Thesis advisor | Author

Radhika Nagpal | **Jason Waterman**

Coordinated Resource Management in Networked Embedded Systems

Abstract

This dissertation shows that with simple programming abstractions, network-wide resource coordination is efficient and useful for programming embedded sensor networks. Existing systems have focused primarily on managing resources for individual nodes, but a sensor network is not merely a collection of nodes operating independently: it must coordinate behavior across multiple nodes to achieve high efficiency. We need tools that can enable system-wide coordination at a higher level of abstraction than what exists today.

We present three core contributions. The first is a service called IDEA that enables network-wide energy management for sensor networks. It unites energy monitoring, load modeling, and distributed state sharing into a single service that facilitates distributed decision making. Using simulation and testbed results, we show that IDEA enables improvements in network lifetime of up to 35% over approaches that do not consider energy distribution.

Our second contribution is Karma, a system for coordinating insect-sized robotic micro-aerial vehicle (MAV) swarms, an emerging class of mobile sensor networks. Karmas system architecture simplifies the functionality of an individual MAV to a sequence of sensing and actuation commands called behaviors. Each behavior has an associated *progress function*, a measure of how much of that behavior has been completed. Programming is done by composing behaviors which are coordinated using input from the progress functions. Through simulation and testbed experiments, we demonstrate Karma applications can run on limited resources, are robust to individual MAV failure, and adapt to changes in the environment.

Our final contribution is Simbeeotic, a testbed for MAV coordination algorithms. MAV sensors must be codesigned with the software and coordination algorithms that depend on them. This requires a testbed capable of simulating sensors to evaluate them before actual hardware is available and the ability to test with real flight dynamics for accurate control evaluation. In addition, simulation should be able to scale to hundreds or thousands of MAVs at a reduced level of fidelity in order to test at scale. We demonstrate that Simbeeotic provides the appropriate level of fidelity to evaluate prototype systems while maintaining the ability to test at scale.

Contents

List of Figures

List of Tables

Acknowledgments

First I'd like to thank Matt Welsh. If it wasn't for his support, I wouldn't be here today. I've learned so much from my time in his group and appreciate his philosophy of building and evaluating real systems. I look forward to continuing that philosophy with my own students.

I'd also like to thank Margo Seltzer, Radhika Nagpal, Eddie Kohler, and Greg Morriset for filling in after Matt left. I lost an adviser but gained a family in the process. It shows how much the department cares about its students. A special thanks to Margo for keeping this dissertation on track by figuratively poking me with her sharp, pointy stick. Her comments and suggestions have greatly improved the clarity and quality of this dissertation.

One of the best parts of being in grad school is the grad students you get to collaborate with. Bor-rong Chen, Konrad Lorincz, Geoffrey Mainland, and Geoffrey Challen (GWA) all welcomed me into Matt's group and showed me the ropes. I am especially grateful for the collaboration on IDEA with GWA. I learned a lot about the entire process of research–starting with a nugget of an idea and taking it all the way to publication–while working with GWA. While I didn't have the pleasure to work directly with Rohan Murty, I am grateful for his friendship and I always look forward to our conversations together.

During my time working on the RoboBees project, I couldnt have asked for better colleagues than Bryan Kate and Karthik Dantu. Our collaboration on Simbeeotic and Karma meant many long hours in the lab together, and I was glad to be able to spend it with them. Bryans experience in writing simulators was very helpful in designing the architecture of Simbeeotic, and I am grateful for his contribution to the Simbeeotic core. In addition to being great to work with, Bryan and Karthik were wonderful traveling companions. I will always remember our conference trips together in Seattle and Vancouver, as well as Bryan and I trying mystery foods in Beijing and climbing the Great Wall of China together.

Last but not least, I'd like to thank my fiancé, Heather Campbell. We first met just a few weeks before I started grad school, and I tried to warn her what dating a grad student was going to be like. However, she was not deterred, and I am a lucky man for that. Heather's sense of humor, inspiration, and support got me through the tough times and I cannot thank her enough for that. I'm looking forward to our next great adventure together.

Dedicated to my grandmother Mae.

Chapter 1

Introduction

Advances in system-on-a-chip manufacturing, low power communication, and micro-electromechanical system (MEMS) sensors are enabling new forms of embedded systems such as wireless sensor networks (WSNs). A typical WSN consists of a large collection of nodes distributed throughout a physical area, sensing the environment through one or more sensors to achieve an application-specific goal. This approach has several advantages over deploying a single or small number of large, more sophisticated sensors. A dense large-scale deployment extends spatial coverage while still achieving high-resolution sensing. Having many deployed nodes increases the fault-tolerance of the system, allowing the system to degrade gracefully in the event of node failure.

Mobile sensor networks are another emerging class of embedded systems. Nodes in these networks have the ability, through self actuation, to move around the environment. Mobility allows nodes to cover a larger spatial area and reposition themselves as conditions change in the environment.

Both classes of these embedded sensor networks present several programming challenges.

In these systems, resources such as energy, sensing, and processing are extremely limited and must be carefully managed across the network. Existing systems have focused primarily on managing resources for individual nodes, but these networks are not merely a collection of nodes operating independently: they must coordinate behavior across multiple nodes to achieve high efficiency. Subtle changes to node-level behavior (such as the radio listening duty cycle or choice of routing path) can have a tremendous impact on the overall efficiency and data yield of the network. As it is usual to deploy these networks with a single purpose in mind, it is natural to conceive of an embedded sensor network as a single programmable entity that operates in a coordinated fashion to achieve some high-level system goal. However, programming these complex coordination behaviors is typically done in an *ad hoc* fashion, using bare-bones APIs provided by the node-level OS. What is needed are tools that can enable system wide coordination at a higher level of abstraction than what exists today. In this dissertation I show that with simple programming abstractions, network-wide resource coordination is efficient and useful for programming embedded sensor networks.

1.1 Wireless Sensor Networks

Wireless sensor networks (WSNs) are an important class of embedded systems with novel applications in areas such as environmental monitoring [76], medicine [51], and industrial monitoring [40]. A prototypical WSN node is the Telos ultra-low power wireless module (mote) [65] shown in Figure 1.1.

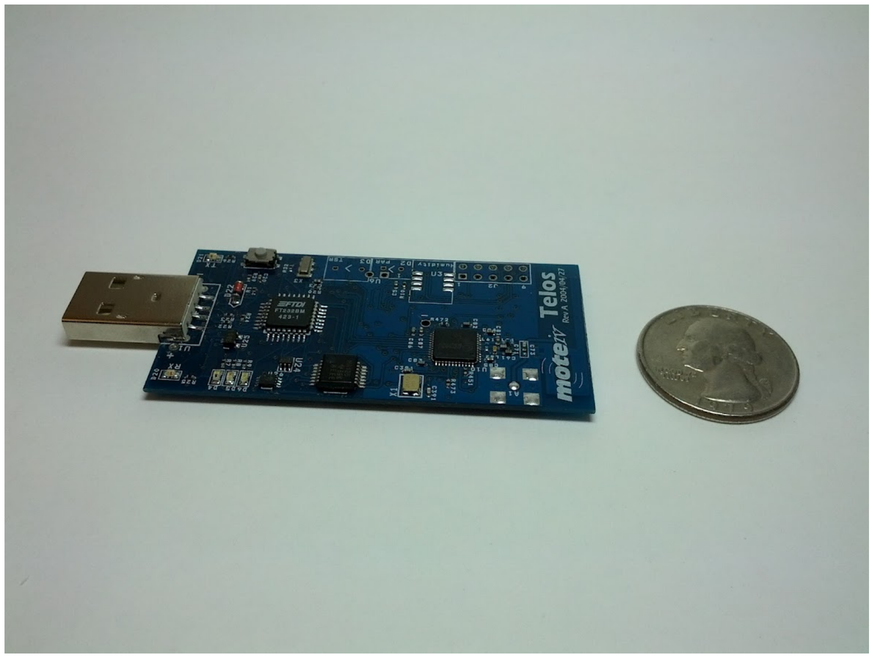


Figure 1.1: A Telos mote.

Telos motes have a low-power microcontroller (MCU) for processing, limited on-board storage, and a low-bandwidth radio for communication. They are powered by batteries, which allows for untethered operation, but it means there are power consumption constraints for nodes. For example, radio communication requires about an order of magnitude more energy than processing, implying that communication should be kept to a minimum.

Typically, nodes in a WSN are deployed in areas where there is no existing infrastructure. The nodes must be able to self-organize, withstand harsh environments, and cope with failure. With the limited resources mentioned above, it is necessary to orchestrate resource management decisions across the network as a whole. For example, consider a network to monitor seismic activity at a volcano [84, 86]. Nodes must decide how much of their limited energy to invest in sampling, storing, and processing local sample data; transmitting signals to the base station; and listening for and routing packets for other nodes deeper in the routing tree. The resource load on each node is a complex function of the activity level of the volcano, quality of the sensor data, and packet-forwarding demand from other nodes. This problem becomes more complex when nodes are powered by solar cells, since the energy budget fluctuates. It is important to note that both resource *load* and resource *availability* fluctuate over time: an offline static solution cannot suffice.

1.2 Mobile Sensor Networks

Micro-aerial vehicles (MAVs) are an emerging class of mobile sensing systems consisting of small autonomous aerial vehicles capable of limited computing, communication, sensing, and actuation. Example research platforms include quadrotors [55], fixed-wing aircraft [30], and small "flying motes" [**?**]. At the forefront of MAV research is the RoboBees project whose goal is to construct *insect-sized* flapping-wing MAVs [1]. Recent advances in airframe construction, flight control, sensor design and high-density power sources are pushing insect-scale MAVs closer to reality. With this class of devices, many novel research directions for mobile distributed sensing applications have emerged.

Insect-scale platforms have two advantages – extremely small size and deployment in large numbers. Insect-scale MAVs will be relatively inconspicuous and can operate in enclosed, close-quartered areas where traditional aerial vehicles cannot fly. In the future, these systems can be used to perform tasks that are challenging for larger platforms, such as landing on a flower and collecting or depositing pollen. Further, hundreds to thousands of MAVs will be able to be deployed in conjunction to achieve a specific task in a massively parallel fashion. In these cases, a

MAV swarm makes up for lack of sophisticated sensing and actuation through scale in deployment, providing parallelism and robustness to failure in the field.

1.3 Coordination Challenges in Embedded Sensor Networks

Designing an embedded sensor network to make efficient use of scarce resources while yielding high-quality data presents a number of challenges. Not only are node resources limited, but small local changes in a node's operation can have a ripple effect throughout the network. For example, if a node shuts down its radio to save energy, many nodes in the network may need to reconfigure their routing tables. Moreover, nodes are mutually dependent upon each other to relay data, maintain time synchronization, perform collaborative event detection, and maintain spatial sensor coverage. It is not enough to conceive of a network as a mere collection of independent nodes, yet that is the dominant programming abstraction supported by existing sensor network operating systems [29, 33, 52].

As applications increase in complexity, reasoning about the global effects of local changes to node behavior can be difficult. The most common form of resource management is simple duty cycling, in which a (usually static) period is assigned to each node to achieve a given target lifetime. This works fine for applications with simple periodic behavior and few configuration changes over time. However, applications with more dynamic resource requirements, such as the energy and bandwidth needed for a high-fidelity body sensor network [50], need more sophisticated approaches involving adaptation over time as well as both local and global knowledge of resource availability.

To achieve the greatest efficiency, nodes cannot simply make local decisions on how to allocate their resources. Rather, it is necessary to perform resource adaptation in a coordinated fashion, where nodes are assigned tasks and are allocated node-level resources to achieve the greatest common good. Such coordination can be done in a distributed fashion within local clusters of nodes done centrally network-wide. While network-wide coordination has the potential for greater optimization, this must be traded off against the higher latency for communicating demand and availability of a centralized controller.

Mobility allows a MAV swarm to cover a much larger area than a stationary sensor network and to reposition the coverage as the features of interest change in the environment. However, mobility also presents a unique set of challenges. Actuation dominates the weight and power budgets for these devices, keeping sensing and control to the bare minimum. The limits on sensing and control imply that MAVs will often fail in the field. A successful system must be able to detect the failure of individual of MAV nodes and reallocate resources to gracefully degrade the performance of the system.

1.4 Contributions and Structure of the Dissertation

In this dissertation we show that with simple programming abstractions, network-wide resource coordination is efficient and useful for programming embedded sensor networks.

We show this by presenting two programming frameworks, IDEA and Karma. IDEA is a distributed sensor network service enabling effective network-wide energy decision making, and Karma is a centralized system for programming and managing MAV swarms. We also present Simbeeotic, a simulator and testbed for MAV swarm experiments, developed to support our MAV research.

The rest of this dissertation is organized as follows. Chapter 2 presents Integrated Distributed Energy Awareness (IDEA), a network service enabling effective network-wide energy decision making. Managing limited energy is a key challenge in sensor networks. While previous research has addressed reducing the energy usage of individual nodes, these nodes do not act in isolation. For example, the choice of how often a node listens for radio packets affects the energy usage of any node that wants to communicate with that node. Programming these complex distributed behaviors is still done in an *ad hoc* fashion, using bare-bones APIs provided by the node-level OS. IDEA *integrates* into the sensor network application by providing an API allowing components to evaluate their impact on other nodes. IDEA *distributes* information about each node's load rate, charging rate, and battery level to other nodes whose decisions affect it. Finally, IDEA enables *awareness* of the connection between the behavior of each node and the application's energy goals, guiding the network toward states that improve performance. We also describe the IDEA architecture and demonstrate its use through two case studies. Using both simulation and testbed experiments, we evaluate each IDEA application by comparing it to simpler approaches that do not integrate distributed energy awareness. We show that using IDEA can significantly improve performance compared with solutions operating with purely local information.

Chapter 3 describes Simbeeotic, our simulator and testbed for MAV swarm experiments. MAV swarms are an emerging class of mobile sensing systems. Simulation and staged deployment to prototype testbeds are useful in the early stages of large-scale system design, when hardware is unavailable or deployment at scale is impractical. To faithfully represent the problem domain, an MAV swarm simulator must be able to model the key aspects of the system: actuation, sensing, and communication, as well as be able to scale to simulate thousands of MAVs. During a search of existing simulators, we were unable to find one that adequately met our needs, so we developed

Simbeeotic. Simbeeotic enables algorithm development and rapid MAV prototyping through pure simulation and hardware-in-the-loop experimentation. We demonstrate that Simbeeotic provides the appropriate level of fidelity to evaluate prototype MAV systems while maintaining the ability to test at scale.

Chapter 4 presents Karma, a framework for programming MAV swarms. We propose a novel system architecture based on a *hive-drone* model that simplifies the functionality of an individual MAV to a sequence of sensing and actuation commands coordinated by a centralized controller. This decision simplifies the hardware and software complexity of individual MAVs by moving the complexity of coordination entirely to a central hive computer. Through simulation and testbed experiments we demonstrate how applications in Karma can run on limited resources, are robust to individual MAV failure, and adapt to changes in the environment.

Chapter 5 discusses related work in coordinated resource management for programming clusters, coordinated energy management for embedded sensor networks, and resource coordination for swarms. We show how our work relates to and builds upon this work. Wrapping up, Chapter 6 concludes with possible future extensions to IDEA, Simbeeotic, and Karma.

Chapter 2

IDEA: Integrated Distributed Energy Awareness for Wireless Sensor Networks

Managing limited energy resources is a key challenge in sensor networks. Every action that a node takes reduces its finite energy store and must be carefully chosen so that the network is able to meet its application goals and its lifetime target. To achieve high efficiency, it is necessary to orchestrate energy management decisions across the network. For example, consider a network monitoring seismic activity at a volcano [84, 86]. Nodes must decide how much of their limited energy to invest in sampling, storing, and processing local sample data; transmitting signals to the base station; and listening for and routing packets for other nodes deeper in the routing tree. The resource load on each node is a complex function of the activity level of the volcano, quality of the sensor data, and packet-forwarding demand from other nodes. This problem becomes more complex when nodes are powered by solar cells [16], since the energy budget also fluctuates. It is important to note that both resource *load* and resource *availability* fluctuate over time: an offline static solution cannot suffice.

Coordinating energy resource management in a sensor network has received considerable attention [9, 31]. However, *programming* these complex distributed behaviors is still done in an *ad hoc* fashion, using bare-bones APIs provided by the node-level OS. Subtle changes to node-level behavior (such as the radio-listening duty cycle or choice of routing path) can have a tremendous impact on the overall efficiency and data yield of the network. Existing systems provide few tools to assist developers in designing correct and efficient solutions.

In this chapter, we present a design for sensor networks enabling coordinated energy management while providing the appropriate abstractions and mechanisms to support collaborative energy management. We present Integrated Distributed Energy Awareness (IDEA), a sensor network service enabling effective network-wide energy decision making. IDEA *integrates* into the sensor network application by providing an API allowing components to evaluate their impact on other nodes. IDEA *distributes* information about each node's load rate, charging rate, and battery level to other nodes whose decisions affect it. Finally, IDEA enables *awareness* of the connection between the behavior of each node and the application's energy goals, guiding the network toward states that improve performance. We describe the IDEA architecture and demonstrate its use through two case studies, and show that using IDEA can significantly improve performance compared to solutions operating with purely local information. This work was done jointly with Geoffrey Werner Challen; parts of this chapter also appear in his dissertation [10].

2.1 Motivation

IDEA's architecture is motivated by two observations. First, many sensor network applications require a large portion of the network to meet their application goals. As a result, failures of sensor nodes can deeply affect operation of the system. Indeed, the most heavily-loaded nodes are often those that are most critical to the application. Consider a node near the root of a spanning tree that is responsible for forwarding traffic for a substantial portion of the network. Loss of this single node can have a disproportionate effect on the whole network's operation.

Second, in most applications, some portion of the load at each node is due to interaction with other nodes. This is an external load and cannot be reduced unilaterally. In the case of routing, nodes spend their own energy to listen for and forward packets for other nodes. In such cases, load mitigation must be negotiated with the peer nodes producing the load. For example, a node with a valuable sensor input might do everything possible to reduce its own power consumption, but unless it can move itself off of a high-traffic routing path, it will be unable to reduce energy expenditure beyond a certain point.

Existing approaches to sensor network energy management suffer from several weaknesses. Greedy approaches to local energy minimization assume that each node minimizing its own power consumption is best for the network as a whole [52]. However, this is not always the case. Such approaches also cannot address the external load problem described above, which requires coordination between nodes. Some sensor network protocols embed forms of distributed energy management into their operation [90], but these ad-hoc solutions encode policies that may be unsuitable for other applications. IDEA addresses these deficiencies by providing a distributed service allowing any component controlling distributed load to perform collaborative energy management.

2.1.1 Example: Energy-Aware Routing

As a simple example demonstrating the need for IDEA, consider a four-node routing problem. Figure 2.1 shows the network topology, showing the energy required to reliably transfer a packet over each link. To simplify the example, we assume the energy to receive a packet is the same as that required to send it, so for Node 3 to send Node 1 a packet, it will cost each node 0.5 mJ. We also assume that for correct operation of the application, all four nodes must remain operational, meaning that the loss of a single node will render the network useless.

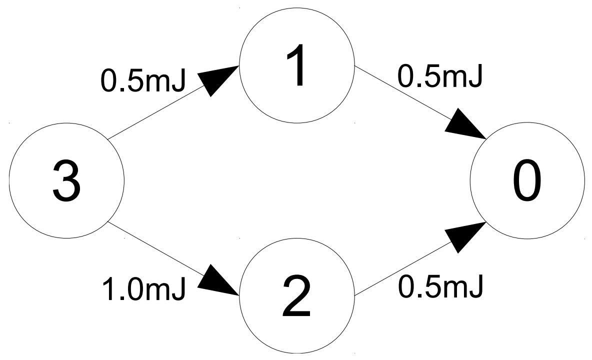


Figure 2.1: An example routing problem. The edges are the energy in mJ to send and receive a packet.

