sensors and DL models, and Footbots that execute rule-based task logic. A single environmental event may therefore trigger different update payloads: DL model updates for Eyebots to handle changes in visual conditions, and task logic updates for Footbots to adjust navigation behavior. We do not consider swarms composed of UAVs performing fundamentally different mission types (e.g., delivery versus search-and-rescue), as such scenarios require higher-level coordination mechanisms to determine update routing and targeting; our dissemination framework would serve as the underlying

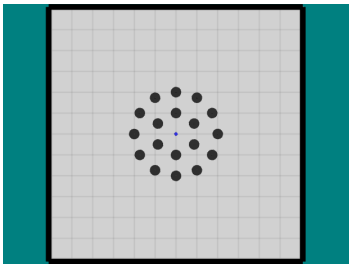


Figure 3.1: 18 drones surrounding the Updater, derived from a hexagonal packing within ARGoS's default communication range of $r = 3$ meters.

transport layer for such systems.

We experiment with swarm sizes $s \in \{20, 100, 200, 500\}$ excluding the Updater, with an equal 1:1 ratio of Footbots to Eyebots. In our simulation, the Updater is deployed stationary at the origin $(0, 0)$, while all Footbots and Eyebots are randomly distributed following a diffusion pattern for surveillance [85]. The topology of the swarm defaults to the cluster topology [86] and the maximum speed of all drones is $V = 1$ m/s, in agreement with prior research [105].

3.2.2 Communication Range and Transmission Speed

As there are physical limitations for the formation of drones (e.g., two drones cannot occupy the same location), and ARGoS uses a default communication range of $r = 3$ meters, we standardize the maximum number of drones simultaneously updating around the Updater to $N = 18$. This number is derived using a hexagonal distribution of drones within ARGoS's communication range, as shown in Fig. 3.1.

Although ARGoS's default data transmission speed is 100 bytes/sec, this is an artificial constraint that does not reflect real-world UAV communication speeds. We instead set a transmission speed of 1 Mbps based on the Bitcraze Crazyradio 2.0

module,[1] which is widely used in swarm robotics research and is also the hardware used in the real-world experiments described in Chapter 7. Since the default ARGoS configuration defines one control step as 100 ms, there are 10 control steps per second. To achieve 1 Mbps (equivalent to 125 kB per second), we set the packet size to 12.5 kB, ensuring that 10 packets per second result in the desired throughput.

3.3 Problem Definition, Convergence, and Evaluation

3.3.1 Problem Definition

Consider a UAV swarm $\mathcal{S} = \{u_1, u_2, \ldots, u_n\}$ where each UAV u_i has a type t_i, runs software version v_i, and has communication range r_i. A software version v is compatible with UAVs whose type belongs to a predefined set $\mathcal{T}_v \subseteq \{t_1, \ldots, t_n\}$. UAVs with $t_i \notin \mathcal{T}_v$ are excluded from receiving it. Let $\mathcal{S}_v \subseteq \mathcal{S}$ represent the subset of UAVs compatible with version v. The goal is to distribute the latest software version v_{new} to all UAVs in $\mathcal{S}_v$, ensuring update coverage with minimal latency and overhead.

In a heterogeneous swarm, updates are not uniform. Depending on the triggering event, an update operation may be of software type, model type, or both. Formally, let $\mathcal{S}_v \subseteq \mathcal{S}$ denote the set of UAVs eligible for the software update, and $\mathcal{S}_m \subseteq \mathcal{S}$ denote those carrying an imaging sensor subject to a model update. These sets are independent: a UAV $u_i \in \mathcal{S}_v \cap \mathcal{S}_m$ is subject to both update operations, each treated separately in terms of payload, propagation, and optimization objectives. Consider a

[1] `https://www.bitcraze.io/products/crazyradio-2-0`

model $\mathcal{M}$ capable of classifying images into classes $L = \{l_1, l_2, \ldots, l_n\}$ with accuracy Acc. Updating the model from $\mathcal{M}$ to $\mathcal{M}^*$ generates a model patch $\mathcal{P}$ with size s. The update propagation problem is defined by two jointly pursued objectives:

Dissemination objective: Deliver the latest software version v_{new} to all UAVs in $\mathcal{S}_v$ and the model patch $\mathcal{P}$ to all UAVs in $\mathcal{S}_m$, with minimal convergence time and transmission overhead.

Payload objective: Minimize the patch size s with minimal degradation in the updated model accuracy Acc^*.

SwarmSync and SwarmMARL both address the dissemination objective through different coordination philosophies. SwarmModelPatch addresses the payload objective independently of the coordination approach and is compatible with either.

3.3.2 Update Convergence

Let P be the set of packets required to update to the new software version v_{new} and $\mathcal{S}_v \subseteq \mathcal{S}$ the set of target UAVs. The swarm is said to have *converged* at control step t if, for every UAV $u \in \mathcal{S}_v$, all packets $p \in P$ have been received and acknowledged by the end of step t. This constitutes the shared termination condition for both SwarmSync and SwarmMARL: no further dissemination or coordination is required post-convergence. Under the assumptions of eventual connectivity and bounded packet loss, and given that no circular acknowledgement dependencies exist among UAVs, deadlock is precluded and convergence is guaranteed in finite time; this is addressed formally for SwarmSync in Section 4.2.2.

In SwarmSync, convergence is explicitly confirmed through a chain of ACK signals propagating from followers to leaders and from leaders to the Updater, producing a verified global convergence event. In SwarmMARL, convergence is implicit: the environment transitions to a terminal state when all agents in $\mathcal{S}_v$ reach the target software version and the episode ends. Both definitions are operationally equivalent, the swarm has converged when every compatible agent holds the complete update, but the mechanisms by which convergence is detected and confirmed differ between the two approaches.

3.3.3 Assumptions

Both SwarmSync and SwarmMARL are designed and evaluated under the following shared assumptions. Deviations from these assumptions are discussed as limitation in Chapter 8.

Eventual Connectivity

While we do not assume that uninterrupted or continuous communication links exist among all UAVs at all times, as we acknowledge that network partitions, mobility, or environmental factors may temporarily disrupt connectivity, we do assume that the system will eventually achieve sufficient connectivity over time [20, 19, 7]. Specifically, every UAV will, with enough time, be able to communicate directly or indirectly with all others in the swarm, enabling the eventual dissemination and acknowledgement of critical information such as update packets. This assumption of eventual connectivity is standard in the distributed systems and mobile ad-hoc network literature, where it

is commonly referred to as "eventual connectivity" or "weak connectivity". In UAV swarm research, many prior works assume more stable connections, such as strong connectivity where "there is a path in each direction between each pair of vertices of the graph" [27], or full connectivity where "every node can exchange messages with every other node" [23]. Eventual connectivity is considered realistic because UAVs are mobile and can actively reposition to restore links, and because multi-hop relaying significantly reduces the likelihood of prolonged isolation. Permanent and irrecoverable partitions are treated as out-of-scope failure modes.

Bounded Packet Loss

The packet loss rate satisfies $f \in [0, 1)$, meaning that at least a nonzero fraction of transmitted packets is successfully delivered over time. This assumption is standard in lossy network models and is well-supported empirically: even in congested or interference-prone wireless environments, measured packet delivery ratios rarely approach total loss [123, 31]. A loss rate of $f = 1$ would imply a permanently broken channel, which falls under the category of irrecoverable failure and is out of scope. We evaluate both approaches across failure rates $f \in \{0, 0.25, 0.50, 0.75\}$, following prior work on homogeneous swarm updates [112].

3.3.4 Evaluation Metrics

Three metrics are used consistently across all experiments in this thesis to enable fair and direct comparison between SwarmSync and SwarmMARL, and to evaluate SwarmModelPatch independently of the coordination approach.

Convergence Time

Convergence time is the number of ARGoS control steps from the Updater's announcement of an available update until global convergence, i.e., when all compatible UAVs in $\mathcal{S}_v$ have applied the update. Each ARGoS step is 100 ms [87], allowing direct conversion to wall-clock time. The update consists of two components: a model patch generated by SwarmModelPatch delivered to Eyebots, and a small task-level configuration update ($\approx$500 bytes) sent to Footbots. The swarm declares convergence only after both types of updates have been fully disseminated and applied across their respective target subsets. To assess scalability, we report both total convergence time and convergence time per UAV, computed by dividing total update time by the number of updated drones.

Transmission Overhead

Transmission overhead is the total data exchanged across the swarm until convergence, reported in megabytes (MB). Overhead includes both payload packets (including retransmissions) and control messages such as acknowledgements, leader election signals, and convergence notifications. Control messages are short, ranging from 1 to 10 bytes and averaging 5 bytes each. Overhead reflects both bandwidth usage and the energy consumed by UAV radios during the update process. We report both total overhead and overhead per drone to support scalability analysis.

Patch Size

Patch size is the serialized difference between the updated model $\mathcal{M}^*$ and the prior model $\mathcal{M}$, measured in MB. Patch size depends on the number of frozen layers during SwarmModelPatch fine-tuning (Chapter 4): freezing more layers reduces the number of changed parameters and therefore the payload size. For comparison, we also report the size of the full updated model, referred to as the "full-size update" baseline. Since transmitted data directly affects both convergence time and overhead, patch size links model-level design choices to system-level performance and applies equally regardless of which coordination approach is used to disseminate the patch.

Chapter 4

SwarmUpdate: Hierarchical Deterministic Coordination

This chapter presents SwarmUpdate, the first of two coordination approaches investigated in this thesis. SwarmUpdate addresses the update propagation problem defined in Chapter 3 through a deterministic, protocol-based philosophy: roles are explicitly assigned, communication is structured into a hierarchy, and delivery is guaranteed through ACK-based flow control. SwarmUpdate comprises two components. SwarmSync governs how the update payload is disseminated across the swarm. SwarmModelPatch governs how the model update payload is generated to be as small as possible before dissemination begins. Together they form an end-to-end pipeline, as illustrated in Fig. 4.1: SwarmModelPatch generates a minimal differential patch at headquarters, and SwarmSync ensures that this patch efficiently reaches only the compatible UAVs in the field.

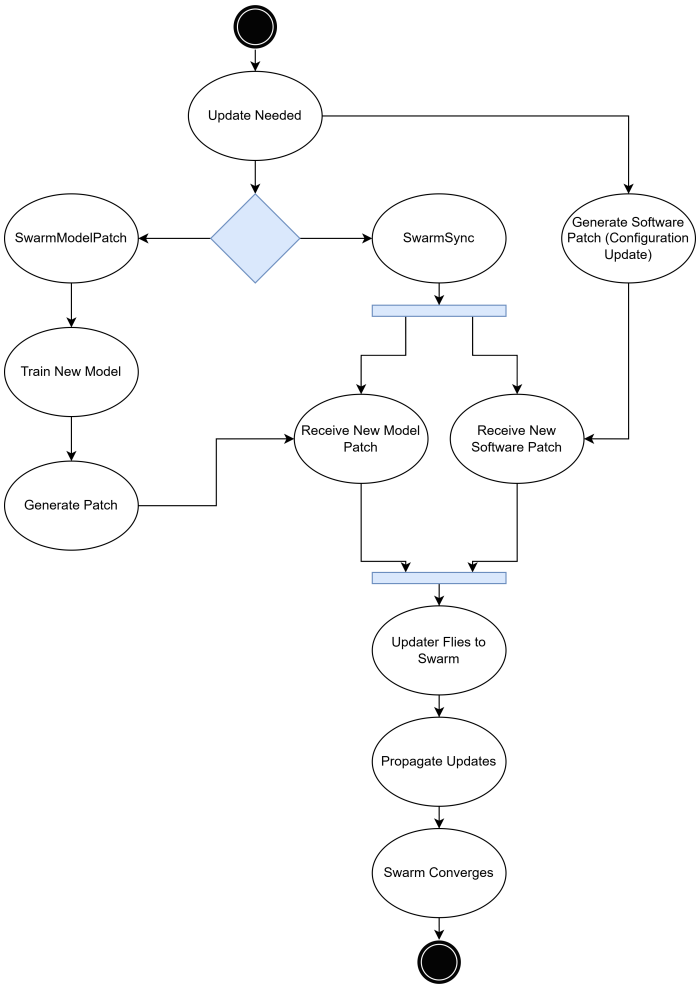


Figure 4.1: The SwarmUpdate pipeline. When an update is needed, SwarmModelPatch generates a minimal differential patch if a DL model update is required. SwarmSync then dispatches an Updater UAV to carry the patch to the swarm and propagate it to all compatible agents.

4.1 SwarmSync

SwarmSync ensures efficient and reliable software update synchronization for heterogeneous UAV swarms, distributing a software update payload whether it represents

an update to a new model version $\mathcal{M}^*$ or a more traditional code or data update. SwarmSync is inspired by the scalable SWARM data replication method [100, 71], where a designated supernode distributes data within server networks to other nodes sharing the same interest. SwarmSync divides the swarm into sub-swarms based on update compatibility: only UAVs in $\mathcal{S}_v$ (i.e., those whose type $t_i \in \mathcal{T}_v$) are included in the sub-swarms for a given update v. Incompatible UAVs receive no update payload. Each UAV belongs to at most one sub-swarm per update, determined by its type. When $|\mathcal{S}_v|$ is large, multiple sub-swarms may be formed for the same update to avoid excessively unbalanced groups, each appointing its own leader either randomly or based on predefined rules such as the lowest UAV identifier. Updates propagate via a structured and hierarchical mechanism based on three distinct roles: Updater, Leader, and Follower.

To ensure guaranteed swarm convergence, SwarmSync uses TCP-style ACK-based flow control [16]: every packet sent requires an acknowledgement before the sender advances to the next packet. Since each leader tracks its assigned followers, this guarantees that all drones receive all update packets, which is critical in mission-critical or safety-critical situations [122]. To mitigate the deadlock risk inherent in ACK-based protocols, SwarmSync implements a timeout mechanism: if a drone is unresponsive within a given time frame, it is removed from the leader's follower list; if a leader is unresponsive, the sub-swarm appoints a new leader in its place. Although a timeout did not occur during simulation, such preventive measures are essential for safety-critical systems.

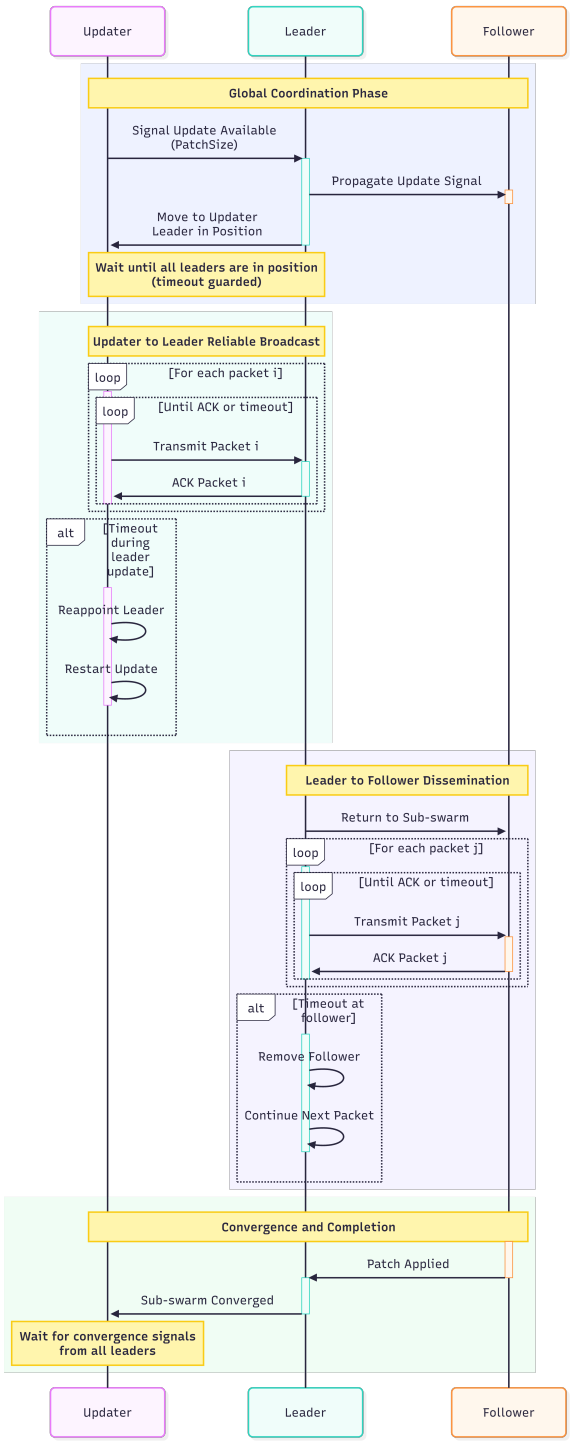


Figure 4.2: Sequence diagram of the SwarmSync process.

Fig. 4.2 shows the sequence diagram [76] for the SwarmSync process. When an update is initiated, the Updater first signals the swarm that an update is available. The swarm propagates this message, after which leaders move toward the Updater to receive the update. Once all leaders are updated, they return to their sub-swarms and distribute the update to their followers. When all leaders and followers have received the update, they signal the Updater that the swarm has converged.

Algorithm 1 SwarmSync's Updater

1: **Signal:** Update is available
2: Wait until all leaders get in position ▷ Until Time-out
3: **for** each packet in packets **do**
4: **repeat**
5: Transmit current packet to all leaders
6: **until** ACK received from all leaders ▷ Until Time-out
7: **end for**
8: **if** Time-out **then**
9: **Signal** Reappoint Leader
10: **Restart** Update Process
11: **end if**
12: Wait until convergence signals received from all leaders

Algorithm 2 SwarmSync's Leader

1: **Receive** signal from Updater
2: $PatchSize \leftarrow$ from signal, propagate signal to swarm
3: Move toward and surround Updater
4: **Signal:** leader in position
5: **Receive** packet from update
6: **while** #packets $<$ $PatchSize$ **do**
7: Wait to receive the packet ▷ Until Time-out
8: **if** packet received successfully **then**
9: **Send** ACK
10: **end if**
11: **if** Time-Out **then**
12: **Abort** Update process, signal others to also abort
13: **end if**
14: **end while**
15: Return to respective sub-swarm
16: **for** each packet in packets **do**
17: **repeat**
18: Transmit current packet to all followers
19: **until** ACK received from all followers ▷ Until Time-out
20: **if** Time-out **then**
21: **Remove** Unresponsive follower from follower list
22: **Continue** Next packet
23: **end if**
24: **end for**
25: **Signal:** Notify Updater that sub-swarm has converged

Algorithm 3 SwarmSync's Follower

```
1: Receive signal from Updater
2: PatchSize ← from signal, propagate signal to neighbours
3: Remain idle until leader returns with update
4: while #packets < PatchSize do
5:     Wait to receive the packet                          ▷ Until Time-out
6:     if Time-out then
7:         Signal Time-out, reappoint leader in the swarm
8:     end if
9:     if packet received successfully then
10:        Send ACK
11:    end if
12: end while
13: Apply Patch
14: Signal: Notify leader completion of update
```

Updater. As presented in Algorithm 1, the Updater signals the swarm that an update is available, broadcasting metadata including version details and update packet size. It then waits for leaders to signal readiness. The Updater sequentially sends update packets and waits for ACKs from all leaders before proceeding. After completing transmission, it waits for convergence signals from all leaders before declaring the swarm converged. Timeouts in either waiting phase trigger leader reappointment and restart of the update process.

Leader. As shown in Algorithm 2, upon receiving an update signal, the leader

of each sub-swarm moves to the Updater's position and forms a predefined spatial formation to get within communication range. Once in position, leaders signal readiness, sequentially receive packets from the Updater, and send an ACK for each packet. After receiving all packets, leaders return to their sub-swarms and distribute the update to followers using the same ACK-based approach. If a leader times out while receiving from the Updater, it aborts and propagates an abort signal. If a leader times out while awaiting follower ACKs, the non-responding follower is removed and the update continues for remaining followers.

Follower. As shown in Algorithm 3, followers propagate the update signal to their neighbors to expedite awareness across the swarm. While leaders travel to the Updater, followers continue their mission activities but remain within the communication range of their leader's last known position. Followers send an ACK for each received packet. Once all packets are received, followers apply the patch and notify their leader. A timeout causes the follower to signal for leader reappointment.

Time and Communication Complexity. In ideal conditions, let $|\mathcal{S}_v|$ denote the number of UAVs requiring an update and N the maximum number of UAVs that can simultaneously communicate with the Updater (determined by communication range and spatial packing; in our setup $N = 18$, as shown in Fig. 3.1). SwarmSync organizes UAVs into a two-level hierarchy: the Updater communicates with at most $\lceil |\mathcal{S}_v|/N \rceil$ leaders, each serving up to N followers. Both Phase 1 (Updater to Leaders) and Phase 2 (Leaders to Followers) complete in $O(P)$ control steps due to parallelism across leaders and followers respectively, giving a total complexity of $O(P)$ for swarms of size $|\mathcal{S}_v| \leq N^2 + N$. For larger swarms, the hierarchy extends recursively,

giving $O(P \cdot \log_N |\mathcal{S}_v|)$. Communication overhead is $O(P \cdot |\mathcal{S}_v|)$, as each of the $|\mathcal{S}_v|$ agents receives P packets exactly once across the two phases. Gossip-based protocols achieve the same asymptotic time complexity but with larger constant factors due to redundant rebroadcasts, while SOUL's sequential grouping incurs $O(P \cdot |\mathcal{S}_v|/N)$ time.

Security. Standard protections such as TLS [56, 103] and DTLS [55] integrate cleanly with SwarmSync. Experiments with and without TLS encryption showed no impact on convergence time beyond a few bytes of per-message overhead (Algorithm 4), confirming that practitioners can adopt well-established cryptographic protections without sacrificing the benefits of SwarmSync.

Timeout Policy. The swarm's behavior upon encountering a timeout is mission-configurable, consistent with established practice in UAV swarm systems [81, 82]. In our experiments, we adopt a liveness-first policy: the update continues while excluding any UAVs that timed out, a posture widely adopted in fault-tolerant swarm frameworks [17, 83, 131]. A rollback mechanism is available for both update types: the model patch is retained until full convergence and can be reversed by applying it in the negative direction; the prior software binary is preserved until convergence is confirmed.

Algorithm 4 SwarmSync's TLS

Precondition: Each node has long-term key pair and certificate; time sync within tolerance
Roles: Updater/Leader acts as TLS Server; Leaders/Followers act as TLS Clients
for each peer in {Server, Client} **do**
    repeat
        Client→Server: ClientHello {ciphers, groups, nonce_C}
        Server→Client: ServerHello, Certificate, CertificateRequest, ServerKeyShare
        Client: Verify Certificate ▷ Until Time-out
        if certificate invalid **or** unsupported cipher/group **then**
            Abort Handshake; **Signal** Security Error
        else
            Client→Server: ClientKeyShare, Certificate, CertificateVerify, Finished
            Server: Verify client Certificate ▷ Until Time-out
            if verification fails **then**
                Abort Handshake; **Signal** Security Error
            else
                Both: Derive shared traffic keys
                Server→Client: Finished
                Both: Switch to encrypted record layer
                Install SessionContext per peer
            end if
        end if
    until TLS session established **or** Time-out
    if Time-out **then**
        Remove Unresponsive peer; **Continue** next peer
    end if
end for
Postcondition: Authenticated, encrypted channels for all active peers

4.2 Formal Analysis of SwarmSync

This section establishes the correctness, termination, and resilience guarantees of SwarmSync under the shared assumptions stated in Chapter 3.

4.2.1 Scalability of SwarmSync

Let $n = |\mathcal{S}_v|$ denote the number of UAVs requiring update, P the number of packets in the payload, and N the maximum number of UAVs that can simultaneously communicate with a single parent (Updater or leader), fixed by communication range and spatial packing as in Section 4.1 ($N = 18$ in our setup). We use the same branching factor N at every level; there is no separate abstract degree k.

Hierarchy height. SwarmSync organizes UAVs into an N-ary dissemination tree rooted at the Updater. The two-level case of Section 4.1 ($n \leq N^2 + N$) is the depth-2 instance; for $n > N^2 + N$ the tree is extended recursively, each follower at the deepest level becoming a sub-leader for up to N children. Since each parent serves at most N children, a tree of depth h holds UAVs at every level, giving a total capacity

$$n_{\max}(h) = N + N^2 + \cdots + N^h = \frac{N\big(N^h - 1\big)}{N - 1}.$$

The exact minimum height is therefore the smallest integer h satisfying $n \leq \dfrac{N\big(N^h - 1\big)}{N - 1}$, which recovers the $n \leq N^2 + N$ boundary at $h = 2$. Because this geometric sum is dominated by its top term N^h up to the constant factor $N/(N-1)$, solving for h

yields $h(n) = \log_N n + O(1)$, so

$$h(n) = \Theta(\log_N n),$$

since each additional level multiplies reachable UAVs by N. The N-packing constraint applies at every internal node: a sub-leader can serve at most N children in range, identical to the constraint on the Updater.

Per-level cost. Within one level, all parents transmit concurrently. This concurrency is well-defined despite the N-packing constraint: the constraint bounds how many *children* a single parent can serve within its own range, not how many parents may be active swarm-wide. Distinct parents at the same level head disjoint, spatially separated subtrees, so their transmissions reuse the medium spatially; where subtrees are close enough to interfere, we assume orthogonal channel assignment among sibling parents (as described in Section 4.1). Each parent still sends its P packets *sequentially*, because ACK-based flow control requires an ACK from all children before advancing to the next packet (Section 4.1). A single level therefore costs $O(P)$ control steps.

Total dissemination time. Multiplying height by per-level cost,

$$T(n) = \Theta(h(n)) \cdot \Theta(P) = \Theta(P \cdot \log_N n),$$

which reduces to $O(P)$ for $n \leq N^2 + N$, matching the two-phase bound of Section 4.2. This is the bound in the idealized regime of Section 4.2 (no packet loss, instantaneous movement); movement and retransmission costs are treated below and

in Section 4.2.2. In particular, with the bounded-loss model of Section 4.2.2, each per-link transmission is retried a geometrically distributed number of times with mean $1/(1-f)$, so the *expected* dissemination time becomes

$$\mathbb{E}[T(n)] = O\left(\frac{P \cdot \log_N n}{1-f}\right),$$

an expected-time bound rather than a deterministic step count. The packet count P is thus retained consistently: it appears in the two-level bound of Section 4.1, in the recursive bound above, and in the lossy-regime expectation.

4.2.2 Almost-sure Termination of SwarmSync

We establish that SwarmSync terminates almost surely [18] in finite expected time under two assumptions, each of which we justify below.

- **Eventual connectivity:** Despite possible temporary disconnections due to mobility, obstacles, or dynamic network conditions, every UAV $u \in S$ will eventually be able to exchange messages with all others, either directly or through multi-hop relays. This assumption is standard in the distributed system and mobile ad-hoc network literature [20, 19, 7], where it is commonly referred to as "eventual connectivity" or "weak connectivity". In UAV swarm research, many prior works assume more stable connections than our assumption, such as "strong connectivity" where "there is a path in each direction between each pair of vertices of the graph" [27], or even fully connected where "every node can exchange messages with every other node" [23]. It is considered realistic in UAV

swarm deployments because UAVs are mobile and can actively reposition to restore links, and because multi-hop relaying significantly reduces the likelihood of prolonged isolation. While transient disconnections are expected, permanent and irrecoverable partitions are treated as out-of-scope failure modes.

- **Bounded Packet Loss:** We model packet loss as independent Bernoulli trials with a stationary per-attempt loss probability $f \in [0, 1)$: each transmission attempt succeeds independently of all others with probability $1 - f$, and f does not vary across attempts. The condition $f < 1$ ensures that a nonzero fraction of transmitted packets is successfully delivered over time. This assumption is standard in lossy network models and is well-supported empirically: even in congested or interference-prone wireless environments, measured packet delivery ratios rarely approach total loss [123, 31]. A loss rate of $f = 1$ would imply a permanently broken channel, which falls under the category of irrecoverable failure. We note that this independent-Bernoulli model is an idealization: real wireless links exhibit temporally correlated, bursty losses, and a correlated-loss model would require a different retry analysis than the geometric one below.

Under these assumptions, SwarmSync terminates almost surely in finite expected time, and deadlock cannot occur, as we show below. We emphasize that this is a probabilistic-liveness guarantee: we establish "almost-sure termination" [18] with finite expected running time, not a deterministic bound on the number of steps. The almost-sure, in-expectation, and high-probability distinction we draw here is standard in the randomized-consensus and gossip literature, where these are treated as separate guarantees [13].

We first show that deadlock is impossible. Suppose for contradiction that a deadlock exists, i.e., there exists a set of UAVs $\mathcal{S}_d \subseteq \mathcal{S}$ such that every $u \in \mathcal{S}_d$ is waiting on some other $u' \in \mathcal{S}_d$ before making progress. In SwarmSync, the only blocking operation is awaiting an ACK. However, ACKs are purely reactive: UAV u' emits an ACK if and only if it has received a packet, an action that requires no response from any other UAV. Therefore u' is never blocked, contradicting the assumption that every member of $\mathcal{S}_d$ is waiting. Hence no such $\mathcal{S}_d$ exists, and deadlock is impossible. However, Updater failure during the Updater-to-Leader phase represents an unhandled failure mode: since all Leaders depend on the Updater as the sole update source, its failure before transmission completes causes the entire update session to abort. We identify this as a limitation and leave fault-tolerant Updater recovery as future work.