Looking at Node 3, we see it has two routes to Node 0: $3 \rightarrow 1 \rightarrow 0$ and $3 \rightarrow 2 \rightarrow 0$. If Node 3 tries to conserve power by making a local greedy decision, it will route through Node 1, since sending a packet to Node 1 consumes 0.5 mJ of energy as opposed to 1.0 mJ for sending to Node 2. Even if we assume Node 3 knows the power consumption of the links $1 \rightarrow 0$ and $2 \rightarrow 0$, with no other information it still chooses the route though Node 1, which consumes less total energy per packet than the route through Node 2. However, there are several conditions where route $3 \rightarrow 1 \rightarrow 0$ actually *harms* application performance.

- **Differences in initial battery levels:** If Nodes 2 and 3 have significantly more energy than Node 1, then routing through Node 2 can increase the lifetime of Node 1, which due to its low battery level defines the lifetime of the network since all nodes must remain operational in our example.

- **Differences in non-routing load rates:** If Node 2 has a higher sampling rate or is receiving packets from another node, it may be draining its battery at a faster rate than the other nodes even with Node 3 using the $3 \rightarrow 2 \rightarrow 0$ route.

- **Differences in charging rates:** If Node 3 and Node 2 are harvesting energy from the environment at a much greater rate than Node 1, routing packets through Node 2 will increase network lifetime, even though it is using more energy overall.

Making a decision at Node 3 to meet application goals requires that it know the load rates, charging rates, and battery levels at Nodes 1 and 2. IDEA addresses this problem by distributing this information across the set of affected nodes. The cases above motivate several features in the IDEA design. In general, the network may want to shift load *toward* nodes that have a great deal of stored energy, low load rates, or high charging rates, and *away* from nodes with low batteries, low charging rates, or that are already highly loaded. In cases where shifting load produces extra overall load for the network, as it does above, changes in load distribution must be managed by the application based on its own goals and requirements. Had our application above been able to tolerate the loss of Node 1, it might have chosen to optimize charging at Nodes 2 and 3 in the third example. Respecting these differences, IDEA is designed to incorporate application-level input into its decision-making process.

2.2 Architecture

We now present the IDEA architecture, beginning with a formal problem definition and brief overview, followed by a detailed description of each major system component.

2.2.1 Problem Definition

IDEA is intended to address the problem of energy-aware tuning in sensor network applications. In IDEA, we use the term *client* to refer to either an application (such as a tracking system) or an individual software component (such as a MAC parameter tuning, routing, or time synchronization protocol) that wishes to perform energy tuning. Clients interact with the IDEA runtime residing on each sensor node to make decisions that affect energy consumption and data fidelity.

Sensor network software components commonly operate by making local decisions. For example, routing protocols [24, 88] typically form a spanning tree by each node picking a parent based on local information, such as the radio link quality or number of hops to the sink. Likewise, duty-cycling MAC protocols [64] decide locally how often to poll the channel and check for traffic. In IDEA, these choices are represented as a universe of possible *states* S that the client can be in at any given time. As an example, a routing protocol's states represent the set of possible parent nodes.

IDEA guides the selection of the optimal state for each client component based on both the inherent value of that state (such as the path quality to the sink in a routing protocol) as well as the *distributed energy impact* of choosing that state. In the case of routing, selecting a given parent affects the energy of the parent as well as each node along the routing path to the sink. The ideal choice of a parent may change over time, for example, based on network load or energy availability. IDEA clients periodically reevaluate their current state and may switch to a new state if it is deemed more desirable.

IDEA quantifies the distributed energy impact of each state using an application-defined *energy objective function*. Each state $s \in S$ has a corresponding energy load vector, $\bar{L}$, where each component $L_i^n(s_n)$ represents the estimated energy load on node i that will result from node n setting its local state to s_n. We represent the current battery level (in joules) at node i by B_i and the current charging rate (in joules per second) at node i by C_i. In networks without charging capability, $C_i = 0$.

Formally, we can define the problem as follows. At a given time, let us denote the global state of all nodes in the network as $S = \{s_1, s_2, \ldots, s_k\}$. The combined energy load at node i induced by this selection of states is

$$L_i(S) = \sum_{j=1}^{k} L_i^j(s_j)$$

Based on the current battery levels B_i and charging rates C_i, we can define an *energy objective function* $O(\bar{L}(S), \bar{B}, \bar{C})$ that represents the global energy impact of the global state assignment S. Likewise, this state assignment has an associated application-defined *utility* $u(S)$ that represents the intrinsic desirability of the state – for example, minimizing path length in a routing protocol. The choice of $u(S)$ can be provided by the application as a static function or learned over time by measuring application quality as it runs. IDEA is agnostic as to its form as it is evaluated online.

The system's goal is to determine the optimal state

$$S^{\star} = \arg\max_{S} O(\bar{L}(S), \bar{B}, \bar{C}) \cdot \alpha + u(S) \cdot (1 - \alpha)$$

where α represents the tradeoff factor between energy impact and intrinsic utility. Setting $\alpha = 1$ optimizes only for energy; $\alpha = 0$ only for application-defined utility.

2.2.2 Energy Objective Functions

Before describing the IDEA system itself, we first consider two energy optimization goals that the system can target. We expect that different applications will allocate energy differently, and the objective function allows the behavior of IDEA to be tuned to meet a variety of needs.

Examples of possible objective functions include:

- **Maximize time to first node death.** Depending on energy load and availability, different nodes may run out of energy at different times. Given the current load and charging rates, one can estimate the projected lifetime of each node i given global state S as

$$\mathrm{T}(i,S) = \begin{cases} \frac{B_i}{L_i(S)-C_i} & C_i < L_i(S) \\ \infty & C_i \geq L_i(S) \end{cases}$$

  To maximize the time to the first node death, we find the state $S^\star$ maximizing $O = \min_i T(i,S)$. This objective function will always choose states that shift load away from the node projected to die first, irrespective of the load that is produced on other nodes, and may be suitable for applications whose fidelity requirements are sensitive to the loss of single nodes.

- **Maximize aggregate charging rate.** Given the charging rate C_i, battery level B_i, and battery capacity P_i on node i, the effective charging rate for node i given global state S ($\mathrm{A}(i,S)$) is

$$\mathrm{A}(i,S) = \begin{cases} C_i - L_i(S) & C_i < L_i(S) \\ C_i - L_i(S) & C_i \geq L_i(S), B_i < P_i \\ 0 & C_i \geq L_i(S), B_i = P_i \end{cases}$$

  This reflects that when the node's battery fills, it is no longer able to collect charge. By maximizing $O = \sum_i \mathrm{A}(i,S)$, we choose the state that leads to the network collecting charge as quickly as possible. When node batteries are all still charging, this objective function will try to find the state minimizing the total system load. However, once batteries begin to fill, it will choose states that shift load toward nodes charging full batteries, since any additional charge these nodes capture cannot be stored. Shifting load towards overcharging nodes allows nodes without full batteries to charge more rapidly. This objective function prioritizes collecting charge over preserving node uptime, and may be well suited to applications that expect to experience periodic charging cycles and can tolerate some nodes running out of energy.

We chose to highlight these objective functions for their simplicity and usefulness. The first objective function keeps all nodes in the network running for as long as possible, treating the loss of the first node as the lifetime of the network. It is straightforward to measure the performance of IDEA using this objective function by simply noting the time at which the first node runs out of energy. For this reason, we use this objective function for the experiments in Section 2.3.2. The second objective function shows an example of a policy that would be useful in a network that is able to harvest energy from the environment, an active area of sensor network research [16, 77].

One of the tradeoffs IDEA objective functions may perform is between increasing the amount of charge collected – which leads to reducing the cumulative network-wide impact of each IDEA component – and periods of node downtime resulting from poor energy distribution. Some applications may weight node downtime differently for each node, depending on the quality of the sensor data it is providing, its location, or other factors. Application goals will differ, but the flexibility provided by the objective function allows IDEA to support a variety of different requirements.

2.2.3 IDEA Overview

Thus far, we have defined the goal of the system as achieving a *globally optimal* assignment of states to each sensor node. Performing such a global optimization would be possible through a central node (such as the base station) collecting load and charge rates from every node and computing the optimal assignment centrally, then informing all nodes of their states. However, in large networks, this approach would induce large communication overheads, reducing energy efficiency. Central control also precludes nodes from making rapid local changes to states, for example, to select a new parent in a routing tree if the current parent dies.

IDEA performs optimization in a *decentralized* fashion, with the goal of closely approximating the globally optimal solution. An important observation is that *most state changes affect only the energy consumption of a node's immediate neighbors*.[1] Hence, nodes can perform a local optimization using information gathered from their neighbors. Although this approach does not ensure that the state assignment will be globally optimal, we show in Section 2.3 that it approximates the optimal solution without requiring each node to have full network state.

[1] In the routing case referenced previously, while a node's choice of parent impacts all nodes between it and the sink, it can only *directly* control the load placed on its parent. The affect on nodes farther downstream is a function of other local choices.

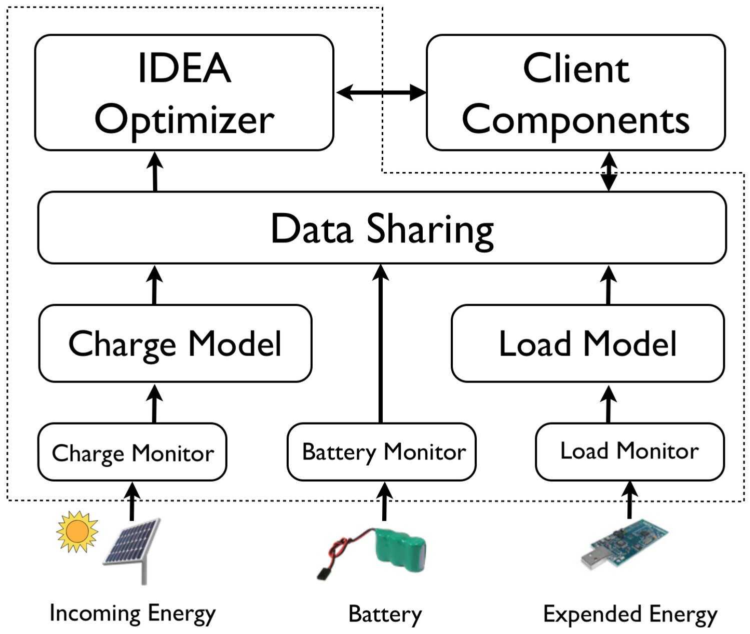


Figure 2.2: Overview of IDEA architecture. IDEA combines load and charge monitoring and modeling, energy data distribution, and an application-provided energy objective function into a single service that can easily be integrated into application components. Client states are evaluated by the energy objective function and also assigned an application utility. These scores are combined by the optimizer to select the state best balancing the application's distributed energy goals against the state's intrinsic desirability.

Figure 2.2 provides an overview of the IDEA architecture. Each node *monitors* its own load rate, charging rate, and battery level. Monitoring output is passed to a *modeling component* that produces models of load and charging behavior. Model parameters are distributed to other nodes via a *data sharing* component, which maintains a distributed table allowing energy information to be queried by energy objective functions. IDEA monitors the accuracy of each node's local model parameters, re-propagating them as necessary to maintain the distributed energy information.

Clients periodically evaluate their current state, which can be driven either by application-specific behaviors (e.g., disconnection from the parent node in the routing tree) or changes to energy availability, triggered by IDEA. The IDEA component residing on each sensor node evaluates the energy objective function O for each possible client state, which is combined with the client utility function u to determine the next state s'. In the following sections we describe each component of the architecture in more detail.

2.2.4 Monitoring and Modeling

IDEA relies on the ability to measure and model load and charging rates at each sensor node. This can be performed using either hardware support, as in systems like Quanto [23], or using software monitoring, as in Pixie [52]. Modularizing these components allows IDEA to easily support multiple node platforms and a variety of energy-harvesting hardware.

IDEA monitors both the energy load on a node as well as the charging rate, both represented as joules per second. The battery level is monitored as well. The raw measurements are used to build *models* that allow IDEA to estimate the projected future energy load and availability. In addition, the model parameters are distributed to other nodes in the network, allowing those nodes to estimate the source node's energy load and charging profile over time.

IDEA provides a component that models load or charging rates by producing an average across a fixed time window, which over time produces a piecewise linear model of varying load or charging rates. To estimate the load on a single node n at time t, $L_n(t)$, we compute $l_n = \frac{\int_{t-\Delta t}^{t} L_n(t)\,dt}{\Delta t}$, and distribute our estimate l_n as the single model parameter. This model is used because it is easy to compute on resource-constrained nodes, and the single model parameter requires little radio bandwidth to share. However, IDEA's modeling architecture is modular, and it would be straightforward to incorporate more sophisticated charging models based on an understanding of the underlying dynamics of the energy harvesting technique being used. A seasonal ARIMA model such as used by PRESTO [48] would provide more accuracy when projecting future charging behavior.

IDEA distributes the battery level $B_n(t_0)$ at time t_0 when it updates the load or charging model parameters. To estimate the battery level at time t_1, $B_n(t_1)$, we integrate the load and charging models such that $B_n(t_1) = B_n(t_0) + \int_{t_0}^{t_1} C_n(t)\,dt - \int_{t_0}^{t_1} L_n(t)\,dt$. Integrating the simple load model is straightforward: $\int_{t_0}^{t_1} L_n\,dt = (t_1 - t_0) * l_n$. Other models may require more complex techniques.

We separate the modeling of load and charging rates for two reasons. First, load and charging rates vary for different reasons: load fluctuates with application demands, whereas charging rates fluctuate with environmental variations. Disentangling energy inputs and outputs facilitates more accurate modeling. For example, if they were not modeled separately, an increase in the load rate could be masked by a simultaneous increase in the charging rate. Moreover, independent modeling of load and charging allows IDEA to accurately model times when a node's battery is exhausted. While a node is running its overall current draw $I_n = C_n - L_n$. If $I_n > \beta$, where β is a threshold current necessary to enable battery recharging, then the node is charging its battery; otherwise it is discharging. Once the node dies, however, we assume that $L_n = 0$ and $I_n = C_n$. Assuming future energy inputs, a node that has completely drained its battery will be able to recharge and rejoin the network once it has charged its battery past a certain threshold.

2.2.5 Data Sharing

In order for nodes to make informed decisions about local state changes, they must have knowledge of the energy profiles of other nodes. IDEA provides a *data sharing* component that distributes this information among nodes in the network. The distribution service maintains a local shared data table allowing estimated energy information for other nodes – including their battery levels B_i, load rate L_i, and charge rate C_i – to be queried.

The data-sharing layer has evolved to minimize the communication overhead of data sharing. Any gains made by making better energy decisions can be wiped out by the cost of sharing the data used to make those decisions. This section describes the data-sharing layer in IDEA and discusses the features we added to reduce the communication overhead of data sharing.

Our first approach was to develop a network-wide state-sharing mechanism. We used Trickle [47] to balance rapid propagation of updates with eventual consistency in the face of link failures. Trickle uses a "polite gossip" policy where nodes periodically share data with their local neighbors but stay quiet if they have recently heard the data. Updates cause immediate data propagation. Nodes hearing the update relay it until the maximum number of retransmissions is reached. We also used broadcast packets to opportunistically retransmit data for other nodes to reduce propagation latency. When retransmission is triggered, a node fills the broadcast packet with other recent updates from its shared data table, which also helps reduce the cost of data sharing.

IDEA clients may piggyback on this mechanism to propagate application-specific data to other nodes. For example, nodes might wish to share information on MAC parameters to enable coordinated communication scheduling. To simplify the implementation of the data-sharing service we limited the amount of space available to client applications to ensure that the total payload fits within a single radio message.

However, for some applications, propagating state across the entire network is not necessary and contributes to extra overhead. If a node's choice of state only affects the energy behavior of its neighbors, then a more limited form of state sharing will suffice. We then added the ability to let the application give input on how far to propagate shared data. IDEA provides a k-hop data-sharing component that disseminates shared data updates using broadcast messages. This approach is similar to neighborhood communication schemes such as Abstract Regions [83] and Hoods [87].

Once our k-hop data-sharing layer was working, it was soon apparent that the raw charge, battery, and load data was changing too quickly to be shared without unbearable communication overhead. This led to the development of the modeling layer described above in Section 2.2.4. For example, a node's battery level can easily be calculated from the load and charge models and does not need to be shared with every query to the node's battery level. However, the calculated battery level is only as good as the accuracy of the load and charging models.

Maintaining the accuracy of load and charging models on external nodes requires the data-sharing layer to periodically distribute updated model parameters. The data-sharing level works with the modeling layer to monitor the accuracy of the model they have previously distributed. Using our simple linear model as an example, if $l_n^{t_0}$ is the model parameter distributed for node n at time t_0, then at time t_1 the model will recompute $l_n^{t_1}$. If the relative model error $\frac{\left|l_n^{t_1} - l_n^{t_0}\right|}{l_n^{t_0}} > E$, where E is an application-configurable error tolerance, then the modeling component will push a new parameter to the data-sharing layer, which is responsible for updating other nodes. For example, if an application only wanted to know if a node's battery was "High," "Medium," or "Low," it would not need the most accurate battery level information. Error tolerance allows applications to reduce data-sharing overhead at the cost of some accuracy. With the addition of the modeling layer and the sharing of the model parameters instead of raw data, we were finally able to manage the overheads of the data-sharing layer.

2.2.6 Client Integration

The interface between client components and IDEA is intended to simplify integration of IDEA with existing software. The IDEA optimizer provides `chooseState()`, an interface that the client can invoke to select a new state in an energy-aware fashion. Normally components may reexamine states periodically to ensure that they respond to changes in network dynamics. IDEA also provides event triggers that indicate when nearby energy conditions have changed significantly, since these may also be opportunities for clients to reevaluate their local state selection. `chooseState()` takes three arguments:

- a list of possible local states $s^n = \left\{s_1^n, s_2^n, \ldots, s_k^n\right\}$ that the client component on node n can enter;
- for each state s_i^n, the intrinsic utility $u(s_i^n)$ of that state, represented as a scalar value; and
- for each state s_i^n, a projected energy load vector $\bar{L}(s_i^n)$ representing the estimated energy impact (in terms of joules/sec) induced by the node entering state s_i^n. $\bar{L}$ has one element for each of the node's neighbors. If it not possible to easily calculate the projected energy load vector, an energy profiling system such as Quanto [23] could be used to empirically determine the projected energy load vector.

IDEA combines this information with knowledge of energy load and availability to determine the ideal state s' the node should enter based on the weighted combination of the objective function O and the utility u. `chooseState()` returns the new state s' selected by the optimizer. To reduce the possibility of two or more nodes oscillating between different states, hysteresis can be added to the objective function to avoid wasting energy through frequent reconfiguration.

In many cases it is straightforward to interface IDEA to existing code. As we demonstrate in Section 2.3.2, IDEA has been used to add energy awareness to tune radio listening intervals with minimal code changes. Existing software components can be supported by wrapping them in code that estimates energy impact, enumerates states, and interfaces to the IDEA service.

2.3 Evaluation

To evaluate IDEA, we built and tested an energy-aware component for adjusting radio listening intervals and one energy-aware application for distributed acoustic localization. For the radio listening component, we compare the performance of our IDEA-based implementations to approaches that are not energy-aware. For the localization application we use IDEA to implement several energy objective functions and compare their performance against each other and against a heuristic that does not consider energy availability.

2.3.1 Experimental Setup

Throughout the evaluation we present results run in several different environments. We have implemented IDEA for TinyOS in order to run experiments on MoteLab [85], our 180 node Telos mote [65] ultra-low power wireless sensor network testbed. The Telos motes consist of a Texas Instruments (TI) MSP430 16-bit microcontroller running at 8 MHz, 48 kB program memory, 10 kB RAM, and 1 MB of on-board flash storage. When the MSP430 is in its low-power standby state, it draws only 1 μA of current and while fully active draws less than 2 mA. For radio communications, Telos has a TI CC2420 2.4 GHz IEEE 802.15.4 with a maximum data rate of 250 kbps. While the radio is active, either while listening or transmitting, the CC2420 uses less that 20 mA.

We also present results obtained using TOSSIM [45], the TinyOS simulator. TOSSIM incorporates a closest-fit pattern-matching noise model to accurately capture complex link dynamics [44]. TOSSIM allows us to run longer experiments incorporating various solar-charging models. To improve the realism of TOSSIM we began with a modified version developed for the Koala project [68] and performed further modifications to correctly simulate the operation of the radio protocol described in section 2.3.2. We use information collected on MoteLab to build a realistic TOSSIM radio model for our simulations. For the acoustic localization application, we built a Python simulator to allow rapid prototyping of various energy objective functions.