We now show termination. Since $f < 1$, each retransmission succeeds independently with probability $1 - f > 0$. The number of retries required to deliver a given packet over a given link is therefore geometrically distributed: the probability that delivery has not yet succeeded after r retries is f^r, which converges to 0 as $r \to \infty$, so delivery occurs after finitely many retries with probability one. Moreover, a geometric random variable with success probability $1 - f$ has finite mean $1/(1 - f)$, so the expected number of retries per link, and hence the expected time to deliver each packet is finite. By the eventual connectivity assumption, every UAV is eventually reachable. Since the swarm is finite ($|\mathcal{S}| < \infty$) and each UAV eventually receives and acknowledges every required packet in finite expected time, the protocol terminates almost surely, with finite expected running time. We note that this is an

almost-sure, expected-time guarantee rather than a deterministic step bound; a deterministic bound would require an additional assumption capping consecutive losses on any link, which we do not impose.

4.2.3 Failure Resilience

SwarmSync employs ACK-based retransmission with timeouts. Under the packet drop model, the probability that a message fails after r retries is f^r. To guarantee delivery with probability at least $1 - \delta$, we require:

$$f^r \leq \delta \implies r \geq \lceil \log_{1/f}(1/\delta) \rceil$$

which gives the minimum number of retries guaranteeing delivery with probability at least $1 - \delta$.

4.3 SwarmModelPatch

Once the payload of a software update is synchronized across all UAVs in a swarm, it needs to be installed. For traditional code/data updates, UAVs would typically replace the previous version's code or data. However, in the case of a deep learning model update, a naive replace strategy would require an entire new model version to be re-trained or fine-tuned, then sent in the payload of an update. This would not scale for typical swarm settings, where only intermittent peer-to-peer communication is available.

To effectively deal with model updates and reduce the costs of retraining from

scratch, we propose SwarmModelPatch, which integrates ideas from lifelong learning [77, 79], transfer learning [49, 74], and incremental learning [88]. As shown in Algorithm 5, SwarmModelPatch maintains a replay buffer that holds representative samples from previously seen training data. When an environmental change occurs, the model update incorporates both the replay buffer and new data to prevent catastrophic forgetting [77] and allows rapid adaptation.

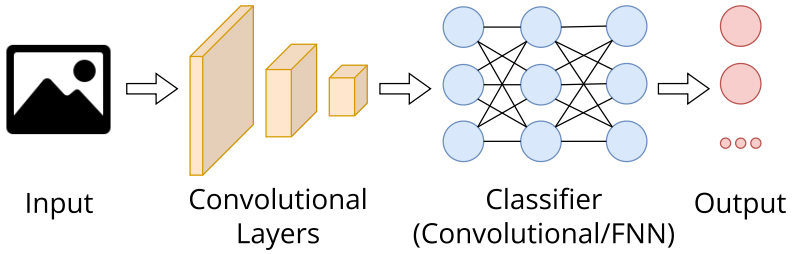


Figure 4.3: Structure of a typical CNN model.

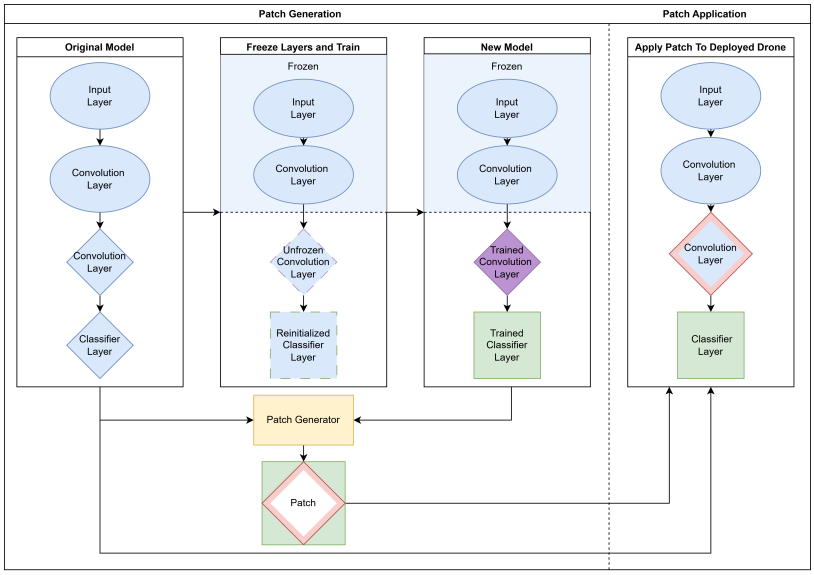


Figure 4.4: The model has its earlier layers frozen (weights unchanged), its later convolutional layers unfrozen and updated during training, and its final classification layer reinitialized with random weights, as opposed to the unfrozen layers, which retain and build on their existing weights.

A typical CNN model is shown in Fig. 4.3. A CNN model usually consists of some trainable convolutional layers, untrainable pooling layers, and a classifier that can be either a feed-forward neural network (FNN) or convolutional layer(s). The output of the classifier is used to determine the classification results. To efficiently enable a given model to classify data according to newly introduced classes, SwarmModelPatch resets the classifier layer [101, 28], while partially freezing the remaining layers and training the rest, as shown in Fig. 4.4. Once a new model version is trained, SwarmModelPatch generates a model patch representing the differences between the new and old model versions, essentially only containing the changed layers and metadata about these layers, while ignoring the frozen layers.

Algorithm 6 and Algorithm 7 present the pseudo-code to create and apply the

Algorithm 5 SwarmModelPatch

Input: Old CNN model $\mathcal{M}$, Replay Buffer Size β, Trainable Layers x
Output: Updated model $\mathcal{M}^*$, Patch file $\mathcal{P}$
Prepare training data:
  $\mathcal{D} \leftarrow$ ReplayBuffer (old data) $\cup$ New Data
Load model $\mathcal{M}$
Reinitialize classifier layer $\mathcal{C}$ in $\mathcal{M}$
Freeze all layers except:
  - Last x convolutional layer(s) ▷ Adjust accordingly
  - Classifier $\mathcal{C}$
$\mathcal{M}^* \leftarrow$ Train $\mathcal{M}$ on $\mathcal{D}$ ▷ Update trainable parameters
Initialize empty dictionary $\mathcal{P}$
Iterate through parameters of $\mathcal{M}^*$:
for each parameter p **in** $\mathcal{M}^*$ **do**
  if $p \in \mathcal{M}$ **and** shape($\mathcal{M}[p]$) = shape($\mathcal{M}^*[p]$) **then**
    if $\mathcal{M}^*[p] \neq \mathcal{M}[p]$ **then** ▷ Store delta for changed params
      $\mathcal{P}[p] \leftarrow \mathcal{M}^*[p] - \mathcal{M}[p]$
    end if
  else ▷ New or reshaped parameter
    $\mathcal{P}[p] \leftarrow \mathcal{M}^*[p]$
  end if
end for
Save $\mathcal{P}$ to patch file
Return Updated model $\mathcal{M}^*$, Patch file $\mathcal{P}$

patch. If a parameter p in the patch $\mathcal{P}$ has the same shape as the corresponding parameter in model $\mathcal{M}$, it is added to $\mathcal{M}[p]$. However, if p does not exist in model $\mathcal{M}$ or has a different shape, $\mathcal{M}[p]$ is replaced by $\mathcal{P}[p]$.

By coupling SwarmSync's hierarchical fault-tolerant dissemination with SwarmModelPatch's differential payloads, the combined framework addresses both traditional software updates and deep learning model evolution. Updates are delivered swiftly via leaders and followers, and only the minimal changed parameters rather

Algorithm 6 Generate Model Patch

Input: Old model $\mathcal{M}$, New model $\mathcal{M}^*$
Output: Patch file $\mathcal{P}$
Initialize empty dictionary $\mathcal{P}$
Iterate through parameters:
for each parameter p **in** $\mathcal{M}^*$ **do**
    if $p \in \mathcal{M}$ **and** shape($\mathcal{M}[p]$) = shape($\mathcal{M}^*[p]$) **then**
        if $\mathcal{M}^*[p] \neq \mathcal{M}[p]$ **then** ▷ Store delta for changed params
            $\mathcal{P}[p] \leftarrow \mathcal{M}^*[p] - \mathcal{M}[p]$
        end if
    else ▷ New or reshaped parameter
        $\mathcal{P}[p] \leftarrow \mathcal{M}^*[p]$
    end if
end for
Save $\mathcal{P}$ to patch file
Return patch file $\mathcal{P}$

than entire model weights traverse the network. This synergy minimizes mission interruption and seamlessly adapts to evolving operational needs in the field.

4.3.1 Formal Analysis: Layer Freezing

Let $L = \{l_1, l_2, \ldots, l_n\}$ denote the ordered set of layers in the CNN, where l_1 is the input layer and l_n is the classifier layer. Let P_i denote the number of trainable parameters in layer l_i and $P = \sum_{i=1}^{n} P_i$ the total parameter count. SwarmModelPatch partitions L into a frozen set $L_f = \{l_1, \ldots, l_k\}$ and a trainable set $L_t = \{l_{k+1}, \ldots, l_n\}$, where $k < n$ is the freezing boundary.

Let $P_f = \sum_{l_i \in L_f} P_i$ denote the number of frozen parameters. The patch size S

Algorithm 7 Apply Model Patch

Input: Base model $\mathcal{M}$, Patch file $\mathcal{P}$
Output: Updated model $\mathcal{M}^*$
for each parameter p **in** $\mathcal{P}$ **do**
    if $p \in \mathcal{M}$ **and** shape$(\mathcal{M}[p])$ = shape$(\mathcal{P}[p])$ **then**
        $\mathcal{M}^*[p] \leftarrow \mathcal{M}[p] + \mathcal{P}[p]$ ▷ Add delta to existing parameters
    else
        $\mathcal{M}^*[p] \leftarrow \mathcal{P}[p]$ ▷ Replace new or reshaped parameters
    end if
end for
Return updated model $\mathcal{M}^*$

transmitted during an update is proportional to the number of trainable parameters:

$$S = \alpha \cdot (P - P_f)$$

where α is a constant encoding factor (e.g., bytes per parameter). Increasing k reduces S at the cost of model expressivity; decreasing k increases adaptability at the cost of larger patch size. This relationship provides a formal basis for mission-aware tuning, as discussed in Chapter 8.

4.4 Illustrative Example: End-to-End SwarmUpdate Workflow

To illustrate the integrated operation of SwarmSync and SwarmModelPatch, we revisit the wildfire monitoring scenario introduced in Section 1.2. A heterogeneous UAV swarm S patrols a remote mountainous region. The swarm consists of two UAV

types: (i) *Footbots*, which execute rule-based navigation and lack visual sensing, and (ii) *Eyebots*, which are equipped with cameras and execute a convolutional neural network (CNN) model $\mathcal{M}$ for visual anomaly detection. Only Eyebots participate in learning-based perception, forming the subset $S^m \subset S$. Initially, the model $\mathcal{M}$ classifies observed scenes into three labels, $L = \{clear, cloud, haze\}$.

During patrol, an Eyebot detects anomalous thermal signatures indicative of a nascent ignition event. The reporting UAV relays this observation to headquarters, where analysts confirm wildfire onset and mandate a model update to recognize smoke plumes and active fire fronts as new semantic classes, denoted l_{smoke} and l_{fire}. The target label set therefore becomes $L^* = L \cup \{l_{smoke}, l_{fire}\}$.

Step 1: Model Patch Generation via SwarmModelPatch. At headquarters, the original model $\mathcal{M}$ is fine-tuned using newly collected wildfire imagery, augmented with a replay buffer containing historical *clear*, *cloud*, and *haze* samples to mitigate catastrophic forgetting. The classifier layer is reinitialized to support five output classes, while only the final convolutional layer and classifier weights are unfrozen during training. This selective update strategy produces a differential model patch $\mathcal{P}$ containing approximately 0.7 MB of parameter deltas, compared to 2.8 MB for the full SqueezeNet model. The patch is packaged with metadata specifying compatibility with Eyebots only and a version identifier v_{fire}.

Step 2: Hierarchical Dissemination via SwarmSync. An Updater UAV departs headquarters carrying $\mathcal{P}$ and initiates the dissemination protocol. Upon broadcasting an update signal, Footbots ignore the transmission as incompatible, while

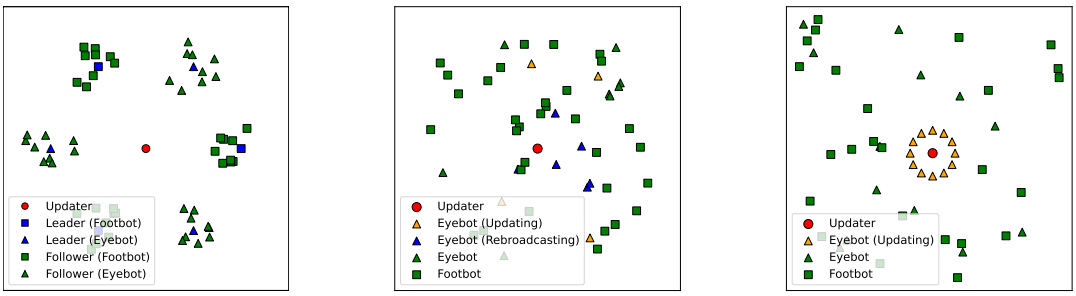
Eyebots self-organize into sub-swarms and elect leaders using a lowest-ID rule.

Dissemination proceeds in two phases. In Phase 1, elected leaders navigate toward the Updater and form a tight ring within communication range. The Updater transmits $\mathcal{P}$ as 64 packets using an ACK-based reliable protocol. If packet loss occurs, such as due to turbulence-induced disconnection, a 20-step timeout triggers leader re-election and resumes transmission. In Phase 2, leaders return to their respective sub-swarms and broadcast $\mathcal{P}$ to followers using the same protocol.

Followers apply the patch in place via weight-delta accumulation, as specified in Algorithm 7, thereby avoiding full model reload overhead. Within a short time, all Eyebots acknowledge successful completion of the update and begin executing the updated model $\mathcal{M}^*$. The swarm can now reliably distinguish smoke from clouds and adapt flight paths to avoid thermal updrafts, demonstrating rapid and bandwidth-efficient adaptation to emergent environmental conditions.

4.5 Empirical Evaluation

This section presents the empirical evaluation of SwarmSync and SwarmModelPatch against their respective baselines. The experimental setup — simulation environment, swarm configuration, communication parameters, and swarm behavior — is described first, followed by the baseline strategies and results.



(a) SwarmSync (b) Gossip (c) SOUL

Figure 4.5: Swarm UAV formations and roles for SwarmSync and the two baseline software update synchronization strategies.

4.5.1 Baseline Strategies

To empirically evaluate SwarmSync and SwarmModelPatch, we select state-of-the-art baseline techniques. For SwarmSync, we use a gossip-based rebroadcast strategy and an auction-based strategy (SOUL), both illustrated in Fig. 4.5. For SwarmModelPatch, we use a full-size update patch as the baseline.

Gossip-Based Rebroadcast Strategy

The gossip-based rebroadcasting method follows the principles of epidemic dissemination, modeled by the classic Susceptible-Infected-Recovered (SIR) epidemic model [21]. This model enables information to spread across the swarm with minimal time consumption, even in unreliable UAV communication networks where packet loss and intermittent connectivity are common.

Varadharajan et al. [112] use such a gossip-based rebroadcasting approach to update a homogeneous swarm. In this strategy, if a receiver fails to compile the update

due to incomplete packets, it requests retransmission from the sender, which resends all packets. Once a drone successfully receives all packets, it transforms into an Updater, actively participating in further dissemination. However, since Updaters are unaware of drones around them, they cannot determine when an update is finished. Gossip-based strategies can also generate high transmission overhead due to redundant rebroadcasting, making them inefficient for bandwidth-constrained environments. Furthermore, these protocols do not differentiate between heterogeneous types of UAVs, leading to unnecessary information being propagated to drones that do not require it.

As Gossip is designed for homogeneous swarms, we adapt it to heterogeneous swarms by introducing a simple modification: drones that are not interested in the update will pass on the update message but will not receive or rebroadcast the update, as shown in Algorithm 8. A similar timeout system is introduced: if none of the broadcasting drones receives an update request in 20 control steps, the swarm is declared to have converged.

Auction-Based Strategy (SOUL)

Auction-based strategies optimize data distribution and overhead by allowing UAVs to bid for content and only transmit the necessary data. SOUL [114] proposes an auction-based method for data sharing in a swarm, where an auctioneer handles the data blob and auctions it out to the swarm; any UAV with available data storage and computation will bid on this blob. Based on the bidding, the auctioneer decides how to split and distribute the blob to UAVs inside the swarm. Although auction-based

Algorithm 8 Gossip

```
1: Receive signal from Updater
2: PatchSize ← from signal, propagate signal to neighbours
3: Receive update packets
4: if #packets < PatchSize then                          ▷ Packet loss detected
5:     Request retransmission
6:     Receive update packets
7: else
8:     rebroadcasting ← true
9: end if
10: if rebroadcasting then
11:     for each packet in update do
12:         Send packet
13:         if request received then
14:             Retransmit packets
15:         else
16:             Signal convergence
17:         end if
18:     end for
19: end if
20: if signal_converged received then
21:     Pass it on
22: end if
23: if all nodes signal_converged then
24:     Converged
25: end if
```

strategies minimize redundant transmissions, they resemble a client-server model: when the number of bidders exceeds the maximum number of connections an auctioneer can handle, update times increase more linearly.

To adapt SOUL for updating a heterogeneous swarm, we set the Updater as the auctioneer and all UAVs as bidders. As shown in Algorithm 9 and Algorithm 10, the Updater broadcasts the entire update, then UAVs that lack certain packets request retransmission through bidding. UAVs whose positions are outside the Updater's

range move toward the Updater to receive the update. If more than N UAVs need the update, only N will be allowed to go to the Updater at a given time, causing the swarm to form egalitarian groups updated sequentially. If no request signal is received within 20 control steps, the Updater assumes the current group has finished receiving the update.

Algorithm 9 SOUL's Updater distributing update

```
1: Signal update is available
2: Signal update first group
3: Wait for signal UAVs are at location
4: Send update
5: while receiving requests do                          ▷ From UAVs
6:     Send requested packets                           ▷ To UAVs
7: end while
8: Proceed to next group
9: if all groups are done then
10:    Converged
11: end if
```

Algorithm 10 SOUL's UAV receiving update

1: **Receive** signal from Updater
2: $PatchSize \leftarrow$ from signal, propagate signal to neighbours
3: **Wait** for group's turn
4: **if** updating this group **then**
5: **Move** to update location
6: **Signal** at_location
7: **end if**
8: **Receive** packets ▷ From Updater
9: **while** #packets $<$ $PatchSize$ **do**
10: **Find** missing packets in self storage
11: **Request** missing packets
12: **Receive** packets
13: **end while**
14: **Signal** complete
15: **Move** away from location

Baseline DL Model Patching Method

No prior research has studied DL model patching for swarms. Since SwarmModelPatch offers an adjustable model patch size by adjusting the number of frozen layers, a SwarmModelPatch variant that freezes no layers during fine-tuning is equivalent to fine-tuning or retraining the entire model, followed by replacing the previously installed model as a whole. We therefore consider the SwarmModelPatch variant with

zero frozen layers as our baseline, which we refer to as the "full-size update". Compared with this worst-case payload, we isolate the benefits of differential patching in terms of reduced update latency and lowered transmission overhead. All DL patch experiments log both the convergence time and the final classification accuracy on the combined weather dataset.

4.5.2 Experimental Design

SwarmModelPatch Models

We select SqueezeNet [43], MobileNetV2 [24], and ResNet50 [37] as representative DL architectures spanning lightweight, mobile-optimized, and moderately deep models respectively, suitable for UAV deployment. SqueezeNet provides a deployment-realistic baseline with minimal computational and memory requirements [72]. MobileNetV2 employs inverted residual blocks and depthwise separable convolutions to achieve higher accuracy at moderate cost and is widely used in embedded vision systems [51, 110]. ResNet50 represents a deeper architecture commonly adopted in production pipelines and similar studies [5], allowing us to evaluate the scalability of SwarmModelPatch to larger models while remaining aligned with practical UAV compute constraints.

SwarmModelPatch Datasets

We evaluate SwarmModelPatch on two image classification datasets to assess its effectiveness under distinct concept drift scenarios. Weather-5 [6] provides a practical outdoor perception task derived from real drone-collected imagery, comprising five

weather conditions: Sunny, Cloudy, Foggy, Rainy, and Snowy. To simulate environmental concept drift, we exclude the Snowy class from the initial training distribution. As the original dataset lacks balanced class representation and a predefined train-test split, we apply data augmentation to balance all classes to 6,274 images, matching the largest class.

MNIST serves as a controlled benchmark to isolate the impact of model architecture and patching strategy from dataset complexity. We simulate concept drift by withholding digit 9 from the training distribution and introducing it during evaluation. Both datasets support image classification rather than more complex tasks such as object detection or semantic segmentation, avoiding the introduction of additional variables that would obscure the core trade-offs between patch size, accuracy recovery, and transmission overhead.

SwarmModelPatch Process

To create a controlled concept drift scenario, we withhold one class from each dataset during initial training (Snowy in Weather-5, digit 9 in MNIST), producing a deployed model with a known blind spot. We initially train on the remaining classes, splitting the data into train/test/val (0.7:0.15:0.15). We then apply SwarmModelPatch with a configurable number of frozen layers to generate a model patch for a new model version that includes the new class. The training data for the new version contains the new class combined with a replay buffer retaining 40% of the old training data. Each model was trained 3 times and the best model is kept. To clearly isolate the impact of SwarmModelPatch on update size and convergence time, we do not apply

quantization or other compression techniques that could further reduce patch sizes.

Communication Failure

We implement packet-loss scenarios as an all-or-nothing model using the ARGoS default configuration. A more detailed discussion of this modeling choice is provided in Chapter 8. We use packet failure rates $f \in \{0, 0.25, 0.50, 0.75\}$ as defined in Chapter 3, following the same values used in prior work on homogeneous swarms [112] to enable direct comparison.

4.6 SwarmUpdate Results

In this section, we provide the setup and findings for each of our research questions.

All reported values are means over $N = 20$ independent runs with different random seeds; error bars and $\pm$ values denote one standard deviation. Pairwise differences between strategies are assessed using the Mann–Whitney U test. To account for multiple comparisons, we apply Bonferroni correction for all 32 tests in each table, giving us $alpha = 0.05/32 \approx 0.00156$ as significance level. We report effect sizes using the rank-biserial correlation r, following the interpretation guidelines of Vargha and Delaney [116], we treat $|r| < 0.11$ as negligible, $0.11 \leq |r| < 0.28$ as small, $0.28 \leq |r| < 0.43$ as medium, and $|r| \geq 0.43$ as large. Any non-significant entries will have an effect size of "N/A" as they have no effect.

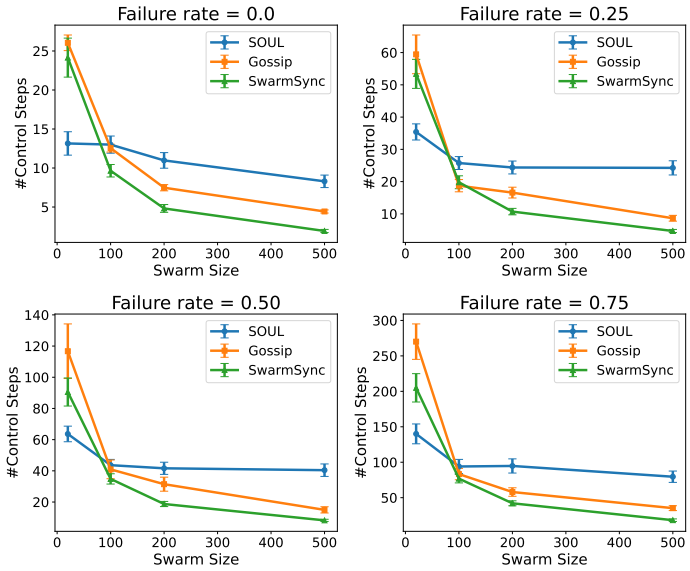


Figure 4.6: The average time taken **per drone** for each synchronization strategy to converge the swarm.

4.6.1 Dissemination Efficiency of SwarmSync

Setup.

To compare the efficiency of SwarmSync with the two baseline update synchronization strategies, we use the baseline DL model update method ("full-size update") instead of SwarmModelPatch, which is evaluated in Section 4.6.3. The efficiency of update synchronization is measured as the number of control steps (i.e., update time) from the start of update synchronization until the swarm declares convergence.

Fig. 4.7 shows the total time taken for the swarm to converge for each strategy, while Fig. 4.6 shows the average time taken per drone (i.e. the time each drone adds to the total convergence time).

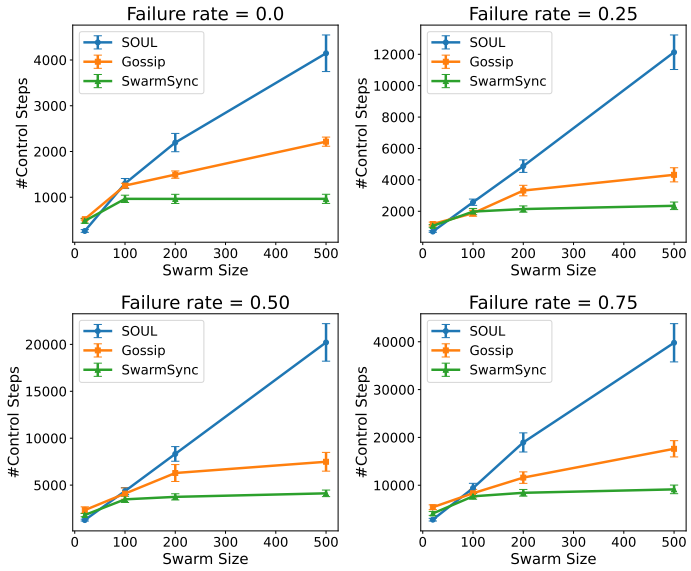


Figure 4.7: The **total** time taken for each synchronization strategy until convergence of the swarm

Findings

SwarmSync achieves the fastest convergence time in all strategies, outperforming SOUL by 78.3% and Gossip by 47.7% on average for the largest swarm size of 500. Fig. 4.6 presents the average time taken across all drones to converge for varying swarm sizes. Compared to SOUL and Gossip, SwarmSync demonstrates efficient update propagation. At the largest swarm size 500, in all failure rates, SwarmSync completes the update on average 78.3% faster compared to SOUL and on average 47.7% faster compared to Gossip. These results reinforce that SwarmSync is an efficient strategy for updating UAV swarms. However, SwarmSync shows minor inefficiencies for small swarms (e.g., 20), which is probably due to a limitation in ARGoS where packets and their ACKs cannot be handled in the same control step, leading to wasted control steps that would not occur in real-life settings;

this is discussed in more detail in Section 8.

SwarmSync scales efficiently with increasing swarm sizes. As the swarm size increases from 100 to 500 for failure rate 0.0, the average number of control steps decreased from 9.7 to 1.9 per drone for SwarmSync, showing a reduction by a factor of 5.1. This means that SwarmSync decreases the update synchronization time per drone approximately inversely proportional to the size increase; when the swarm size increases by a factor of 5 times, the time per drone decreases by a similar factor of 5.1, implying more predictable and efficient scaling. SOUL exhibits higher convergence times across all failure rates, showing a time-per-drone trend almost parallel to the x-axis after it almost stabilizes after swarm size 100. Although Gossip performs better than SOUL, it still shows a higher increase in convergence time as the swarm sizes increase.

In terms of **total** convergence times, shown in Fig. 4.7, while the total convergence time naturally increases with swarm size, SwarmSync's growth is significantly more gradual than both SOUL and Gossip. At swarm size 500 and failure rate 25%, SwarmSync's total convergence time remains 80.7% lower than SOUL and 55.6% lower than Gossip, demonstrating that the per-drone efficiency gains shown in Fig. 4.6 translate directly into overall swarm savings at scale.

Statistical Analysis

Table. 4.1 presents the Mann-Whitney U test results confirming the statistical significance of the convergence speed differences observed above.

SwarmSync's advantage over SOUL is statistically significant at almost

Table 4.1: Mann-Whitney U test results for SwarmSync vs SOUL and SwarmSync vs Gossip convergence speed across swarm sizes and failure rates. Effect size r is the rank-biserial correlation; negative values indicate the competing algorithm converges faster, positive values indicate SwarmSync converges faster. Non significant($p \geq alpha$) entries are highlighted in bold.

		SwarmSync vs SOUL		**SwarmSync vs Gossip**	
Failure Rate	**Swarm Size**	p	r	p	r
$f = 0.00$	20	<0.001	−0.980	**0.028**	**N/A**
	100	<0.001	0.830	<0.001	0.863
	200	<0.001	1.000	<0.001	0.990
	500	<0.001	1.000	<0.001	1.000
$f = 0.25$	20	<0.001	−1.000	**0.047**	**N/A**
	100	<0.001	0.750	**0.114**	**N/A**
	200	<0.001	1.000	<0.001	0.917
	500	<0.001	1.000	<0.001	1.000
$f = 0.50$	20	<0.001	−0.700	**0.081**	**N/A**
	100	**0.003**	**N/A**	**0.099**	**N/A**
	200	<0.001	1.000	<0.001	0.670
	500	<0.001	1.000	<0.001	0.910
$f = 0.75$	20	<0.001	−0.885	<0.001	0.765
	100	<0.001	0.770	**0.860**	**N/A**
	200	<0.001	0.990	<0.001	0.900
	500	<0.001	1.000	<0.001	1.000

all swarm sizes above 20, with no ambiguity about the crossover point. At swarm size 20, SOUL is faster across all failure rates ($r \leq -0.700$), but this reverses decisively from size 100, with effect sizes approaching or reaching $r = 1.000$ at sizes 200 and 500. With the sole exception of the marginal $f = 0.50$, size 100 case ($p = 0.003$), every comparison remains significant after correction, confirming that the transition occurs cleanly between sizes 20 and 100.

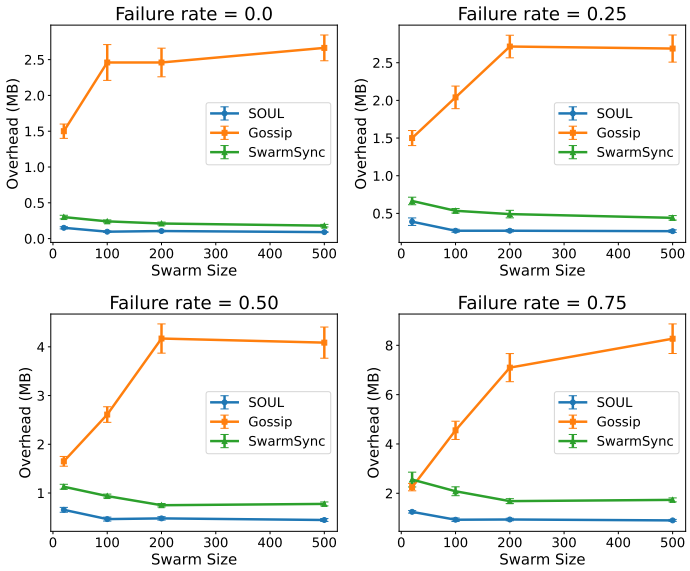


Figure 4.8: The average overhead **per drone** for each synchronization strategy until convergence of the swarm

SwarmSync's advantage over Gossip is significant at scale but narrows at small swarm sizes under high failure rates. Several small-swarm comparisons are non-significant: at $f = 0.50$, sizes 20 and 100 ($p = 0.081$ and $p = 0.099$), at $f = 0.75$, size 100 ($p = 0.860$, $r = -0.035$), and at size 20 under low failure rates ($f = 0.00$ and $f = 0.25$, $p = 0.028$ and $p = 0.047$). This reflects Gossip's redundant broadcast temporarily compensating for packet loss at small scales. From size 200 onwards, SwarmSync dominates across all failure rates with near-perfect effect sizes.

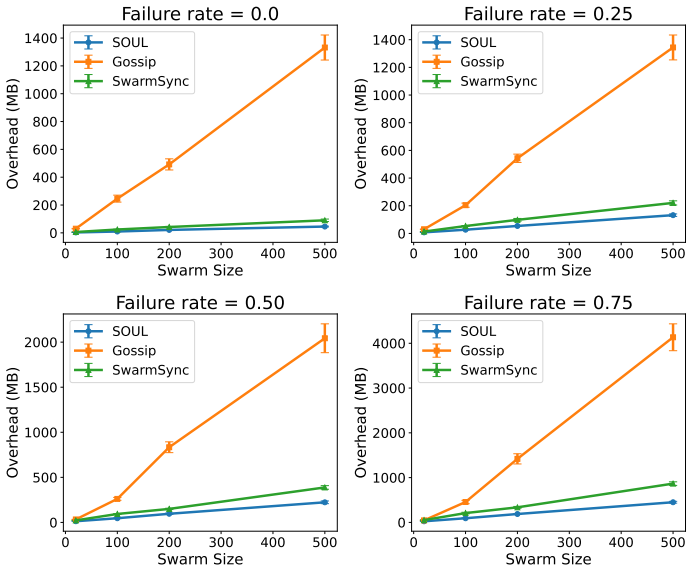


Figure 4.9: The **total** overhead for each synchronization strategy until convergence of the swarm

4.6.2 Transmission Overhead of SwarmSync

Setup

Using the same setup as Section 4.6.1, we evaluate the transmission overhead of the updates, which is accounted for when a drone sends out a packet (12.5 kb), and when a drone sends a communication signal (5 bytes on average). We measure the total overhead from the start of the update synchronization until the swarm converges in Megabytes. The update payloads remain the same as for Section 4.6.1.

Fig. 4.8 and Fig. 4.9 shows the overhead cost to converge. Fig. 4.9 shows the total overhead cost for the swarm to converge for each strategy, while Fig. 4.8 shows the average overhead per drone, which measures the overhead each drone adds to the total overhead.

Findings

SwarmSync generates an average of 87.7% more overhead than SOUL but only produces 19.7% overhead compared to Gossip. Fig. 4.8 shows, for both SwarmSync and SOUL, that transmission overhead decreases as swarm size increases. Although SwarmSync produces an average of 87.7% more overhead than SOUL across all failure rates and swarm sizes, the overhead for both SOUL and SwarmSync shows a decreasing trend as swarm size increases. In contrast, Gossip creates up to 8.3 MB of transmission overhead at a failure rate of 0.75, showing a 72.8% increase in overhead compared to SwarmSync at a swarm size of 20.

As shown in Fig. 4.9, the **total** overhead accumulated across the swarm outlines the difference between the three strategies, while SwarmSync produces more overhead per drone than SOUL, its total overhead remains far below Gossip at scale. This confirms that SwarmSync strikes a practical balance, where it produces an acceptable extra overhead cost relative to SOUL, but substantially more bandwidth-efficient than Gossip when deployed at scale.

Higher failure rates increase overhead in all three strategies, at swarm size 20, SOUL and SwarmSync show similar overhead growth (828.4% and 850.5%) as the failure rate increases from 0.0 to 0.75. As shown in Fig. 4.8, the overhead of all strategies increases with increased failure rates in different swarm size settings. In particular, the relative overhead ratios among the strategies remain similar as the failure rate increases. Both SwarmSync and SOUL almost double in terms of overhead with each increase in the failure rate.

Interestingly, while the overhead of both SwarmSync and SOUL grew by more

than 100% from failure rate 0 to 0.25, the overhead of Gossip remained relatively stable. This behavior is likely attributable to the nature of the Gossip protocol, which inherently involves multiple drones redundantly disseminating the same data packet to one or more receivers. This creates transmission redundancy, even in the presence of packet loss, the probability that all copies of a packet are dropped remains low, so receivers typically accumulate sufficient copies to reconstruct the update. This effect will likely exist as long as the per-packet drop probability has not increased to a level where simultaneous loss across all concurrent transmissions becomes likely. In contrast, this trend is not observed in SwarmSync or SOUL, both of which adopt more centralized dissemination strategies. In these protocols, each receiver typically obtains a given packet from a single designated transmitter (e.g., a leader or parent node), eliminating the built-in redundancy of Gossip. Consequently, they lack the same resilience to packet loss under identical channel conditions, making their performance more sensitive to individual transmission failures.

However, both SOUL and SwarmSync maintain a relatively stable overhead increase (around 15 times) when the swarm size grows from 20 to 500 across all failure rates. The overhead for Gossip does not remain stable; with the same increase in swarm size, the increase in overhead goes from around 40 times for failure rate 0 and 0.25, to 62 times at failure rate 0.5, and 91 times at failure rate 0.75. This is likely due to the formation of Gossip, where lower sizes have multiple Updaters broadcasting to the same receivers, making the receiver get the update with minimal packet loss. With increasing swarm size, the length between Updater and drones on the edge of the swarm increases, creating the need to get the update relayed through the entire

swarm, growing in overhead.

Table 4.2: Mann-Whitney U test results for SwarmSync vs SOUL and SwarmSync vs Gossip across swarm sizes and failure rates. Effect size r is the rank-biserial correlation; negative values indicate the competing algorithm causes less overhead, positive values indicate SwarmSync causes less overhead. Non significant($p \geq alpha$) entries are highlighted in bold.

		SwarmSync vs SOUL		**SwarmSync vs Gossip**	
Failure Rate	**Swarm Size**	p	r	p	r
$f = 0.00$	20	<0.001	−0.825	<0.001	0.858
	100	<0.001	−0.728	<0.001	0.693
	200	**0.004**	**N/A**	<0.001	0.885
	500	**0.007**	**N/A**	<0.001	0.830
$f = 0.25$	20	<0.001	−0.610	<0.001	0.765
	100	<0.001	−0.835	<0.001	0.820
	200	<0.001	−0.782	<0.001	0.958
	500	**0.014**	**−N/A**	<0.001	1.000
$f = 0.50$	20	<0.001	−0.613	**0.159**	**N/A**
	100	<0.001	−1.000	<0.001	0.968
	200	<0.001	−0.730	<0.001	0.685
	500	<0.001	−0.635	<0.001	0.900
$f = 0.75$	20	<0.001	−0.702	**0.358**	**N/A**
	100	0.001	−0.607	**0.036**	**N/A**
	200	<0.001	−0.815	<0.001	0.900
	500	<0.001	−0.855	<0.001	0.900

Statistical Analysis

Table. 6.2 presents the Mann-Whitney U test results confirming the statistical significance of the overhead differences observed above.

SOUL produces significantly lower overhead than SwarmSync in nearly every condition. SOUL's overhead advantage over SwarmSync is statistically sig-

nificant with large effect sizes across most conditions ($r \leq -0.455$), consistent with the 87.7% average overhead gap reported above. The exceptions are the larger swarms at low failure rates, where the advantage does not survive correction: $f = 0.00$ at sizes 200 and 500 ($p = 0.004$ and $p = 0.007$) and $f = 0.25$ at size 500 ($p = 0.014$).

SwarmSync produces significantly lower overhead than Gossip across nearly all conditions, with Gossip only remaining competitive at small swarms under high failure rates. At $f = 0.50$, size 20 is non-significant ($p = 0.159$), and at $f = 0.75$, sizes 20 and 100 are also non-significant ($p = 0.358$, $r = -0.173$ and $p = 0.036$, $r = 0.390$), with Gossip marginally lower at size 20. This aligns with the findings above: at small swarm sizes under high failure rates, Gossip's redundant broadcast keeps overhead low by ensuring delivery without retransmission. From size 200 onwards, SwarmSync's overhead advantage over Gossip is significant in every condition, confirming that Gossip's bandwidth cost grows substantially faster than SwarmSync's as the swarm scales.

4.6.3 Model Patch Size and Accuracy Trade-off

Setup

To evaluate SwarmModelPatch's ability to reduce model patch size while maintaining model accuracy, we progressively freeze the model layers up to the last layer, where the last classification layer is unfrozen by default, performing model patching for each number of frozen layers and comparing the trade-off between accuracy and update size with the baseline "full-size update", which was used in Section 4.6.1 and Section 4.6.2. Each model is trained for 10 epochs and repeated 3 times, the best model is kept for

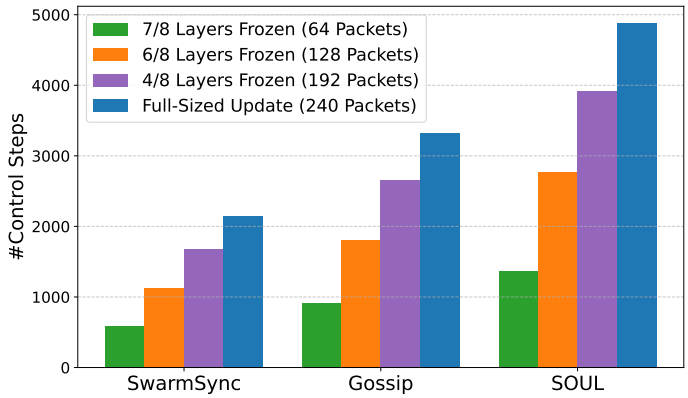


Figure 4.10: The average number of control steps for swarm convergence across varying patch sizes.

the study. We evaluate the accuracy and size trade-offs for all three models across the two datasets, 6 configurations in total.

Simulation Setup

We use SqueezeNet as the main model for our UAV simulations to study SwarmSync, due to its fast inference speed and low parameter count, as well as a linear size decrease, the other two models are only used for the study of SwarmModelPatch, and not SwarmSync. In SqueezeNet, the layers closer to the classifier contain more weights, which means that freezing those layers reduces the update size of the patch more than the other (more shallow) layers.

In particular, we select three configurations for our simulation experiment: freezing 4, 6, and 7 fire modules, other configurations are studied for their accuracy trade offs but not used in the update simulation. These configurations produce balanced reductions in patch size, with the configuration freezing four fire modules producing a patch size approximately 300% larger than the one obtained by freezing seven mod-

ules, and freezing six fire modules resulting in a patch size approximately 200% larger than the 7-module case.

For example, the full-size update used in which was used in Section 4.6.1 and Section 4.6.2 requires 240 packets for one model version update. Freezing 4 fire modules reduces the model patch size to 192 packets, whereas freezing 6 and 7 fire modules further reduces the update sizes to 128 and 64 packets, respectively. We evaluated the updated models based on two metrics: overall classification accuracy (across all classes, including the new class data) and accuracy specifically on the newly introduced class data.

We evaluate the efficiency and overhead of different patch sizes in a swarm of 200 UAVs with a failure rate of $f = 0.25$. We select swarm size = 200 to get 100 drones to be updated with the DL patch according to previous work [112]. We selected $f = 0.25$ because it is a common failure rate for sophisticated UAV systems [80]. We measure efficiency and overhead with the same measurements as Section 4.6.1 and Section 4.6.2. Similarly to Section 4.6.1 and Section 4.6.2, we study all three strategies and the impact of patch sizes on them.

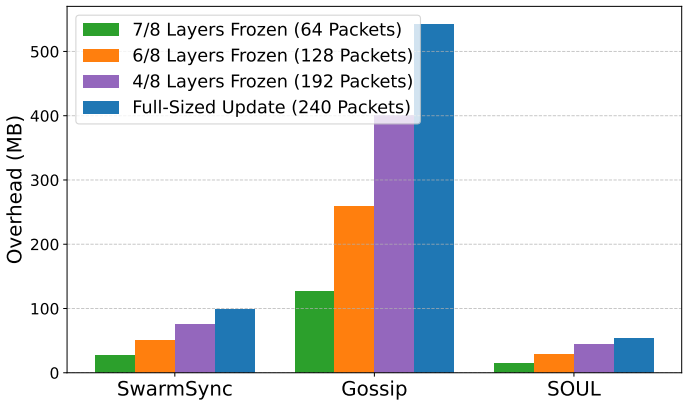
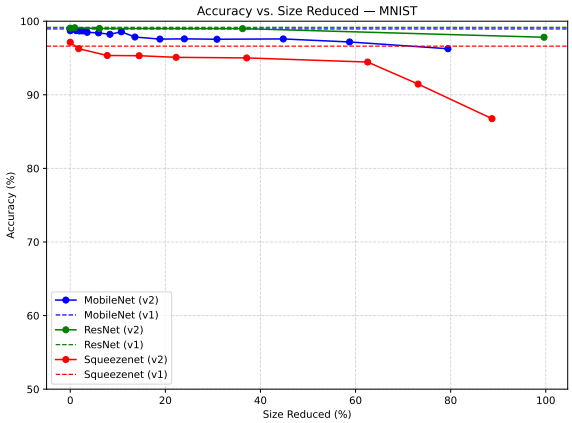
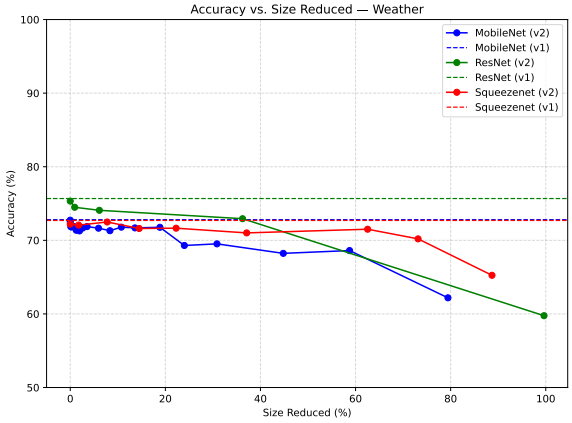


Figure 4.11: The average overhead for swarm convergence across varying patch sizes.



(a) MNIST



(b) Weather

Figure 4.12: The model accuracy vs size reduction for the two datasets, the dashed line represents the model's performance before learning the new class.

Findings

SwarmModelPatch with frozen layers improves to an average convergence speed of 72.3% in 7/8 frozen layers compared to the full-size update baseline described in Section 4.5.1. Fig. 4.10 compares the number of control steps required for different update sizes with a failure rate of 0.25 with a swarm size of 200 (100 Eyebots and 100 Footbots). The results indicate that the use of SwarmModelPatch with frozen layers improves convergence speed by an average of up to 72.3%, with an error margin of 1%, compared to updating the whole model.

SwarmModelPatch with frozen layers has an average transmission overhead of 74.3% lower at 7/8 frozen layers in all strategies. Fig. 4.11 presents the average transmission overhead (in megabytes) evaluated under the same parameters as Fig. 4.10. Similarly to Fig. 4.10, having a reduced-size patch reduces the overhead transmitted during the update. The overhead for both SOUL and SwarmSync was reduced by up to 73.0%(72.9% and 73.0%) for the 7/8 frozen layer update, while the overhead for Gossip was reduced by up to 76.7%, which is likely due to UAVs in Gossip receiving packets from multiple senders at the same time.

SwarmModelPatch reduces patch size by up to 99.6% across models, with accuracy trade-offs varying by architecture. Fig. 4.12 presents the accuracy–size trade-off when applying SwarmModelPatch with increasing numbers of frozen layers. Results on the *weather* dataset show that MobileNet achieves a 79.4% reduction in patch size (shrinking from 8.75 MB to 1.8 MB), at the cost of a 10.6 percentage point drop in accuracy (from 72.8% to 62.2%). ResNet demonstrates the most aggressive compression, reducing patch size by 99.6% (from 90.13 MB to 0.33

MB), but suffers a larger accuracy loss of 15.9 percentage points (from 75.7% to 59.8%). In contrast, SqueezeNet offers the best balance, it cuts patch size by 88.6% (2.83 MB to 0.32 MB) while incurring only a modest 7.5 percentage point accuracy drop (from 72.7% to 65.3%).

On the simpler MNIST dataset, models are more robust but still exhibit notable degradation under extreme compression. MobileNet maintains high performance (98.95% to 96.26%, -2.7%) even at 79.4% size reduction. ResNet retains strong accuracy (99.17% to 97.82%, -1.4%) despite 99.6% compression. However, SqueezeNet shows a substantial drop: from 96.61% down to 86.76% (-9.9%) when compressed by 88.6%, indicating that lightweight architectures may be more sensitive to parameter freezing on digit recognition tasks despite their small footprint.

Chapter 5

SwarmMARL: Learned Decentralized Coordination

This chapter presents SwarmMARL, the second coordination approach investigated in this thesis. Where SwarmSync addresses the update propagation problem defined in Chapter 3 through a deterministic, protocol-based philosophy with explicitly assigned roles and guaranteed delivery semantics, SwarmMARL takes a fundamentally different stance: it trains agents to autonomously learn *when* to broadcast, request, or remain idle during an update dissemination cycle, without any pre-defined role assignment or explicit protocol rules. Coordination behavior emerges from a reward signal that incentivizes rapid convergence and penalizes wasteful or misdirected transmissions, allowing the policy to adapt to dynamic swarm conditions that a fixed protocol cannot respond to — such as critically low battery, in-progress transfers, or a neighborhood that is already fully updated.

The motivation for pursuing a learned coordination approach arose directly from SwarmSync's most significant limitation: its inability to suppress contextually inappropriate transmissions. SwarmSync's protocol rules determine what each agent does based solely on its assigned role and version state, a fully updated agent with neighbors in range will always broadcast, regardless of whether its battery is nearly depleted, whether a better-positioned neighbor is already covering the same receivers, or whether a concurrent transfer is already underway on the same channel. These are not edge cases; in a realistic heterogeneous swarm operating over extended missions, they are the norm. The insight that motivated SwarmMARL is that the three underlying coordination behaviors: broadcast, request, idle, are not themselves the problem. The problem is the rigidity of the rules that determine when each is executed. A learned policy can, in principle, condition the same three actions on the full richness of local context, suppressing transmissions that are technically valid but practically wasteful without requiring those suppressions to be explicitly enumerated as protocol rules.

SwarmMARL is not a replacement for SwarmSync. It is an alternative coordination philosophy for the same dissemination objective, with a different set of strengths and trade-offs. SwarmSync provides formal termination guarantees, interpretable behavior, and ACK-based reliability that is provably resilient under high packet loss. SwarmMARL is structurally closer to gossip-based dissemination, but adds a learned suppression policy that reduces the redundant rebroadcasting that makes naive gossip inefficient. The design is broadly inspired by the observation in the gossip protocol that the primary cost of epidemic dissemination is not the first wave of transmis-

sions but the redundant retransmissions that propagate after most agents are already updated; SwarmMARL replaces the fixed suppression rules used in push-pull gossip variants with a learned suppression policy that can condition on local context. A direct empirical comparison of the two approaches is presented in Chapter 6. SwarmModelPatch is compatible with SwarmMARL in the same way it is compatible with SwarmSync: the payload it generates can be disseminated by either coordination approach without modification.

The remainder of this chapter is organized as follows. Section 5.1 presents the SwarmMARL design, including the observation space, action space, reward function, and actor-critic architecture. Section 5.2 describes the ARGoS integration through the file-based state-action bridge. Section 5.3 describes the MAPPO training procedure and hyperparameter choices. Section 5.4 provides an illustrative walkthrough of the SwarmMARL coordination process using the wildfire scenario from Chapter 1. Section 5.5 presents the empirical evaluation of SwarmMARL as a standalone coordination approach.

5.1 SwarmMARL Design

SwarmMARL formulates the update dissemination problem as a fully cooperative Dec-POMDP (see Section 2.1), where each UAV agent u_i observes a local state o_i, selects a coordination action a_i, and receives a shared reward signal r that reflects progress toward global swarm convergence. All agents share a single policy network trained centrally using MAPPO [126] and executed independently at runtime us-

ing only local observations, following the CTDE framework described in Section 2.1. The choice of MAPPO as the base algorithm was motivated by its strong empirical performance on cooperative multi-agent tasks with shared rewards [126], its compatibility with the parameter-sharing variant of CTDE that is natural for homogeneous or near-homogeneous swarms, and its relative stability compared to value decomposition methods such as QMIX in settings where the global reward signal is dense and per-agent credit assignment is not the primary challenge.

5.1.1 Observation Space

The observation space was designed around a single guiding principle: include the minimum set of features that allows the policy to distinguish between situations where each of the three actions is appropriate, without exposing variable-length or global information that would either require architectural changes to handle or break the locality constraint of the Dec-POMDP formulation. Early design iterations included the full neighbor list as input, but this required either a fixed maximum neighborhood size with zero-padding, which introduces spurious features for sparse neighborhoods, or a permutation-invariant aggregation step that added architectural complexity without a corresponding benefit given the small neighborhood sizes observed in the ARGoS simulation. The two scalar summaries, neighbor count and mean distance, were found to carry sufficient topology signal for the policy to distinguish between the broadcast-suppression and broadcast-encouragement regimes that the reward function targets.

Each agent u_i constructs a local observation vector $o_i \in \mathbb{R}^{11}$ at each control step from the state information written to its JSON state file by the ARGoS controller. The

observation encodes the information most relevant to deciding whether to broadcast, request, or remain idle at the current moment. The full observation vector is:

$$o_i = \begin{bmatrix} \underbrace{\mathbf{1}_{\text{type=0}},\ \mathbf{1}_{\text{type=1}}}_{\text{type one-hot}} \\ \underbrace{\delta_v}_{\text{version gap}} \\ \underbrace{b}_{\text{battery}} \\ \underbrace{|\mathcal{N}_i|}_{\text{num neighbors}} \\ \underbrace{\bar{d}}_{\text{mean neighbor dist}} \\ \underbrace{\rho}_{\text{transfer progress}} \\ \underbrace{\mathbb{1}_{\text{rx}}}_{\text{is receiving}} \\ \underbrace{\mathbb{1}_{\text{tx}}}_{\text{is broadcasting}} \\ \underbrace{\eta}_{\text{neighbor update ratio}} \end{bmatrix}$$

where each component is defined as follows.

Type one-hot $[\mathbf{1}_{\text{type=0}},\ \mathbf{1}_{\text{type=1}}]$ encodes the UAV type as a two-dimensional one-hot vector, allowing the policy to learn type-specific behavior without requiring separate policies per type. In the experimental setup of this thesis, type 0 corresponds to Footbots and type 1 to Eyebots.

Version gap $\delta_v = v_{\text{target}} - v_{\text{current}}$ encodes how far the agent is from the target software version. A value of zero indicates the agent is fully updated; a positive value

indicates it still requires the update. This is the primary signal driving the agent's decision between broadcasting (if updated) and requesting (if outdated).

Battery $b \in [0, 1]$ encodes the agent's current battery level as a normalized scalar. **Battery urgency** $1 - b$ is its complement. Including both b and $1 - b$ in the observation vector is a deliberate redundancy: a single battery feature is mathematically sufficient, since the network could compute $1 - b$ internally. However, representing the urgency regime explicitly as a high-valued feature when battery is low provides a direct gradient signal into the battery conservation term of the reward function, without requiring the network to learn that a small value of b maps to a large penalty. In preliminary experiments with only b as input, the policy was slower to learn the battery conservation behavior, since the relevant signal appeared at the low end of the feature's range where gradients are weakest. Adding $1 - b$ resolved this by ensuring that critical battery states produce high-magnitude features regardless of scale.

Number of neighbors $|\mathcal{N}_i|$ and **mean neighbor distance** $\bar{d}$ encode the local communication topology as two scalar summaries derived from the raw neighbor list reported by the ARGoS controller. $|\mathcal{N}_i|$ is the count of agents currently within communication range, and $\bar{d}$ is the mean of their distances. Broadcasting when no neighbors are in range wastes energy; remaining idle when neighbors are close and outdated misses dissemination opportunities. Together these features allow the policy to condition transmission decisions on neighborhood density without requiring the full neighbor list as input.

Transfer progress ratio $\rho = p_{\text{received}}/p_{\text{total}}$ encodes what fraction of the incoming update has been received, where p_{total} is the total number of packets in the update.

This feature is zero when no transfer is in progress and rises toward 1.0 as a transfer nears completion. It allows the policy to learn that remaining idle during an active transfer is productive behavior rather than wasted time.

Is receiving $\mathbb{1}_{\text{rx}}$ and **is broadcasting** $\mathbb{1}_{\text{tx}}$ are binary flags indicating whether the agent is currently involved in an active transfer as receiver or sender respectively. These complement the progress ratio by providing an explicit binary signal about current transmission state; ρ is a continuous quantity that rises slowly, while $\mathbb{1}_{\text{rx}}$ switches immediately and cleanly at transfer start and end, giving the policy a lower-latency signal about whether a transfer is active at all.

Neighbor update ratio η is a scalar derived from the `neighbor_versions` list reported by the ARGoS controller, computed as the fraction of visible neighbors that have already received the full update:

$$\eta = \frac{|\{u_j \in \mathcal{N}_i : v_j \geq v_{\text{target}}\}|}{|\mathcal{N}_i|}$$

This feature is particularly important for broadcasting decisions: an updated agent surrounded by already-updated neighbors has little reason to broadcast, while an updated agent surrounded by outdated neighbors should prioritize transmission. The inclusion of this feature was directly motivated by the known inefficiency of naive gossip protocols in the late stages of dissemination, where redundant rebroadcasting among already-updated agents dominates the overhead. By making η visible to the policy, the reward signal for broadcasting to an already-updated neighborhood can propagate directly through this feature, allowing the policy to learn broadcast suppression without that suppression needing to be hard-coded as a rule. When

neighbor version information is unavailable (i.e., the C++ ARGoS controller does not expose `neighbor_versions` in the state JSON), η defaults to 0.0, making the feature backward compatible with earlier controller versions.

The full observation dimension is $|o_i| = 11$, consistent with the `OBS_DIM = 11` constant in the SwarmMARL implementation.