IDEA is designed to tune components in the face of variations in both load and charging rates, and to test this we present experiments using solar-charging data collected off of a solar panel deployed on an Arlington, Massachusetts rooftop in March 2009. Battery levels are calculated using a charging model based on a nickel-metal hydride battery technology with a 66% charging efficiency. We attenuate this data to simulate the charging produced by solar panels of several different sizes in order to evaluate IDEA's performance as available energy changes. We also perform experiments with a randomly attenuated charging profile to simulate bad solar panel placement or obstacles to incident sunlight affecting the spatial distribution of collected energy.

For our MoteLab experiments we determine the system's ability to span periods without charging inputs. We use two sets of initial conditions based on the interaction between the charging data we collected and the capacity of the batteries deployed. If the solar panel is large enough and it is sufficiently sunny, the panel will provide considerable charging input and completely charge small batteries during the day so all nodes will begin the night with full batteries. If the solar panel is not large enough to completely charge the batteries nodes will begin the night with varying amounts of charge depending on their load rates during the day.

Energy tracking is done by IDEA using a software-only approach developed for the Pixie [52] project. The component captures state transitions and applies an energy consumption model for each state based on current consumption measured offline. The short lifetimes for some experiments are explained by the use of extremely small batteries, which were chosen to allow experiments to complete in reasonable amounts of time. We expect that application developers will want to use a battery size and charging technology suitable to allow their system to achieve a desired level of performance, and the improvements in energy efficiency possible using IDEA will allow smaller batteries or solar panels to be used, reducing the size and cost of the hardware package.

Experiments for the radio listening intervals use the "maximize time to first node death" energy objective function described in Section 2.2.2, and therefore we evaluate the network lifetime as the time at which the first node runs out of energy. Our distributed localization application illustrates the process of designing an effective energy objective function when the overall goal of the system is known.

2.3.2 Low-Power Listening Parameter Tuning

Low-power listening (LPL) enables radio duty-cycling without requiring nodes to arrange fixed transmission schedules [59]. It is well-suited for environments where network topologies and traffic patterns are highly variable, since these variations challenge duty-cycling techniques that assume *a priori* knowledge of traffic patterns.

When using LPL, nodes poll the radio channel at a fixed rate, listening for packets addressed to them. The radio is shut off when not polling or sending packets. To send a packet to another node, the sender must know that node's polling interval, and repeatedly send the packet with reduced MAC backoffs until either the packet is acknowledged, ending the packet train and indicating a successful transmission, or the length of the packet train exceeds the receiver's polling interval, at which point the transmission fails.

The choice of LPL polling rate at a given node affects the continuous energy drain required to periodically poll the channel as well as the cost to other nodes to communicate with the given node. Assuming we model the radio as drawing I_{listen} and $I_{transmit}$ mA of current in listen and transmit modes, respectively, then, given an interval between radio checks of γ sec, the current draw required to poll the channel is $\frac{1}{\gamma} \cdot t_{check} \cdot I_{listen}$, where t_{check} is the time the radio must remain on to detect channel activity. The cost to transmit a packet to a node using an LPL interval of γ is, on average, $\frac{\gamma}{2} \cdot I_{transmit}$. We can observe then that increasing γ or polling the channel less frequently *reduces* the current draw on the receiving node while *increasing* the communication cost on sending nodes.

On the CC2420 radio used in our experiments, $I_{listen} = 18.9\ mA$ and $I_{trasmit} = 17.4\ mA$. The radio can rapidly leave and return to a low-power state so t_{check} is short, on the order of 10 ms, allowing the continuous receive cost to be minimized. Adjusting LPL intervals offers a way of changing the energy consumption for communication between two nodes, and an opportunity for IDEA to tune the intervals to match the availability of energy within the network. To develop intuition about the tuning process, we consider a simple example where Node 1 is transmitting packets to Node 2. If Node 1 has a lot of energy while Node 2 has little, then Node 2 should poll the channel slowly and let Node 1 pay the high per-packet penalty. On the other hand, if Node 2 has a lot of energy while Node 1 has little, then Node 2 should poll the channel rapidly, increasing its own energy consumption but reducing the per-packet cost to Node 1.

IDEA allows us to build a component to tune the LPL parameters on each node adaptively. Our local state space is $s_n = \{s_n^5, s_n^6, \ldots, s_n^{10}\}$, where s_n^j corresponds to polling at intervals of 2^j on node n, so the smallest interval is 32 and the largest interval is 1024. For each state s_n^j, we construct the projected energy load vector $\bar{L}(s_n^j)$ out of two components: one measuring the receive cost to node n, the other measuring the transmission cost to other nodes to send to node n. The receive cost on node n, $\bar{r}_n$, has only a single component for node n, $r_n^n(s_n^j) = \frac{1024}{2^j} \cdot 0.010\text{sec} \cdot 18.9$ mA, where 0.010 second is the check interval and 18.9 mA is the radio receive current. The transmission cost to nodes sending to node n, $\bar{t}_n$, has components of the form $t_n^i(s_n^j) = \frac{1}{2} \cdot \frac{2^j}{1024}\text{sec} \cdot \delta(i,n) \cdot 17.4$ mA, where $\delta(i,n)$ is the rate at which node i is sending packets to node n, and 17.4 mA is the radio transmission current.

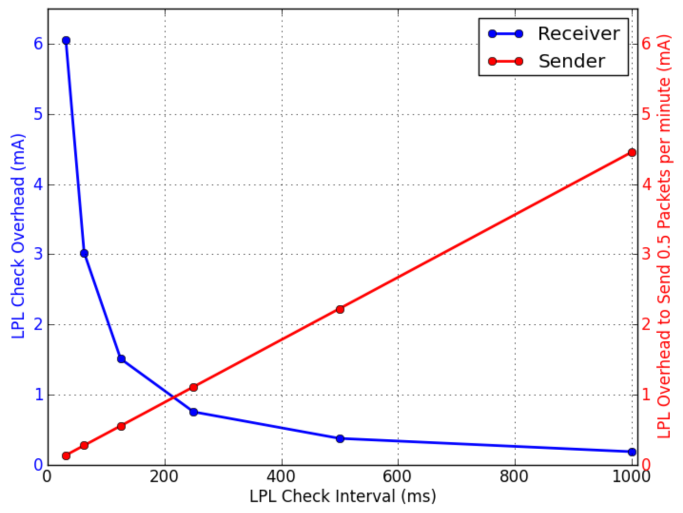


Figure 2.3: LPL overheads for sending and receiving packets. By lowering its LPL check interval, a node increases its own energy usage but decreases the amount of energy other nodes need to send packets to the node. Conversely, when a node increases its LPL check interval, its own energy usage is reduced, but it also increases the energy other nodes need to send packets to the node.

We construct the total energy load vector $\bar{L}_n(s_n^j)$ as the component-wise sum of $\bar{r}_n$ and $\bar{t}_n$, and pass this information to IDEA to evaluate each state. Figure 2.3 shows the LPL check overhead for receiving LPL packets and the LPL overhead to transmit packets at the data rates used in the experiments described in Section 2.3.2. From the figure it is easy to see how changing a node's LPL interval shifts the energy burden either to the node (by decreasing its LPL check interval) or from the node (by increasing its LPL check interval).

The tuning component intercepts outgoing transmissions, queries IDEA for the correct LPL interval to use for the given destination, and sets the packet's LPL interval accordingly. When the LPL tuning component switches states, it must propagate this information to nearby nodes that might be sending it data. We use the ability of IDEA to propagate component state to disseminate this information as described in Section 2.2.5. It is worth noting that this data, and all state information, is propagated by using a radio broadcast packet. The node does not necessarily know the LPL intervals of all other nodes that may be in broadcast range, so it must transmit the broadcast packet for the longest LPL check interval, which is 1 second for our experiments. To put this in perspective, this broadcast packet is roughly the equivalent of 16 minutes of sending data at an LPL interval of 250 ms at the data rates used in the experiments below. This also helps explain why the modeling component of IDEA, which helps to minimize the amount of updates that are needed, is important to keeping overheads down.

Changing the LPL interval also affects the total throughput possible over the link, which provides the component-specific measure of desirability, although the relationship is complicated by the ability of LPL to bunch transmissions to amortize the cost of awakening the receiver. For our evaluation we chose to set the tradeoff factor $\alpha = 1$ and optimize only for energy, since the throughput of the link was not a limiting factor at the data rates we tested.

IDEA is designed as a service, so it is up to the application to decide when to evaluate a new state. IDEA will select the state that best satisfies the weighted combination of the energy objective function and utility of the state, but it does not take into consideration any costs associated with changing to a new state. This can affect how often the application chooses to evaluate a new state. At one extreme, if there is no overhead for the application to change its state, then the cost of evaluating the state is just the cost of computing the energy objective function and utility for all of the possible states. For the energy objective functions and utilities we have experimented with, this cost is negligible so increasing the evaluation rate should not negatively affect the network.

However, there is almost always some cost associated with changing state. In the case of LPL tuning, a node's new LPL state must be shared with its neighbors, which is done with a broadcast message at the longest LPL interval in the system. For example, in the LPL experiments below, if IDEA suggests a node increases its LPL check interval from 250 ms to 500 ms to conserve energy, the cost to share the new LPL interval (assuming no errors in communication) will offset the savings for the first 46 seconds in the new state. This implies that how often IDEA evaluates state should be weighted against the cost of changing to a new state.

Experimental Results

We first look at the overhead of the IDEA LPL-tuning component as we vary the rate at which updates are performed in the system. IDEA can vary how often nodes evaluate their LPL intervals as well has how often to evaluate load and model parameters. A more frequent evaluation of LPL intervals allows the system to react more quickly to changes in the network with the potential for higher energy costs as more state changes may need to be propagated across the network. By the same token, more frequent evaluation of load and charge model parameters allow IDEA to react quickly to fluctuations in energy in the network, but may result in more energy consumed as new models must be propagated via the data sharing mechanism.

We begin by using TOSSIM to configure 20 nodes into a collection tree based on topology and link quality obtained from data collection experiments on Motelab. In this experiment each node sends messages to the sink (Node 118) once every 2 minutes. Figure 2.4 shows the topology used for this and all of the other LPL tuning experiments.

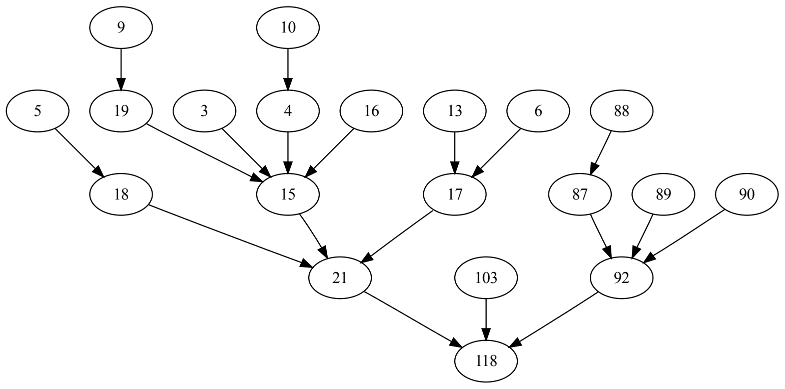


Figure 2.4: Topology used for LPL tuning experiments.

For this experiment, all nodes start out with the same battery level, and there is no incoming charge on the nodes (i.e., they are not hooked up to a solar panel). IDEA is using the "maximize time to first node death" energy objective function, and the experiment runs until the first node runs out of battery. We vary the update rate of IDEA (for both propagating new state information and evaluating new LPL intervals) from 5 to 50 minutes in 5-minute increments and compare the lifetime to an optimal solution.

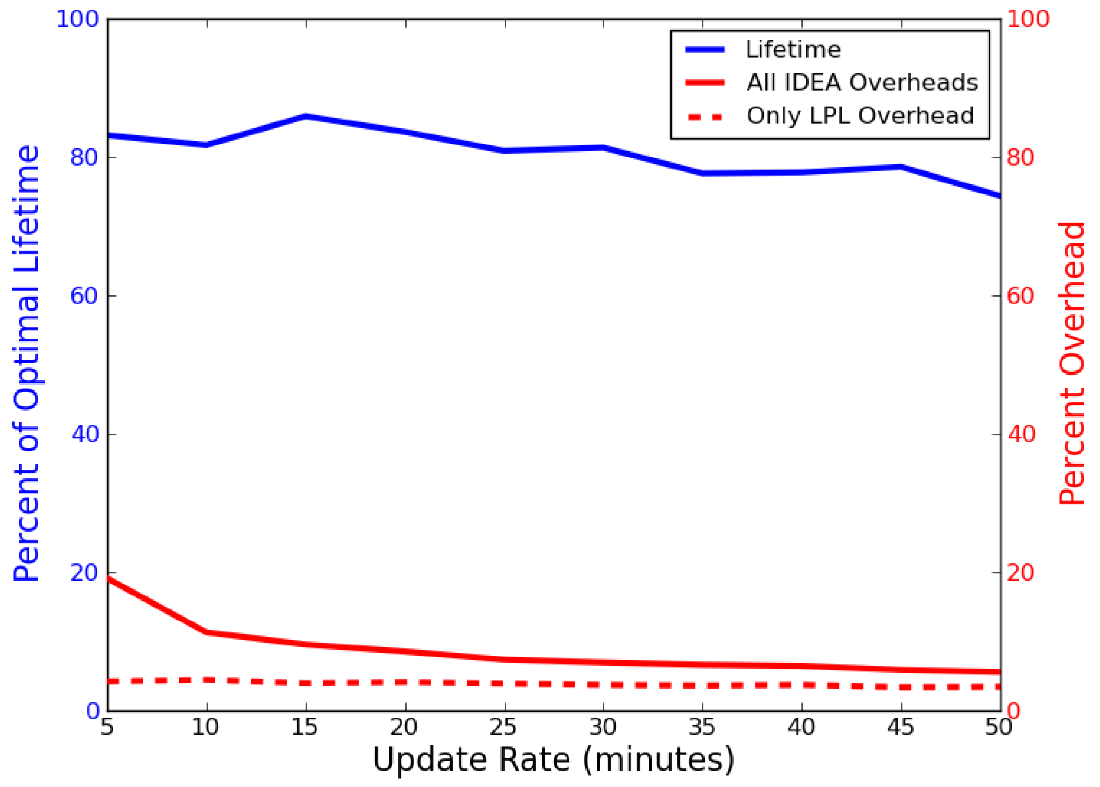


Figure 2.5: Optimality and overhead. IDEA consumes energy in order to propagate load, charge, and state information. For the LPL-tuning component the energy overhead is related to the rate at which we re-tune the local LPL interval, load model, and charge model. This plot shows both the IDEA overhead and the degree of optimality achieved as the update rate is varied.

We obtain an offline-optimal estimate of the lifetime performance by treating the problem as a multi-dimensional, multiple-choice knapsack problem and computing the solution offline with a linear program solver. In the optimal case, every node has the up-to-date load rate and current battery level for every node in the system, i.e., every node has the entire global network state. We assume in the optimal case that there is no overhead in sharing this information. The battery size was chosen such that in the optimal case the network lifetime was just under 14 hours, which allowed the TOSSIM simulations to be run in a reasonable length while still being able to understand how

IDEA performs.

Figure 2.5 shows the variation in lifetime, plotted as percent of the optimal solution, and the percent overhead used by IDEA overall, as well as the subset of IDEA energy used for tuning the LPL parameters as we vary both the LPL and load model evaluation rate.

Most of the gap between IDEA and the offline-optimal solution is due to the overhead of propagating state information and new LPL parameters. As we decrease the update rate, model parameters are shared less frequently and the network consumes less energy, causing the overhead to decrease. LPL tuning overhead remains relatively constant since the workload for this application is static and most evaluation periods do not produce a change in LPL intervals. For this application the lifetime curve shows the best results with an update rate of 15 minutes. At the left end of the curve with a rapid update rate the overheads associated with data sharing reduce the system's lifetime. At the right end of the curve the system is slower to find the optimal state and may spend some time with sub-optimal intervals, and the lifetime again suffers. Across the entire range, however, the achieved network lifetime remains above 74% of the optimal offline solution.

We can also use the optimal solution as a qualitative point of comparison. Figure 2.6 shows the differences between intervals picked by the IDEA-driven and optimal solutions in the case of 15-minute update intervals for the above experiment. IDEA chooses near-optimal intervals for Node 21, the energy bottleneck (within 1% of optimal) and the worst case, Node 118 (the sink), was still within 15% of optimal. Node 118 does the worst because as IDEA is sub-optimally lowering its LPL interval to assist Node 21, the energy bottleneck.

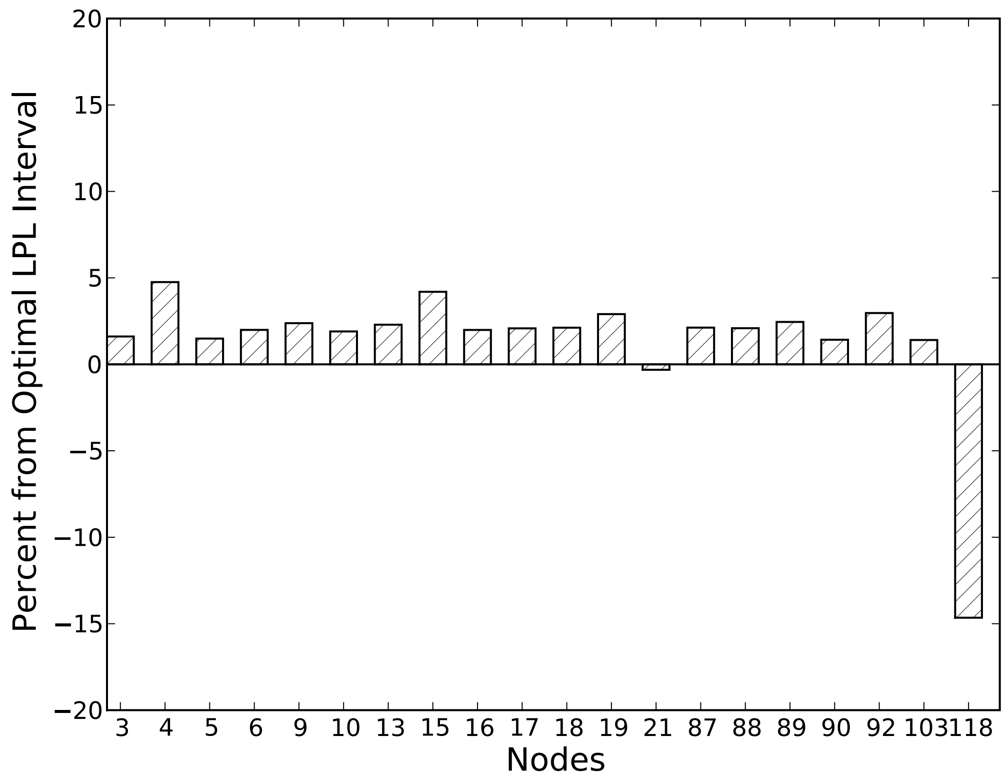


Figure 2.6: LPL interval comparison with optimal. To assess the degree to which the IDEA-driven approach finds a near-optimal global state, we plot the percent difference between the intervals chosen by the IDEA-tuned and offline optimal systems. The plot demonstrates that IDEA sets the LPL intervals of nodes similarly to the optimal solution and helps explain its performance.

For our next set of experiments we ran IDEA on a 20-node subset of Motelab with the same topology and settings as the previous experiments. As a point of comparison, we ran experiments using static intervals assigned a priori, with all nodes using the same LPL interval. We compared the results from all six LPL intervals IDEA could choose from and picked the one that performed the best. Note that this experimentation itself is a form of tuning and would be difficult to do beforehand. We ran one-hour MoteLab experiments and used each node's rate of energy consumption to compute a projected lifetime. Table 2.1 summarizes the results of experiments.