5.1.2 Action Space

Each agent selects one of three discrete actions at each control step:

Action 0 — Idle: The agent takes no communication action this step. This is appropriate when a transfer is already in progress, when no neighbors are in range, or when battery is critically low.

Action 1 — Broadcast: The agent transmits the update to neighbors within communication range. This is appropriate when the agent holds the complete and current update and has outdated neighbors within range.

Action 2 — Request: The agent sends a request for the update to neighbors within range. This is appropriate when the agent does not yet hold the update and is not currently receiving it.

These three actions map directly to the coordination behaviors that both SwarmSync and gossip-based protocols implement through explicit rules. The key difference is that SwarmMARL learns *when* each action is appropriate from experience rather than having it hard-coded into the protocol.

5.1.3 Reward Function

The reward function is designed to incentivize rapid convergence while penalizing wasteful or counterproductive transmissions. It is computed per-agent per-step using the agent's current state and the action taken, and is conditioned on the agent's previous state to detect the transition step at which an update completes. The full reward for agent u_i taking action a_i in state s_i is:

$$r_i = r_{\text{completion}} + r_{\text{action}} + r_{\text{progress}} + r_{\text{step}} + r_{\text{battery}}$$

The overall structure was shaped by a practical concern: sparse rewards in the form of a single completion signal are sufficient in principle but slow to propagate useful gradient information through long episodes, particularly when the episode may span hundreds of control steps before any agent receives the update. The design therefore combines a large one-time completion bonus with a continuous step penalty, a dense progress signal during active transfers, and a set of action-shaping terms that provide immediate feedback on clearly correct or clearly incorrect actions. The relative magnitudes of these components were determined through iterative tuning, guided by the principle that the completion bonus should dominate the total expected reward for a well-behaved episode, while the continuous terms should be large enough to provide useful gradient signal but not so large that they distort the policy toward locally rewarding behaviors that are globally suboptimal. The specific values that follow reflect the final configuration after this tuning process; where key calibration decisions were made, they are noted explicitly.

Completion bonus $r_{\text{completion}}$ fires once, on the control step at which the agent transitions from outdated to fully updated:

$$r_{\text{completion}} = \begin{cases} +5.0 & \text{if } v_{\text{current}} \geq v_{\text{target}} \text{ and } v_{\text{prev}} < v_{\text{target}} \\ 0 & \text{otherwise} \end{cases}$$

The magnitude of $+5.0$ was calibrated against the step penalty of -0.01 per outdated step: an agent that converges in T steps incurs a cumulative step penalty of $-0.01T$, so the completion bonus outweighs the step cost for any episode completing within 500 steps. In practice, well-trained episodes converge in 100–300 steps, so the bonus reliably dominates. The one-time nature of this bonus is equally important: rewarding the agent every step it remains updated would encourage passive behavior rather than active dissemination toward neighbors. By firing only on the transition step, it incentivizes the agent to complete its own update as quickly as possible without creating a persistent incentive to stay idle once updated.

Action shaping r_{action} rewards correct action selection relative to the agent's

current update status:

$$r_{\text{action}} = \begin{cases} +0.3 & \text{if } v_{\text{current}} \geq v_{\text{target}} \text{ and } a_i = 1 \text{ and } |\mathcal{N}_i| > 0 \\ -0.05 & \text{if } v_{\text{current}} \geq v_{\text{target}} \text{ and } a_i = 1 \text{ and } |\mathcal{N}_i| = 0 \\ +0.01 & \text{if } v_{\text{current}} \geq v_{\text{target}} \text{ and } a_i = 0 \\ -0.3 & \text{if } v_{\text{current}} \geq v_{\text{target}} \text{ and } a_i = 2 \\ +0.2 & \text{if } v_{\text{current}} < v_{\text{target}} \text{ and } a_i = 2 \\ -0.3 & \text{if } v_{\text{current}} < v_{\text{target}} \text{ and } a_i = 1 \\ +0.1 & \text{if } v_{\text{current}} < v_{\text{target}} \text{ and } a_i = 0 \text{ and } \not\Vdash_{\text{rx}} = 1 \\ 0 & \text{otherwise} \end{cases}$$

Several design decisions in this component warrant explanation. The magnitudes $+0.3$ and -0.3 for the primary correct and incorrect action cases were chosen to be large enough to drive rapid early learning of the basic action-state correspondence (updated agents broadcast; outdated agents request) while remaining smaller than the completion bonus, so that the policy does not learn to spam broadcasts or requests purely for the per-step shaping reward. The asymmetry between the broadcast-to-empty-neighborhood penalty (-0.05) and the mismatch penalties (-0.3) is intentional: broadcasting to an empty neighborhood is wasteful but not fundamentally misconceived, whereas an updated agent requesting or an outdated agent broadcasting represents a category error that should be discouraged more strongly. The small positive reward for an updated agent that idles ($+0.01$) was added to prevent the

policy from learning that any non-broadcast action while updated is neutral — idling when already updated and surrounded by updated neighbors is mildly productive, since it conserves channel capacity. An outdated agent that remains idle while actively receiving a transfer is rewarded (+0.1), since interrupting a transfer with a redundant request would be counterproductive; this case was added after observing in early training runs that agents would intermittently issue requests during active transfers, disrupting the transfer and resetting the progress ratio.

Progress reward r_{progress} provides a dense signal proportional to how much of the incoming update has been received, but only when a transfer is actively in progress:

$$r_{\text{progress}} = \begin{cases} 0.15 \cdot \rho & \text{if } \mathbb{1}_{\text{rx}} = 1 \\ 0 & \text{otherwise} \end{cases}$$

This component prevents the sparse completion bonus from being the only learning signal during long transfer periods, providing a continuous gradient that guides the policy toward behaviors that sustain rather than disrupt active transfers. The coefficient 0.15 was chosen so that a completed transfer ($\rho = 1.0$) contributes +0.15 per step of the transfer — comparable in magnitude to the action shaping reward — but integrates to a total well below the completion bonus across a typical transfer duration.

Step penalty r_{step} applies a small negative reward every step that the agent remains outdated, creating urgency to complete the update rather than idling indef-

initely:

$$r_{\text{step}} = \begin{cases} -0.01 & \text{if } v_{\text{current}} < v_{\text{target}} \\ 0 & \text{otherwise} \end{cases}$$

This term complements the completion bonus by providing a continuous gradient signal toward convergence throughout the episode, rather than only at the moment of completion. Its magnitude of -0.01 was deliberately kept small relative to the action shaping terms: early experiments with a larger step penalty (-0.05) led the policy to prioritize reducing the step cost through any available transmission — including redundant broadcasts — rather than learning to coordinate efficiently.

Battery conservation r_{battery} penalizes any transmission action when the agent's battery falls below a critical threshold:

$$r_{\text{battery}} = \begin{cases} -0.2 & \text{if } b < 0.1 \text{ and } a_i \neq 0 \\ 0 & \text{otherwise} \end{cases}$$

The threshold $b < 0.1$ (10% battery remaining) was chosen to correspond to the critical low-battery threshold used in the ARGoS battery model, below which the simulator models significantly degraded transmission reliability. The penalty magnitude -0.2 is comparable to the mismatch penalties in the action shaping term, reflecting that transmitting on critically low battery is approximately as counterproductive as a category-mismatched action. This component encodes the constraint that critically low-battery agents should conserve energy rather than transmit — a behavior that a fixed protocol such as SwarmSync cannot enforce without explicit

battery-aware logic added to the protocol specification.

5.1.4 Actor-Critic Architecture

The choice of a shared actor-critic architecture with parameter sharing across all agents follows naturally from the structure of the problem: all agents face structurally identical decision problems (the observation vector is the same format for every agent regardless of type, and the type one-hot provides sufficient information for the network to differentiate behavior by type), and parameter sharing substantially reduces the number of parameters that must be learned, improving sample efficiency in the large-swarm setting ($N = 100$). An alternative would have been to train separate policies for Footbots and Eyebots, but this would have halved the effective training data per policy and required coordination between the two policies at runtime — a complexity that parameter sharing avoids without sacrificing expressive power, since the type one-hot allows the shared network to learn type-specific action distributions internally.

SwarmMARL uses a shared actor-critic network in which all agents share the same policy and value function parameters, following the parameter-sharing variant of MAPPO [126]. The network architecture consists of an input projection, a residual block, and separate policy and value necks and heads:

- **Input projection:** Linear ($11 \to 128$), LayerNorm, ReLU. LayerNorm is applied here rather than batch normalization because the observation features span substantially different scales: version gap $\delta_v \in \{0, 1\}$, battery $b \in [0, 1]$, number of neighbors $|\mathcal{N}_i| \in \{0, \ldots, 18\}$, and mean neighbor distance $\bar{d}$ in meters. LayerNorm normalizes across the feature dimension independently for

each sample, making it more suitable than batch normalization in this setting where feature scale variance is high and batch composition varies across training steps. Manual pre-normalization of the input features was considered as an alternative but rejected because it would require maintaining running statistics across training and would not adapt to distribution shift in neighbor distance as the swarm deployment scenario changes.

- **Residual block:** Two sequential Linear($128 \to 128$) layers, each followed by LayerNorm and ReLU, with a skip connection that adds the block input to the block output before a final ReLU activation. Formally, if $\mathbf{h}$ is the input to the block and $F(\mathbf{h})$ is the output of the two linear layers, the block computes $\text{ReLU}(\mathbf{h} + F(\mathbf{h}))$. The residual connection may seem heavyweight for an 11-dimensional input problem, but it serves a specific purpose in the shared trunk architecture: the policy head and value head impose conflicting gradient signals on the shared representation, and without a residual connection the shared trunk tended to collapse toward representing whichever objective had the larger gradient magnitude at any given point in training. The skip connection allows the network to learn identity mappings easily, stabilizes gradient flow, and prevents this representational collapse without requiring separate trunks for the policy and value streams.

- **Policy neck:** Linear($128 \to 64$), ReLU.

- **Value neck:** Linear($128 \to 64$), ReLU. Separate necks for the policy and value heads allow each to specialize its 64-dimensional representation without

interference from the other's training signal, which is particularly important in PPO where the value loss and policy loss can pull the shared representation in conflicting directions.

- **Policy head:** Linear($64 \to 3$), producing one logit per action. The weights are initialized orthogonally with a gain of 0.01 and the bias is initialized to zero, ensuring that the initial action probability distribution is near-uniform across the three actions. This prevents the policy from collapsing to a single action at the start of training before any useful gradient signal has been received; in early experiments without this initialization, the policy would frequently collapse to always selecting Action 1 (Broadcast) within the first few rollout cycles, from which recovery was slow.

- **Value head:** Linear($64 \to 1$), producing a scalar value estimate used to compute GAE advantages.

During training, actions are sampled from the categorical distribution defined by the softmax of the policy head logits. Action probabilities are clipped to $[10^{-6}, 1 - 10^{-6}]$ before computing log-probabilities to prevent numerical instability in the PPO surrogate loss. During inference, the `act` method runs a single forward pass with `torch.no_grad()` to avoid unnecessary gradient computation, which is important given that the method is called once per agent per control step across all 100 agents.

The hyperparameters used for training SwarmMARL are summarized in Table 5.1. Two design choices in the hyperparameter schedule are worth noting. First, the entropy coefficient is annealed from 0.05 at the start of training down to 0.005 over

50,000 gradient steps. The initial value of 0.05 was chosen to be higher than the 0.01 default commonly used in single-agent PPO, reflecting the larger exploration challenge in the three-action cooperative setting where the policy must learn not just what action to take but how its action interacts with the simultaneous choices of 99 other agents. The annealing schedule was motivated by the observation that a high entropy bonus in late training prevented the policy from committing to the high-confidence broadcast/idle decisions that efficient dissemination requires, while a low entropy bonus from the start led to premature convergence to a near-deterministic policy before the value function had learned to provide reliable gradient signal. The 50,000-step annealing horizon was calibrated to correspond approximately to the first 100 rollout-update cycles, by which point the policy had consistently learned the basic action-state correspondence. Second, a linear learning rate schedule decays the learning rate from 3×10^{-4} to 3×10^{-5} over 200 rollout-update cycles, following the schedule used in [126] and adjusted to the smaller action space. This prevents large policy updates in late training when the policy is near convergence, complementing the PPO clipping mechanism for training stability.

5.2 ARGoS Integration

SwarmMARL integrates with ARGoS through a file-based state-action bridge that deliberately decouples the Python-based MAPPO trainer from the C++-based ARGoS simulation. This design choice is motivated by three considerations. First, it avoids requiring any modification to the ARGoS simulator itself, maintaining compatibility

Table 5.1: SwarmMARL hyperparameters.

Hyperparameter	Value
Number of agents (N)	100
Observation dimension	11
Action dimension	3
Rollout horizon (T)	128
PPO epochs per update (K)	6
Minibatch size	512
Discount factor (γ)	0.99
GAE λ	0.95
PPO clip ϵ	0.2
Learning rate	3×10^{-4}
LR schedule	Linear decay to 3×10^{-5} over 200 updates
Entropy coefficient	Annealed from 0.05 to 0.005 over 50,000 steps
Value function coefficient	0.5
Gradient norm clip	0.5
Hidden layer sizes	[128 (residual block), 64 (policy/value necks)]
Policy head initialization	Orthogonal, gain = 0.01, zero bias

with the same simulation environment used for SwarmSync evaluation. Second, it makes the MAPPO controller simulator-agnostic: the bridge protocol could be replicated for any simulator or physical hardware platform that can write JSON files to a shared filesystem. Third, it mirrors the architecture used in the real-world experiments described in Chapter 7, where policy inference is offloaded to a ground server communicating with physical drones over a network interface.

5.2.1 State-Action Protocol

At each control step t, the ARGoS C++ controller for each agent u_i writes its current state to a JSON file at a fixed path:

```
/tmp/argos_agent_{i}_state.json
```

The state file contains the following fields:

```
{
  "type":                <int>,
  "current_version":     <int>,
  "target_version":      <int>,
  "battery":             <float>,
  "neighbors":           [<float>, ...],
  "neighbor_versions":   [<int>, ...],
  "incoming_progress":   <int>,
  "incoming_size":       <int>,
  "is_receiving":        <bool>,
  "is_broadcasting":     <bool>,
  "episode_reset":       <bool>
}
```

The MAPPO Python process polls these files once per control step, reads the state for each of the $N = 100$ agents, computes the observation vector o_i using the `state_to_obs` function, and runs a forward pass through the shared actor-critic network to sample an action a_i for each agent. The selected action is written atomically to:

`/tmp/argos_agent_{i}_action.json`

using a write-then-rename pattern (`.tmp` intermediate file followed by `os.replace`) to prevent the ARGoS controller from reading a partially written file. The ARGoS

controller reads the action file at the start of the next control step and executes the corresponding communication behavior.

5.2.2 Episode Boundaries

The `episode_reset` flag in the state JSON is set to `true` by the ARGoS controller on the first control step of a new episode, signaling to the MAPPO trainer that the previous episode has ended and per-agent memory (specifically, the previous state used for transition-based reward computation) should be cleared. This prevents the trainer from computing rewards that span episode boundaries, which would corrupt the GAE advantage estimates. When `episode_reset` is `true` for agent u_i, the agent's mask is set to zero in the rollout buffer, ensuring that GAE does not bootstrap across the boundary.

5.2.3 Timing and Synchronization

The MAPPO process sleeps for `SLEEP_SEC = 0.05` seconds between control steps to avoid busy-waiting on the state files. Since ARGoS operates at a simulated 100 ms per control step, this sleep interval is calibrated to allow the ARGoS controller to complete its state write before the MAPPO process attempts to read. In practice, the file-based bridge introduces a one-step lag between the state observed by the MAPPO process and the action executed by the ARGoS controller, mirroring the one-step communication delay already present in the ARGoS simulation model and described in Chapter 3. This lag is consistent across all agents and does not introduce asymmetric timing artifacts.

5.3 Training Procedure

SwarmMARL is trained from scratch using MAPPO with the hyperparameters summarized in Table 5.1. Training proceeds in rollout-update cycles: the MAPPO process collects a rollout of $T = 128$ control steps across all $N = 100$ agents, computes GAE advantages and returns, and performs $K = 6$ epochs of PPO updates over the collected data using minibatches of size 512.

5.3.1 Rollout Collection

At each control step t within a rollout, the MAPPO process reads the state of all N agents in parallel using a thread pool executor, computes observations, samples actions, writes actions back to ARGoS atomically, and computes per-agent rewards using the reward function defined in Section 5.1. Parallel state reading and action writing is important at scale: with 100 agents each requiring a file read and write per control step, sequential I/O would introduce latency that could desynchronize the MAPPO process from the ARGoS simulation. Agents for which no state file is available or for which `episode_reset` is true receive a mask value of 0 in the rollout buffer and are excluded from the PPO update. After $T = 128$ steps, the process computes bootstrap values for all active agents by running a forward pass through the value head on the final observed state, then computes GAE advantages and discounted returns.

The training loop also handles three types of episode boundary within a rollout. The first is a convergence-triggered reset: when all 100 agents are readable and fully

updated, the MAPPO process writes a reset signal file that the ARGoS controller reads to reinitialize all agents, beginning a new episode. The second is a timeout-triggered reset, when nothing happens for 50 ticks (including agents switching between broadcasting and requesting), a timeout triggers and the episode is considered failed to converge, the third is a time limit reset: the controller enforces a maximum episode length (5000) and sets `episode_reset` to true when this limit is reached, although this did not happen during training. All boundary types are logged to `episode_log.csv` along with the convergence step, total overhead, and whether the episode converged, enabling post-hoc analysis of training progress at the episode level.

5.3.2 Advantage Normalization

Advantages are normalized after flattening and masking, so that only valid (mask = 1) transitions contribute to the normalization statistics. This is important because including masked zeros in the normalization would distort the mean and standard deviation, producing incorrect advantage estimates. The normalized advantage for transition (s, a) is:

$$\hat{A} = \frac{A - \mu_A}{\sigma_A + \epsilon}$$

where μ_A and σ_A are computed over valid transitions only and $\epsilon = 10^{-8}$ prevents division by zero.

5.3.3 PPO Update

For each of the $K = 6$ epochs, the valid transitions are randomly shuffled and processed in minibatches of 512. For each minibatch, the network computes new log-probabilities and entropy for the sampled actions, and the clipped surrogate loss is computed as:

$$\mathcal{L} = \mathcal{L}^{\text{CLIP}} + c_v \mathcal{L}^{\text{VF}} - c_e(t) \cdot \mathcal{H}$$

where $\mathcal{L}^{\text{CLIP}}$ is the clipped policy loss, $\mathcal{L}^{\text{VF}} = \mathbb{E}[(V_\theta(s) - R)^2]$ is the value function loss, $\mathcal{H}$ is the mean entropy of the policy distribution, $c_v = 0.5$ is the value coefficient, and $c_e(t)$ is the annealed entropy coefficient at global step t, defined as:

$$c_e(t) = 0.05 + \min\left(1, \frac{t}{50000}\right) \cdot (0.005 - 0.05)$$

The entropy bonus encourages exploration during training by preventing premature convergence to a deterministic policy, with the annealing schedule allowing the policy to commit to higher-confidence decisions as training progresses. Gradients are clipped to a maximum norm of 0.5 before the optimizer step to prevent destabilizing updates. After each rollout-update cycle, the learning rate scheduler takes one step, linearly decaying the learning rate toward 3×10^{-5} over 200 cycles.

5.3.4 Checkpointing and Logging

After each rollout-update cycle, the policy and optimizer state are saved to:

`checkpoints/policy_latest.pt`

The checkpoint includes the model state dict, optimizer state dict, and the global step count, enabling training to be resumed from any point. Four log files are maintained throughout training. `loss_log.csv` records the actor loss, value loss, entropy, and total loss per update step. `metrics_log.csv` records per-control-step swarm metrics including the convergence ratio, the number of agents currently broadcasting, and the number currently receiving, indexed by both global step and episode step. `episode_log.csv` records per-episode outcomes including convergence step, total overhead, and whether the episode converged within the time limit. A loss graph across all four loss components is saved periodically to `loss_graph.png` using matplotlib, providing a visual summary of training stability that complements the raw CSV logs.

5.4 Illustrative Example: SwarmMARL Coordination in the Wildfire Scenario

To illustrate how SwarmMARL coordinates update dissemination in practice, we revisit the wildfire monitoring scenario introduced in Section 1.2 and extended in Section 4.4. Recall that a heterogeneous swarm of Eyebots and Footbots is patrolling a mountainous region when a wildfire is detected. Headquarters has generated a differential model patch $\mathcal{P}$ using SwarmModelPatch and dispatched an Updater UAV carrying $\mathcal{P}$ to the field swarm. The Updater arrives and signals that an update is available. What happens next illustrates the key difference between SwarmSync and SwarmMARL.

In SwarmSync, the protocol takes over: leaders are elected, move to the Updater, receive the patch in a structured sequence, and return to distribute it to followers. Every agent's role and behavior at each step is determined by the protocol. In SwarmMARL, there are no pre-assigned roles. Instead, each agent observes its local state and queries its learned policy to decide what to do at each control step.

Consider three representative agents at the moment the Updater arrives:

Agent A — Eyebot, fully charged, no update yet, three outdated neighbors nearby. Agent A's observation encodes $\delta_v = 1$, $b = 0.95$, $|\mathcal{N}| = 3$, $\nVdash_{\text{rx}} = 0$, $\eta = 0.0$. The policy selects **Action 2 (Request)**. Agent A broadcasts a request signal to its neighbors, and Agent B — a nearby fully updated agent — responds by beginning transmission of $\mathcal{P}$.

Agent B — Eyebot, fully charged, already updated, covers both Agent A and Agent C in its neighborhood. Agent B's observation encodes $\delta_v = 0$, $b = 0.92$, $|\mathcal{N}| = 2$, $\eta = 0.0$, $\nVdash_{\text{tx}} = 1$. The policy selects **Action 1 (Broadcast)**. Agent B begins transmitting $\mathcal{P}$ directly to its neighbors. Because Agent B is fully charged and has outdated neighbors in range, the policy correctly identifies it as the most productive agent to handle dissemination at this moment.

Agent C — Eyebot, low battery ($b = 0.08$), already updated, same neighborhood as Agent B. Agent C's observation encodes $\delta_v = 0$, $b = 0.08$, $1-b = 0.92$, $\nVdash_{\text{tx}} = 0$, $\eta = 0.5$. Although Agent C holds the complete update and has outdated neighbors nearby, the policy selects **Action 0 (Idle)**: the battery urgency feature

signals a critically low battery, and the neighbor update ratio η indicates that a healthier agent, Agent B, is already actively broadcasting to the same neighborhood. A fixed protocol such as SwarmSync or naive gossip cannot make this distinction: a fully updated agent with neighbors in range would always broadcast regardless of whether a better-resourced neighbor is already covering the same receivers. SwarmMARL suppresses the redundant transmission from Agent C, conserving its battery and avoiding the channel contention that simultaneous broadcasts to the same receivers would cause.

As the dissemination progresses over subsequent control steps, agents that complete their update transition from requesting to broadcasting, and the neighbor update ratio η rises across the swarm as more agents become updated. The policy has learned to interpret a high η as a signal to reduce broadcast frequency, there is little to be gained from broadcasting when all visible neighbors are already updated. This emergent behavior reduces redundant transmissions in the later stages of dissemination, which is a known inefficiency in gossip-based protocols but is not explicitly programmed into any rule.

This walkthrough illustrates the core behavioral property that SwarmMARL contributes relative to SwarmSync: context-sensitive, adaptive action selection that suppresses wasteful transmissions without any explicit rule encoding those suppressions. The trade-off is that these behaviors are learned rather than guaranteed: the policy may occasionally make suboptimal decisions, and its behavior is not formally provable in the way that SwarmSync's termination guarantee is. The comparative evaluation in Chapter 6 quantifies this trade-off empirically.

5.5 MARL Results

This section evaluates SwarmMARL as a standalone coordination approach using the metrics defined in Chapter 3. The experimental evaluation uses the same ARGoS simulation environment, swarm sizes $s \in \{20, 100, 200, 500\}$, failure rates $f \in \{0, 0.25, 0.50, 0.75\}$, and packet size of 12.5 kB as the SwarmSync evaluation to enable direct comparison in Chapter 6. SwarmMARL is trained under a single configuration of 100 agents and $f = 0.25$; the evaluation across other configurations tests generalization rather than in-distribution performance.

5.5.1 Convergence Time

Figure 5.1 shows the convergence time of SwarmMARL compared against SwarmSync and Gossip across all swarm sizes and failure rates.

SwarmMARL achieves the fastest convergence (up to a 45.6% reduction) at low failure rates and small-to-medium swarm sizes. At $f = 0.0$ and swarm size 20, SwarmMARL converges in 312 control steps, compared to 483 for SwarmSync and 521 for Gossip. At swarm size 100 under the same failure rate, SwarmMARL converges in 525 steps compared to 965 for SwarmSync and 1,255 for Gossip, a 45.6% and 58.2% reduction respectively. This speed advantage stems from SwarmMARL's flat dissemination structure: because any updated agent can immediately broadcast to its neighbors without waiting for leader election, phase sequencing, or role assignment, multi-source dissemination begins from the first control step. The learned policy amplifies this advantage by suppressing redundant broadcasts once

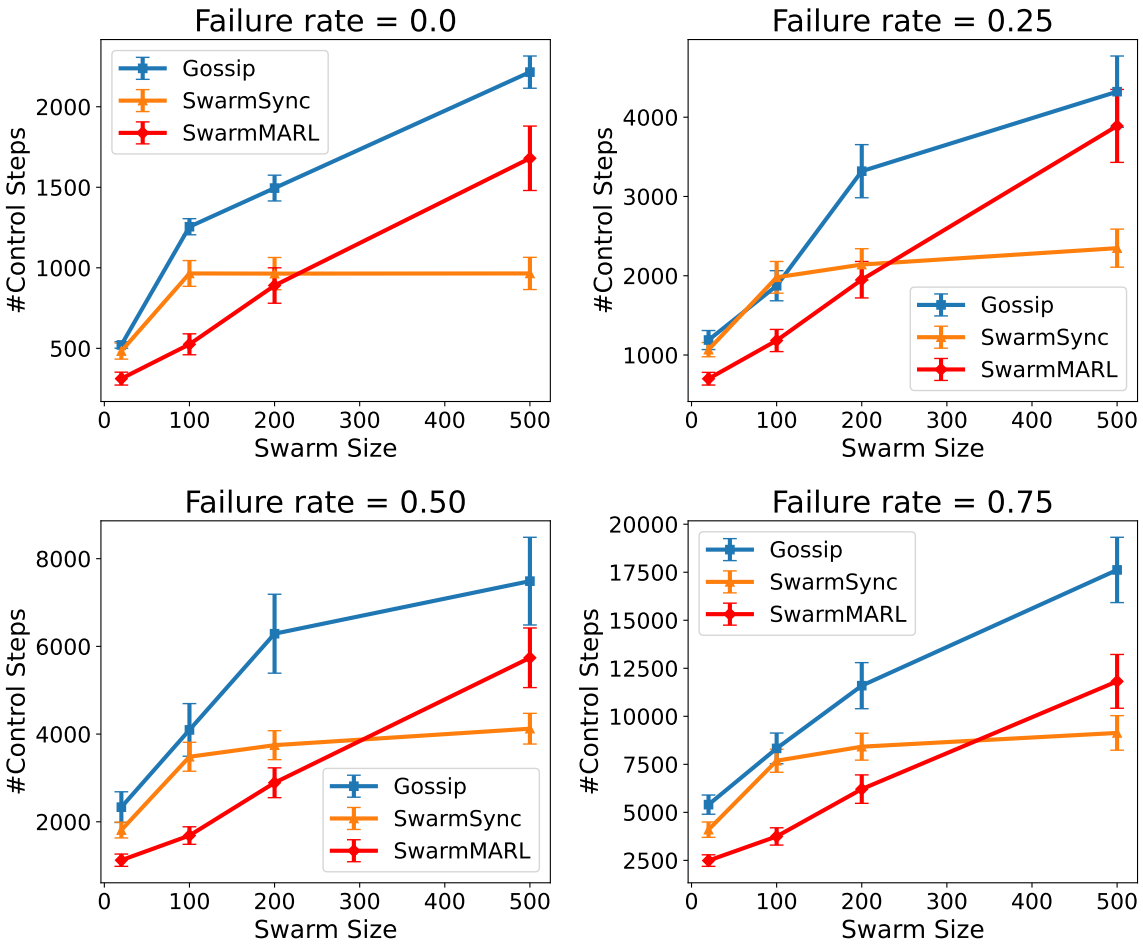


Figure 5.1: SwarmMARL convergence time (control steps) across swarm sizes $s \in \{20, 100, 200, 500\}$ and failure rates $f \in \{0, 0.25, 0.50, 0.75\}$, compared against SwarmSync and Gossip.

neighborhoods are covered, avoiding the accumulation of wasted transmissions that makes naive gossip slower in the later stages of dissemination.