Initial Battery Levels	**Lifetime (hours)**		**Increase (%)**
	Static	**Idea**	
Uniform	4.6	5.6	22%
Random	2.8	3.0	7%

Table 2.1: LPL tuning performance on MoteLab. The table shows results for MoteLab experiments comparing the performance of the IDEA-driven LPL tuning component against the best static parameter solution. IDEA shows gains for both the case where all nodes start with the same battery level and randomly initialized battery states.

The table shows that the tuned LPL intervals produce improvements in projected lifetimes when compared with the best static interval under both non-charging scenarios discussed in Section 2.3.1. We observe an improvement of 22% for the case where nodes start with the same initial charge and 7% when random initial battery levels are used. This is in spite of the fact that the LPL tuning component produces significant overhead propagating new states early in the experiment as nodes are moving from their initial states into their IDEA-tuned intervals.

Table 2.2 summarizes results from experiments performed on TOSSIM that include solar-charging inputs discussed above. IDEA provides 5% and 13% performance improvements for cases in which all nodes see the same input charging profile and a 35% improvement in the case where charging inputs are randomly attenuated. This is due to the increased difference in battery levels due to the random attenuation, which creates more diversity in the amount of available charge. The table also shows numbers that indicate the best that IDEA can do when its overhead is artificially eliminated, showing that future work on improving the load and charge modeling and more efficient data sharing will continue to improve performance.

Solar Charging Pattern	**Lifetime (hours)**			**Increase (%)**
	Static	**Idea**	**No Overhead**	
Large Panel	22.7	23.8	24.0	5%
Small Panel	16.8	18.9	21.2	13%
Randomly Attenuated	13.8	18.6	20.4	35%

Table 2.2: LPL tuning performance with solar charging. This table displays results for TOSSIM experiments comparing IDEA-based LPL parameter tuning with the best static interval and an overhead-free version of IDEA. IDEA shows gains over the non-tuned approaches across a range of different solar-charging profiles. The % increase column reflects the improvement between the best static LPL interval and the IDEA-tuned intervals

2.3.3 Distributed Localization

This application illustrates how to use IDEA to control a system designed to perform acoustic source localization. Several previous systems have explored this application in different contexts, including urban sniper localization [73] and localizing animals based on mating calls [3]. Using IDEA, it is possible to carefully manage the energy load at each sensor node to prolong battery lifetime while maintaining high localization accuracy.

Acoustic source localization calculates the location of an acoustic source by collecting arrival times at several stations and performing a back-azimuth computation [63]. We assume a dense sensor network deployment, so that an acoustic event is detected by many sensors. We also assume that for each event, any set of four sensors that heard the event can correctly perform the localization to within the application's error tolerance.

A centralized approach to localization requires nodes to transmit data to a base station where the computation is performed. Because we assume that nodes cannot accurately compute an arrival time by considering only their own sampled data, they must transmit a sizable amount of data to the base station to implement the centralized strategy, with the bulk data transfer required producing a significant load on the nodes that heard the event as well as nodes required to route data. This approach also does not scale well as the size of the network increases.

To avoid the overheads of centralization, we want to perform the localization inside the network. However, the cost to transmit signals and perform the computation are still high, so it is important that localization be done in a way sensitive to the availability of energy within the network.

When an event occurs, the goal is to select a single *aggregator* node and three *signal provider* nodes from the set of nodes that detected the event. The signal providers will transmit a portion of the acoustic signal to the aggregator, which performs the localization computation using a time-of-arrival and angle-of-arrival computation [61]. For each event we expect multiple valid aggregator and signal provider sets to exist, each with its own energy consumption signature. We refer to a selection of four such nodes as a *localization plan*.

Nodes that heard the signal participate in a leader election process, seeded by the value of the IDEA energy objective function for each proposed localization plan. Each candidate aggregator computes the energy objective function for the localization plan or plans for which they are the aggregator. If more than three nodes within a single hop of an aggregator heard the event, then the aggregator will have multiple plans to consider. The aggregator chooses the local plan with the best score and broadcasts a message advertising that score, which is propagated to all nodes that heard the event. If the aggregator does not hear a broadcast with a better score, it assumes that it won the leader election and proceeds to perform the localization as planned.

Experimental Results

To evaluate the distributed localization application, we built a Python simulator, which improves significantly on TOSSIM performance at this scale and allowed rapid iteration and experimentation with different energy objective functions. Our simulator models acoustic event sources within the sensor network, each of which triggers a distributed localization operation. The energy overheads of communication, both the leader election process and the subsequent data transfer, are modeled in the simulator based on empirical measurements taken on our MoteLab testbed.

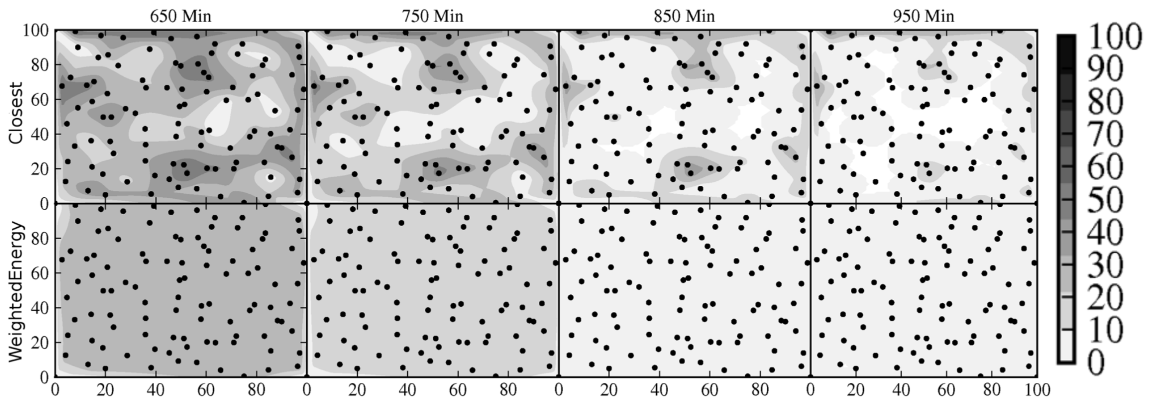


Figure 2.7: Energy density over time. Energy densities for the `Closest` heuristic and IDEA using the `WeightedEnergy` objective function are shown at four points in time. The event distribution is uniform. IDEA enables better load distribution, which leads to a longer application lifetime.

For these experiments we arranged 100 nodes into a 100 m by 100 m area, resulting in the placements shown in Figure 2.7. We simulate a sensing range equal to the communication range, each set to 20 m, and randomize the reliable transfer protocol bandwidth across each link to between 768 and 1280 bytes/sec, a feasible range based on results from data transfer protocols such as Flush [39] and Fetch [84]. Events are simulated using a uniform random distribution so that events have equal probability of occurring anywhere in the sensor field.

To evaluate network performance, we define the *capability* of the network as the percent of the last 100 operations that succeeded, where success is defined as localizing the event. We assume that the application requires that the network be able to localize 90% of events that occur, and design our energy objective functions with this in mind. We quote the system lifetime as the the 90% capability time, that is, the time at which the network's capability drops below 90%.

We experimented with several approaches to choosing a localization plan, one that does not use IDEA and three that do, using different energy objective functions:

1. **`Closest`:** produces a localization plan with the node closest to the event source as the aggregator and the next three closest nodes as signal providers. We assume a real solution would use an imperfect estimate of proximity such as total signal energy or signal-to-noise ratio, but for the simulations we use the known simulated event location to choose the closest nodes. `Closest` does not require energy state information and so could be implemented without IDEA. It is implemented as an example of a plausible non-energy-aware solution.

2. **`MaxEnergy`:** chooses the node with the most energy (that heard the event) as aggregator and the next three highest-energy nodes as signal providers.

3. **`TotalEnergy`:** chooses the localization plan that consumes the lowest amount of total energy summed across all nodes in the network.

4. **`WeightedEnergy`:** weights the total energy consumption using a weighted version of the cosine similarity index [71] to measure the degree to which the energy vector for the localization plan is a good "fit" given the current energy availability.

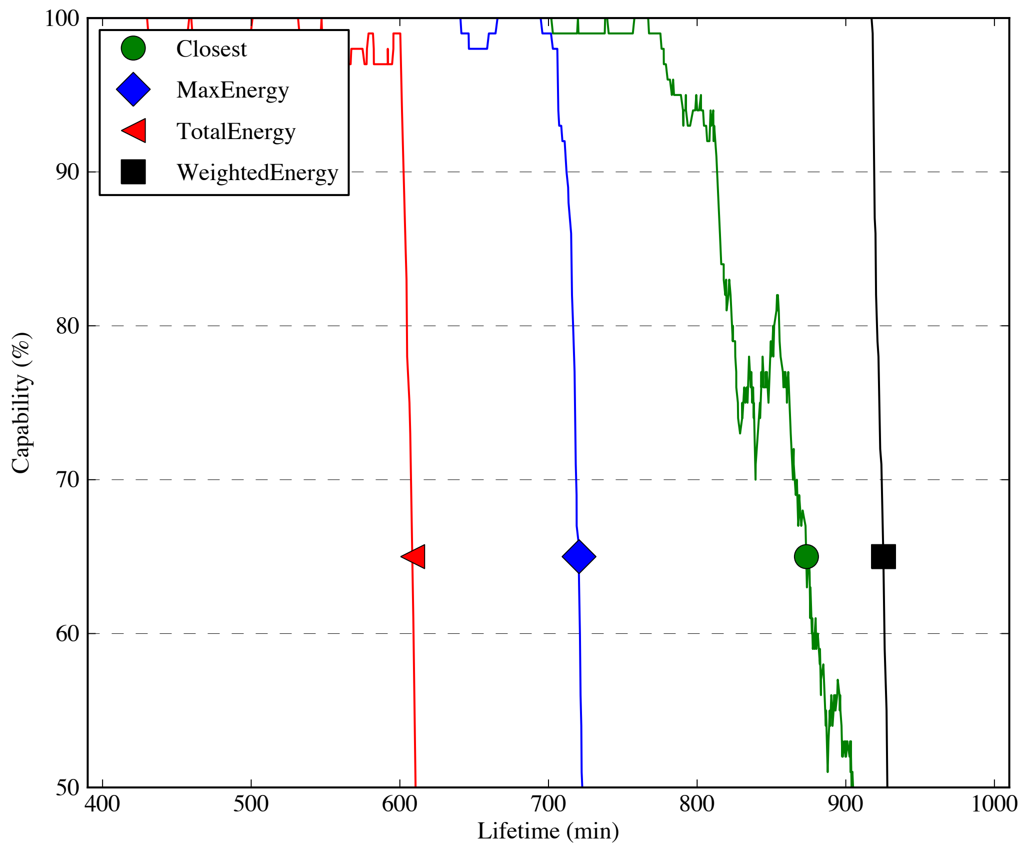


Figure 2.8: Performance of IDEA objective functions and heuristic. Simulation results are shown for the localization application. The graph compares the `Closest` heuristic, implemented without using IDEA, against three different IDEA objective functions: `MaxEnergy`, `TotalEnergy` and `WeightedEnergy`. The `WeightedEnergy` approach using IDEA outperforms the non-energy-aware approach while the other objective functions perform more poorly.

We began by experimenting with the `Closest`, `MaxEnergy` and `TotalEnergy` approaches. As Figure 2.8 shows, the `Closest` heuristic outperformed the two IDEA-based approaches. However, when examining the energy density plot shown in Figure 2.7 for the `Closest` heuristic, we could see that it led to concentrations of available energy on nodes at dense locations on the irregular grid. This is despite the uniform distribution of acoustic event sources, which one might expect to produce good energy load distribution without the need for tuning. After exploring several additional approaches, we found an energy objective function capable of producing extremely good load distribution, the `WeightedEnergy` approach described above. Figure 2.8 shows that it outperforms `Closest`, increasing the network's lifetime by 15%, while Figure 2.7 illustrates how it uses all the nodes' available energy. Our experience with the localization application illustrates the role of the proper energy objective function in enabling good application performance, and points to the increases in system lifetime possible through better energy distribution.

In this chapter we looked at IDEA, a system that enables effective network-wide energy decision making. IDEA automatically distributes the state of incoming and outgoing energy to a node, as well as the state of its battery, to nodes throughout the network. With input from the application on how application states affect energy usage , IDEA chooses states that best maximize energy optimization goals set by the application. We looked at two case studies and have shown, through simulation and testbed experiments, that IDEA can improve energy performance compared with solutions that operate with local information.

Chapter 3

Simbeeotic: A Testing Platform for Micro-Aerial Vehicle Swarms

MAV swarms are an emerging class of mobile sensing systems. These MAV swarms are connected to the environment through sensing and actuation and benefit from testing in a physical environment. However, developing programming models that can coordinate the resources of hundreds or thousands of MAVs requires simulation in order to easily test at scale. We need a testing environment that can span both of these domains, from testing individual MAVs with real flight dynamics to simulating a swarm of hundreds or thousands of MAVs. In this chapter we present Simbeeotic [36], a testing platform that facilitates the development of micro-aerial vehicle (MAV) swarms, an active area of our current research. Simbeeotic supports both pure simulation and hardware-in-the-loop (HWIL) experimentation with a radio controlled (RC) helicopter testbed. The simulator relies on modular software design principles and a commitment to deployment-time configuration to provide modeling flexibility and ease of use. It is highly extensible and is designed for repeated experimentation. With Simbeeotic we demonstrate that whole-system modeling is feasible for the MAV swarm domain. This chapter first explains why a new testing platform is needed for our research, then describes the design of Simbeeotic. Next, we detail hardware testbed and then evaluate both the simulator and testbed. Finally, we wrap up with a summary of the chapter. This work was done jointly with Bryan Kate, who developed the core simulator, while I developed the MAV testbed for evaluating hardware using HWIL simulation.

3.1 Motivation

Our research is focused on the emerging class of MAV swarms comprised of hundreds to thousands of insect-sized mechanical RoboBees [1]. In this subset of the MAV swarm space, the challenges MAV nodes face are similar to the challenges stationary sensor nodes face: storage and computation are limited, communication bandwidth is minimal, and energy is scarce. However, mobility places even greater demands on these limited resources, especially energy, as the limited thrust provided by these robots limit the weight of the energy source they can carry.

Autonomous mobility at this size and scale is a new domain for sensor networks. Current devices exist only as prototypes, capable of a few seconds of flight while tethered to a power source, but advances are pushing these MAVs closer to reality. The small size of these MAVs introduces new complexity for development and testing. Existing sensing technologies are too big or too heavy to be used at this scale, so new sensors need to be developed and tested, leading to the codevelopment of the hardware and software for these devices.

In these cases, simulation can be used to design and test control algorithms before the physical sensors are available for use. Simulation can also be used to rapidly evaluate new sensing technologies for inclusion to the platform. For both of these cases, accurate real-time flight dynamics are required for proper testing and evaluation. However, for evaluating MAV swarm algorithms, modeling accurate flight dynamics may not be important to the final results. In this case we can trade fidelity for speed, allowing simulations to scale to hundreds or thousands of MAVs with the fidelity only in the areas important to the experiment at hand. An effective testing platform should be able to span these two areas, from testing a small number of MAVs with real flight dynamics, to simulating large numbers of MAVs for effective swarm evaluation.

To enable MAV research that spans both of these domains, we have built Simbeeotic, a simulator capable of modeling thousands of MAVs, tightly coupled with a physical MAV testbed for testing with real flight dynamics. As we describe in Section 3.1.2, existing solutions are inadequate for our testing needs, failing in one or more of our core requirements: scalability, completeness, variable fidelity, or staged deployment. The next section talks about these core requirements in more detail.

3.1.1 Core Requirements

The core requirements of our MAV swarm simulator are similar to those of other simulators [46] and are defined as follows:

- **Scalability**: The simulator must be able to simulate thousands of MAVs in a single simulation. Scale of deployment is an important aspect of swarm research. Without the ability to study algorithms at true swarm scale, some of the hard problems will be missed.
- **Completeness**: Simulations should model as much of the problem domain as possible. Though research may be conducted on a subset of swarm design (e.g., flight control or networking),

it is advantageous to construct a holistic view of the problem in which complex interactions are revealed. For MAV swarms, this means modeling *actuation*, *sensing*, and *communication* for each application.

- **Variable Fidelity**: The desire to improve the accuracy of models is often at odds with simulation performance (scalability in this case). Users should be free to construct models with the appropriate level of fidelity to capture the subtleties of their problem. For example, researchers working on emergent algorithms may not require realistic flight control loops, whereas those working on controls will require accurate sensor and flight dynamics models but may not be concerned with network protocols. Using the same simulator, these researchers can work to improve the modeling of their domain while retaining the ability to combine their efforts and simulate the system as a whole.

- **Staged Deployment** No matter how detailed, simulation cannot completely capture every situation that will be encountered in the real world. While the ultimate goal is to deploy a swarm of MAVs, building hardware can be expensive and time consuming. The simulator can facilitate the development of control software and inform the hardware design process by providing a staged deployment feature, allowing prototype hardware to respond to both real and simulated inputs.

3.1.2 Why Other Simulators Could not Meet our Needs

The MAV swarm domain intersects with other research areas, including biologically-inspired algorithms, robotics, and sensor networks. There are high-fidelity simulators that exist in each of these communities; prior to implementing our own simulator, we investigated the possibility of using these tools. In general we were unable to find a simulator that satisfied our completeness and staged deployment requirement. We considered combining multiple simulators to satisfy this goal but determined that performance would suffer due to the high fidelity of some of the tools. Each simulator uses considerable machine resources to model its own domain for thousands of agents, making our scalability goal untenable with this approach. Finally, we considered the engineering cost of repurposing multiple simulators to be too high, given that these tools are written in a number of languages and are not uniformly maintained.

The first set of tools we consider come from the multiagent systems and swarm intelligence communities. These simulators are appealing because they can generally model thousands of agents at once. Swarm [32] and MASON [53] are two such tools. The main drawback of these simulators is that the they do not faithfully model the environment and actuation, opting for cell-based or 2D continuous worlds. For both Swarm and MASON, a significant amount of effort would be put into modeling a 3D, physics-based world that is accurate enough to support the staged deployment requirement. MASON provides a built-in 3D space (known as a field in MASON-speak) but leaves manipulation of objects in the field (e.g., kinematics, collision detection) to the modeler. Breve [41] is a discrete event simulator with an embedded physics engine. Unfortunately, models are written in a domain-specific language called Steve (there is limited support for Python), which hinders adoption and limits the number of existing math and science packages available to modelers.

The robotics community has long used simulators as design tools, since building hardware is often expensive and time consuming. Similar to our situation, the hardware and software are often codesigned, driving the need for accurate modeling of the physical environment. Thus, the strength of robot simulators is generally in modeling the interaction of the robot with the environment (e.g., actuation and sensing). Two commonly used tools are Webots [57] and Player-Stage [25] [79]. Webots models the environment as a 3D continuous space and has physics-based sensor models. It is an excellent teaching tool with support for many commercial robot platforms, but it fails to meet our scalability requirements. In addition, its commercial nature does not allow for arbitrary modification, as would likely be the case for modeling communication networks and bridging with our testbed. Player-Stage consists of a robot driver interface, Player, and a simulated environment, Stage. Player is used in a client/server fashion to control robot and sensor hardware. Stage is used to simulate robots in a virtual environment but exports a Player interface so that code can be migrated to a hardware platform. Stage is a 2.5D simulator that scales to handle hundreds of robots in real time for realistic workloads and thousands of robots for simple workloads. Its key limitation as mentioned by the authors [79] is that it is a first-order geometric simulator that does not model acceleration or momentum. Our approach to MAV swarm simulation requires a more comprehensive treatment of vehicle dynamics.

The Robot Operating System (ROS) is a collection of hardware drivers, algorithms, and tools for building robotic applications [69]. ROS users compose agent behaviors from a large set of open source packages that provide functionality for data acquisition and processing, planning, and locomotion. For the most part, ROS is a complementary technology to Simbeeotic. It is primarily used to construct a fully functioning software stack that can be deployed on one or more robots. There are packages that integrate ROS with simulators (including Player-Stage) to execute a robot in a virtual world, but these packages are insufficient for our needs due to shortcomings mentioned above.