SwarmMARL is faster than SwarmSync in most configurations, with the gap narrowing at large swarm sizes under high failure rates. Across most swarm sizes and failure rates, SwarmMARL achieves lower convergence times than

SwarmSync. At $f = 0.75$ and swarm size 500 however, SwarmMARL requires 11,820 steps compared to SwarmSync's 9,135, the only configuration in which SwarmSync converges faster. This narrowing reflects the fact that as swarm size and packet loss both increase simultaneously, the hop-by-hop propagation across a large flat topology becomes less reliable, and SwarmSync's structured retransmission begins to close the gap. In all other tested configurations, SwarmMARL converges faster. **SwarmMARL consistently outperforms Gossip across all tested configurations.** While both SwarmMARL and Gossip use flat multi-source dissemination, Gossip broadcasts indiscriminately regardless of whether neighbors have already received the update. SwarmMARL's learned suppression policy prevents this redundancy, resulting in faster convergence at every swarm size and failure rate. At $f = 0.75$ and swarm size 500, SwarmMARL converges in 11,820 steps compared to Gossip's 17,620, a 32.9% reduction, demonstrating that the learned policy retains a meaningful advantage over unstructured rebroadcasting even under the most challenging tested conditions.

5.5.2 Transmission Overhead

Figure 5.2 shows the transmission overhead of SwarmMARL compared against SwarmSync and Gossip.

SwarmMARL generates more overhead than SwarmSync but less than Gossip across all tested configurations. At $f = 0.0$ and swarm size 100, SwarmMARL produces 81.0 MB compared to 24.1 MB for SwarmSync and 246.0 MB for Gossip. This ordering is consistent across all swarm sizes and failure rates. Swar-

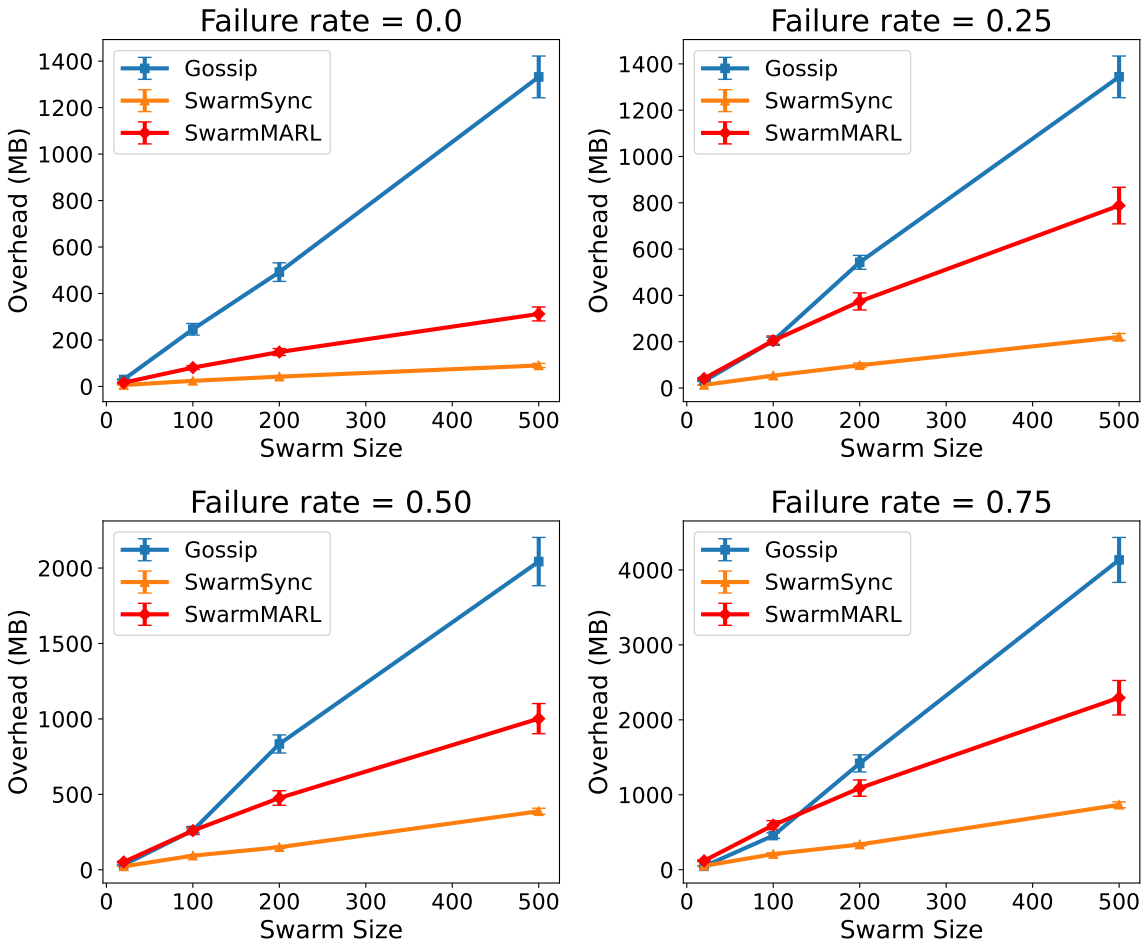


Figure 5.2: SwarmMARL transmission overhead (MB) across swarm sizes $s \in \{20, 100, 200, 500\}$ and failure rates $f \in \{0, 0.25, 0.50, 0.75\}$, compared against SwarmSync and Gossip.

mMARL's overhead exceeds SwarmSync's because its flat multi-source broadcasting structure causes multiple updated agents to transmit simultaneously to overlapping neighborhoods, whereas SwarmSync routes each packet through a single designated sender per phase. SwarmMARL's overhead remains below Gossip's because the learned policy suppresses broadcasts once the neighbor update ratio η rises, avoiding

the indiscriminate rebroadcasting that Gossip performs regardless of neighborhood coverage.

SwarmMARL's overhead grows more steeply with failure rate than Gossip's. At swarm size 100, SwarmMARL overhead grows from 81.0 MB at $f = 0.0$ to 595.0 MB at $f = 0.75$, a factor of 7.3×. Gossip grows from 246.0 MB to 455.3 MB over the same range, a factor of only 1.9×. Gossip's relative stability under packet loss reflects its inherent redundancy: because multiple agents broadcast the same packet simultaneously, the probability that all copies are dropped remains low, so receivers accumulate sufficient copies without additional retransmission. SwarmMARL lacks this built-in redundancy — when the learned policy suppresses a broadcast and that packet is subsequently lost by the remaining sender, a retransmission cycle is required, contributing to overhead growth under high failure rates.

SwarmMARL's overhead scales linearly with swarm size, consistent with SwarmSync and unlike Gossip. At $f = 0.0$, SwarmMARL overhead grows from 16.2 MB at swarm size 20 to 312.0 MB at swarm size 500, a factor of 19.3× for a 25× increase in swarm size. SwarmSync grows from 6.0 MB to 90.2 MB, a factor of 15.0×. Gossip grows from 30.0 MB to 1,332.0 MB, a factor of 44.4×, reflecting its inability to suppress redundant transmissions as the number of agents and propagation paths increases. The near-linear scaling of both SwarmMARL and SwarmSync confirms that both approaches avoid the super-linear overhead growth that makes unstructured gossip impractical at large swarm sizes.

5.5.3 Training Convergence

Training stability is evidenced by Figure 5.3, which shows all four loss components over 58,350 gradient steps. Figure 5.4 shows the swarm convergence time over training episodes.

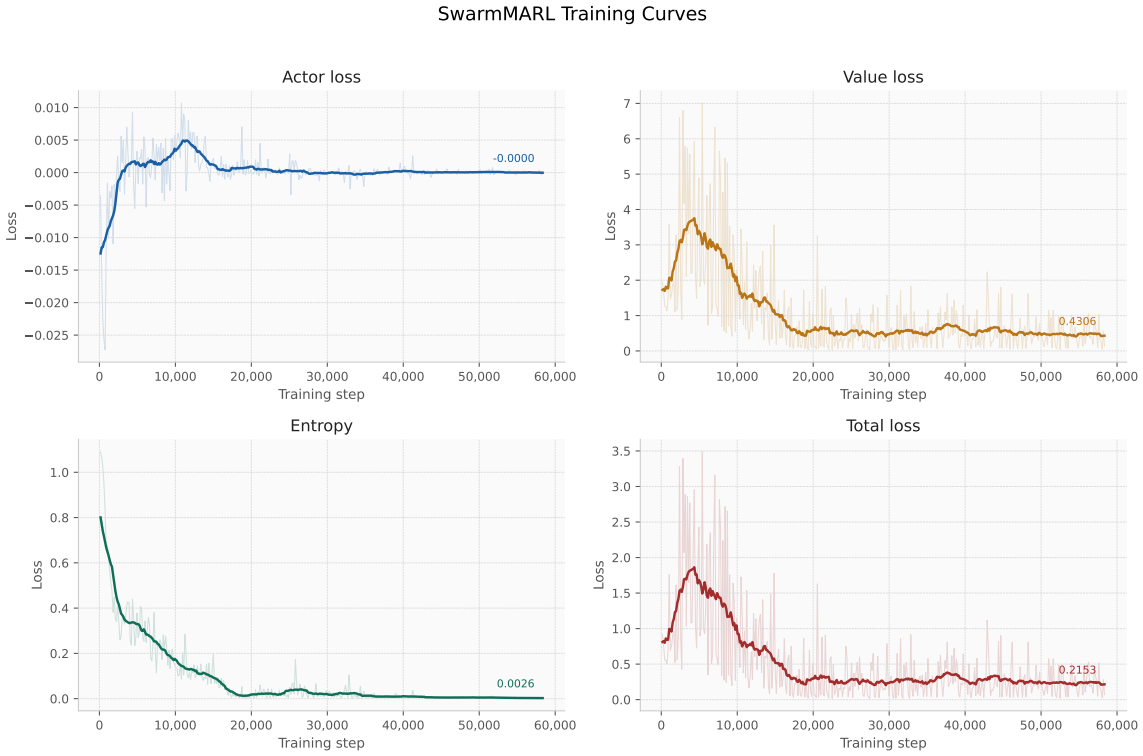


Figure 5.3: SwarmMARL training loss curves over 58,350 gradient steps. Faint lines show raw per-step values; solid lines show a rolling average. Value loss and total loss decline sharply in the first 10,000 steps as the critic learns to predict returns, then stabilise near zero. Entropy decays monotonically from 1.09 to near zero, consistent with the annealing schedule, indicating the policy transitions from near-uniform exploration to confident action selection. Actor loss remains close to zero throughout, reflecting PPO's clipping constraint preventing large policy updates. The stable plateau after approximately step 20,000 indicates policy convergence.

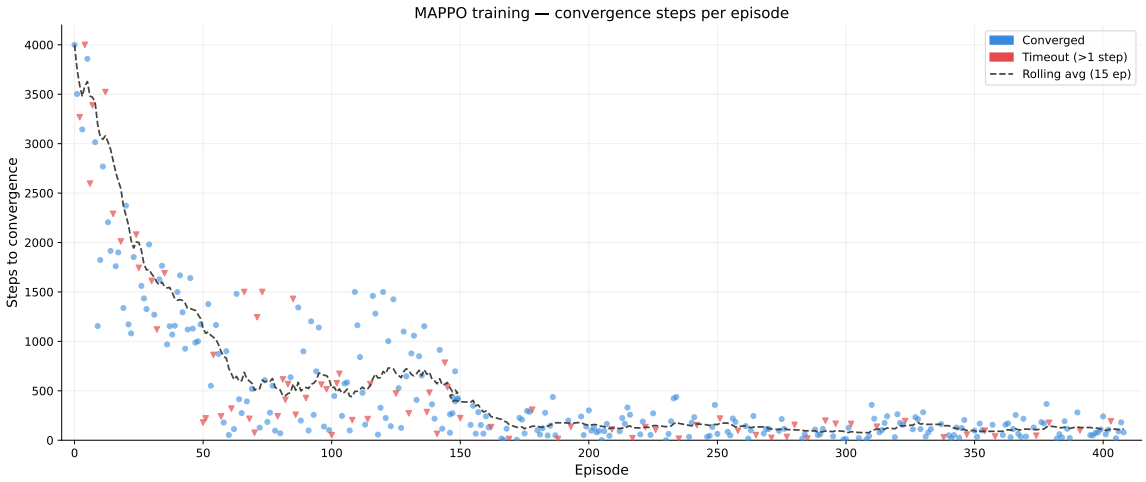


Figure 5.4: SwarmMARL convergence time over training episodes, showing the number of control steps required for all agents to reach the target software version within each episode. Episodes that timed out without full convergence are marked with red indicators.

5.5.4 Discussion of Standalone Results

The standalone evaluation of SwarmMARL demonstrates that a shared MAPPO policy trained under a single configuration of 100 agents and $f = 0.25$ learns coordination behaviors that generalize meaningfully across a range of swarm sizes and failure rates not seen during training.

The learned policy produces sensible coordination behavior without explicit rules. The convergence and overhead results confirm that the policy has learned the core coordination behaviors targeted by the reward function: updated agents broadcast to outdated neighbors, outdated agents request the update, agents suppress transmissions when their neighborhood is already covered, and critically low-battery agents reduce their transmission activity. None of these behaviors were

programmed explicitly, they emerged from the reward signal over the course of training.

Training converged stably. The training curves in Figure 5.3 show that value loss and total loss decline sharply in the first 10,000 gradient steps and stabilise near zero thereafter. Entropy decays monotonically from 1.09 to near zero, consistent with the annealing schedule, indicating the policy moved from broad exploration of the three-action space toward confident, context-sensitive action selection. The stable plateau after approximately step 20,000 confirms that the policy reached a stable equilibrium rather than oscillating or collapsing.

Performance degrades gracefully under conditions further from the training distribution. SwarmMARL was trained at swarm size 100 and $f = 0.25$, yet the evaluation results show meaningful coordination at swarm sizes 20 and 500 and failure rates up to $f = 0.75$. Convergence was achieved in all tested configurations, and the overhead remained bounded. This suggests that the observation features — particularly the neighbor update ratio η and the version gap δ_v — provide sufficient local context for the policy to make reasonable decisions even when the global swarm state differs from what was seen during training. That said, performance does degrade at the most extreme configuration of $f = 0.75$ and swarm size 500, which is the furthest from the training distribution and the only condition where convergence time exceeds SwarmSync's.

The absence of formal delivery guarantees is the primary limitation. SwarmMARL provides no equivalent to the termination proof established for SwarmSync in Section 4.2.2. The evaluation results show that convergence was achieved

in all tested configurations, which provides empirical evidence of reliability under these specific conditions. However, this does not constitute a formal guarantee, and the policy's behavior under conditions not represented in the evaluation — including adversarial observations, hardware faults, or significantly larger swarms — remains unverified. This is the fundamental trade-off that the comparative analysis in Chapter 6 examines more directly.

Chapter 6

Unified Evaluation and Comparison

6.1 Comparison of SwarmSync and SwarmMARL

This section directly compares SwarmSync and SwarmMARL across the shared experimental conditions established in Chapter 3, using the results from Chapters 4 and 5. The comparison addresses three dimensions: convergence time, transmission overhead, and practical deployment considerations.

6.1.1 Convergence Time

SwarmMARL achieves faster convergence than SwarmSync in most tested configurations. The speed advantage is most pronounced at small-to-medium swarm sizes, where SwarmMARL's flat multi-source dissemination structure allows multiple up-

dated agents to broadcast simultaneously from the first control step, while SwarmSync must first complete its leader election and phase sequencing before dissemination begins. At $f = 0.0$ and swarm size 100, SwarmMARL converges in 525 steps compared to SwarmSync's 965, a 45.6% reduction.

The time difference gets smaller as the size of the swarm increases. At $f = 0.75$ and swarm size 500, SwarmSync converges in 9,135 steps compared to SwarmMARL's 11,820. This reflects the interaction between swarm scale and SwarmMARL's flat topology, which requires updates to propagate hop by hop across the full swarm diameter. As the swarm grows, this propagation path lengthens, compounding coordination overhead in a way that SwarmSync's hierarchical phases avoid by confining synchronisation to bounded leader-to-follower links. Larger swarms therefore favour SwarmSync, while smaller swarms reduce this topological disadvantage and closes the gap between the two approaches.

In terms of scaling behavior, SwarmSync is more predictable. At $f = 0.25$, SwarmSync's convergence time grows by a factor of 2.2× from swarm size 20 to 500, while SwarmMARL grows by a factor of 5.6× over the same range. For deployments where swarm size will grow significantly over time, SwarmSync's more stable scaling behavior is a meaningful practical advantage even if its per-episode convergence time is higher.

6.1.2 Transmission Overhead

SwarmSync produces lower overhead than SwarmMARL in every tested configuration. At $f = 0.0$ and swarm size 100, SwarmSync generates 24.1 MB compared

to SwarmMARL's 81.0 MB. At $f = 0.75$ and swarm size 500, SwarmSync generates 866.2 MB compared to SwarmMARL's 2,295.0 MB. The consistent overhead advantage of SwarmSync reflects its fundamental structural difference: each packet is transmitted exactly once per designated sender per phase, so the total number of transmissions is tightly bounded by the update payload size and the swarm size. SwarmMARL's flat multi-source structure inherently produces more transmissions because multiple agents broadcast to overlapping neighborhoods simultaneously, and while the learned policy suppresses a significant fraction of these redundant transmissions, it cannot eliminate them entirely.

Both approaches scale near-linearly with swarm size in terms of overhead growth rate, which distinguishes them from Gossip, whose overhead grows much faster. However, SwarmMARL's overhead grows more steeply with failure rate than SwarmSync's in relative terms at smaller swarm sizes, while at larger swarm sizes the two approaches converge in their failure-rate sensitivity. This is consistent with the interpretation that SwarmMARL's overhead under packet loss is dominated by retransmission cycles triggered when the policy's suppression of a broadcast turns out to have been incorrect — a cost that Gossip avoids through redundancy and that SwarmSync manages through ACK-guaranteed single-sender delivery.

6.1.3 Statistical Analysis

Convergence Time

To verify that observed differences in convergence steps between SwarmSync and SwarmMARL are statistically significant, Mann-Whitney U tests were conducted for

each combination of swarm size and failure rate condition. The Mann-Whitney U test was selected as it makes no assumption of normality and is appropriate for comparing two independent samples. Each configuration was evaluated over 20 independent runs. Effect size was quantified using rank-biserial correlation r, similar to Section 4.6, we follow interpretation guidelines of Vargha and Delaney [116], we treat $|r| < 0.11$ as negligible, $0.11 \leq |r| < 0.28$ as small, $0.28 \leq |r| < 0.43$ as medium, and $|r| \geq 0.43$ as large. Any non-significant entries will have an effect size of "N/A" as they have no effect. All tests were two-tailed with a significance threshold of $alpha = 0.05/16 \approx 0.00313$ from applying apply Bonferroni correction for the 16 tests in the table.

Results are summarised in Table 6.1. The pattern of results confirms that swarm size is the primary driver of relative performance between the two algorithms. At small swarm sizes, SwarmMARL consistently converges faster than SwarmSync, with significant negative effect sizes observed across all failure rates at sizes 20 and 100. This advantage diminishes and eventually reverses as swarm size increases, with SwarmSync outperforming SwarmMARL at size 500 across most conditions.

The crossover point varies by failure rate. At $f = 0.00$, the difference at size 200 is non-significant ($p = 0.304$, $r = 0.193$), indicating neither algorithm holds a clear advantage at that scale. At $f = 0.75$ and size 500, the result is also non-significant ($p = 0.060$, $r = 0.350$), suggesting that at the highest failure rate the transition to SwarmSync's advantage is less decisive, likely because the retransmission costs at extreme packet loss partially offsets SwarmSync's hierarchical efficiency gains.

Overall, the statistically confirmed reversal in relative performance across swarm sizes supports the conclusion that SwarmMARL is better suited to small swarms

while SwarmSync becomes the more efficient protocol as swarm size grows.

Table 6.1: Mann-Whitney U test results for SwarmSync vs SwarmMARL across swarm sizes and failure rates. Effect size r is the rank-biserial correlation; negative values indicate SwarmMARL converges faster, positive values indicate SwarmSync converges faster. (entries where it is statistically non-significant is highlighted in bold)

Failure Rate	**Swarm Size**	p	r
$f = 0.00$	20	**0.009**	**N/A**
	100	<0.001	−0.890
	200	**0.304**	**N/A**
	500	0.002	0.570
$f = 0.25$	20	<0.001	−0.720
	100	**0.011**	**N/A**
	200	<0.001	−0.665
	500	<0.001	0.680
$f = 0.50$	20	**0.010**	**N/A**
	100	<0.001	−0.930
	200	0.001	−0.615
	500	**0.029**	**N/A**
$f = 0.75$	20	0.003	−0.560
	100	<0.001	−0.895
	200	0.003	−0.555
	500	**0.060**	**N/A**

Overhead

While SwarmMARL demonstrates faster convergence at smaller swarm sizes, this advantage comes at a consistent communication cost. Table 6.2 presents the Mann-Whitney U test results comparing the total communication overhead, measured in megabytes, between SwarmSync and SwarmMARL across all swarm sizes and failure

rate conditions.

Unlike the convergence results, where relative performance reversed depending on swarm size, the overhead comparison reveals a uniform pattern: SwarmSync produces significantly lower communication overhead than SwarmMARL in every condition tested ($p \leq 0.017$, $r \geq 0.443$). This consistency across all failure rates and swarm sizes suggests that SwarmSync's hierarchical structure fundamentally reduces message redundancy, regardless of scale or network reliability.

SwarmMARL's higher overhead is a direct consequence of its flat topology. Without a hierarchical coordination layer, updates must propagate hop by hop across the full swarm, requiring each agent to repeatedly broadcast to its neighbours until the update reaches the entire swarm. As failure rates increase, this propagation demands additional retransmissions, compounding the overhead further. SwarmSync, by contrast, confines coordination to bounded leader-to-follower links, limiting the total volume of messages required regardless of how many retransmissions packet loss induces.

This trade-off has practical implications for deployment. In bandwidth-constrained environments, SwarmMARL's faster convergence at small scales may not justify its higher communication cost, particularly as even modest increases in swarm size erode its speed advantage while its overhead remains elevated. SwarmSync's consistent efficiency across all tested conditions makes it the more scalable choice as swarm size and failure rates grow.

Table 6.2: Mann-Whitney U test results for SwarmSync vs SwarmMARL across swarm sizes and failure rates for communication overhead. Effect size r is the rank-biserial correlation; negative values indicate SwarmMARL has lower overhead, positive values indicate SwarmSync has lower overhead.

Failure Rate	**Swarm Size**	p	r
$f = 0.00$	20	<0.001	0.662
	100	0.001	0.603
	200	<0.001	0.853
	500	0.001	0.597
$f = 0.25$	20	<0.001	0.647
	100	0.001	0.520
	200	0.002	0.443
	500	<0.001	0.790
$f = 0.50$	20	<0.001	0.720
	100	0.002	0.580
	200	<0.001	0.645
	500	0.002	0.565
$f = 0.75$	20	<0.001	0.853
	100	<0.001	0.883
	200	0.002	0.570
	500	0.002	0.585

6.1.4 Practical Deployment Considerations

Beyond the quantitative results, the two approaches differ along several practical dimensions that are relevant to deployment decisions.

Formal guarantees. SwarmSync provides a provable termination guarantee under the assumptions of Chapter 3: every compatible agent eventually receives every required packet. SwarmMARL provides no equivalent guarantee. The empirical results show convergence in all tested configurations, but this does not extend to untested conditions, hardware faults, or adversarial observation perturbations. For

mission-critical deployments where an incomplete update is more dangerous than a slow one, SwarmSync's formal guarantee is a meaningful advantage.

Configurability. SwarmSync requires a defined topology, either through dynamic leader election in simulation or through predetermined configuration. SwarmMARL requires no topology specification: coordination behavior emerges from the learned policy applied independently at each agent. This makes SwarmMARL easier to deploy in swarms where topology is unknown or changes frequently, at the cost of requiring a trained policy that may not generalize to significantly different conditions.

Interpretability. SwarmSync's behavior at every step is determined by explicit protocol rules and is fully interpretable: an operator can trace why a particular packet was sent or retransmitted. SwarmMARL's behavior is determined by a neural network policy and is not directly interpretable. Diagnosing unexpected coordination behavior in SwarmMARL requires inspecting the observation vector and policy output at each step, which is more demanding than reading a protocol trace.

Real-world validation. Both approaches were validated on physical Crazyflie 2.1 Brushless hardware in a controlled laboratory environment, as described in Chapter 7. Both achieved convergence in all physical runs, confirming that neither approach is a purely simulation artifact. The physical experiments also revealed that SwarmMARL required server-side observation construction and policy inference due to the limited sensing and compute capabilities of the physical platform, while SwarmSync operated with the orchestrator running entirely on the ground server with no additional inference overhead.

Summary. Table 6.3 summarizes the key trade-offs between the two approaches

Table 6.3: Summary comparison of SwarmSync and SwarmMARL across key evaluation dimensions.

Dimension	SwarmSync	SwarmMARL
Convergence speed	Slower (most configs)	**Faster (most configs)**
Transmission overhead	**Lower**	Higher
Scaling with swarm size	**More stable**	Steeper growth
Formal guarantees	**Yes**	No
Topology required	Yes	**No**
Interpretability	**High**	Low
Training required	**No**	Yes
Physical validated	**Yes**	**Yes**

across the dimensions evaluated in this thesis.

Neither approach dominates the other across all dimensions. SwarmSync is preferable when formal reliability guarantees, low overhead, and predictable scaling are the primary concerns. SwarmMARL is preferable when convergence speed and deployment flexibility matter more, and when the swarm operates under conditions sufficiently similar to the training distribution. The choice between them is ultimately mission-dependent, and both can be combined with SwarmModelPatch to reduce the update payload regardless of which coordination approach is used. Table 6.4 summarizes when to choose SwarmSync versus SwarmMARL based on the deployment scenario.

Table 6.4: Decision guide for choosing between **SwarmSync** and **SwarmMARL** based on deployment scenario.

Scenario / Priority	**Choose**	**Why**
Small-to-medium swarm	**SwarmMARL**	Up to 45.6% faster convergence
Large swarm ($\geq$500 agents)	**SwarmSync**	Avoids hop-by-hop cost
Swarm expected to grow	**SwarmSync**	More predictable, stable scaling
Bandwidth-constrained	**SwarmSync**	Lower transmission overhead
High failure rate situations	**SwarmSync**	Better at handling failure
Mission-critical, guaranteed delivery needed	**SwarmSync**	Provable termination
Topology unknown or changes frequently	**SwarmMARL**	No topology required
Want to audit/debug decisions	**SwarmSync**	Fully interpretable trace
No training pipeline available	**SwarmSync**	Requires no policy training
Fastest convergence	**SwarmMARL**	Speed and deployment flexibility
Payload size reduction	**SwarmSync** & **SwarmMARL**	Either with SwarmModelPatch

Chapter 7

Real-World Validation

The simulation experiments presented in Chapters 4 and 5 establish the performance characteristics of SwarmSync and SwarmMARL under controlled and reproducible conditions. However, simulation environments make idealized assumptions such as synchronous control steps, binary packet loss, circular communication ranges, and instantaneous state reads, that do not hold in physical deployment. This chapter describes the real-world validation experiments conducted on a fleet of Crazyflie 2.1 Brushless drones in a controlled laboratory environment, demonstrating that both SwarmSync and SwarmMARL transfer from simulation to physical hardware and that the core coordination mechanisms operate correctly in the presence of real-world communication characteristics.

The real-world experiments are designed as a proof-of-concept validation rather than a quantitative benchmark. As discussed in Section 7.2, the physical environment introduces conditions that make formal metric collection impractical and, more

importantly, not directly comparable to the simulation results: packet loss is negligible under laboratory wireless conditions, convergence occurs faster than any of the simulated failure rate scenarios, and transmission overhead is not measurable at the level of granularity used in simulation. The primary questions the real-world experiments answer are therefore qualitative: do both approaches operate correctly on physical hardware, does the hierarchy work end-to-end on real drones, and does the learned SwarmMARL policy produce sensible coordination behavior when inference is offloaded to a ground server?

7.1 Hardware and Software Setup

7.1.1 Hardware Platform

The physical experiments use a fleet of six Crazyflie 2.1 Brushless drones, a lightweight open-source quadrotor platform developed by Bitcraze with a total weight of approximately 35 g. The Brushless variant was selected over the standard Crazyflie 2.1 for its improved thrust-to-weight ratio, which improves stability during hover sequences. All drones communicate wirelessly using the Bitcraze Crazyradio 2.0 USB dongle, which operates at 1 Mbps in the 2.4 GHz band, the same transmission speed used in the simulation experiments, as described in Chapter 3.

Position tracking is provided by a Vicon motion capture system, which tracks retroreflective markers attached to each drone and publishes pose estimates at high frequency over a local network. The Vicon system is used for flight stabilization during the hover-on-convergence sequences rather than for the update dissemination process

itself: update packets are transmitted over the Crazyradio RF link independently of the Vicon infrastructure, meaning the update protocol does not depend on the motion capture system to function. The Vicon data is consumed by the Crazyflie onboard EKF for state estimation during hover, with an 8-second warmup period before arming to allow the filter to converge on an accurate position estimate.