The construction of the GRASP Micro UAV Testbed [56] is similar to Simbeeotic in that an off-board computer remotely controls the vehicles, relying on accurate position and orientation information from a motion capture system. One difference between the two testbeds is fidelity. The researchers using the GRASP testbed are interested in vehicle control, so the simulation includes a dynamics model and accounts for aerodynamic effects. Though we have performed a system identification on our helicopters and constructed a dynamics model, our efforts in simulation have focused on modeling larger swarms with lower fidelity vehicle movements. If researchers are interested in the aggregate behavior of a large swarm, foregoing the simulation of control loops can significantly improve simulation scalability.

The wireless networking and sensor network communities have invested heavily in simulation tools. GloMoSim [92] and ns-3 [62] are widely adopted simulators that model the OSI seven-layer architecture. While they do an excellent job of implementing RF propagation, radio models, and network protocols, these tools are singularly focused on networking. A significant effort would be needed to model actuation and sensing to meet our completeness requirement. Rather, our approach is to start with a physical simulation and add networking fidelity as needed. This strategy allows us to selectively integrate the parts of these tools that are useful in our domain.

TOSSIM [45] and EmStar [26] are two popular wireless sensor network simulators. TOSSIM takes the completeness and bridging requirements to an extreme by providing a virtual environment in which the embedded mote software (running TinyOS) is executed. Though the whole-system approach is appealing, TOSSIM restricts users to writing applications in TinyOS. We borrow the idea of staged deployment from EmStar, which allows virtual models (e.g., radios) to be replaced by physical hardware in a testbed. Our staged deployment goal is derived from a desire to iterate on software and hardware designs using virtualized representations prior to building a deployable system. We do not consider EmStar a viable starting point for a MAV swarm simulator because the software is no longer maintained.

Implementing a new testing platform has several advantages. We can ensure that our requirements are satisfied and make design decisions that suit our needs. Our approach also allows us to evolve the fidelity of each subdomain (e.g., actuation, sensing, communication) as more accuracy is needed. However, we do not want to reinvent what is considered state-of-the-art in each domain. Whenever possible, we leverage open source tools and learn from existing models to avoid duplication of effort.

3.2 Simulator Design

At its core, Simbeeotic is a general purpose discrete event simulator tightly coupled with a physical MAV testbed. A simulation execution consists of one or more *models* that schedule *events* to occur at a future point in time. The virtual time of the simulation is moved forward by an executive that retrieves the next event from a queue of causally ordered pending events and passes it to the intended recipient for processing. In effect, time passes in between events – the events themselves represent discrete points in time. Since we are interested in modeling the MAV swarm domain, Simbeeotic builds upon the basic discrete event mechanism to provide convenient abstractions for building MAV swarm simulations, such as a virtual environment, robotic platforms, sensors, and radios.

The Simbeeotic simulator is written in the Java programming language. Java was chosen for a number of reasons. First, it is widely understood among our team and easily learned by neophytes. Second, it is for the most part a cross-platform language. We have confidence that Simbeeotic can be compiled on or distributed in binary form with little effort to the most popular (and some esoteric) operating systems. Third, there exists a large repository of high quality, open source libraries that can be leveraged by our modelers.

At present, Simbeeotic consists of 13,387 lines of Java code in 148 classes and 506 lines of XML schema. Of this code base, 48% makes up the core (including the simulation executor, modeling interfaces, base classes, and common model implementations), 26% is for testbed integration, 13% is example code, 6% defines tools that generate random enclosed environments (such as mine shafts and office buildings), 6% is for visualization components, and 1% is for the main entry point. This codebase builds atop a collection of open source libraries that provide support for the simulator. The major libraries used are shown in Table 3.1

Library	**Version**	**Description**
jbullet	20101010	Physics engine
guice	2.0	Dependency injection for configuration
commons-math	2.1	Math library for statistics
jopt-simple	3.1	Command line parsing
jfreechart	1.0.12	Graph visualization
protobuf-java	2.3.0	Serialization for testbed integration
jama	1.0.2	Linear algebra package
j3d-core	1.5.2	3D Visualization
j3d-utils	1.5.2	3D Visualization
vecmath	1.5.2	Vector library
log4j	1.2.13	Logging library
junit	3.8.1	Unit testing

Table 3.1: Open source libraries utilized in Simbeeotic.

3.2.1 Architecture

We constructed Simbeeotic to fulfill the requirements established in 3.1 (scalability, completeness, variable fidelity, and staged deployment). In addition, we are careful to provide repeatability and promote ease of use and extensibility throughout our design. Simulation repeatability is of utmost importance; experiments must be reproducible given identical inputs. As such, our framework provides seeded random number streams to models and causally orders scheduled events using a set of deterministically generated tiebreak fields. We also have a simple XML-based configuration system for setting simulation parameters.

Figure 3.1 shows an overview of the simulation architecture (lefthand side) and a partial class diagram of the modeling abstractions (righthand side). All objects in Simbeeotic are represented by models. Models can represent physical objects such as MAVs and obstacles in the environment. Abstract models can also be used to represent things such as weather or wind (which do not have a physical entity associated with them).

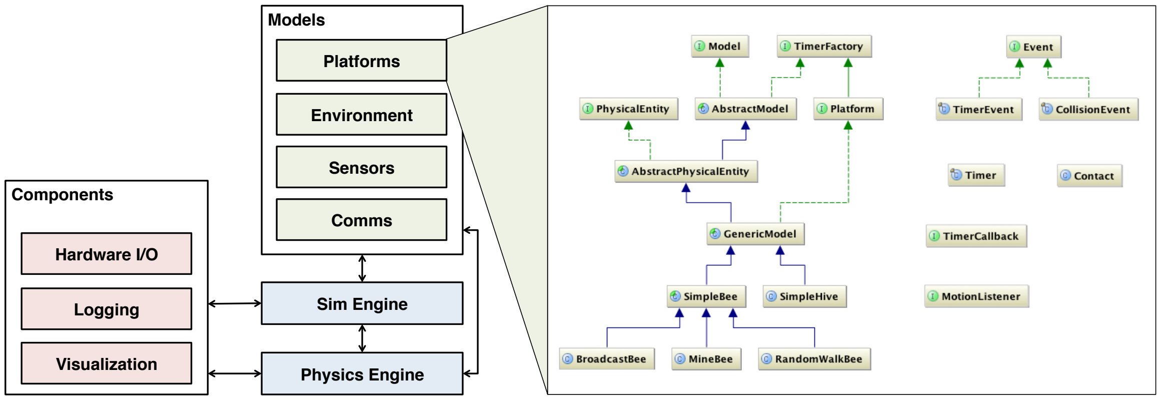


Figure 3.1: The Simbeeotic architecture. Domain models are plugged into a discrete event simulation engine. The kinematic state of models with physical presence is managed by an integrated physics engine. Several levels of abstraction in the model layer provide flexibility and convenience to modelers. The simulation architecture can be augmented by user-supplied plugin components.

The heart of the simulator is the simulation engine, which manages the discrete event queue and dispatches events to models, pushing virtual time forward. Prior to the commencement of the simulation, the simulation engine populates the virtual world from a supplied configuration file, called a scenario, and initializes all of the models by calling a model-specific initialization routine. The simulation engine is also responsible for answering queries about the model population. It provides an API for locating models based on type or ID.

The model layer sits on top of the simulation engine. The majority of user-supplied code uses model layer interfaces to implement features of the target domain. Simbeeotic employs a layered strategy to provide extension points within the model space. The layered approach API is one way that Simbeeotic fulfills the variable fidelity design goal outlined in Section 3.1. Users model particular applications by specializing model classes to match objects (either physical or virtual) in their scenario. The model class hierarchy is intentionally detailed so that an application designer can select the right layer at which to specialize.

At the very bottom are the `Model` and `Event` interfaces. All models implement the `Model` interface, but few do so directly. The `AbstractModel` base class provides a default implementation that introduces other useful mechanisms, such as a seeded random number generator and a timer abstraction. We have committed to a continuous, 3D representation of space in Simbeeotic. The `PhysicalEntity` interface is defined to standardize the representation of a physical object (its size, shape, and mass), the information that can be queried about its kinematic state, and how its state can be manipulated (by applying forces, torques, and impulses). While it is possible for users to directly implement the `PhysicalEntity` interface, there exists a base class, `AbstractPhysicalEntity`, that implements the interface by delegating to a rigid body physics engine (described below).

The next level of abstraction, the `GenericModel` class, treats the established physical body as a robotic platform, allowing equipment (e.g., sensors and radios) to be associated with the platform. The attached equipment models do not implement the `PhysicalEntity` interface.

Rather, they are granted access to the host platform's physical presence and are attached using a body-relative position and orientation (e.g., antenna position and pointing direction). It is possible for a modeler to develop a new robotic platform by extending `GenericModel`, attaching sensors and radios, and defining custom agent logic using the timer mechanism. We introduce a final abstraction layer with the `SimpleBee` base class. This class provides a simple actuation API (e.g., `turn`, `set-LinearVelocity`, `setHovering`) that makes the simulation more accessible to modelers who do not require high-fidelity actuation modeling. The `SimpleBee` carries out the actuation commands with an internal kinematic update loop, translating the desired motion into the appropriate forces and torques and applying them to the body.

Modelers do not generally use the event-scheduling mechanism directly. Rather, they implement agent logic using the `Timer` mechanism introduced by the `AbstractModel` class. Timers are a familiar abstraction that most modelers are comfortable using. Timers are scheduled periodically or for single use. The modeler provides a custom callback, which is fired when the timer expires (Figure 3.2). Timers are implemented with a self-scheduled `TimerEvent` under the covers.

We also discourage the use of events for inter-model communication. In-domain communication mechanisms (e.g., the radios) should be used for the sake of realism and consistency. These mechanisms expose a familiar API to the modeler and are implemented internally with events.

In addition to building models in the target domain, users can extend the functionality of the simulator by providing *components*. Component implementations can interact with the simulation engine and physics engine directly, or with models by scheduling events. Two components that have received heavy use in our research are the 3D visualization component and a component used to communicate with our MAV testbed (discussed in Section 3.3). Component instances are created

```
compass = getSensor("compass", Compass.class);


// a timer that takes a compass reading periodically
Timer compTimer = createTimer(new TimerCallback() {


    public void fire(SimTime time) {


        float h = compass.getHeading();
        ...
    }
}, 0, TimeUnit.SECONDS,
   sensorTimeout, TimeUnit.MILLISECONDS);
```

Figure 3.2: Code from a model initialization routine. This code demonstrates how to query for attached equipment and schedule a periodic timer (starting immediately and firing every `sensorTimeout` ms).

prior to model initialization and can operate in a separate thread of execution. This way it is possible to provide asynchronous I/O components, such as buffered loggers.

The final piece of the Simbeeotic architecture is the physics engine. As described above, the physics engine is used as the backing implementation for the `PhysicalEntity` interface, which is implemented by all models with a physical presence in the simulation. The physics engine we use is JBullet [35], a six-degrees-of-freedom (6DoF) rigid-body physics engine written in pure Java. JBullet provides a number of features that are useful in modeling MAV swarms at high fidelity:

- **Rigid Bodies**: The MAV platforms and the virtual environment are composed from simple shapes (e.g., box, sphere, cone) and complex geometries (e.g., convex hull, triangular mesh).

- **Dynamics Modeling**: The kinematic state of every object is maintained by integrating the forces and torques (e.g., rotor thrust, gravity, wind) applied to physical entities over time.
- **3D Continuous Collision Detection**: Physical interactions between objects, such as environmental manipulation by a robot or bump sensors, are easily modeled.
- **Ray Tracing**: Used primarily to implement sensors, such as range finders and optical flow.

When a descendant of `AbstractPhysicalEntity` is initialized, a representative rigid body is registered with the physics engine. The information associated with the body includes its size, shape, mass, inertial properties, initial position, and orientation. As the rigid body is manipulated over time, its kinematic state is updated. During the course of an event, a model can query the kinematic state of an `AbstractPhysicalEntity`, which delegates the request to the rigid body. The simulation engine invokes JBullet in between events to push the dynamics simulation forward to the time of the next discrete event. We modified the JBullet library to break out of the dynamics simulation if a collision is detected during an update. In this case the simulation engine checks a registry of interested collision listeners (registered by the models). If found, an event is generated to inform the listener (e.g., a bump sensor) of the collision. If no listener is interested in the collision, the dynamics simulation is resumed.

JBullet integration enables high-fidelity actuation and sensor models, but this fidelity comes at a cost. Most of the routines in JBullet execute sequentially; therefore the performance of the simulator is explicitly coupled with the size of the swarm and complexity of the environment (i.e., the number of states that must be integrated and bodies checked for collisions). Section

3.4.1 evaluates the effect of environmental complexity and swarm size on simulation performance. Our conclusion is that the performance tradeoff is acceptable given the corresponding increase in fidelity.

3.2.2 MAV Domain Models

Modelers contribute new functionality to the community codebase using the extension points described above. Simbeeotic constructs the virtual world from the rigid bodies defined by the physical entities and object definitions supplied in a world configuration file. The configuration file contains definitions of obstacles, structures, and environmental features to be inserted into the environment. Weather is modeled in the simulation by an abstract model (one without physical presence) that can be queried for the current weather state with respect to location. High fidelity models can simulate the effects of weather on themselves (e.g., by applying a wind force to a physical entity) or other models using the information provided (e.g., wind speed and direction).

Most of the built-in sensors provided by Simbeeotic are based on information provided by the physics engine. At present, interfaces and default implementations exist for inertial (accelerometer, gyroscope, optical flow), navigational (position, compass), and environmental (camera, range, bump) sensors. The inertial and navigational sensors use the kinematic state of the host platform, whereas the environmental sensors (and the optical flow sensor) use advanced features of the physics engine, such as ray tracing and collision detection. All of the default sensor models can be configured to produce inaccurate readings from truth state using a Gaussian noise model. Modelers can introduce new implementations of sensors that closely reflect the accuracy, precision, and error profile of real hardware.

Modeling RF communication is something that is done well by community standard simu-

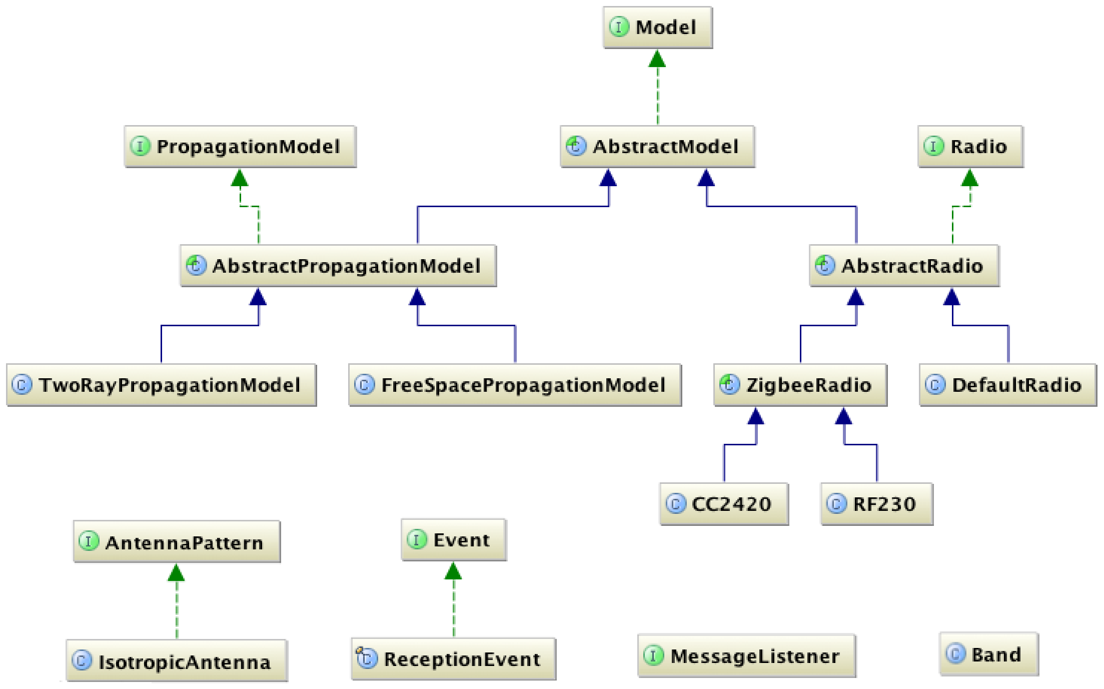


Figure 3.3: A class diagram for the RF communications package in Simbeeotic. The abstraction defines a physical layer packet-driven radio. Interfaces are indicated with a Ⓘ and classes are indicated with a Ⓒ .

lators [62]. As such, the philosophy for RF in Simbeeotic has been to implement the smallest portion of the OSI seven-layer architecture possible and evolve the fidelity of the models (or integrate another simulator) when the need arises. Figure 3.3 shows a class diagram for the communications package in Simbeeotic. We implement a simple physical layer abstraction that includes the radio, antenna, and path loss model interfaces. Modelers are free to implement layers on top of the packet-driven radio abstraction.

3.2.3 Software Engineering Tricks

Simbeeotic relies on two features of the Java programming language, reflection and run-time annotation processing, to provide convenient interfaces to the end user. Though not necessary to achieve our original design goals, this section describes how these two features enhance the usability of our implementation.

Both reflection and runtime annotation processing are used to provide a flexible configuration system in Simbeeotic. Our design treats models and components as plugins to the simulator and configures them through dependency injection. Specifically, we use Java reflection to construct scenarios from an arbitrary number and type of models. We define an XML schema for our scenario configuration file that allows users to specify the fully qualified name of Java classes they wish to load and execute. When the scenario file is parsed, the user-supplied type is checked for compliance (that it implements the required interfaces), and the specified number of instances are instantiated, registered with the simulation engine, and initialized. Other simulation frameworks, such as Player-Stage, allow for an arbitrary number of user-defined scenarios to be loaded based on a configuration file. However, users are restricted to a pre-existing set of known model types. By using reflection, any class or component on the Java classpath is eligible for inclusion in the simulation.

The second part of configuration is parameterization. As a convenience to the user, we allow for a set of key-value pairs to be associated with each model or component definition in the scenario file. We use an open source dependency injection library, Google Guice [27], to configure the newly instantiated objects using the supplied parameters. After an object is instantiated, Guice inspects the instantiated class for injection sites (annotated fields or setters). To identify parameters for a model or component, users simply annotate their classes with the `@Inject` annotation, which can be attached to fields and methods. Guice uses the type of the field or method argument to match the injection site with a supplied configuration parameter. An additional `@Named` annotation is used to disambiguate between parameters of the same type. Figure 3.4 depicts the usage of these

```
@Inject
private double maxVel;
private boolean useRadio = false;


@Inject(optional=true)
public void setUseRadio(@Named("use-radio")
                        boolean use) {
    this.useRadio = use;
}
```

Figure 3.4: Simbeeotic code demonstrating the use of the `@Inject` annotation for model parameterization.

annotations on fields and methods to prepare a model for parameter injection. With the ability to load arbitrary model and component implementations and inject parameters, many decisions regarding scenario construction can be pushed to deployment time.

3.3 MAV Testbed

In addition to the Simbeeotic simulator, we maintain an indoor MAV testbed for conducting small-scale experiments primarily used to test sensing and control with realistic flight dynamics. Despite our best effort, the simulator cannot form a complete representation of the real world. Our approach is to develop new systems and algorithms at scale in simulation and experiment with smaller deployments in the testbed where they can be tested under realistic conditions.

As RoboBees are not ready to be deployed at this time, we chose the E-flite Blade mCX2 [17] RC helicopter as the aerial platform for the testbed. The mCX2 is a low-cost ($100), off-the-shelf MAV that is quite limited in its capabilities. It has a payload of up to 5 grams and a flight time on the order of 7 minutes. It carries a proprietary control board that processes RC commands and stabilizes flight with an embedded gyroscope (yaw axis only). As a stock system it has no other processors, sensors, or radios. There are several advantages to using this platform. First, building a swarm from these helicopters is not prohibitively expensive. Second, the small size (20 cm in length) allows multiple helicopters to be flown in our 7 x 6 meter laboratory space. The helicopter serves as a convincing prototype for the intended target of our research, insect-scale MAVs, in terms of flight time and capability. One disadvantage of the mCX2 is that it is a toy, not a research robot. Processing, sensing, and communication hardware must be added to make the vehicle into an autonomous swarm agent. However, the lack of on-board sensing allows for easy integration with the sensing and control directly from the simulator.

3.3.1 Remote Control

The helicopter testbed is instrumented with a Vicon [80] motion capture system. The Vicon sensors are capable of capturing the position and orientation of an object (adorned with reflective markers) in our testbed with sub-millimeter accuracy at 100 Hz. This information is made available to programs that remotely control the helicopters. We achieve computer control by disassembling the supplied joystick and removing the radio transmitter daughterboard. Though the wireless protocol between the transmitter and helicopter is proprietary, the transmitter board is driven by a serial interface. The input signal to the transmitter is composed of four 10-bit RC command values; yaw, pitch, roll, and throttle. We connect the transmitter to a PC with a USB-serial cable and allow the RC commands to be generated programmatically.

A testbed gateway machine mediates access to the observed helicopter state and RC transmitters. For helicopter state (measured by Vicon) the gateway provides a publish-subscribe mechanism for pushing updates to interested clients. Clients receive updates via messages that are serialized using Google Protocol Buffers [28]. The information in each update includes the Vicon frame number (essentially a timestamp) along with the object's identifier, position, orientation, and an occlusion flag (indicating that Vicon has lost track of the object in this frame). The gateway also provides a server for controlling each helicopter, which accepts <`yaw, pitch, roll, throttle`> command tuples. The server ensures that at most one client is connected to each helicopter and sends the latest RC commands to the transmitter at the required 50 Hz. Clients communicate with the testbed gateway machine over a Gigabit Ethernet LAN.

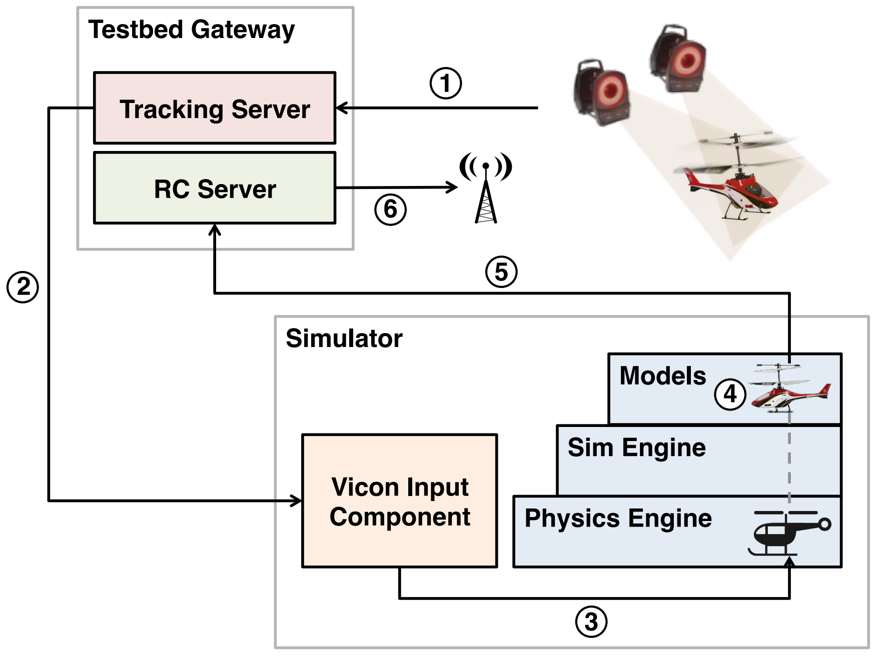


Figure 3.5: The HWIL cycle in Simbeeotic. Vicon cameras track the position and orientation of a helicopter and push frames to a tracking server (1), which pushes updates (2) to registered clients. A Vicon input component in Simbeeotic receives the update and overrides the kinematic state (3) of the corresponding object in the physics engine. When the ghost model executes an event (4), it has the most recent state of the helicopter. If a command is issued, it is sent to the RC command server (5) where it is dispatched by the RF transmitter (6) to the helicopter.

3.3.2 Simulator Integration

It is possible to write a standalone program that communicates with the testbed gateway to control the helicopters in the testbed. However, we realize that writing such programs would result in significant overlap with the Simbeeotic simulator, given that virtual sensor outputs would need to be constructed from the absolute position and orientation information provided by Vicon. We chose instead to integrate the helicopter testbed with the simulator, allowing the modeler to leverage the virtual sensor implementations that already exist and conduct hybrid experiments with simulated and real MAVs. We refer to this operating mode as hardware-in-the-loop (HWIL) simulation. This technique is similar to the staged deployment mechanism in EmStar [26], which allows a simulated network to be transparently backed by real hardware.

We accomplish the testbed integration, depicted in Figure 3.5, by introducing *ghost models* in the simulator for physical objects that are tracked by Vicon. The ghost models implement the same `PhysicalEntity` interface as the simulated models, so interaction between the two is unchanged. The difference is that the ghost model's kinematic state is derived from the Vicon input, not the physics engine. However, the virtual sensors and other models that interrogate the virtual environment rely on the presence of an object in the physics engine for every physical entity. To fulfill this requirement, we simply create an object with the correct size, shape, and mass in the physics engine and periodically override its kinematic state with the information from Vicon. We introduce a new component that is responsible for connecting to the testbed gateway and receiving state updates. The simulation allows for the internal state of tracked objects in the physics engine to be updated prior to executing each event. Thus, whenever an event is executed, the state of all physical entities in the simulation is correctly represented by the physics engine. Some minor modifications to JBullet were required to allow the state to be set and to integrate the new state forward correctly in between Vicon updates.

Sending RC commands is similar. Upon initialization, each ghost MAV model opens a socket that connects to the testbed gateway. The RC commands are fed over the wire to the transmitter, which controls the helicopter in turn. The effects of the commands are witnessed by the Vicon, and the loop is closed.

The Simbeeotic simulator processes events as quickly as possible when executing a pure simulation. However, the simulator must make an effort to run in realtime when hardware is attached. We make the assumption that the wallclock time necessary to execute an event is less than the virtual time between the current event and its immediate successor. If this assumption holds, then it is trivial to maintain a soft realtime schedule by delaying the processing of an event until a corresponding system time has passed. When event processing violates this assumption, events are processed as quickly as possible to catch up. This approach works in practice, though it compels modelers to keep events simple (arguably a good thing) and avoid scheduling simultaneous events.

3.3.3 Hardware in the Loop

The testbed integration allows one to fly real MAVs using virtual sensors in a simulated environment. This arrangement allows for the transformation of a laboratory space into an arbitrarily complex proving ground with virtual obstacles and features. Sensors can be easily tested in different positions and with varying amounts of sensor error to test the limits of control and navigation algorithms. For these cases, pure simulation is not sufficient since the physics engine cannot always capture subtleties such as aerodynamic ground effects and servo actuation error, so the HWIL tests can aid in the iterative design process by observing these phenomena early on using staged deployment.

We have modeled several sensors for navigation: optical flow, accelerometer, gyroscope, and compass. We also have developed a flower sensor, which indicates when a flower is within range and in the sensor's field of view. In addition, we have also implemented a position and orientation sensor, which gives three-dimensional position and orientation, respectively, of the MAV in space. These two sensors are in essence ideal sensors, since they come directly from the physics engine in simulation or the Vicon motion capture system in the testbed.

The first sensors evaluated in the testbed were the position and orientation sensors. As these are the most accurate sensors, this is a natural place to start for evaluating the control algorithms of the MAV. Control is performed by four proportional-integral-derivative (PID) controllers [4], one each for thrust, yaw, pitch, and roll. PID controllers were chosen because they can be implemented with minimal computing resources and are being used for control on the actual RoboBees themselves. Using the four PID controllers and the position and orientation sensors, we were able to develop several navigation primitives: takeoff, hover, move to a location, and land. In addition to the navigation primitives, we implemented a simple obstacle avoidance system using potential fields [38] and have tested it by flying up to five MAVs at one time, avoiding both virtual obstacles and each other while using the flower sensor to search for virtual flowers to pollinate.

While developing these algorithms, the testbed's tight integration with the simulator proved very useful. We were able to use Simbeeotic's 3D visualization tools to track the MAVs as they navigated through the virtual world. When an MAV crashed or ran into an obstacle, Simbeeotic's logging facilities made it easier to go back and diagnose what happened.

Using HWIL we were able to determine some key properties of our system that would have been difficult to determine otherwise. First, we were able to determine the limits of safe flight for our MAVs. In open space the MAVs can travel at 2 meters per second. At this speed it takes about 1 second to slow to a hover, which gives us a good idea at what density we can deploy MAVs to a given area if we wish to travel at maximum speed. Also, while evaluating our flower sensor, we found that slowing down from 0.25 to 0.5 meters per second gave the most reliable detections. We use these numbers when modeling our swarm simulations.

Because the main focus of our research is resource coordination for swarms, we have only currently evaluated the position, orientation, and flower sensors using HWIL. However, by designing and testing control algorithms around these sensors, we have shown that HWIL is a viable platform for evaluating sensing and control algorithms with real flight dynamics.

Our HWIL arrangement has a couple of disadvantages. First, we are coupled to Vicon. Without a very accurate measurement of position and orientation, we would not be able to write sensors that convey the truth about the physical object. As we are tied to the motion capture system, we cannot fly outdoors. This is not a severe limitation at the moment, but our laboratory can only accommodate a handful of physical helicopters. However, Simbeeotic is modular in its design, and it is possible to replace Vicon with some other form of motion tracking. To demonstrate Simbeeotic outside of our lab [82], we replaced the Vicon motion tracking system with a Microsoft Kinect [58] sensor. The range and accuracy of the motion tracking was greatly reduced under this setup, but it was still possible to have a single MAV fly and hover in a space of about two cubic meters.

Another potential disadvantage is that the control software for the helicopter is running in the simulator on a PC-class system. There is a risk of developing software that uses far too many resources for the eventual platform to handle. A TOSSIM-like approach to whole-system simulation may be needed to keep the modelers honest. Finally, our current setup does not allow for any processing or sensing to occur on the physical helicopter. This is why we refer to the remote-control HWIL solution as staged deployment. It is merely a stepping stone to truly autonomous MAVs. Chapter 6 discusses the possibility of extending the HWIL approach to communication hardware and how Simbeeotic can facilitate a move toward autonomous MAVs.

There are multiple sources of latency in the HWIL loop described above, including capture and processing time for Vicon frames, the transmission of MAV tracks to Simbeeotic over the LAN, processing events in Simbeeotic, sending RC commands to the testbed gateway over the LAN, and broadcasting the RC commands via the wireless link. If needed, the tracking server, RC server, and simulation could be co-located, eliminating the LAN. Our experiments have shown that the round-trip loop latency in the testbed does not cause control instability or a substantive delay in MAV reaction time. The latency introduced by the processing loop is absorbed by the relatively slow update rate of the RC helicopters (50 Hz). In addition, if a command is delayed there is not a noticeable impact on the position and orientation of the MAV. Unlike the GRASP testbed, which focuses on fast, complicated maneuvers, our MAVs typically move at a rate of 0.25–2.0 $\frac{m}{s}$. At this velocity a 20 ms latency might result in a positional drift of a few centimeters. Since the HWIL loop latency is not an observable hindrance to our experiments, little effort has been put into characterizing and minimizing the delay in our testbed.

3.4 Evaluation

We have used Simbeeotic for almost three years (two years with HWIL) to conduct research on MAV swarms. In this section we evaluate the performance of the simulator and present two applications that use Simbeeotic to explore the MAV swarm domain.

3.4.1 Simulation Performance

Since Simbeeotic is used in daily experimentation, we can state from experience that the tool meets our needs. However, it is beneficial to know the limits of the simulator, how modeler and user decisions can affect performance, and how the tool might be improved with future work. We evaluate the performance of the simulator and our ability to meet our scalability objectives based on three challenges:

- **Environment Complexity**: The number of objects defined in the environment (e.g., obstacles, structures) determines how much collision checking is necessary during each physics update. Complicated scenarios can slow down the simulator.
- **Swarm Size**: As more MAVs are introduced there is more work to be done by the physics engine to maintain the kinematic states of the moving objects. In addition, each new MAV represents an additional workload (events to process) to execute the agent's logic.
- **Model Complexity**: Higher fidelity agent logic is likely to affect performance since complex events take longer to simulate.

Defining a single performance goal for the simulator is difficult given that modelers can construct scenarios that contain models of varying fidelity and execute in arbitrarily complex environments. Our motivation for constructing the simulator was to study large swarms of less-capable MAVs. Thus, we focus on a performance goal of simulating one thousand MAVs executing a typical workload in soft realtime or better. The scalability goal is lowered for HWIL scenarios to ensure that RC commands are issued as close as possible to a realtime schedule so helicopters do not crash. Our experiments show that Simbeeotic is capable of simulating thousands of MAVs executing a typical workload and hundreds of MAVs executing a complex workload in soft realtime. As with other discrete event simulators, Simbeeotic is capable of simulating faster than realtime when there is no testbed hardware in the loop.

We define a typical MAV workload to consist of a random walk (10 Hz kinematic update rate) and a periodic sensor reading (1 Hz compass). This models the performance we expect from our target MAV. In all of the following experiments, the MAVs operate for 100 virtual seconds and start from random locations within 20m of the origin. The operation time chosen allows enough work be done to evaluate Simbeeotic while still allowing the simulations to be run in a reasonable length of time. The starting distance from the origin was selected to give us a wide range of MAV densities as the number of MAVs are increased. We instrument Simbeeotic to record the amount of wallclock time necessary to simulate the physics (in between events) and run the agent logic (the events themselves). All measurements are taken on a 2.2 GHz quad-core laptop with 8 GB of RAM using the HotSpot JVM version 1.6.0_26.

We begin with an experiment that addresses the environmental complexity challenge. We measure the overhead of collision detection by simulating a small swarm (32 MAVs) executing the workload defined above. A variable number of static obstacles are introduced into the environment at each iteration of the experiment. As the number of obstacles grows, we expect the collision detection routines to take more time. Performing naive collision detection is $O(n^2)$ in time. Fortunately, JBullet employs more sophisticated collision detection routines that reduce the number of compared objects. Since the kinematic state of a static object in JBullet is not integrated forward at each time step, we can attribute any increase in the physics simulation time to increased collision checks (and likely some added overhead). Further, we expect this increase to be linear with respect to the number of obstacles (as opposed to quadratic) because two statically placed objects are not checked for collisions. Thus, the only collisions being checked are between the MAVs and the obstacles. The results in Figure 3.6 show that the amount of time to execute the events (agent logic) is constant through the course of the experiment. This is to be expected, as the swarm size does not change and, for this experiment, our event logic ignores the collision notification (i.e., colliding MAVs will pass right through each other). However, the overall time spent in the physics simulator increases linearly with the number of objects introduced. MAV swarm modelers must be informed that environmental complexity, not just swarm size, can have a significant impact on the performance of the simulation.

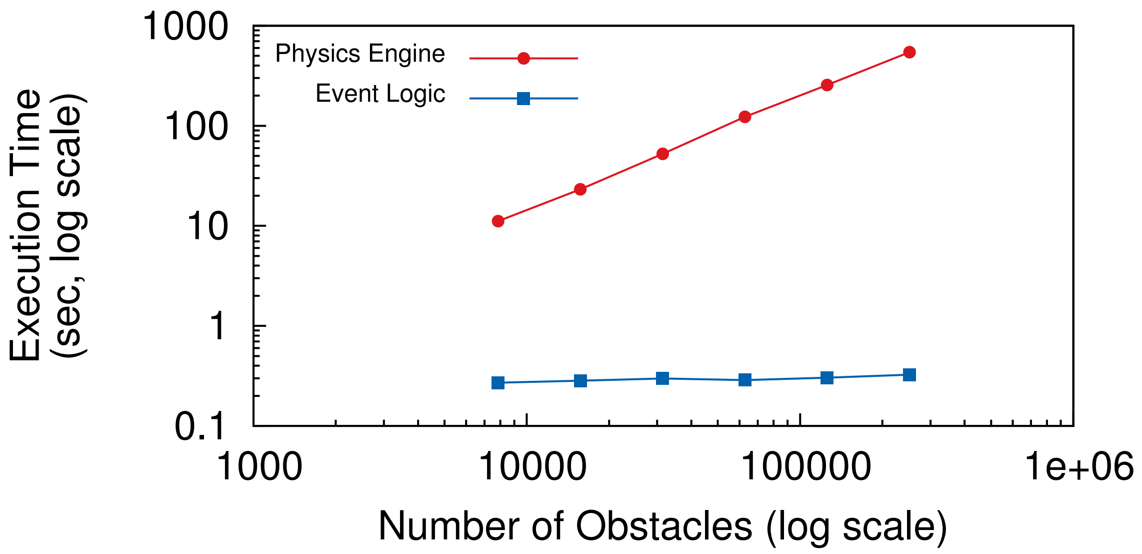


Figure 3.6: The overhead of collision detection in Simbeeotic. A fixed number of MAVs are simulated with a varying number of static obstacles. The amount of time to execute the event logic is constant. The number of required collision checks between MAVs and obstacles (and the time spent in the physics engine) grows linearly as obstacles are introduced.

The next experiment characterizes the scalability of the simulator with respect to swarm size. With each iteration we vary the number of MAVs deployed into a constant environment (no obstacles). The MAVs execute the workload defined above. We expect increased collision checks (between MAVs) and a linear increase in the time needed to update the kinematic states of the MAVs. Similar to the static obstacle experiment, JBullet's collision detection routines reduces the actual number of collision checks that are needed, improving performance. Figure 3.7 shows the results of this experiment. The simulation scales roughly linearly as the swarm size is increased. The number of events (and the corresponding event execution time) scales linearly as well. Using this workload, it is possible to simulate 3,074 MAVs in soft realtime. These scalability results are comparable to the performance of Player-Stage using a similarly defined "simple" workload [79].

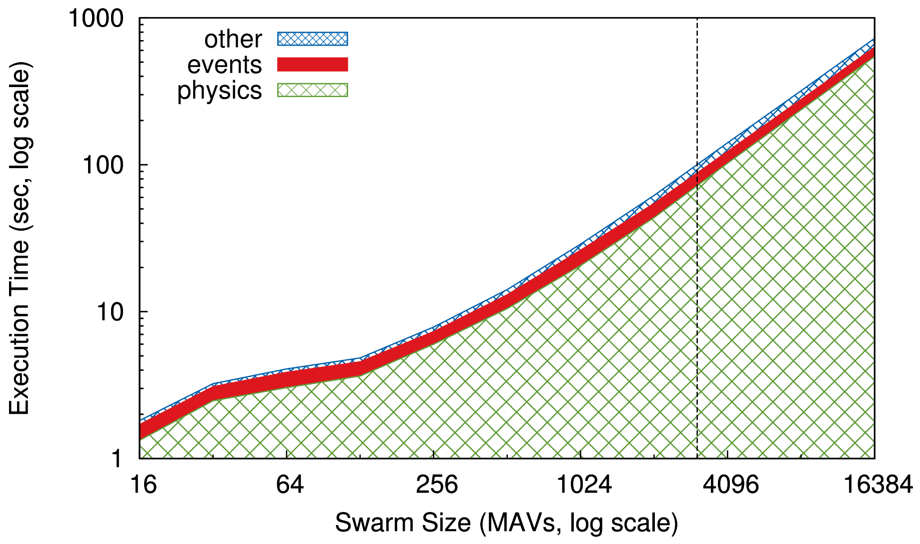


Figure 3.7: Scalability of Simbeeotic with respect to swarm size. The number of events to process and kinematic states to integrate increases linearly with swarm size. The corresponding event and physics execution times reflect this increase. The dashed vertical line indicates the point above which soft realtime cannot be achieved with this workload (3,074 MAVs).

We address the final performance challenge, model complexity, by introducing an additional element to the workload – each MAV broadcasts a radio message at 1 Hz. The result of this addition is a significant increase in the event execution time. The increase in event time has two main causes, event complexity and message explosion. The former refers to the nontrivial amount of work that must be done to send each packet.

The propagation model considers every other radio-equipped model as a potential recipient and performs path loss calculations between the two radios. This includes determining the antenna positions and orientations, extracting the gains from the antenna patterns, and computing the signal strength at the recipient. Though there is a cutoff distance in the path loss model, this optimization is not useful in the scenario under test because the MAVs are closely spaced.