7.1.2 Software Stack

The ground server runs Ubuntu 22.04 with ROS 2 Humble and the CrazySwarm2 [90] framework, which provides the ROS 2 interface to the Crazyflie fleet including the `crazyflie_interfaces` service definitions for arming, takeoff, and landing. The update dissemination logic runs as a ROS 2 node (`radio_orchestrator`) on the ground server, communicating with per-drone ROS 2 nodes (`drone_update_node`) over ROS 2 topics using a reliable QoS profile. The overall communication architecture is shown in Figure 7.1.

Each drone runs a `drone_update_node` that subscribes to its own dedicated packet topic (`/swarm/rab/<robot>`) and a shared broadcast topic (`/swarm/rab_broadcast`), and publishes ACKs to `/swarm/ack/<robot>`. Packets are represented as `RabPacket` messages containing a sender ID, target ID, and a fixed-size data array encoding the robot type, message type, version number, page count, and page index. This message format is intentionally minimal to match the constrained communication bandwidth of the Crazyradio link.

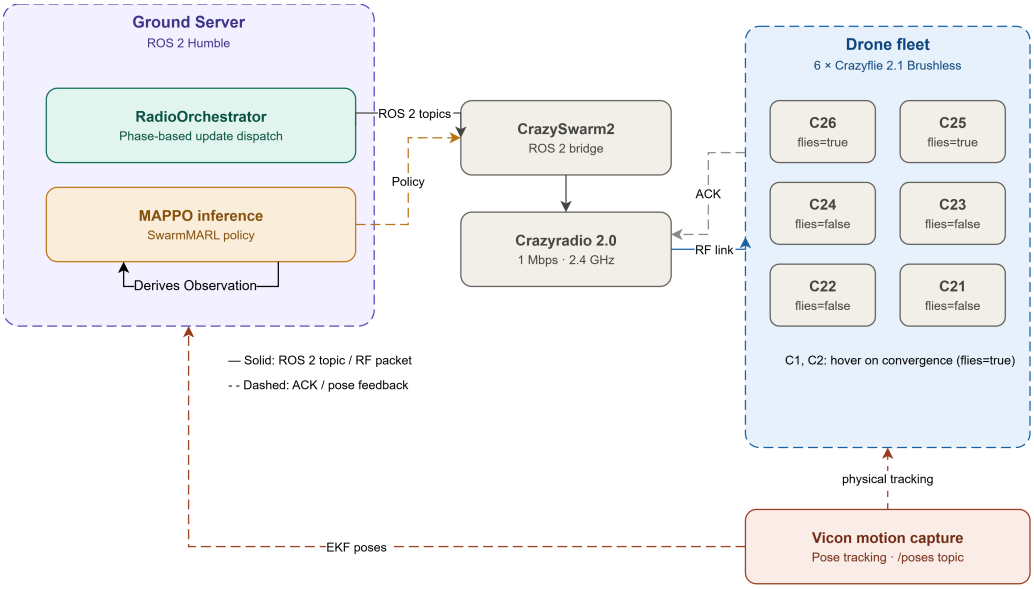


Figure 7.1: Real-world system architecture. The ground server runs the RadioOrchestrator node for SwarmSync experiments and the MAPPO inference process for SwarmMARL experiments. CrazySwarm2 bridges ROS 2 topics to the Crazyradio 2.0 RF link. The Vicon motion capture system provides pose estimates used by the Crazyflie onboard EKF during hover sequences. Two drones (C26, C25) are configured with `flies=true` and execute a hover sequence as a physical convergence signal; the remaining four drones stay grounded throughout the update.

7.1.3 Topology Configuration

Unlike the simulation, where SwarmSync's sub-swarm structure and leader election emerge dynamically from the swarm's spatial configuration, the physical experiments use a pre-defined topology specified in a YAML configuration file (`swarm_update_config.yaml`). This configuration defines the update payload size, the version number, the maximum number of simultaneous links from the orchestrator and from each relay drone, and the hierarchical tree structure of the fleet, specifying which drones act as relay nodes and which are leaf receivers. The orchestrator reads this configuration at startup and auto-generates an ordered list of dissemination phases using breadth-first traversal of

the tree, executing each phase sequentially and waiting for all targets in a phase to ACK before advancing.

This pre-defined topology is a practical adaptation of SwarmSync to the physical setting. With only six drones and a single Crazyradio dongle constrained to one simultaneous link (`updater_links: 1`), the entire fleet fits within three update phases. Dynamic leader election and spatial sub-swarm formation of the simulation are unnecessary due to small swarm size. The YAML-driven design also allows the hierarchy to be reconfigured for different fleet compositions without modifying the orchestrator code, making it straightforward to extend to larger fleets.

7.2 Real-World Experiment Objectives

The real-world experiments serve a distinct and complementary purpose to the simulation study. Where the simulation provides controlled, repeatable quantitative benchmarking across a wide range of swarm sizes and failure conditions, the physical experiments address a different set of questions: can both protocols operate correctly on real hardware, do they produce the expected coordination behaviours end-to-end, and does the MARL policy transfer from simulation to physical deployment without retraining? These are feasibility and validation questions, and they are the primary motivation for conducting hardware experiments at all.

Both SwarmSync and SwarmMARL were deployed on the physical platform, allowing a direct side-by-side validation of both protocols under identical hardware conditions. The goal was not to reproduce the simulation's performance comparison,

but to confirm that each protocol behaves correctly in a physical setting and that the convergence signal: **drone hover**, is observable in practice for both approaches.

Formal quantitative metrics were not collected for three reasons. First, packet loss was negligible under laboratory wireless conditions with six drones in close proximity to a single Crazyradio dongle, meaning retransmissions were never triggered. Second, with zero packet loss and a 1 Mbps Crazyradio link, all 64 pages were delivered to all six drones in seconds, consistent with the simulation results at $f = 0$ but not extending the findings to the more challenging conditions that differentiate the strategies. Third, the ROS 2 topic interface and Crazyradio link do not expose per-packet transmission counts at the granularity used by the simulation's overhead metric, making a direct comparison of overhead figures impossible.

The quantitative performance characterization of both protocols is therefore provided entirely by the simulation experiments in Chapters 4 and 5, which offer the controlled conditions and metric granularity necessary for a meaningful comparison. The real-world experiments establish that the protocols are physically viable and behave as expected, a necessary foundation for trusting the simulation results, rather than providing comparative results.

7.3 SwarmSync Real-World Experiments

7.3.1 Experimental Protocol

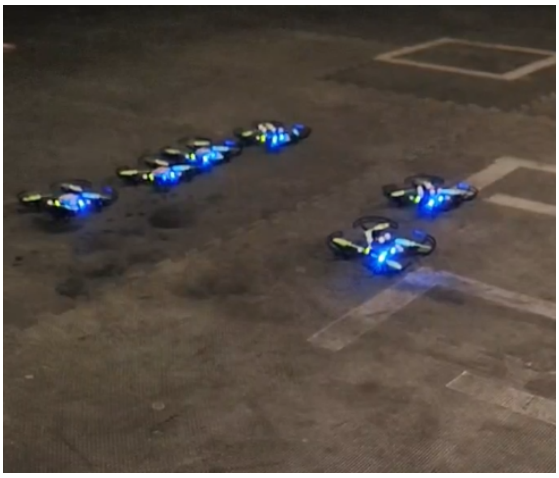
SwarmSync real-world experiments were conducted using the full fleet of six Crazyflie 2.1 Brushless drones. Each experiment began with a trigger published to `/swarm/trigger_update`,

causing the RadioOrchestrator to broadcast a version announcement to all drones and begin sequential page transmission. The update payload was configured to match the simulation: 64 pages per update, with each page corresponding to one RabPacket transmission. Multiple update runs were conducted by re-triggering the orchestrator after convergence, which automatically incremented the version number and reset all drone states.

To demonstrate the convergence signal physically, two of the six drones were configured with `flies: true` in the YAML topology. Upon receiving the `/swarm/converged` topic published by the orchestrator after all six drones ACKed all pages, these two drones executed a hover sequence: a Vicon-stabilized takeoff to 0.3 m, a 10-second hover, and a controlled landing. This hover sequence serves as an unambiguous physical indicator that the update protocol reached global convergence, a behavior that is straightforward to observe and record on video.

7.3.2 Results

All six drones successfully received all 64 pages and ACKed them to the orchestrator in every run conducted. The orchestrator's delivery metrics confirmed a total of 64 packets sent per run under clean laboratory conditions, indicating that no retransmissions were required, every page was delivered on the first attempt. Following convergence, the two designated flying drones executed the hover sequence correctly, taking off within 10 seconds of the `/swarm/converged` signal, hovering stably at 0.3 m for the configured duration, and landing cleanly. The remaining four grounded drones reset their state and confirmed readiness for the next update run, demonstrating that



(a) Fleet of six Crazyflie 2.1 Brushless drones on the laboratory floor before the update begins.

```
── PHASE 1 ── radio → C26
[radio→C26] page 1/64
[radio→C26] page 16/64
[C26] ACK page 16 ✓
[radio→C26] page 32/64
[C26] ACK page 32 ✓
[radio→C26] page 64/64
[C26] ACK page 64/64 — DONE ✓
[radio] Phase 1 complete — C26 ready to relay
── PHASE 2 ── C26 → C24, C23 (simulated radio) | radio → C25
[C26→C24] page 1/64 [C26→C23] page 1/64
[radio→C25] page 1/64
[C26→C24] page 16/64 [C26→C23] page 16/64
[radio→C25] page 16/64
[C24] ACK page 16 ✓ [C23] ACK page 16 ✓ [C25] ACK page 16 ✓
[C26→C24] page 32/64 [C26→C23] page 32/64
[radio→C25] page 32/64
[C24] ACK page 32 ✓ [C23] ACK page 32 ✓ [C25] ACK page 32 ✓
[C26→C24] page 64/64 [C26→C23] page 64/64
[radio→C25] page 64/64
[C24] ACK page 64/64 — DONE ✓ (via C26 relay)
[C23] ACK page 64/64 — DONE ✓ (via C26 relay)
[C25] ACK page 64/64 — DONE ✓
[radio] Phase 2 complete — C25 ready to relay
```

(b) RadioOrchestrator terminal output showing sequential page delivery and per-drone ACK confirmation. **Output shown is a condensed excerpt**; the full log was omitted as it is too verbose with non significant entries.

(c) Two designated drones hovering at 0.3 m as a physical convergence signal after all six drones completed the update.

(d) Fleet after landing and disarm, confirming successful completion of the full update and hover sequence.

Figure 7.2: Real-world experiment. Four stages of a single update run on six Crazyflie 2.1 Brushless drones in a controlled laboratory environment.

the protocol supports repeated update cycles without manual intervention.

The orchestrator's per-phase logging confirmed that the BFS-ordered phase structure operated as configured: the orchestrator advanced through each phase only after all targets in the current phase had fully ACKed, and the convergence signal was published only after the final phase completed. No timeouts were triggered in any run, consistent with the zero packet loss observed under laboratory wireless conditions.

7.4 SwarmMARL Real-World Experiments

7.4.1 Adaptations for Physical Deployment

The SwarmMARL real-world experiments required two significant adaptations relative to the simulation setup, both arising from the limited sensing capabilities of the Crazyflie 2.1 Brushless platform.

First, the observation space could not be populated directly from onboard drone sensors. In simulation, each agent constructs its local observation vector from state information written to a JSON file by the ARGoS controller, including update progress, neighbor versions, battery level, and transfer status. The Crazyflie 2.1 Brushless hardware has no capability to measure or report these values autonomously: it cannot sense its neighbors, assess its own update progress relative to others, or communicate peer-to-peer with other drones. Instead, all observation features were derived centrally on the ground server. Neighbor positions and distances were computed from Vicon pose data, update progress and version status were tracked by the RadioOrchestrator, and battery estimates were approximated from flight time.

The MAPPO inference process assembled these derived values into the observation vector for each drone, effectively constructing what each drone *would* observe if it had the sensing capabilities assumed in simulation. This means the physical SwarmMARL setup operates closer to centralized execution than the truly decentralized execution the simulation models: the server constructs observations and selects actions on behalf of each drone, rather than each drone observing and deciding independently. This is an important distinction that is discussed further as a limitation in Chapter 8.

Second, the file-based state-action bridge used in ARGoS was replaced with direct function calls within the ground server process, since both observation construction and action inference happen on the same machine. The RadioOrchestrator assembles the derived observation vectors at each control step, passes them to the MAPPO inference process, receives the selected actions, and dispatches the corresponding update packets to each drone via the Crazyradio link. This architecture preserves the logical separation between the coordination protocol and the inference process while eliminating the filesystem I/O of the simulation bridge entirely.

7.4.2 Experimental Protocol

SwarmMARL physical experiments used the same six-drone fleet as the SwarmSync experiments, with the ground server running both the CrazySwarm2 communication stack and the MAPPO inference process simultaneously. At each control step, the inference process read the state of all active drones, computed the observation vector o_i for each drone using the same `state_to_obs` function used in simulation, ran a forward pass through the trained actor-critic network, and published the selected

action (broadcast, request, or idle) to each drone. Drones executed the corresponding communication behavior and published their updated state for the next inference step.

Aligning with the SwarmSync experiments, two drones were configured to hover on convergence, providing a physical convergence signal. Convergence in the SwarmMARL experiments was determined by the inference process detecting that all active drones reported `current_version >= target_version`, at which point it published a convergence signal and the hover sequence began.

7.4.3 Results

The SwarmMARL policy successfully coordinated update dissemination across the physical fleet in all runs conducted. All six drones reached the target version in every run, and the two designated drones executed the hover convergence sequence correctly following completion. The policy's action outputs were qualitatively consistent with the expected learned behavior: drones that had not yet received the update predominantly selected the request action, drones that held the complete update predominantly selected the broadcast action, and drones with critically low battery or actively receiving a transfer predominantly selected idle. This action distribution confirms that the observation features encoding update status, transfer progress, and battery level are functioning correctly in the physical deployment and that the policy is reading them as intended.

No cases were observed in which the policy caused a drone to idle indefinitely or to select an action that prevented convergence, although the non-determinism of this

approach did cause delays in some runs. This is an important qualitative finding: one of the primary concerns about transferring a simulation-trained MARL policy to physical hardware is that out-of-distribution observations could cause the policy to collapse to a degenerate action (e.g., all agents idling). The physical experiments provide evidence that the policy is robust to the distribution shift between the simulated and physical observation spaces, at least under the clean wireless conditions of the laboratory environment.

7.5 Hardware Constraints

The real-world experiments surfaced several practical engineering constraints that are not apparent from simulation alone.

Drone-to-drone communication is absent in the physical setup, requiring the orchestrator to simulate the relay. In the ARGoS simulation, update propagation relies on peer-to-peer communication between drones: in SwarmSync, leaders receive the update from the Updater and then relay it directly to their followers over the swarm's wireless mesh; in SwarmMARL, each drone receives the update and relays it to its neighbours as needed. In the physical setup, the Crazyflie drones cannot communicate with each other directly, all communication is mediated by the ground server via the Crazyradio dongle. This means the multi-hop relay structure does not exist on physical hardware. To preserve the dissemination behavior, the orchestrator was configured to simulate the relay cycles itself: rather than transmitting the full update to all drones in a single broadcast, the orchestrator splits delivery into

sequential phases defined in the YAML topology configuration, transmitting to each group in turn and waiting for ACKs before advancing, mimicking the peer-to-peer propagation that would occur naturally in the simulation. This restraint is developed due to the small deployment size; in actual large-scale swarm deployment, the environment is closer to the simulation than our deployment in the lab. With only six drones available, a single cycle would have been sufficient to update all of them directly, so the phased structure was introduced deliberately to exercise and validate the dissemination logic rather than out of hardware necessity. In a real large-scale deployment, peer-to-peer capable hardware would be required to realise the parallelism and scalability advantages of propagation that the simulation demonstrates.

The Vicon EKF warmup period introduces a fixed overhead before hover. The `drone_update_node` includes a delay after receiving the convergence signal to allow the Crazyflie's onboard EKF to converge on a stable position estimate before arming. This delay is specific to the Vicon-based positioning setup and would not be present in GPS-based or UWB-based positioning systems. It does not affect the update dissemination process itself but adds latency between update completion and the physical convergence demonstration.

Server-side MARL inference introduces a communication dependency that must be managed in deployment. In the real-world SwarmMARL experiments, the policy inference server communicates with the drones over the Crazyradio 2.0 RF link via CrazySwarm2. This introduces a single point of failure that does not exist in the simulation: if the server becomes unreachable, drones have no fallback coordination mechanism. However, the server-side MARL inference was only

a trade-off made to compensate the weak computational power of the hardware on hand. In practice, practitioners deploying on hardware with sufficient onboard compute can execute the policy locally, eliminating the server dependency. The server-side architecture should be viewed as a practical expedient for hardware-constrained deployment rather than a fundamental requirement of the SwarmMARL approach.

7.6 Deployment Lessons

Beyond hardware constraints, the physical experiments also yielded broader lessons about the gap between simulation and deployment, and the practical trade-offs between the two approaches.

Physical validation reveals the limits of simulation as a proxy for deployment readiness. The ARGoS simulation was valuable for stress-testing dissemination logic under controlled failure conditions at scale, but it abstracted away several layers of real-world complexity: hardware-specific communication constraints, EKF convergence delays, and the operational overhead of managing physical devices such as battery swaps, firmware compatibility checks, and radio interference. As a result, the path from a working simulation to a working physical deployment required non-trivial re-engineering effort that the simulation results alone gave no indication of.

Both approaches benefit from real-world conditions. A finding that unifies the SwarmSync and SwarmMARL physical experiments is that both approaches performed better than their worst-case simulation results under real laboratory con-

ditions, primarily because the laboratory wireless environment produced negligible packet loss. This confirms the interpretation from the simulation results: both approaches are effective under benign channel conditions, and their performance difference is most significant under the stressed conditions (high failure rates, large swarm sizes) that the simulation is specifically designed to test. Practitioners deploying in environments with reliable wireless infrastructure can expect both approaches to perform well; the choice between them becomes most consequential in degraded channel conditions.

Chapter 8

Discussion and Limitations

This chapter discusses the implications of the findings presented in Chapters 4, 5, and 6. We begin in Section 8.1 with a high-level comparative framing of the two coordination philosophies investigated in this thesis, identifying when each is preferable and what the existence of both approaches reveals about the update propagation problem. Section 8.2 discusses implications specific to SwarmSync and SwarmModelPatch. Section 8.3 discusses implications specific to SwarmMARL. Section 8.4 discusses the real-world implementation and what the hardware validation reveals about the gap between simulation and deployment. Section 8.5 covers cross-cutting implications that apply to both approaches: security, energy consumption, environmental obstacles, generalization to other simulators, and dynamic transfer rates. Section 8.6 consolidates the methodological limitations and scope constraints of the work.

8.1 Deterministic vs. Learned Coordination

The central comparative question of this thesis is not which approach is better in absolute terms, but under what conditions each is preferable. The results in Chapter 4 and Chapter 6 provide empirical grounding for this question; this section interprets those results in terms of the underlying trade-offs between the two philosophies.

SwarmSync is preferable when reliability guarantees matter most. The ACK-based delivery semantics of SwarmSync provide a formally provable termination guarantee under the assumptions of Chapter 3: every compatible agent will eventually receive every required packet, regardless of individual agent behavior. This guarantee is particularly valuable in mission-critical deployments where an incomplete update is more dangerous than a slow one, for example, when a DL model update is required for hazard avoidance and a partially updated swarm would produce inconsistent perception outputs. No learned policy can provide an equivalent guarantee without additional formal verification, because the policy's behavior in out-of-distribution scenarios cannot be bounded by construction.

SwarmMARL is preferable when dissemination speed matters most, despite its higher overhead. As seen in Section 5.1, across small to moderate swarm sizes, the learned policy consistently achieves faster update propagation than SwarmSync. However, this speed advantage diminishes as swarm size grows: at large scales (¿200 UAVs), SwarmSync's hierarchical propagation structure allows it to close the gap and eventually match or surpass SwarmMARL's completion time, while SwarmMARL's communication overhead remains consistently higher throughout. The practical implication is that SwarmMARL is best suited to smaller, dynamic deploy-

ments where propagation speed is the primary objective, while SwarmSync becomes the more efficient choice as fleet size increases. As with any learned policy, SwarmMARL also sacrifices formal reliability guarantees and interpretability: it is not straightforward to reason about why the policy makes a particular decision in a given state, and behaviour may degrade in scenarios not well represented in the training distribution.

The comparison reveals a fundamental tension in decentralized update coordination. Protocol-based approaches encode knowledge about correct behavior explicitly, making them interpretable and formally analyzable but rigid. Policy-based approaches learn correct behavior from experience, making them flexible and adaptive but opaque and unverifiable. This tension is not unique to the UAV swarm update problem. In wireless network protocol design, traditional protocols such as ALOHA [3] and CSMA [53] are products of hand-crafted heuristics that underperform under traffic heterogeneity and topological diversity, motivating RL-based approaches that sacrifice predictability for adaptability. Similarly, in swarm robotics, hand-crafted state-machine controllers are valued for their reliability and simplicity but often require extensive manual tuning and struggle in dynamically changing environments [70, 50].

8.2 Implications of SwarmSync and SwarmModel-Patch

SwarmSync's hierarchical structure provides better scalability than flat dissemination. The logarithmic dissemination time established formally in Section 4.2.1 and confirmed empirically in Chapter 4 demonstrates that organizing UAVs into role-based sub-swarms with designated leaders fundamentally changes the scaling behavior of the update process. Gossip-based protocols achieve the same asymptotic complexity in theory but with substantially larger constant factors due to redundant rebroadcasting; SOUL's sequential grouping produces linear scaling in swarm size. For practitioners deploying update strategies in large swarms, the scalability advantage of SwarmSync compounds with swarm size and becomes most significant in the 200 and 500 UAV configurations evaluated in this thesis.

Selecting resource-rich UAVs as leaders can further reduce synchronization time in real-world deployments. In our experiments, leaders are assigned randomly within each sub-swarm, since UAVs of the same type are identical in computational capability in the simulation. In practice, selecting leaders based on factors such as communication bandwidth, battery life, computational power, and spatial position within the sub-swarm could shorten synchronization paths, reduce retransmissions, and amortize coordination costs [113]. Practitioners should prioritize leader assignment as a deployment-time optimization rather than leaving it to random selection.

Standard cryptographic protections can be layered onto SwarmSync

without affecting performance. Experiments with TLS encryption applied to SwarmSync showed no impact on convergence time beyond a few bytes of per-message overhead. Similarly, incorporating HMAC-based integrity verification into the data transmission protocol increased both efficiency and overhead by less than 5%, varying between different swarm sizes. These results confirm that practitioners can adopt well-established cryptographic protections — TLS for TCP-based communication and DTLS for UDP-based communication — without sacrificing the performance benefits of SwarmSync. Security integration should therefore be considered a standard deployment practice rather than an optional enhancement.

The accuracy-update speed trade-off introduced by SwarmModelPatch should be tuned per mission. The formal relationship $S = \alpha \cdot (P - P_f)$ established in Section 4.3.1 makes the patch size trade-off explicit and tunable: freezing more layers reduces patch size at the cost of model expressivity, while freezing fewer layers preserves adaptability at the cost of larger payload. The appropriate operating point on this trade-off curve depends on mission criticality and time constraints. In time-sensitive scenarios such as disaster response or wildfire monitoring, rapid dissemination of a smaller patch may enable the swarm to adapt faster despite a modest accuracy reduction. In precision-critical applications such as infrastructure inspection or collaborative mapping, maintaining higher model accuracy may justify the overhead of a larger patch. SwarmModelPatch makes this trade-off explicit and configurable without requiring changes to the dissemination protocol, enabling mission-by-mission policy without redesigning the update pipeline.

Combining SwarmModelPatch with model compression techniques may

further reduce payload size, though this remains untested. SwarmModelPatch is conceptually complementary to compression techniques such as pruning [39], quantization [29, 132, 93], and knowledge distillation, since these techniques reduce model weight precision or remove redundant parameters through mechanisms that are, in principle, orthogonal to layer freezing. This thesis deliberately excludes such combinations to isolate the contribution of differential patching to patch size reduction, and as a result we do not empirically know whether combining SwarmModelPatch with these techniques yields the expected additional payload reduction in practice, since interactions between layer freezing and, for example, quantization noise could behave unpredictably, particularly in the unfrozen layers that SwarmModelPatch relies on to recover accuracy. Even if the combination is beneficial, it is also unclear whether the resulting size reduction compounds linearly, sub-linearly, or super-linearly: our formal model $S = \alpha \cdot (P - P_f)$ characterizes only the effect of layer freezing in isolation and makes no claims about its interaction with orthogonal compression methods. Exploring the interaction between SwarmModelPatch and model compression, empirically evaluating at least one representative technique from each of pruning, quantization, and distillation across the models and datasets used in this thesis, is therefore a natural and necessary direction for future work.

Future work should explore alternative layer-freezing criteria for SwarmModelPatch. This thesis freezes layers sequentially by depth, which is a natural and effective choice given that early layers in the CNN architectures evaluated tend to encode general, transferable features while later layers encode task-specific representations. A promising direction for future work is to explore complementary

freezing criteria, such as ranking layers by the magnitude of their gradient updates or weight change during an initial fine-tuning pass, and freezing the least-changed layers first. Such sensitivity-based freezing could offer swarm operators an additional lever for tuning the accuracy-to-patch-size trade-off alongside depth-based freezing, particularly for architectures or tasks where task-relevant adaptation is not strongly correlated with depth.

Future work should explore extending SwarmModelPatch to other deep learning architectures beyond CNNs. Transformer-based architectures, for example, offer richer structural choices for what to freeze, attention blocks, feed-forward blocks, or individual attention heads, and could draw on parameter-efficient fine-tuning techniques such as LoRA [41], which show that a small, targeted subset of parameters is often sufficient to adapt a model. Architectures with weight sharing or dynamic computation graphs similarly invite a more flexible definition of a "freezable unit," beyond a whole layer, such as individual weight matrices or low-rank adapters. We identify generalizing the freezing granularity and re-deriving the patch-size relationship $S = \alpha \cdot (P - P_f)$ under these finer-grained units as a concrete direction for extending SwarmModelPatch beyond the CNN setting evaluated in this thesis.

8.3 Implications of SwarmMARL

SwarmMARL is best understood as a learned improvement over gossip, not an alternate version of SwarmSync. Structurally, SwarmMARL shares more in common with gossip-based protocols than with SwarmSync. Like gossip, there are

no pre-assigned roles: any agent that holds the update can broadcast it, and any agent that lacks it can request it. This flat, fully-distributed structure means that dissemination can proceed from multiple sources simultaneously, which SwarmSync's hierarchical design cannot do. In SwarmSync, only designated leaders broadcast to their sub-swarms, serializing dissemination through a fixed bottleneck. The multi-source broadcasting of SwarmMARL significantly accelerates convergence speed at the cost of higher transmission overhead, since more agents are transmitting concurrently. What SwarmMARL adds over naive gossip is that the learned policy suppresses broadcasts that gossip would make unconditionally: an agent does not broadcast simply because it holds the update; it broadcasts only when its local observations, such as battery level, neighbor update ratio, and active transfer status, suggest that doing so is productive. This selective suppression reduces gossip's primary weakness of redundant rebroadcasting while preserving the speed advantage of multi-source dissemination.

The overhead-speed trade-off of SwarmMARL is distinct from both SwarmSync and gossip. SwarmSync minimizes overhead by restricting transmissions to a structured hierarchy but pays a speed cost because dissemination is serialized through leaders. Gossip maximizes speed through multi-source broadcasting but generates high overhead through redundant rebroadcasting to already-updated agents. SwarmMARL occupies a middle ground: it retains the speed advantage of multi-source broadcasting while using the learned policy to suppress a significant fraction of the redundant transmissions that gossip would produce. The resulting overhead is higher than SwarmSync's because more agents are transmitting, but lower

than naive gossip's because the policy has learned not to broadcast to already-updated neighborhoods or during active transfers. Practitioners choosing between the three approaches should therefore consider SwarmMARL when convergence speed is the primary concern and overhead is a secondary constraint, SwarmSync when overhead and formal reliability guarantees are paramount, and gossip only as a simple baseline without the intelligence to suppress redundant transmissions.

SwarmMARL's performance advantage over static strategies is most pronounced under high failure rates. Under clean channel conditions, both SwarmMARL and SwarmSync converge quickly, and the speed advantage of multi-source broadcasting in SwarmMARL is offset by the additional overhead it generates. Under high packet loss rates, however, SwarmMARL's ability to adapt its transmission behavior to observed channel conditions becomes more significant, likely due to the gossip-based features of SwarmMARL, where the sheer volume of data transmission overshadows the failure rate. This finding has practical implications for deployment in environments where communication reliability varies significantly, such as urban canyons, dense forest canopy, or electromagnetically contested environments, where the adaptability of a learned policy provides a meaningful advantage over a static protocol.

SwarmMARL should be extended to other resource-constrained domains. The problem structure addressed by SwarmMARL: distributing a payload across a set of resource-constrained agents that must coordinate using only local observations under unreliable communication, is not unique to UAV swarms. Internet of Things (IoT) networks, edge computing platforms, and satellite constellations face

structurally similar update dissemination challenges. SwarmMARL could be adapted to these domains by redefining the observation space to reflect the domain-specific resource constraints (e.g., memory usage, computational load, or orbital position) while preserving the MAPPO training framework and reward structure. This generalization is identified as a promising direction for future work.

8.4 Real-World Implementation and Validation

Both approaches transfer from simulation to physical hardware, with faster convergence under clean laboratory conditions. The real-world experiments conducted in a controlled laboratory environment using a fleet of six physical drones confirm that both SwarmSync and SwarmMARL operate correctly on physical hardware as a proof of concept. Convergence was achieved in all trials, validating that the core coordination mechanisms are not artifacts of the idealized ARGoS simulation environment. Absolute convergence times in the real-world experiments are faster than their simulation counterparts, primarily because the laboratory environment operates without the artificially injected packet loss rates ($f \in \{0.25, 0.50, 0.75\}$) used in simulation to stress-test reliability. Under clean wireless conditions with no injected failures, both SwarmSync and SwarmMARL can complete update dissemination without retransmissions, removing the dominant source of latency in the high-failure-rate simulation scenarios. This finding reinforces the interpretation that both approaches perform well in benign deployment conditions, and that the performance gap between them narrows when packet loss is low, consistent with the simulation results at $f = 0$

reported in Chapter 6.

The real-world experiments validate correctness rather than scale. The physical evaluation uses six drones, constrained by available hardware, rather than the 20–500 agent configurations evaluated in simulation. This means the real-world results are not intended as a direct comparison to large-swarm simulation performance; instead, they confirm that the communication protocols, file-based state-action bridge, and SwarmModelPatch application mechanism all function correctly on physical hardware, and that no fundamental incompatibility exists between the simulation design and physical deployment. The scalability properties of both approaches are established through simulation, where fleet size can be varied systematically.

8.5 Cross-Cutting Implications

8.5.1 Energy Consumption

While our experiments on SwarmSync do not simulate battery dynamics directly, we estimate per-UAV energy consumption during update propagation using a standard first-order energy model [40]:

$$E_{\text{total}} = E_{\text{base}} + E_{\text{tx}} + E_{\text{rx}}$$

where $E_{\text{base}} = t \times P$ is the energy spent during the update propagation period due to active flight, $E_{\text{tx}} = D_{\text{tx}} \times e_{\text{tx}}$ is the energy consumed transmitting D_{tx} bytes, and $E_{\text{rx}} = D_{\text{rx}} \times e_{\text{rx}}$ is the energy consumed receiving D_{rx} bytes. We adopt $e_{\text{tx}} = e_{\text{rx}} = 75$ nJ/bit

= 0.6 J/MB based on Bluetooth Low Energy (BLE), which is widely used in UAV communication systems [117, 64], and $P = 0.32$ W for idle IoT communication [127].

Table 8.1: Estimated energy consumption per UAV at 200 UAVs with failure rate $f = 0.5$ and a full update payload (≈3 MB).

Strategy	Avg. TX (MB)	Avg. RX (MB)	Control Steps	Active Time (s)	Est. Energy (J)
SwarmSync	0.75	**3**	18.7	1.87	≈**2.85**
SOUL	**0.48**	**3**	41.6	4.16	≈3.40
Gossip	4.17	**3**	31.4	3.14	≈5.31
SwarmMarl	2.38	**3**	**14.5**	**1.45**	≈3.69

Results in Table 8.1 show that SwarmSync achieves the lowest total energy consumption despite not having the lowest transmission overhead. SOUL transmits less data per drone (0.48 MB vs. 0.75 MB) but incurs higher total energy due to its longer active time (4.16 s vs. 1.87 s), indicating that minimizing flight energy E_{base} outweighs the energy saved by reducing transmission volume in this comparison. Gossip incurs the highest energy cost, driven primarily by redundant rebroadcasting (4.17 MB per drone), confirming that communication overhead outweighs its shorter convergence time relative to SOUL. These results suggest that minimizing convergence time is the more effective optimization target for energy-constrained UAV deployments.

8.5.2 Environmental Obstacles

Although not a primary focus of our study, we conducted preliminary experiments incorporating environmental obstacles to explore their potential impact on swarm coordination. We ran $N = 10$ trials under two density configurations: sparse (10

obstacles) and dense (50 obstacles); obstacle types are drawn randomly from boxes and cylinders, with positions uniformly randomized across the arena and orientation randomized for box entities. All obstacles were static and non-movable throughout each trial. We treat these results as an initial probe rather than an establishment of real-world robustness, for which a larger number of trials and physical deployment would be required. In the sparse configuration, obstacles had no measurable effect on convergence time or coordination performance across the trials. UAVs successfully navigated around obstacles without disruption, and all three strategies maintained their relative performance ordering, consistent with the obstacle-free results. In the dense configuration, all three strategies (SwarmSync, Gossip, SOUL) exhibited increased convergence times, though their relative performance ordering remained unchanged. In roughly 10% of these trials, UAVs became trapped within obstacle clusters, ultimately triggering a timeout. Notably, removing obstacles in otherwise identical trials restored normal performance, indicating that the degradation is attributable to the navigation mechanism rather than the coordination protocol itself. This interpretation is further supported by the fact that none of the three protocols contains obstacle-aware logic: their coordination operates independently of pathfinding, so any failure introduced by obstacles originates from navigation rather than coordination. Developing a more sophisticated path-planning or obstacle-avoidance algorithm represents a natural direction for future work.

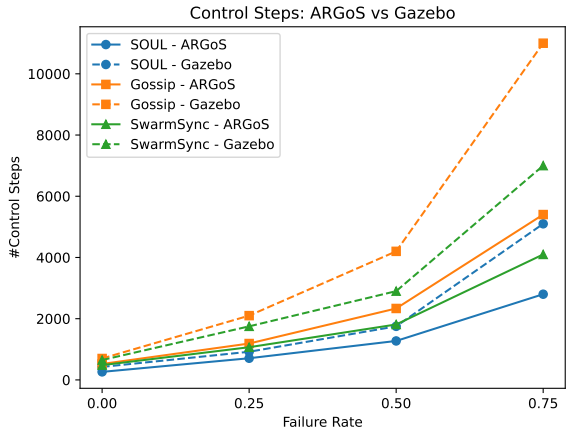
8.5.3 Generalization to Other Simulators

To assess generalization beyond the idealized conditions of ARGoS, we conducted a low-fidelity validation in Gazebo Harmonic integrated with ROS 2. In this Gazebo simulation, we did not include security measures, data corruption modeling, or obstacle scenarios. We attempted to replicate the ARGoS environment as closely as possible, though fundamental differences between the two simulators introduce nontrivial discrepancies. Notably, Gazebo lacks a global time step, meaning that communication events such as signal transmissions and acknowledgements are not synchronized optimally, potentially affecting coordination timing.

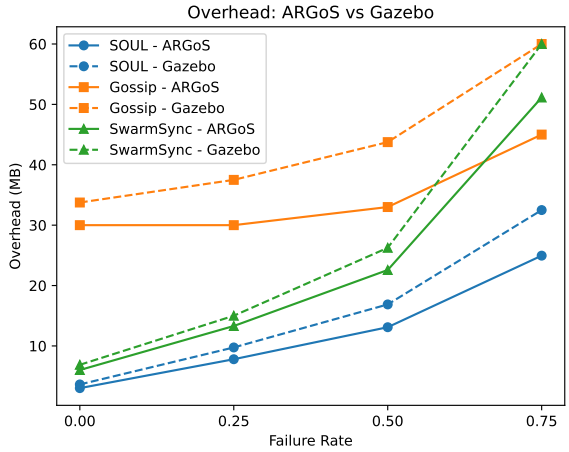
Due to Gazebo's higher computational demands, we scaled the simulation to 20 UAVs. Results in Figure 8.1 show that the relative ordering of the three strategies is preserved in Gazebo, confirming that SwarmSync's advantages over SOUL and Gossip are not artifacts of ARGoS's idealized conditions. Absolute convergence times are approximately 15-25% longer than their ARGoS counterparts and exhibit higher variance, attributable to Gazebo's asynchronous physics and network update mechanisms, which lack ARGoS's global time step and therefore introduce timing variability in signal transmission and ACK synchronization.

8.5.4 Dynamic Transfer Rates

As an additional exploration beyond our core study, we experimented with dynamic packet transfer rate schemes. The static transfer rate follows the description in Chapter 3 at 1 Mbps. The dynamic scheme varies the transfer rate between 0 and 2 Mbps, with the constraint that the time-averaged throughput matches the static baseline to



(a) Convergence time (control steps)



(b) Transmission overhead (MB)

Figure 8.1: ARGoS vs. Gazebo results for 20 UAVs under failure rates $f \in \{0, 0.25, 0.50, 0.75\}$.

ensure a fair comparison. While the majority of experiments yielded results consistent with our original findings, a subset of runs exhibited noticeably different behavior when dynamic transfer rates were introduced, with both acceleration and deceleration observed across different runs. This variability is likely due to transient effects such as bursty transmission or timing mismatches between the dynamic rate and the fixed

ARGoS control step duration. When aggregated across all experiments, performance metrics averaged to values similar to the static baseline configuration, suggesting that the choice of transfer rate policy does not significantly alter overall system behavior under the conditions tested.

8.5.5 Security Considerations for Heterogeneous Swarms

Our study demonstrates that standard cryptographic protections such as TLS and HMAC integrate cleanly with SwarmSync without affecting convergence performance. However, security threats specific to heterogeneous swarms during software updates remain an open research problem. Prior work by Al Blooshi and Han [4] investigated security threats during homogeneous swarm updates, but heterogeneous swarms introduce additional attack vectors that have not been studied. Specifically, role-based update targeting — where only certain UAV types receive certain updates — creates opportunities for adversaries to exploit the targeting mechanism: an attacker who can spoof a UAV's type identifier could cause incompatible updates to be delivered to the wrong agents, potentially corrupting mission-critical software on perception or navigation systems. Future research should focus on two directions: understanding the vulnerabilities specific to role-based update distribution in heterogeneous swarms, and developing role-based authentication mechanisms that prevent type spoofing without introducing prohibitive coordination overhead.

For SwarmMARL specifically, the learned policy introduces an additional security surface that does not exist in protocol-based approaches: adversarial perturbations to agent observations could cause the policy to select incorrect actions. An attacker

who can manipulate the state JSON files written by the ARGoS controller could potentially cause agents to idle indefinitely or broadcast incorrect versions. Robustness of MARL policies to adversarial observation perturbations is an active research area [128] and should be considered in any security-critical deployment of Swarm-MARL.

8.5.6 Timeout Policy Selection

Depending on the nature of the task and the specifics of the update, the swarm's behavior upon encountering a timeout can vary. Prior to initiating an update, the operator defines a recovery policy dictating system behavior upon timeout, consistent with established practice in UAV swarm systems [81, 82]. Possible strategies include proceeding without the unresponsive UAV, rolling back to a prior consistent state, or degrading functionality gracefully. In our experiments, we adopted the liveness-first approach: continuing the update process while excluding any UAVs that timed out, a posture widely adopted in fault-tolerant swarm frameworks [17, 83, 131]. A rollback mechanism is available for both update components: the model patch is retained until full convergence and can be reversed by applying it in the negative direction; the prior software binary is preserved until convergence is confirmed. Practitioners should select the timeout policy that matches their mission's tolerance for partial updates versus delayed convergence.

8.6 Limitations and Scope

This section consolidates the methodological constraints and scope boundaries of the work. These are not critiques of the experimental design but characterizations of the conditions under which the results hold and the directions in which the work should be extended.

8.6.1 Simulation Fidelity

ARGoS one-step communication delay. ARGoS simulates time in discrete control steps, but its implementation of message transmission and reception introduces a systematic delay: a receiver can only read signals transmitted during the previous control step. When a packet is sent at time t, the receiver can only receive it at step $t + 1$. Hence, even if the receiver sends an ACK immediately, the sender receives the ACK at $t + 2$. In an optimized control system at failure rate $f = 0$, when a packet is sent at t, the ACK would also be sent back at t and the sender could advance to the next packet at $t + 1$; ARGoS currently does not allow this, causing SwarmSync to take longer in simulation than it would on a real system without this constraint. This effect does not affect Gossip or SOUL, as those strategies do not wait for a response before sending the next packet. It also does not affect SwarmMARL, which reads state files written at t and writes actions consumed at $t + 1$ — the same one-step lag is already accounted for in the bridge design. The practical implication is that SwarmSync's simulation results represent a conservative lower bound on its real-world performance, and the advantage of SwarmSync over baselines may be even larger in

deployment.

Packet loss model. Packet loss is modeled as an all-or-nothing drop: packets are either delivered intact or completely dropped. In real-world wireless environments, partial corruption or bit errors can occur without total loss. We incorporated HMAC-based integrity verification to guard against undetected corruption, but this mechanism yielded no measurable benefit under ARGoS's binary failure semantics. In practice, a more realistic channel model would require the receiver to detect partial corruption and request retransmission, which would increase overhead relative to our simulation results. Similarly, the static 1 Mbps transfer rate is not realistic in all deployment environments; adaptive bitrate selection based on interference and range limitations would be needed in practice, and the relative performance of the three strategies may shift under adaptive rates.

Simulation-to-reality gap. The primary evaluation is conducted in ARGoS. The real-world validation experiments on six physical drones confirm correctness under clean laboratory conditions, but the laboratory environment does not reproduce the interference, dynamic topology changes, or environmental obstacles present in field deployment. The real-world results should therefore be interpreted as a proof-of-concept validation rather than a comprehensive assessment of field performance.

8.6.2 SwarmMARL Scope

Training generalization. SwarmMARL is trained under a single fixed configuration: a swarm of 100 agents, a packet failure rate of $f = 0.25$, and an update payload of 64 packets. The evaluation therefore does not optimize to other swarm sizes, fail-

ure rates, or payload sizes. A policy trained under $f = 0.25$ and 100 agents may coordinate sub-optimally under high-stress conditions, since the observation distributions and coordination challenges differ substantially across configurations. This is a significant scope limitation relative to SwarmSync, which is a deterministic protocol that requires no training and generalizes by construction. Extending SwarmMARL training across multiple configurations, or investigating zero-shot generalization of a single policy to unseen conditions, is identified as an important direction for future work.

Reward function design and misspecification. The reward function used to train SwarmMARL was hand-crafted based on domain knowledge about the update dissemination problem. Hand-crafted reward functions carry the risk of misspecification: the policy may learn to maximize the reward signal through behaviors that satisfy the reward's letter but not its intent. For example, the step penalty $r_{\text{step}} = -0.01$ per step while outdated could in principle cause agents to issue rapid sequential requests to avoid the penalty, saturating the channel without improving convergence. Similarly, the progress reward could incentivize agents to maintain a receiving state indefinitely rather than completing transfers. We monitored the `metrics_log.csv` convergence ratio during training to detect such behaviors, but formal verification of reward alignment is not provided. This is a known limitation of hand-crafted reward functions in MARL and is identified as a direction for future work.

Seed agent assumption. The SwarmMARL implementation designates two specific agents (agent IDs 0 and 1, one of each type) as seed agents that are forced to broadcast when they hold the update, regardless of the policy's action. This is a

practical expedient to ensure that the update begins propagating at the start of each episode, preventing the degenerate case where all agents idle and no dissemination occurs. However, it means that SwarmMARL's behavior is not fully decentralized: two agents are effectively operating under a hard-coded rule rather than the learned policy. This does not affect the generality of the approach — removing the seed constraint and instead using a high initial entropy to encourage early broadcasting is a natural extension — but it means the reported results reflect a partially seeded initialization rather than a fully learned coordination policy from scratch.

Simulation-to-hardware transfer. The SwarmMARL policy is trained entirely within the ARGoS simulation environment. The file-based state-action bridge assumes that state files written at control step t are available to the MAPPO process before actions for step $t+1$ are needed, with a 50 ms sleep interval calibrated to the ARGoS 100 ms step duration. In physical deployment, the control loop frequency of the drone hardware may differ, requiring recalibration of this interval. More fundamentally, the observation feature `neighbor_versions` — which requires the C++ ARGoS controller to expose the version status of all visible neighbors — may not be available or accurate in physical deployment where version information must be communicated explicitly over the wireless channel. If this feature defaults to 0.0 in deployment (as designed for backward compatibility), the policy will operate without the neighbor update ratio signal, which may degrade the quality of broadcasting decisions in the later stages of dissemination. Additionally, the real-world experiments offload policy inference to a ground server due to onboard computational constraints, introducing a communication dependency that does not exist in simulation. Future

work should evaluate SwarmMARL with onboard inference to eliminate this dependency.

8.6.3 SwarmSync and SwarmModelPatch Scope

Leader selection strategy. We explored alternative leader selection strategies beyond random assignment, including position-based leader selection using K-means clustering to identify sub-swarm centers and appointing the central drone as leader. This strategy yielded little difference in efficiency (an average difference of 4.1%) compared to random selection, since leader choice primarily affects the initial movement phase rather than the core data transmission process. However, in real-world deployments, leader selection may have a greater impact due to factors such as UAV speed, battery capacity, and spatial distribution, all of which vary in physical hardware in ways the simulation does not capture.

Swarm configuration and agent types. We used Footbots and Eyebots to model a heterogeneous UAV swarm, which is a common setup in ARGoS for heterogeneous swarm studies [26, 61, 109]. However, we maintained an equal 1:1 ratio of Footbots to Eyebots, although practical applications may use different ratios or more than two agent types. Additionally, ARGoS represents the drone communication range as a circular area in two dimensions, whereas in real-world scenarios this range is typically spherical, which could affect the number of agents within communication range at any given time and therefore influence both dissemination speed and overhead.

SwarmModelPatch generalization. SwarmModelPatch was evaluated using

SqueezeNet, MobileNetV2, and ResNet50 on the Weather-5 and MNIST datasets. The trend of model performance observed with different numbers of frozen layers may differ when applying SwarmModelPatch to other DL architectures or datasets, particularly those with substantially different layer structures (e.g., transformer-based models) or data distributions that are further from the pre-training distribution. The relationship $S = \alpha \cdot (P - P_f)$ established in Section 4.3.1 assumes that patch size scales linearly with unfrozen parameters, which holds for the CNN architectures evaluated but may not hold for architectures with weight sharing, sparse connectivity, or dynamic computation graphs.

An early exploration of sensitivity-based layer selection did not transfer reliably, motivating our use of depth-based freezing instead. Before settling on depth-based freezing, we prototyped a sensitivity-based layer selection strategy, freezing layers based on the magnitude of their accuracy change during the first pass of fine-tuning, rather than their position in the network, on the intuition that this might identify a more effective set of layers to freeze than depth alone. This early prototype suggested that sensitivity rankings measured early in training did not reliably transfer to later stages of fine-tuning: layers that appeared to change the least, and therefore seemed like good freezing candidates in early epochs, *sometimes* went on to have much bigger performance impact than other layers as training progressed. Because this behavior undermined the basic premise of using an early sensitivity measurement to decide what to freeze for the remainder of training, we did not pursue this direction further and adopted the simpler, more predictable depth-based freezing strategy used throughout this thesis. We note that this exploration was limited in

scope, a small number of quick prototype runs rather than a systematic study, and it remains possible that a more carefully designed sensitivity metric, computed over a longer warm-up period or averaged across multiple fine-tuning runs, could avoid this instability. We therefore identify sensitivity-based layer selection as an open direction for future work rather than a dead end, and flag this early negative result so that future work does not need to rediscover it.

Heterogeneity scope. This thesis evaluates heterogeneity in hardware capabilities and update payloads, specifically vision-capable Eyebots requiring DL model updates versus task-oriented Footbots receiving logic updates. This two-type setup is sufficient to demonstrate role-dependent update handling while keeping the evaluation focused on synchronization behavior rather than role-assignment or task-planning variables. However, it does not address high-level mission diversity such as concurrent delivery, inspection, and relay operations, nor does it evaluate swarms with more than two agent types. Swarms with richer role diversity may exhibit more complex update dependency patterns that the current dissemination framework does not handle.

8.6.4 Shared Constraints

Single updater liveness dependency. Both SwarmSync and SwarmMARL rely on a single Updater UAV to initiate dissemination. This design introduces a liveness dependency: if the Updater fails or loses connectivity before delivering the update to any swarm member, dissemination cannot commence because no agent possesses the new payload. This failure mode is more severe than partial dissemination scenarios where intermediate relays can continue propagation independently. For SwarmSync,

this risk can be mitigated through parallel dispatch of a backup Updater or satellite-based heartbeat monitoring to trigger failover from headquarters. For SwarmMARL, the same mitigation applies at the dissemination initiation stage; once at least one agent holds the update, the learned policy can continue propagation without the Updater. Comprehensive fault-tolerant Updater recovery is beyond the current scope of this thesis.

Chapter 9

Conclusion

This thesis investigated software update propagation in heterogeneous UAV swarms through two complementary coordination philosophies: deterministic hierarchical protocols and learned decentralized policies. We presented three contributions: SwarmSync, SwarmModelPatch, and SwarmMARL, and validated all three in simulation and on physical Crazyflie 2.1 Brushless hardware in a controlled laboratory environment.

SwarmSync and SwarmModelPatch. SwarmUpdate, comprising SwarmSync and SwarmModelPatch, is the first framework for software updates in heterogeneous UAV swarms. We empirically evaluated SwarmSync by comparing it with two baseline synchronization strategies (SOUL and Gossip) and one baseline model update approach (full-size update) across heterogeneous swarm simulations at sizes $s \in \{20, 100, 200, 500\}$ and failure rates $f \in \{0, 0.25, 0.50, 0.75\}$. Our results show that SwarmSync is the most efficient and scalable of the three synchronization strategies, achieving up to 78.3% faster convergence times than SOUL and 47.7% faster

convergence than Gossip, while maintaining a reasonable transmission overhead of 19.7% above Gossip. SwarmModelPatch reduces update time and overhead by up to 72.3% and 74.3% respectively compared to a full-size model update, without substantially compromising model accuracy, and exposes a tunable trade-off between patch size and accuracy that operators can configure per mission.

SwarmMARL. SwarmMARL can replace SwarmSync in the SwarmUpdate framework depending on the situation and designer goals, and it is the first application of multi-agent reinforcement learning to software update coordination in heterogeneous UAV swarms. Trained using MAPPO under a configuration of 100 agents and a packet failure rate of $f = 0.25$, SwarmMARL learns context-sensitive broadcast and request policies that suppress misdirected transmissions, broadcasts to already-updated neighborhoods, requests during active transfers, and transmissions from critically low-battery agents, without any of these behaviors being explicitly programmed.

Structurally, SwarmMARL resembles an intelligent gossip protocol: like gossip, all agents can broadcast simultaneously, which accelerates convergence through multi-source dissemination, but the learned policy suppresses the redundant rebroadcasting that makes naive gossip inefficient. The comparison between SwarmSync and SwarmMARL reveals that the two approaches occupy complementary positions: SwarmSync offers formal termination guarantees and reliable delivery under high packet loss, while SwarmMARL offers adaptability and context-sensitivity that a fixed protocol cannot provide.

Real-world validation. Both SwarmSync and SwarmMARL were validated on a fleet of six Crazyflie 2.1 Brushless drones using ROS 2 and CrazySwarm2 in a con-

trolled laboratory environment. All six drones successfully received the full update in every run under clean laboratory conditions, and two designated drones executed a hover sequence as a physical convergence signal, providing an unambiguous hardware demonstration that both coordination approaches operate correctly beyond simulation. The real-world experiments also surfaced important engineering constraints: the absence of drone-to-drone communication required the orchestrator to simulate the relay hierarchy centrally, and the limited onboard sensing capabilities of the Crazyflie platform meant that SwarmMARL observations had to be derived from Vicon pose data and orchestrator state rather than measured autonomously by each drone. These constraints represent the gap between the simulation design and physical deployment that future work must address.

Implications for practitioners. This thesis provides empirical guidance for practitioners choosing between coordination approaches for heterogeneous UAV swarm updates. SwarmSync is preferable when reliability guarantees and formal convergence bounds are the primary concern, particularly under high packet loss rates. SwarmMARL is preferable when convergence speed and resource efficiency under moderate failure conditions are prioritized, and when the swarm operates in environments where channel conditions vary dynamically. SwarmModelPatch is compatible with either coordination approach and should be adopted whenever bandwidth or energy constraints make transmitting full model updates impractical.

Future work. Several directions emerge from this thesis. First, combining the two coordination philosophies, using SwarmSync's hierarchical structure for guaranteed delivery while using a MARL policy to determine when leaders should initiate

transmission to followers, represents a natural next step that could inherit the formal guarantees of SwarmSync and the adaptability of SwarmMARL simultaneously. Second, extending SwarmMARL training across multiple swarm sizes and failure rates would establish whether a single policy can generalize across deployment conditions, eliminating the need to retrain per configuration. Third, porting SwarmSync's dynamic leader election and sub-swarm formation to the ROS 2 implementation would enable self-organizing hierarchies in physical deployment, removing the current dependency on a hand-defined YAML topology. Fourth, deploying on hardware with peer-to-peer communication capability would allow the full parallelism advantage of SwarmSync's hierarchical dissemination to be realized physically, not just in simulation. Finally, applying SwarmModelPatch in combination with model compression techniques such as quantization and pruning could further reduce patch sizes beyond what layer freezing alone achieves, and extending the approach to transformer-based architectures would broaden its applicability to modern UAV perception systems.

As UAV technology continues to advance and swarms are deployed in increasingly demanding mission-critical scenarios, ensuring robust, efficient, and scalable software updates will be essential. This thesis establishes that both deterministic and learned coordination approaches are viable, identifies the conditions under which each is preferable, and provides a validated foundation for future research on software update propagation in heterogeneous autonomous systems.

Table Summary of our contributions.

Table 9.1: Summary of thesis contributions, mapped to their corresponding objective and chapter.

Contribution	**Description**	**Objective**	**Chapter**
SwarmSync	Hierarchical, ACK-guaranteed software update synchronization strategy achieving logarithmic dissemination time and outperforming gossip-based and auction-based baselines.	1	Ch. 4
SwarmModelPatch	DL model patching method using selective layer freezing to produce a controllable, predictable trade-off between patch size, convergence time, and accuracy.	2	Ch. 4
SwarmMARL	First application of multi-agent reinforcement learning (MARL) to software update coordination, learning context-sensitive broadcast/request behavior without predefined roles.	3	Ch. 5
Comparative evaluation	Direct comparison of deterministic and learned coordination philosophies under shared experimental conditions, identifying when each is preferable.	4	Ch. 6
Real-world validation	Physical implementation of both SwarmUpdate and SwarmMARL on UAV hardware, characterizing the simulation-to-deployment gap.	4	Ch. 7
Replication package	Simulation environment setups, scripts, data, and DL models.	—	`https://github.com/SAILResearch/replication-25-allen-update-swarms`

Bibliography

[1] Alexandre Abadie, Said Alvarado-Marin, Filip Maksimovic, Mališa Vučinić, and Thomas Watteyne. 2024. RobOTAP: Over-the-Air Programming of Robotic Swarms. In *2024 IEEE Workshop on Crystal-Free/-Less Radio and System-Based Research for IoT (CrystalFreeIoT)*. Hong Kong, Hong Kong. doi:10.1109/CrystalFreeIoT

[2] Mohamed Abid, Said El Kafhali, Abdellah Amzil, and Mohamed Hanini. 2025. Optimization of UAV flight paths in multi-UAV networks for efficient data collection. *Arabian Journal for Science and Engineering* 50, 10 (2025), 7207–7232.

[3] Norman Abramson. 1970. The ALOHA system: Another alternative for computer communications. In *Proceedings of the November 17-19, 1970, fall joint computer conference*. 281–285.

[4] Shamma Al Blooshi and Kyusuk Han. 2022. A study on employing UPTANE for secure software update OTA in drone environments. In *2022 IEEE international conference on omni-layer intelligent systems (COINS)*. IEEE, 1–6.

[5] Hassan J Al Dawasari, Muhammad Bilal, Muhammad Moinuddin, Kamran

Arshad, and Khaled Assaleh. 2023. DeepVision: Enhanced drone detection and recognition in visible imagery through deep learning networks. *Sensors* 23, 21 (2023), 8711.

[6] Ammar Alfaifi. [n. d.]. 5-class weather status image classification. `https://www.kaggle.com/datasets/ammaralfaifi/5class-weather-status-image-classification`

[7] Yunes Alqudsi and Murat Makaraci. 2025. UAV swarms: research, challenges, and future directions. *Journal of Engineering and Applied Science* 72, 1 (2025), 12.

[8] Huzaifa Arif, Keerthiram Murugesan, Ching-Yun Ko, Pin-Yu Chen, Payel Das, and Alex Gittens. 2025. Patching LLM Like Software: A Lightweight Method for Improving Safety Policy in Large Language Models. *arXiv preprint arXiv:2511.08484* (2025).