Message explosion refers to the number of receive events that will be generated as a result of each packet transmitted. It is possible that n^2 events are generated each second in the simulation. In this case, some events are not generated due to low signal strength at the recipient. Despite the relative simplicity of the receive event processing, the sheer number that need to be processed can add significant overhead. The results of this experiment are shown in Figure 3.8. The overhead of creating and enqueuing these events is likely the source of the increase in the "other" category. With this workload, we can simulate 550 MAVs in soft realtime.

We set out to create a complete simulator for the MAV swarm domain. These experiments demonstrate that Simbeeotic meets our scalability goals for typical workloads (thousands of MAVs in soft realtime). They also reaffirm the premise that environment complexity and model fidelity can significantly affect performance. Chapter 6 discusses potential modifications to improve the scalability of the simulator.

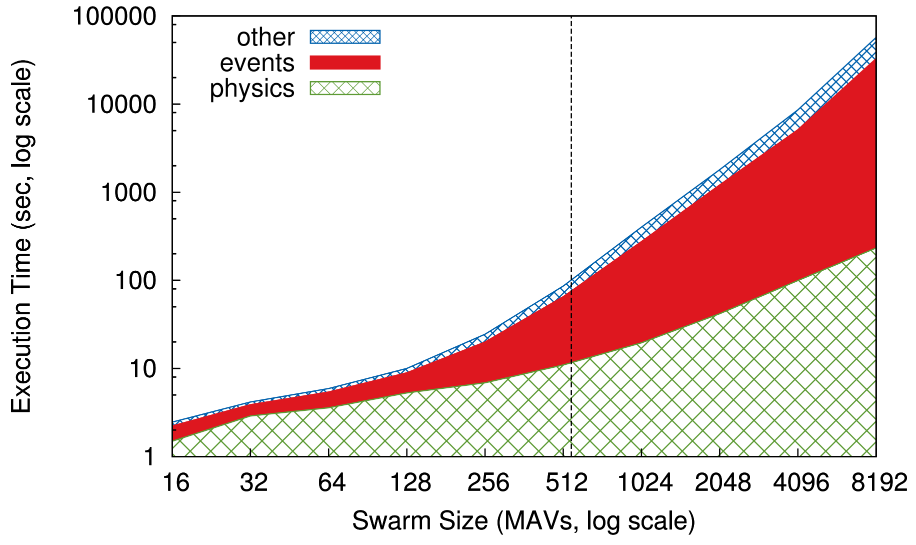
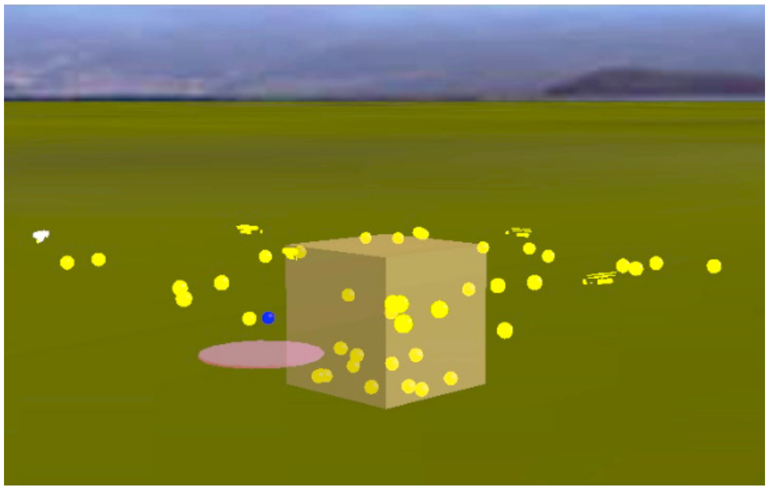


Figure 3.8: Simbeeotic performance with radio broadcasts. The simulation runtime does not increase linearly for the broadcast scenario. A nontrivial amount of work is undertaken for each radio transmission event, which may also generate reception events on all other MAVs. The event execution time dominates this scenario as the swarm scales. The dashed vertical line indicates the point above which soft realtime cannot be achieved with this workload (550 MAVs).

3.4.2 Example Scenarios

We describe two MAV swarm scenarios that we have simulated using Simbeeotic. The main goal of the first scenario is coverage. The MAV swarm is deployed to search a space for features of interest (e.g., flowers) and manipulate the environment where the features are located (e.g., chemical sampling, pollination). There are many possible solutions to the swarm coordination problem, including static task assignment, cooperative planning, and emergent behavior. We employ Karma [11], a system that coordinates the actions of the swarm from a centralized location called the hive. We discretize the world into cells and dispatch MAVs from the hive to perform a specific task until they are low on energy, at which point they return to recharge. A planner at the hive analyzes the results of the trip (the information collected) and determines which cells require more attention. Figure 3.9 shows a snapshot of our swarm management system executing a search and survey scenario using 45 virtual MAVs and 5 testbed helicopters. The top picture shows a Simbeeotic visualization of the virtual world, while the bottom picture shows the helicopters flying under PC control. This example demonstrates that Simbeeotic has adequate modeling fidelity in actuation and sensing to fly real hardware, and that the staged deployment goals are satisfied.


(a) Virtual World

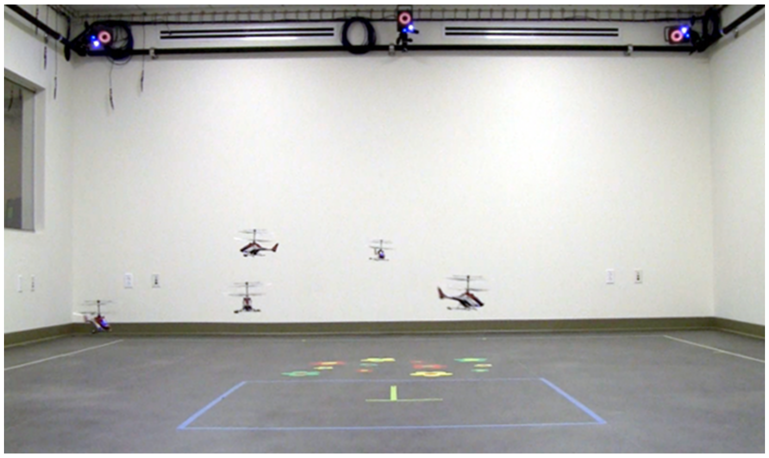

(b) Helicopter Testbed

Figure 3.9: A HWIL deployment of a MAV swarm. Five testbed MAVs are deployed alongside 45 simulated MAVs to search a space for flowers. The circle in the virtual world represents a flower patch (also visible in the testbed floor), and the box at the center denotes the MAV hive.

The second scenario explores the possibility of using RF beacons embedded in the environment as navigational aids for flying MAVs. Figure 3.10 shows an overhead trace of MAVs using a biased random walk algorithm in a gradient field [15] to navigate along two preferred paths. The MAVs and beacons are equipped with virtual CC2420 radios and isotropic antennas. The two-ray RF propagation model is used to calculate path loss. The MAVs use the value and signal strength of beacon packets to determine the direction of travel in the gradient. This example demonstrates one way that RF communication can be used in a MAV swarm.

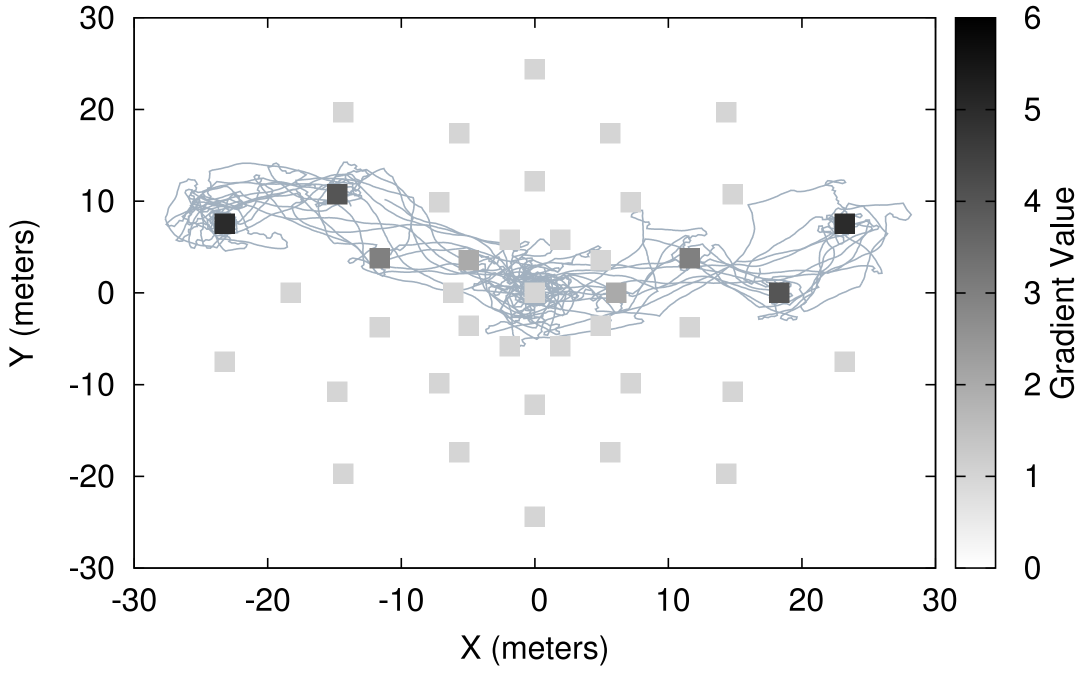


Figure 3.10: An overhead trace of five simulated MAVs navigating through the environment with the assistance of a gradient field provided by RF beacons (square dots). The gradient in this case specifies two paths away from the center. The MAVs use the value and the signal strength of beacon packets as input to a biased random walk (chemotaxis) algorithm. The MAVs are successful in traveling between the hive and the edge of the gradient field along the two paths.

3.5 Summary

MAV swarms are an emerging class of mobile sensor systems with strong ties to the robotics, sensor networking, and swarm intelligence communities. This chapter presents Simbeeotic, a simulation environment and testbed for MAV swarms to support research effort in this area. Simbeeotic is designed to be flexible and easy to use. The domain modeling interfaces are designed to cover a complete view of the application space, including actuation, sensing, and communication. We show that Simbeeotic is capable of simulating MAV swarms at scale, and demonstrate its usefulness in exploring new concepts with real flight dynamics. Simbeeotic is available as open source at `http://github.com/RoboBees/simbeeotic`. Our next chapter uses Simbeeotic to develop and evaluate Karma, our programming and coordination framework for MAV swarms.

Chapter 4

Karma: A Framework for Coordinating Micro-Aerial Vehicle Swarms

In this chapter we turn our attention to the unique set of challenges that face insect-sized MAVs. Actuation dominates the weight and power budgets for these devices, keeping sensing and control to the bare minimum. MAVs are required to move around in and to interact with an unpredictable environment, and MAVs will often fail in the field. This can be mitigated by adding redundancy to the swarm, but as the size of the swarm increases, it becomes harder to reason about the swarm as a whole, and the complexity of coordination increases. Furthermore, the extreme resource limitations of this MAV platform restrict the complexity of the programs that can be executed on the MAVs.

To address these problems, we present Karma, a centralized system architecture that coordinates MAV swarm resources with simple programming abstractions. Karma applications are composed of simple sensing and actuation commands allocated by a centralized controller. This decision simplifies the hardware and software complexity of individual MAVs and allow the application developer to focus on the tasks to perform, while Karma handles the complexity of resource coordination and allocation.

This chapter describes the architecture and implementation of Karma. Throughout the chapter we use a running example of an alfalfa crop monitoring and pollination application to show the ease of developing Karma applications and to demonstrate how Karma coordinates MAVs swarm resources. We evaluate Karma with simulation and testbed experiments and demonstrate how applications in Karma run on limited resources, are robust to individual MAV failure, and adapt to changes in the environment.

4.1 Architecture

The Karma system architecture is based on a *hive-drone model*, in which individual MAVs, called *drones*, perform simple sensing and manipulation tasks required to fulfill the goals of the swarm. To simplify programming, drones do not communicate with each other in the field and operate without precise knowledge of their location, relying on proprioception or periodic external localization for navigation. These restrictions simplify drone programs and allows swarm-scale behavior that is coordinated by a centralized *hive*, which contains sufficient sensing, computation, and storage capabilities to manage the swarm. With the majority of the computational burden pushed to the hive, the application programmer can focus on implementing the correct behaviors for the application (what to do) while the system reasons about allocation and coordination of the resources to carry out those behaviors (where and when to do it).

To achieve this decoupling, the Karma programming model allows the application programmer to compose programs from simple MAV-level behaviors by relating the behaviors that produce information to the ones that consume it. This model allows for an easy and flexible composition of programs and enables Karma to reason about MAV coordination efficiently.

4.1.1 Motivating Application: Alfalfa Crop Monitoring and Pollination

Throughout this chapter we use an example application to illustrate the design and operation of our system. The application, alfalfa crop monitoring and pollination [14], represents a typical application of MAV swarms in that it relies on both information gathering and micromanipulation behaviors in a relatively static environment. Alfalfa is an important food crop for cattle and requires an external pollinator (e.g., bees) to produce seeds. In recent years, colony collapse disorder [75] has devastated honeybee populations and jeopardized the cultivation of important crops. A swarm of insect-scale MAVs is well-suited to performing this type of pollination.

In addition to pollination, alfalfa crops require periodic monitoring for pests and disease. These tasks need to be performed at least three times a week and are normally done with visual spot checks. We envision a full service application that not only pollinates the crop when it is in bloom, but monitors the crop throughout the growing season.

To meet these requirements, the application consists of three periodic behaviors: searching for pests, searching for diseases, and looking for flowers in bloom. Pest infestation is typically detected by inspecting the leaves of the crop for damage caused by feeding insects. Diseases can be detected by looking at the color of the leaves, which turn greenish-white or brown in the presence of disease. If pests or diseases are found, the application notifies farmers so that they can treat the infected area. If the flowers are in bloom, the system should start a one-time pollination behavior. Together, the application has four behaviors in total. We will use this application to describe our design and evaluate our system.

4.1.2 Hive-Drone Model

A key challenge in designing swarm applications is to efficiently allocate resources to perform a desired task, such as crop pollination. Deciding what resources to allocate and where to allocate them is a complex function of available resources, conditions in the environment, deployed resources, and the desired goals of the application. All these conditions change dynamically over time, making offline static solutions inadequate.

The separation of concerns between the behaviors that run on the MAV and where and when these behaviors are carried out is a driving influence in the design of Karma. We propose a *hive-drone model*, which moves the resource coordination complexity to a centralized computer, allowing the system to have an up-do-date and global view of the network. In this model, MAVs are stationed at the *hive*, which has a physical presence in the environment and the capability to recharge MAV batteries. The hive computer determines how MAV resources be allocated to accomplish the swarm objectives and dispatches MAVs as *drones* to execute specific behaviors in the environment. This system design is coupled with a programming model that allows the application programmer to specify the desired swarm behaviors as sequences of sensing, sensor processing, and actuation commands without concern for coordination.

In Karma, the complexity of coordination and fault tolerance becomes the responsibility of the system, not the application programmer. We minimize the program complexity of individual MAVs by eliminating in-field communication and restricting programs to a simple set of commands. Therefore, by design, the application programmer is granted the freedom to describe how a MAV behaves when it is dispatched but has no need to explicitly coordinate its actions in the field. While these restrictions prohibit the expression of some programs, they allow us to accomplish our original goal of presenting a simple swarm interface to the user. Our approach is analogous to the philosophy of the MapReduce programming model [13]: we provide a simple, powerful abstraction to the user with the caveat that not all computations can be expressed using the abstraction. However, if the invariants of the model are followed, the system will handle the inherent complexity of coordinating the massively parallel operation.

Further, the hive-drone model fits well with the target hardware. In addition to the processing and sensing limitations, there is a rather severe restriction on flight-time. Current estimates of insect-scale MAV flight times suggest that a drone could operate for 5–10 minutes before its energy source is depleted. If the drone must return to the hive to recharge after a short period of time, it makes sense to perform coordination centrally at the hive. There are few scenarios that would necessitate in-field communication given the flight time restrictions and inherent burden of complexity.

The benefits of the hive-drone model can be summarized as follows:

- **Simplify MAV programming**: The drones need not coordinate among themselves, make tasking decisions, or deal with MAV loss. Thus, their software complexity is reduced.
- **Better decision making:** The centralized hive is more informed than an individual drone would be, making local greedy decisions based on partial information. The overhead for sharing information in the field is eliminated. The hive has the advantage of collective intelligence and more computation to better allocate resources.

In any sensing paradigm there will be a delay from the time an event of interest occurs in the environment to when the end user is notified of the event. We call this delay the *information* *latency* in the system. In traditional sensor network deployments, this latency may be on the order of a few seconds, depending on the duty cycle of the sensor and the properties of the network that propagates the information to a base station. One drawback of the hive-drone model is that it introduces an additional latency because drones retain information collected in the field until they return to the hive rather than routing it through a network. It is up to the user to decide if this latency (on the order of minutes) is tolerable given the nature of his or her application. Section 4.2.3 discusses optimizations that can be made in the system to reduce this latency at runtime.

To facilitate the description of our system and the discussion of the hive-drone model, we introduce the following definitions that will be used throughout the chapter.

- **Sortie**: One round trip from the hive to the area of intrest and back to the hive, in which a drone executes a single behavior.
- **Behavior**: A sequence of sensing, sensor processing, and actuation commands that are followed by a drone on a sortie. For example, a disease-detection behavior for the alfalfa application consists of performing a random walk to search for alfalfa leaves, acquiring an image with a simple image sensor, checking for diseases with a color matching algorithm, and recording the location if a disease was found.
- **Application**: A composition of low-level drone behaviors and high level-goals that is submitted by a user for execution on the swarm.

4.1.3 Spatial Decomposition

Given the inherent spatial nature of swarm applications, it is necessary to coordinate the allocation of swarm resources throughout the target space. At a high level, the user deploying the swarm application has some notion of how it should be distributed. For example, a farmer may desire uniform coverage of the pest monitoring behavior, but want targeted execution of the pollination behavior. The Karma system provides an abstraction that establishes a shared spatial context between the hive and the application behaviors. At the time of deployment, the target area in which the swarm will operate is divided into *regions*. The spatial decomposition may be influenced by application parameters (e.g., the size of the field and the desired sampling resolution) but is ultimately controlled by the hive. Behaviors are written in a location agnostic manner so that they can be applied to any region in the target space. However, they are given access to a location service at runtime that provides the current region for accounting purposes.

The choice to decompose space into regions benefits our system in multiple ways. First, it transforms the area of deployment from a continuous space into discrete regions, making it easier to reason about MAV allocation. Second, it aligns the localization primitives available to the drone behaviors with the likely capabilities of the MAV platform. Given the extreme limitations on computation and sensing, it is unlikely that the MAVs will have access to high-resolution location services in the field. The MAVs will rely on a combination of proprioception and exteroception (e.g., odometry using inertial sensors and a polarized light compass [43]) to navigate in the field. It may also be possible to externally localize the MAVs from the hive using RF triangulation or harmonic radar [66]. This information could be used to update the MAVs in flight or correlate sensor readings with a location when the drones return. We assume that it is possible to localize MAVs to the resolution of a region through a combination of these techniques, though we do not solve this problem directly.

Finally, it is not necessary for the regions in the spatial abstraction to be defined using a Cartesian coordinate system. The abstraction will work just as well if the regions are defined topologically or as nodes in a graph. For example, it may be possible to embed beacons into a target area and allow the drones to localize to the region that is defined by the closest beacon signal. Though it is an exciting prospect, we do not explore the use of non-Cartesian decompositions in this work.

4.1.4 Data Model

As the hive is the place where drones return to recharge, it is a natural location to store the information that is collected in the field. For this purpose, the hive maintains a key-value repository called the *Datastore*. Updates to this data structure are asynchronous, occurring when drones return from a sortie. The value for each key is structured as a log, appending data and metadata describing the time and location at which it was collected by the drone. Thus, the Datastore can be queried both temporally and spatially (at the resolution of a region). The information collected in the Datastore is used by the hive to track the progress of the application and make resource allocation decisions.