[9] Muhammad Ashfaq, Ahmed R Sadik, Teerath Das, Muhammad Waseem, Niko Mäkitalo, and Tommi Mikkonen. 2025. Runtime composition in dynamic system of systems: A systematic review of challenges, solutions, tools, and evaluation methods. *Journal of Systems and Software* (2025), 112661.

[10] Foozhan Ataiefard, Mohammad Jafar Mashhadi, Hadi Hemmati, and Neil Walkinshaw. 2022. Deep State Inference: Toward Behavioral Model Inference of Black-Box Software Systems. *IEEE Transactions on Software Engineering* 48, 12 (2022), 4857–4872. doi:`10.1109/TSE.2021.3128820`

[11] Yanis Bardes, Hassan Soubra, Zineb Noumir, and Amar Ramdane-Cherif. 2025. Federated Learning for UAV Perception Task: A Survey. *Authorea Preprints* (2025).

[12] Ilker Bekmezci, Ozgur Koray Sahingoz, and Şamil Temel. 2013. Flying ad-hoc networks (FANETs): A survey. *Ad Hoc Networks* 11, 3 (2013), 1254–1270.

[13] Stephen Boyd, Arpita Ghosh, Balaji Prabhakar, and Devavrat Shah. 2006. Randomized gossip algorithms. *IEEE transactions on information theory* 52, 6 (2006), 2508–2530.

[14] Stephen Brown and Cormac J Sreenan. 2013. Software updating in wireless sensor networks: A survey and lacunae. *Journal of Sensor and Actuator Networks* 2, 4 (2013), 717–760.

[15] Yann Busnel, Marin Bertier, Eric Fleury, and Anne-Marie Kermarrec. 2007. Gcp: Gossip-based code propagation for large-scale mobile wireless sensor networks. *arXiv preprint arXiv:0707.3717* (2007).

[16] Vinton Cerf and Robert Kahn. 1974. A protocol for packet network intercommunication. *IEEE Transactions on communications* 22, 5 (1974), 637–648.

[17] Indu Chandran and Kizheppatt Vipin. 2024. Network analysis of decentralized fault-tolerant UAV swarm coordination in critical missions. *Drone Systems and Applications* 12 (2024), 1–15.

[18] Krishnendu Chatterjee, Hongfei Fu, and Amir Kafshdar Goharshady. 2016. Ter-

mination analysis of probabilistic programs through Positivstellensatz's. In *International Conference on Computer Aided Verification*. Springer, 3–22.

[19] Imrich Chlamtac, Marco Conti, and Jennifer J-N Liu. 2003. Mobile ad hoc networking: imperatives and challenges. *Ad hoc networks* 1, 1 (2003), 13–64.

[20] Scott Corson and Joseph Macker. 1999. *Mobile ad hoc networking (MANET): Routing protocol performance issues and evaluation considerations*. Technical Report.

[21] Daryl J Daley and David G Kendall. 1964. Epidemics and rumours. *Nature* 204, 4963 (1964), 1118–1118.

[22] Supratim Deb, Muriel Médard, and Clifford Choute. 2006. Algebraic gossip: A network coding approach to optimal multiple rumor mongering. *IEEE Transactions on Information Theory* 52, 6 (2006), 2486–2507.

[23] Kapel Dev, Yash Madhwal, Sofia Shevelo, Pavel Osinenko, and Yury Yanovich. 2025. SwarmRaft: Leveraging Consensus for Robust Drone Swarm Coordination in GNSS-Degraded Environments. *IEEE Internet of Things Journal* (2025).

[24] Ke Dong, Chengjie Zhou, Yihan Ruan, and Yuzhi Li. 2020. MobileNetV2 model for image classification. In *2020 2nd International Conference on Information Technology and Computer Application (ITCA)*. IEEE, 476–480.

[25] Frederick Ducatelle, Gianni A Di Caro, and Luca M Gambardella. 2010. Cooperative self-organization in a heterogeneous swarm robotic system. In *Proceed-*

ings of the 12th annual conference on Genetic and evolutionary computation. 87–94.

[26] Frederick Ducatelle, Gianni A Di Caro, Carlo Pinciroli, and Luca M Gambardella. 2011. Self-organized cooperation between robotic swarms. *Swarm Intelligence* 5 (2011), 73–96.

[27] Xiao Fan, Peiran Wu, and Minghua Xia. 2024. Air-to-ground communications beyond 5G: UAV swarm formation control and tracking. *IEEE Transactions on Wireless Communications* 23, 7 (2024), 8029–8043.

[28] Yaël Frégier and Jean-Baptiste Gouray. 2021. Mind2Mind: transfer learning for GANs. In *Geometric Science of Information: 5th International Conference, GSI 2021, Paris, France, July 21–23, 2021, Proceedings 5*. Springer, 851–859.

[29] Karan Goel, Albert Gu, Yixuan Li, and Christopher Ré. 2020. Model patching: Closing the subgroup performance gap with data augmentation. *arXiv preprint arXiv:2008.06775* (2020).

[30] Jian Gu, Aldeida Aleti, Chunyang Chen, and Hongyu Zhang. 2023. Neuron Patching: Semantic-based Neuron-level Language Model Repair for Code Generation. *arXiv preprint arXiv:2312.05356* (2023).

[31] Yafeng Guo and Tianhong Pan. 2012. Robust stability of uncertain systems over network with bounded packet loss. *Journal of Applied Mathematics* 2012, 1 (2012), 945240.

[32] Bernhard Haeupler, Asaf Cohen, Chen Avin, and Muriel Médard. 2012. Network coded gossip with correlated data. In *2012 IEEE International Symposium on Information Theory Proceedings*. IEEE, 2616–2620.

[33] Ruidong Han, Siqi Ma, Juanru Li, Surya Nepal, David Lo, Zhuo Ma, and JianFeng Ma. 2024. Range specification bug detection in flight control system through fuzzing. *IEEE Transactions on Software Engineering* 50, 3 (2024), 461–473.

[34] Ruidong Han, Shangzhi Xu, Juanru Li, Elisa Bertino, David Lo, JianFeng Ma, and Siqi Ma. 2025. Real-time Rectifying Flight Control Misconfiguration Using Intelligent Agent. *ACM Trans. Softw. Eng. Methodol.* 34, 4, Article 92 (April 2025), 29 pages. doi:10.1145/3702994

[35] Ruidong Han, Chao Yang, Siqi Ma, JiangFeng Ma, Cong Sun, Juanru Li, and Elisa Bertino. 2022. Control parameters considered harmful: detecting range specification bugs in drone configuration modules via learning-guided search. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 462–473. doi:10.1145/3510003.3510084

[36] Manzoor Ahmed Hashmani, Syed Muslim Jameel, Hitham Alhussain, Mobashar Rehman, and Arif Budiman. 2019. Accuracy performance degradation in image classification models due to concept drift. *International Journal of Advanced Computer Science and Applications* 10, 5 (2019), 422–425.

[37] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[38] Peng He, Zhihong Zhao, and Xinyue Liu. 2025. Quick-UAV OCSORT: a fast and robust UAV-based tracking-by detection algorithm. *Signal, Image and Video Processing* 19, 7 (2025), 577.

[39] Yang He and Lingao Xiao. 2023. Structured pruning for deep convolutional neural networks: A survey. *IEEE transactions on pattern analysis and machine intelligence* 46, 5 (2023), 2900–2919.

[40] Wendi Rabiner Heinzelman, Anantha Chandrakasan, and Hari Balakrishnan. 2000. Energy-efficient communication protocol for wireless microsensor networks. In *Proceedings of the 33rd annual Hawaii international conference on system sciences*. IEEE, 10–pp.

[41] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2021. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685* (2021).

[42] Maximilian Hüttenrauch, Adrian Šošić, and Gerhard Neumann. 2019. Deep reinforcement learning for swarm systems. *Journal of Machine Learning Research* 20, 54 (2019), 1–31.

[43] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. 2016. SqueezeNet: AlexNet-level accuracy with 50x

fewer parameters and¡ 0.5 MB model size. *arXiv preprint arXiv:1602.07360* (2016).

[44] Gabriel Ilharco, Marco Tulio Ribeiro, Mitchell Wortsman, Suchin Gururangan, Ludwig Schmidt, Hannaneh Hajishirzi, and Ali Farhadi. 2022. Editing models with task arithmetic. *arXiv preprint arXiv:2212.04089* (2022).

[45] Gabriel Ilharco, Mitchell Wortsman, Samir Yitzhak Gadre, Shuran Song, Hannaneh Hajishirzi, Simon Kornblith, Ali Farhadi, and Ludwig Schmidt. 2022. Patching open-vocabulary models by interpolating weights. *Advances in Neural Information Processing Systems* 35 (2022), 29262–29277.

[46] Sadaf Javed, Ali Hassan, Rizwan Ahmad, Waqas Ahmed, Rehan Ahmed, Ahsan Saadat, and Mohsen Guizani. 2024. State-of-the-art and future research challenges in uav swarms. *IEEE Internet of Things Journal* 11, 11 (2024), 19023–19045.

[47] Chijung Jung, Ali Ahad, Jinho Jung, Sebastian Elbaum, and Yonghwi Kwon. 2021. Swarmbug: debugging configuration bugs in swarm robotics. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 868–880. doi:10.1145/3468264.3468601

[48] Gal A Kaminka and Yinon Douchan. 2025. Heterogeneous foraging swarms can be better. *Frontiers in Robotics and AI* 11 (2025), 1426282.

[49] Andrej Karpathy, Justin Johnson, and Fei-Fei Li. [n. d.]. Transfer Learning and Fine-tuning Convolutional Neural Networks. `https://cs231n.github.io/transfer-learning/`.

[50] S Kernbach et al. 2021. Towards applied swarm robotics: current limitations and enablers. *Swarm Intelligence* 15, 4 (2021), 223–245.

[51] Sarika Khandelwal, Archana Raut, Harsha Vyawahare, Dipti Theng, and Sheetal Dhande. 2024. Optimizing performance in mango plant leaf disease classification through advanced machine learning techniques. *Engineering, Technology & Applied Science Research* 14, 6 (2024), 18476–18480.

[52] James Kirkpatrick, Razvan Pascanu, Neil Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A Rusu, Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, et al. 2017. Overcoming catastrophic forgetting in neural networks. *Proceedings of the national academy of sciences* 114, 13 (2017), 3521–3526.

[53] Leonard Kleinrock and Fouad Tobagi. 1975. Packet switching in radio channels: Part I-carrier sense multiple-access modes and their throughput-delay characteristics. *IEEE transactions on Communications* 23, 12 (1975), 1400–1416.

[54] Nathan Koenig and Andrew Howard. 2004. Design and Use Paradigms for Gazebo, An Open-Source Multi-Robot Simulator. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*. Sendai, Japan, 2149–2154.

[55] Thomas Kothmayr, Corinna Schmitt, Wen Hu, Michael Brünig, and Georg

Carle. 2013. DTLS based security and two-way authentication for the Internet of Things. *Ad Hoc Networks* 11, 8 (2013), 2710–2723.

[56] Hugo Krawczyk, Kenneth G Paterson, and Hoeteck Wee. 2013. On the security of the TLS protocol: A systematic analysis. In *Annual Cryptology Conference*. Springer, 429–448.

[57] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep learning. *nature* 521, 7553 (2015), 436–444.

[58] Joao Leitao, Jose Pereira, and Luis Rodrigues. 2007. Epidemic broadcast trees. In *2007 26th IEEE International Symposium on Reliable Distributed Systems (SRDS 2007)*. IEEE, 301–310.

[59] Zhizhong Li and Derek Hoiem. 2017. Learning without forgetting. *IEEE transactions on pattern analysis and machine intelligence* 40, 12 (2017), 2935–2947.

[60] Xiaozhou Liang, John Henry Burns, Joseph Sanchez, Karthik Dantu, Lukasz Ziarek, and Yu David Liu. 2021. Understanding Bounding Functions in Safety-Critical UAV Software. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 1311–1322. doi:10.1109/ICSE43902.2021.00119

[61] Marc Espinós Longa, Antonios Tsourdos, and Gokhan Inalhan. 2022. Human–machine network through bio-inspired decentralized swarm intelligence and heterogeneous teaming in SAR operations. *Journal of Intelligent & Robotic Systems* 105, 4 (2022), 88.

[62] Ryan Lowe, Yi I Wu, Aviv Tamar, Jean Harb, OpenAI Pieter Abbeel, and Igor Mordatch. 2017. Multi-agent actor-critic for mixed cooperative-competitive environments. *Advances in neural information processing systems* 30 (2017).

[63] Nikoleta Manakitsa, George S Maraslidis, Lazaros Moysis, and George F Fragulis. 2024. A review of machine learning and deep learning for object detection, semantic segmentation, and human action recognition in machine and robotic vision. *Technologies* 12, 2 (2024), 15.

[64] Patrick Mannion. 2017. Comparing low power wireless technologies (Part 2). *Digi-Key. Dec* 14 (2017).

[65] Arya Marda, Shubham Kulkarni, and Karthik Vaidhyanathan. 2024. Switch: An exemplar for evaluating self-adaptive ml-enabled systems. In *Proceedings of the 19th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. 143–149.

[66] M McCloskey and NJ Cohen. 1989. Catastrophic interference in connectionist networks: the sequential learning problem. Psychol Learn Motiv 24: 109–165.

[67] Patrick E McKnight and Julius Najab. 2010. Mann-whitney U test. *The Corsini encyclopedia of psychology* (2010), 1–1.

[68] Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Aguera y Arcas. 2017. Communication-efficient learning of deep networks from decentralized data. In *Artificial intelligence and statistics*. PMLR, 1273–1282.

[69] O. Michel. 2004. Webots: Professional Mobile Robot Simulation. *Journal of Advanced Robotics Systems* 1, 1 (2004), 39–42. http://www.ars-journal.com/International-Journal-of-Advanced-Robotic-Systems/Volume-1/39-42.pdf

[70] Anup Mishra, Čedomir Stefanović, Petar Popovski, and Israel Leyva-Mayorga. 2025. Reinforcement Learning-Based Policy Optimization for Heterogeneous Radio Access. *IEEE Wireless Communications Letters* 15 (2025), 1050–1054.

[71] Behnaz Mohammadi and Nima Jafari Navimipour. 2019. Data replication mechanisms in the peer-to-peer networks. *International Journal of Communication Systems* 32, 14 (2019), e3996.

[72] Minu MS, Subashka Ramesh SS, et al. 2022. Optimal Squeeze Net with Deep Neural Network-Based Arial Image Classification Model in Unmanned Aerial Vehicles. *Traitement du Signal* 39, 1 (2022).

[73] Aiman Munir, Ayan Dutta, and Ramviyas Parasuraman. 2024. Energy-aware coverage planning for heterogeneous multi-robot system. In *International Symposium on Distributed Autonomous Robotic Systems*. Springer, 210–227.

[74] Derrick Mwiti. [n. d.]. Transfer Learning Guide: A Practical Tutorial With Examples for Images and Text in Keras. https://neptune.ai/blog/transfer-learning-guide-examples-for-images-and-text-in-keras.

[75] Leandro Nahabedian, Victor Braberman, Nicolás D'Ippolito, Shinichi Honiden, Jeff Kramer, Kenji Tei, and Sebastián Uchitel. 2020. Dynamic Update of Dis-

crete Event Controllers. *IEEE Transactions on Software Engineering* 46, 11 (2020), 1220–1240. doi:10.1109/TSE.2018.2876843

[76] Object Management Group. 2017. OMG Unified Modeling Language (OMG UML), Version 2.5.1. https://www.omg.org/spec/UML/2.5.1/.

[77] Doriane Olewicki, Sarra Habchi, Mathieu Nayrolles, Mojtaba Faramarzi, Sarath Chandar, and Bram Adams. 2024. On the Costs and Benefits of Adopting Lifelong Learning for Software Analytics-Empirical Study on Brown Build and Risk Prediction. In *Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Practice*. 275–286.

[78] Frans A Oliehoek, Christopher Amato, et al. 2016. *A concise introduction to decentralized POMDPs*. Vol. 1. Springer.

[79] German I Parisi, Ronald Kemker, Jose L Part, Christopher Kanan, and Stefan Wermter. 2019. Continual lifelong learning with neural networks: A review. *Neural networks* 113 (2019), 54–71.

[80] Enrico Petritoli, Fabio Leccese, and Lorenzo Ciani. 2018. Reliability degradation, preventive and corrective maintenance of UAV systems. In *2018 5th IEEE International Workshop on Metrology for AeroSpace (MetroAeroSpace)*. IEEE, 430–434.

[81] Abhishek Phadke and F Antonio Medrano. 2022. Towards resilient UAV swarms—A breakdown of resiliency requirements in UAV swarms. *Drones* 6, 11 (2022), 340.

[82] Abhishek Phadke and F Antonio Medrano. 2023. Examining application-specific resiliency implementations in UAV swarm scenarios. *Intelligence & Robotics* 3, 3 (2023), 453–478.

[83] Abhishek Phadke and F Antonio Medrano. 2024. Increasing operational resiliency of uav swarms: An agent-focused search and rescue framework. *Aerospace Research Communications* 1 (2024), 12420.

[84] Carlo Pinciroli and Giovanni Beltrame. 2016. Buzz: An extensible programming language for heterogeneous swarm robotics. In *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 3794–3800.

[85] Carlo Pinciroli and contributors. 2015. ARGOS3 Examples. `https://github.com/ilpincy/argos3-examples`

[86] Carlo Pinciroli, Adam Lee-Brown, and Giovanni Beltrame. 2016. A tuple space for data sharing in robot swarms. (2016).

[87] Carlo Pinciroli, Vito Trianni, Rehan O'Grady, Giovanni Pini, Arne Brutschy, Manuele Brambilla, Nithin Mathews, Eliseo Ferrante, Gianni Di Caro, Frederick Ducatelle, Mauro Birattari, Luca Maria Gambardella, and Marco Dorigo. 2012. ARGoS: a modular, parallel, multi-engine simulator for multi-robot systems. *Swarm Intell.* 6, 4 (Dec. 2012), 271–295.

[88] Robi Polikar, Lalita Upda, Satish S Upda, and Vasant Honavar. 2001. Learn++: An incremental learning algorithm for supervised neural networks. *IEEE trans-*

actions on systems, man, and cybernetics, part C (applications and reviews) 31, 4 (2001), 497–508.

[89] James A Preiss, Wolfgang Honig, Gaurav S Sukhatme, and Nora Ayanian. 2017. Crazyswarm: A large nano-quadcopter swarm. In *2017 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 3299–3304.

[90] James A. Preiss*, Wolfgang Hönig*, Gaurav S. Sukhatme, and Nora Ayanian. 2017. Crazyswarm: A large nano-quadcopter swarm. In *IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 3299–3304. doi:10.1109/ICRA.2017.7989376 Software available at https://github.com/USC-ACTLab/crazyswarm.

[91] Tabish Rashid, Mikayel Samvelyan, Christian Schroeder De Witt, Gregory Farquhar, Jakob Foerster, and Shimon Whiteson. 2020. Monotonic value function factorisation for deep multi-agent reinforcement learning. *Journal of Machine Learning Research* 21, 178 (2020), 1–51.

[92] Sylvestre-Alvise Rebuffi, Alexander Kolesnikov, Georg Sperl, and Christoph H Lampert. 2017. icarl: Incremental classifier and representation learning. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*. 2001–2010.

[93] Babak Rokh, Ali Azarpeyvand, and Alireza Khanteymoori. 2023. A comprehensive survey on model quantization for deep neural networks in image classification. *ACM Transactions on Intelligent Systems and Technology* 14, 6 (2023), 1–50.

[94] Sara Romiti, Christopher Inskip, Viktoriia Sharmanska, and Novi Quadrianto. 2022. Realpatch: A statistical matching framework for model patching with real samples. In *European Conference on Computer Vision*. Springer, 146–162.

[95] Fabrice Saffre, Hanno Hildmann, Hannu Karvonen, and Timo Lind. 2022. Monitoring and cordoning wildfires with an autonomous swarm of unmanned aerial vehicles. *Drones* 6, 10 (2022), 301.

[96] Danilo Sato, Arif Wider, and Christoph Windheuser. 2019. Continuous delivery for machine learning: Automating the end-to-end lifecycle of machine learning applications. *Martin Fowler* (2019).

[97] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. 2015. High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv:1506.02438* (2015).

[98] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347* (2017).

[99] Manju Shanmugam, Prabha M, Farithkhan A, Sivagurunathan G, and Michael Mahesh K. 2025. Decentralized control design for UAV swarms communication. *Discover Applied Sciences* 7, 2 (Feb. 2025). doi:10.1007/s42452-024-06408-w

[100] Haiying Shen, Guoxin Liu, and Harrison Chandler. 2015. Swarm Intelligence Based File Replication and Consistency Maintenance in Structured P2P File

Sharing Systems. *IEEE Trans. Comput.* 64, 10 (2015), 2953–2967. doi:10.1109/TC.2015.2389845

[101] Zhiqiang Shen, Zechun Liu, Jie Qin, Marios Savvides, and Kwang-Ting Cheng. 2021. Partial is better than all: Revisiting fine-tuning strategy for few-shot learning. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 35. 9594–9602.

[102] Ali Shoker, Fernando Alves, and Paulo Esteves-Verissimo. 2023. ScalOTA: Scalable Secure Over-the-Air Software Updates for Vehicles. doi:10.48550/arXiv.2307.02032 arXiv:2307.02032.

[103] Seung Woo Son, Dong Woo Kim, Woong Gi Sung, and Jae Jin Yu. 2020. Integrating UAV and TLS approaches for environmental management: A case study of a waste stockpile area. *Remote sensing* 12, 10 (2020), 1615.

[104] David St-Onge, Vivek Shankar Varadharajan, Ivan Švogor, and Giovanni Beltrame. 2020. From design to deployment: decentralized coordination of heterogeneous robotic teams. *Frontiers in Robotics and AI* 7 (2020), 51.

[105] Daniel H Stolfi and Grégoire Danoy. 2024. An ARGoS plug-in for the Crazyflie drone. *arXiv preprint arXiv:2401.16948* (2024).

[106] Sainbayar Sukhbaatar, Rob Fergus, et al. 2016. Learning multiagent communication with backpropagation. *Advances in neural information processing systems* 29 (2016).

[107] Richard S Sutton, David McAllester, Satinder Singh, and Yishay Mansour. 1999. Policy gradient methods for reinforcement learning with function approximation. *Advances in neural information processing systems* 12 (1999).

[108] Ming Tan et al. 1993. Multi-agent reinforcement learning: Independent vs. cooperative agents. In *Proceedings of the tenth international conference on machine learning*. 330–337.

[109] Jun Tang, Haibin Duan, and Songyang Lao. 2023. Swarm intelligence algorithms for multiple unmanned aerial vehicles collaboration: A comprehensive review. *Artificial Intelligence Review* 56, 5 (2023), 4295–4327.

[110] Frimpong Twum, Charlyne Carol Eyram Ahiable, Stephen Opoku Oppong, Linda Banning, and Kwabena Owusu-Agyemang. 2025. Employing transfer learning for breast cancer detection using deep learning models. *PLOS Digital Health* 4, 6 (2025), e0000907.

[111] Georgios Tzoumas, Lenka Pitonakova, Lucio Salinas, Charles Scales, Thomas Richardson, and Sabine Hauert. 2023. Wildfire detection in large-scale environments using force-based control for swarms of UAVs. *Swarm Intelligence* 17, 1 (2023), 89–115.

[112] Vivek Shankar Varadharajan, David St Onge, Christian Guß, and Giovanni Beltrame. 2018. Over-the-Air Updates for Robotic Swarms. *IEEE Software* 35, 2 (2018), 44–50. doi:10.1109/MS.2018.111095718

[113] Vivek Shankar Varadharajan, Karthik Soma, Sepand Dyanatkar, Pierre-Yves

Lajoie, and Giovanni Beltrame. 2024. Hierarchies define the scalability of robot swarms. *arXiv preprint arXiv:2405.02417* (2024).

[114] Vivek Shankar Varadharajan, David St-Onge, Bram Adams, and Giovanni Beltrame. 2020. Soul: Data sharing for robot swarms. *Autonomous Robots* 44, 3 (2020), 377–394.

[115] Vivek Shankar Varadharajan, David St-Onge, Bram Adams, and Giovanni Beltrame. 2020. Swarm relays: Distributed self-healing ground-and-air connectivity chains. *IEEE Robotics and Automation Letters* 5, 4 (2020), 5347–5354.

[116] András Vargha and Harold D Delaney. 2000. A critique and improvement of the CL common language effect size statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics* 25, 2 (2000), 101–132.

[117] Anna Maria Vegni, Valeria Loscrí, Carlos T Calafate, and Pietro Manzoni. 2022. Communication technologies enabling effective UAV networks: A standards perspective. *IEEE Communications Standards Magazine* 5, 4 (2022), 33–40.

[118] Michael Vierhauser, Sean Bayley, Jane Wyngaard, Wandi Xiong, Jinghui Cheng, Joshua Huseman, Robyn Lutz, and Jane Cleland-Huang. 2021. Interlocking Safety Cases for Unmanned Autonomous Systems in Shared Airspaces. *IEEE Transactions on Software Engineering* 47, 5 (2021), 899–918. doi:10.1109/TSE.2019.2907595

[119] Dinghua Wang, Shuqing Li, Guanping Xiao, Yepang Liu, Yulei Sui, Pinjia He, and Michael R. Lyu. 2024. An Exploratory Investigation of Log Anomalies in

Unmanned Aerial Vehicles. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 210, 13 pages. doi:10.1145/3597503.3639186

[120] Karl Weiss, Taghi M Khoshgoftaar, and DingDing Wang. 2016. A survey of transfer learning. *Journal of Big data* 3 (2016), 1–40.

[121] Mitchell Wortsman, Gabriel Ilharco, Samir Ya Gadre, Rebecca Roelofs, Raphael Gontijo-Lopes, Ari S Morcos, Hongseok Namkoong, Ali Farhadi, Yair Carmon, Simon Kornblith, et al. 2022. Model soups: averaging weights of multiple fine-tuned models improves accuracy without increasing inference time. In *International conference on machine learning*. Pmlr, 23965–23998.

[122] Liudong Xing and Barry W Johnson. 2022. Reliability theory and practice for unmanned aerial vehicles. *IEEE Internet of Things Journal* 10, 4 (2022), 3548–3566.

[123] Junlin Xiong and James Lam. 2007. Stabilization of linear systems over networks with bounded packet loss. *Automatica* 43, 1 (2007), 80–87.

[124] Huanyu Yang, Jun Wang, and Jiacun Wang. 2023. Efficient detection of forest fire smoke in UAV aerial imagery based on an improved YOLOv5 model and transfer learning. *Remote Sensing* 15, 23 (2023), 5527.

[125] Jason Yosinski, Jeff Clune, Yoshua Bengio, and Hod Lipson. 2014. How trans-

ferable are features in deep neural networks? *Advances in neural information processing systems* 27 (2014).

[126] Chao Yu, Akash Velu, Eugene Vinitsky, Jiaxuan Gao, Yu Wang, Alexandre Bayen, and Yi Wu. 2022. The surprising effectiveness of ppo in cooperative multi-agent games. *Advances in neural information processing systems* 35 (2022), 24611–24624.

[127] Mehmet Erkan Yüksel. 2020. Power consumption analysis of a Wi-Fi-based IoT device. *Electrica* 20, 1 (2020), 62–70.

[128] Kaiqing Zhang, Tao Sun, Yunzhe Tao, Sahika Genc, Sunil Mallya, and Tamer Basar. 2020. Robust multi-agent reinforcement learning with model uncertainty. *Advances in neural information processing systems* 33 (2020), 10571–10583.

[129] Kaiqing Zhang, Zhuoran Yang, and Tamer Başar. 2021. Multi-agent reinforcement learning: A selective overview of theories and algorithms. *Handbook of reinforcement learning and control* (2021), 321–384.

[130] Xiangyu Zhong, Xiaojun Yuan, Huiyuan Yang, and Chenxi Zhong. 2022. UAV-assisted hierarchical aggregation for over-the-air federated learning. In *GLOBECOM 2022-2022 IEEE Global Communications Conference*. IEEE, 807–812.

[131] Xinxin Zhou, Yun Huang, Guanghan Bai, Bei Xu, and Junyong Tao. 2024. The resilience evaluation of unmanned autonomous swarm with informed agents under partial failure. *Reliability Engineering & System Safety* 244 (2024), 109920.

[132] Yiren Zhou, Seyed-Mohsen Moosavi-Dezfooli, Ngai-Man Cheung, and Pascal Frossard. 2018. Adaptive quantization for deep neural network. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 32.

[133] Zihao Zhou, Jie Tang, Wanmei Feng, Nan Zhao, Zhutian Yang, and Kai-Kit Wong. 2024. Optimized routing protocol through exploitation of trajectory knowledge for uav swarms. *IEEE Transactions on Vehicular Technology* 73, 10 (2024), 15499–15512.