Figure 4.1 is an illustration of the hive-drone data model. A drone flies out with a blank local store (called a *scratchpad*) that is populated as it executes a behavior. Upon its return, the information it collected is uploaded to the hive Datastore. Thus, the Datastore at the hive has a partial view of the environment at any given time, which is dependent on the information brought back by drones that have completed their sorties.

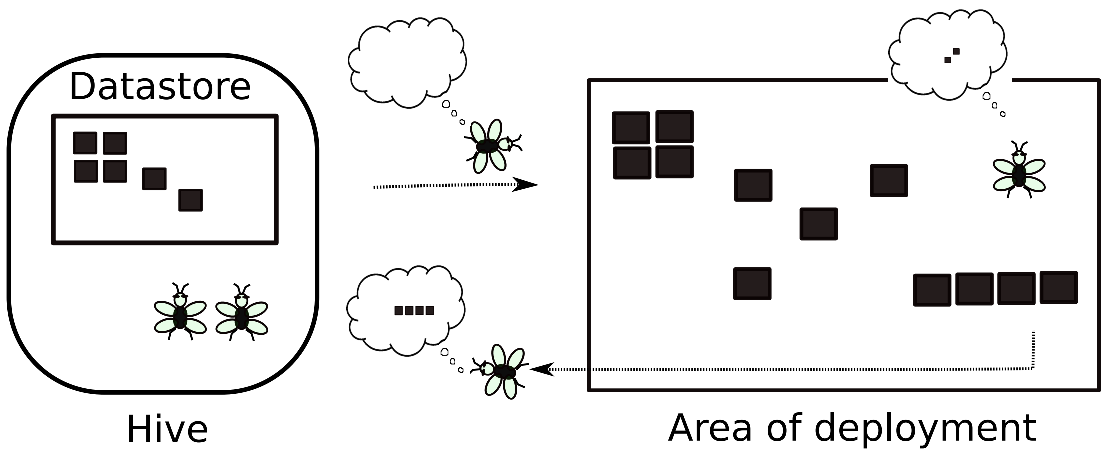


Figure 4.1: The hive-drone data model in action. Drones are dispatched with blank scratchpads. As they execute a behavior, they populate their scratchpad. On return to the hive, the scratchpad is appended to the Datastore.

4.1.5 Programming Model

A Karma application is a composition of low-level drone behaviors and high-level goals. We restrict drone behaviors to be location agnostic so they can be applied to any region as determined by the hive. However, we do not restrict the actions taken by the behaviors; we only explore simple algorithms in this work (e.g., random walks, open loop patterns, and periodic sensing).

The high-level goals of an application are more abstract. In general, the user will need to specify the sequencing of behaviors and the area over which the swarm will operate. For our alfalfa example, a farmer may wish to specify a portion of crops to be monitored and that a pollination behavior should only be executed following the detection of alfalfa in bloom.

From the perspective of the hive, the sole purpose of a drone behavior is to populate the Datastore with new information. We take this notion one step further and require that every behavior produce some type of information under normal execution. For monitoring behaviors this requirement is trivial to fulfill – a new piece of information can be produced with each sensor reading. A similar approach is taken by behaviors that manipulate the environment. For instance, a pollination behavior could record attempted landings. All behaviors produce information that is added to the Datastore when the drone returns to the hive. The hive uses this information to reason about the state of the application at runtime, and this information is used to define relationships between behaviors and selectively apply them to regions. In this abstraction, the programmer defines an application as a set of simple drone behaviors along with two functions for each behavior:

- **Activation Predicate**: A boolean function based on the information in the Datastore. The hive can allocate a drone to execute this behavior if the function evaluates to true.
- **Progress Function**: A function based on the information in the Datastore that evaluates to a real number between 0 and 1, indicating the progress made toward the application goal associated with the behavior. When this function returns a value of 1, the application has achieved the behavior's goal.

These function definitions enable the hive to make decisions about resource allocation over time and space. Using the activation predicates, the hive can determine *when* it is appropriate to execute each behavior. Since the Datastore can be queried spatially (at a regional resolution), the context of the information passed to this function can be narrowed to determine *where* the behavior is activated. This allows a programmer to specify a data dependency that defines the selective execution of a behavior at runtime. Further, the information used by the activation predicate defines the prerequisites for execution. That is, an implicit dependency is created between a behavior that produces the information used by another behavior's activation predicate. By defining the activation predicates this way, a programmer can sequence the execution of behaviors. Since each activation predicate is evaluated independently, our model allows multiple behaviors to be activated concurrently, including behaviors that have a dependent relationship.

For example, consider our alfalfa application; the pollination behavior is dependent on the bloom-monitoring behavior, but both could be activated once some flowers have been found. Pollination could begin on the known flowers while more are sought. We would like to note that the implicit behavior graph created by this representation may contain cycles. At this point, we make no effort to detect cycles in an application.

The progress function is used by the hive to reason about resource allocation. By tracking the rate of change of this function, it can determine an estimate for the number of drones that are required to complete the behavior. Like the activation predicate, the Datastore query can be narrowed to a regional context, allowing the hive to make more targeted drone allocations.

4.1.6 Scheduling Problem

The hive-drone model and our programming abstraction transform the problem of executing an application on a MAV swarm into a problem of scheduling behaviors on drones. There are many policies for determining the allocation of drones to behaviors. We have chosen the *shortest time to application completion* as one objective. There is an economic argument for wanting to finish an application as soon as possible – swarm maintenance (e.g., powering the hive computer and charging drones) may be expensive. Other reasons might be environmental in nature; for example, MAV flight will not be possible when it is raining, so an application may be racing against an unfavorable weather forecast. This objective advocates greedily scheduling all available drones. It could, however, lead to a policy that would execute behaviors that are concurrently activated in a batch sequence. That is, the scheduling objective makes no distinction between a schedule that allocates all drones to behavior `A` until it is completed, followed by all drones to behavior `B`, and one that interleaves allocations for `A` and `B`. Concurrently activated behaviors should be scheduled fairly (without starvation) in the absence of prioritization information. A second objective is introduced to *achieve fairness between behaviors*. Fairness can be defined as parity in the output of the progress functions of individual behaviors. Therefore, the second objective of our scheduling function is to minimize the difference in progress between any two activated behaviors.

More formally, let $B = \{b_i \mid 1 \leq i \leq n\}$ be the set of behaviors in the application. Let $prev_i$ be the total number of drones that have previously executed behavior b_i. At the current time, let $prog_i(S)$ be the progress made toward the goal of behavior b_i and S the state of the Datastore. Let the estimate of the rate of progress made for behavior i per drone be defined as $rate_i = \frac{prog_i(S)}{prev_i}$. Let $curr_i$ be the number of drones currently running this behavior. Let d be the number of drones currently available for dispatch.

To solve our objectives, we need to allocate $alloc_i$ drones to behavior i such that

$$max\{rate_k * (prev_k + curr_k + alloc_k)\} \; \forall \; k \in [1,n]$$

and

$$\begin{aligned} min \; \{&[rate_i * (prev_i + curr_i + alloc_i)] \\ &- [rate_j * (prev_j + curr_j + alloc_j)]\} \; \forall \; (i,j) \; \in \; [1,n], i \neq j \end{aligned}$$

such that

$$\sum_{i \in (1,n)} \{alloc_i\} \leq d$$

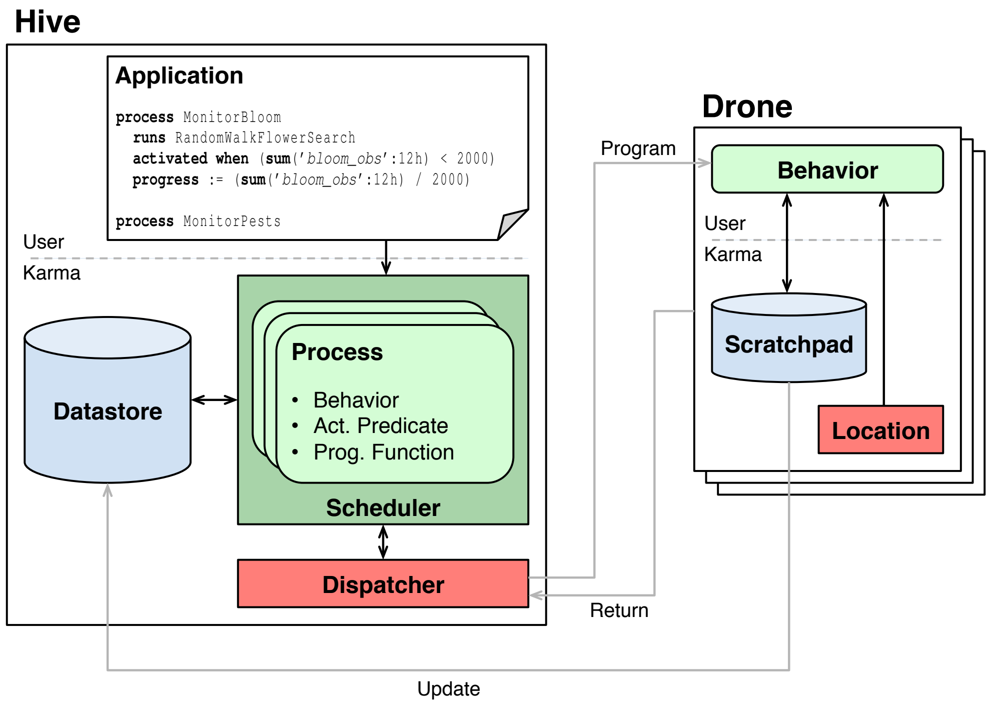
Given these objectives, the hive will allocate drones to behaviors as resources become available.

4.2 Karma Implementation

We have built Karma, a resource coordination system for MAV swarms based on the hive-drone model. Figure 4.2 depicts the functional block diagram of Karma. The Karma runtime at the hive consists of a hive Controller, Scheduler, Dispatcher, and Datastore. The hive Controller is the overall manager of the runtime and invokes the other modules when needed. When a user submits an application to Karma, the hive Controller determines the set of active behaviors (using the activation predicates), and invokes the Scheduler to allocate the available drones to them. The hive Controller monitors the progress of each process and considers the application complete when the set of active processes is empty. The Scheduler is periodically invoked by the Controller to allocate drones to each active process. The Dispatcher is responsible for tracking the status of the physical resources (the MAVs). It programs the drones with the allocated behavior prior to a sortie, tracks the size of the swarm, and notifies the Controller when a drone returns to the hive and is ready for redeployment.

To accomplish any goal with the MAV swarm, a user must submit an application. Karma applications are collections of *processes* that are executed at the hive. Each process defined by the application has an associated behavior that is executed on the drone. Each process also specifies the information that it yields when its behavior is executed, as well as the information that is required to activate the process. This is accomplished by enumerating the Datastore keys for the information that is used and yielded. The activation predicate and progress function are evaluated by the Scheduler in the context of queries made against the Datastore using these keys. As mentioned in Section

4.1.4, these queries can be temporally bounded to suit the needs of the application.



```
process MonitorBloom
  runs RandomWalkFlowerSearch
  activated when (sum('bloom_obs':12h) < 2000)
  progress := (sum('bloom_obs':12h) / 2000)

process MonitorPests
```

Figure 4.2: Block diagram of the Karma design. Applications containing sets of processes are submitted to the Karma hive by a user. Each process definition contains an activation predicate, a progress function, and a drone behavior. The Scheduler allocates resources (available drones) to processes. The Dispatcher consumes the allocation and programs behaviors on to drones and dispatches them on sorties. Upon their return, drones transfer the contents of their scratchpad to the hive Datastore.

4.2.1 Programming the Swarm

Figure 4.3 shows pseudocode for part of our alfalfa crop monitoring and pollination application described in Section 4.1.1. The code defines four processes, two of which are omitted for brevity but are defined similarly to `MonitorBloom`. In this case, the pseudocode references library behaviors that are specified outside of the application code. With this approach it is possible to define common routines that execute on the drone and share them among applications.

```
process MonitorBloom
  runs RandomWalkFlowerSearch
  uses ()
  yields ('bloom_obs', 'bloom_det')
  activated when ('bloom_obs':12h < 2000)
  progress := ('bloom_obs':12h / 2000)

process MonitorPests
  ...

process MonitorDisease
  ...

process Pollinate
  runs RandomWalkPollinate
  uses ('bloom_det')
  yields ('pollinated')
  activated when ('bloom_det':12h > 0)
  progress := (if isNull('bloom_det':12h)
                 then 1
                 else ('pollinated':12h /
                        (2 * 'bloom_det':12h)))
```

Figure 4.3: Pseudocode for the process definitions that make up the alfalfa crop monitoring and pollination application.

The behavior associated with the `MonitorBloom` process, `RandomWalkFlowerSearch`, executes a random walk pattern while periodically using an optical sensor to detect the presence of flower blooms. Each time the sensor is read, a counter in the drone's local scratchpad with the key *bloom_obs* is incremented. After analyzing the reading, a counter with the key *bloom_det* is incremented if a flower bloom was detected.

When the drone returns to the hive, this information is propagated to the hive Datastore, where it can be queried by the Scheduler to evaluate activation predicates and progress functions. In this case, it uses the total number of observations in the past 12 hours to determine if the `MonitorBloom` process should be activated, as well as how much progress has been made toward its completion. The `Pollinate` process references a similar behavior that executes a random walk and lands the drone on flowers, collecting and depositing pollen through incidental contact. However, unlike the `MonitorBloom` process, which is activated by a lack of observations in the past 12 hours, this process is only activated when flower blooms have been detected (in the past 12 hours).

The information dependency on *bloom_det* creates an implicit sequence of operations. For the `Pollinate` process to activate, the Datastore must contain information for the key `bloom_det`. In our application, this means that a drone running the `RandomWalkFlowerSearch` behavior for the `MonitorBloom` process must observe flowers on a prior sortie. Alternatively, should the need arise to bootstrap the `Pollinate` process, the user could inject this information into the Datastore manually. The information dependency also exists in the progress function defined by the `Pollinate` process. The function is defined such that the process is considered complete in regions where there is no *bloom_det* information. This allows the progress of the process to be evaluated regionally and prevents drones from being allocated to regions where no blooms have been detected. Finally, recall that the activation predicate for each process is evaluated independently, potentially resulting in multiple activated processes. In this case, the monitoring processes may execute concurrently with the pollination process, and the Scheduler must allocate resources appropriately.

4.2.2 Karma Scheduler

Karma implements a fair, work-stealing scheduler that solves the optimization problem illustrated in Section 4.1.6. It operates in two steps. First, it estimates the total workload to be performed for the set of n active processes per region. Let $P = \{p_i \mid 1 \leq i \leq n\}$ be the set of active processes. Let R be the set of regions the area of operation is divided into. Let S_t be the state of the Datastore at the hive at time t. The progress of a process i in a given region r at time t can be evaluated using the progress function provided ($PF_i(S_t^r)$). Let us denote this progress as $prog^t_{(i,r)} = PF(S_t^r)$. At a future time t' ($t' > t$) when the scheduler is invoked, let there be k drones executing sorties for process i in region r that have returned to the hive.

We can compute the rate of progress per drone-sortie for process i ($q_{(i,r)}$) as $q_{(i,r)} = \frac{(prog^{t'}_{(i,r)} - prog^{t}_{(i,r)})}{k}$. Given the rate of progress per drone per sortie $q_{(i,r)}$, we can estimate the amount of work remaining to complete process i in region r (in terms of number of drone-sorties) as $N^{t'}_{(i,r)} = \frac{(1-prog_{(i,r)})}{E[q_{(i,r)}]}$. Since the value of $q_{(i,r)}$ can vary significantly over time and with environmental conditions, we compute the value $E[q_{(i,r)}]$ using a weighted average of historical $q_{(i,r)}$ values. Note that the progress rates $q_{(i,r)}$ cannot be computed at $t = 0$. We bootstrap the progress estimation mechanism by sending a fixed number of *scout drones* to regions where there is insufficient progress information. Once the progress of the scout drones is known, $q_{(i,r)}$ can be computed as defined above. We can then estimate the total work to be done across all active processes as $N^{t'} = \sum_{i \in (1,n), r \in R} N^{t'}_{(i,r)}$. Note that the work estimate $N^{t'}$ assumes a linear relationship between the number of drones dispatched and the amount of progress made.

The second step is to allocate the available drones to the set of active processes fairly. Karma takes a work-stealing approach to allocation, using a sorted queue with each element in the queue representing an active process requesting drones in a region. The queue is sorted in ascending order according to the *service level* of each request. We define service level as the ratio of remaining amount of work to complete process i in region r ($N^{t'}_{(i,r)}$) to the total work to be done by the application for all active processes across all regions at the current time ($N^{t'}$). As drones become available, Karma allocates them iteratively by servicing the request at the head of the queue. If m drones are available for allocation, this results in an allocation of $alloc^{t'}_{(i,r)}$ for each process i in region r at time t'

$$alloc^{t'}_{(i,r)} = \frac{N^{t'}_{(i,r)}}{N^{t'}} * m$$

This formulation meets our objective of fairness across processes, as illustrated in Section 4.1.6.

The above formulation ensures that resources are divided fairly across processes and regions. However, there may be applications in which the processes must be executed in a predetermined order. To address this class of applications, we allow for the specification of a process priority in the application description. To accommodate these requirements, we first sort the queue used for drone allocation by process priority and then by service level. In a resource-constrained situation, we allocate drones to the higher priority processes, allowing lower priority processes to starve.

4.2.3 Dispatcher

The Dispatcher is responsible for carrying out the allocation decisions made by the Scheduler. Specifically, it manages the drone inventory and prepares drones for sorties by programming the specified behavior onto the drone and parameterizing the starting region. When a drone returns, the Dispatcher invokes a process to merge the drone's scratchpad with the hive Datastore and initiates a charge cycle. When drones are fully charged and ready to be dispatched, the Dispatcher notifies the Controller of the resource availability.

In Section 4.1.2, we define the term information latency as the difference in time between an event occurring in the world and the hive being informed of the occurrence. This latency is a function of the swarm deployment (e.g., swarm size, sortie time, charge time, maximum velocity) and dispatch policy. In general, it is desirable to minimize the information latency in the system, but it is particularly important for applications that track highly dynamic or continuous phenomena. In Karma we manipulate the dispatch policy to better fit the application and minimize the information latency. The Controller in Karma operates in two phases; it first determines how many resources each process-region pair should be granted by invoking the Scheduler, and then dispatches drones (using the Dispatcher) to fill the allocation requests. The algorithm for determining the amount of resources to grant is fixed, but the dispatch policy may vary. As such, we propose two dispatch policies. The goal of the *continuous dispatch policy* in Karma is to ensure a constant presence of drones in the field and minimize the information latency in the system by amortizing the total allocation of drones to a region over a period of time (sortie time + charge time). In contrast, the *greedy dispatch policy* dispatches drones opportunistically. We investigate the effects of these policies in our system evaluation.

Reducing information latency can have a significant impact on the quality of data collected by applications that track continuously changing phenomena, such as chemical plumes. The application programmer should decide how information latency affects the application at hand. We allow the application programmer to add this as part of the program specification, and the corresponding dispatch policy is selected accordingly.

4.2.4 Execution Walkthrough

We conclude the description of the system by walking through an execution trace of a simplified application. The pseudocode for this application is given in Figure 4.4.

```
process Search
  runs RandomWalkSearch
  uses ()
  yields ('obs', 'feature')
  activated when ('obs' < 250)
  progress := ('obs' / 250)

process Survey
  runs RandomWalkSurvey
  uses ('feature')
  yields ('studied')
  activated when ('feature' > 0)
  progress := (if isNull('feature')
               then 1
               else ('studied' / (2 * 'feature')))
```

Figure 4.4: Psuedocode for the definition of processes that make up the walkthrough application.

We illustrate key features of the system by examining the decisions made by the Controller over the course of the execution. The example application consists of two processes, `Search` and `Survey`. Process `Search` runs a behavior executed uniformly over the target area and produces information relating to interesting features observed in the field. Process `Survey` runs a behavior that is triggered to execute only in areas identified as interesting by the `Search` process.

We execute this application in simulation using 200 drones in a target area that is 75 x 75 meters. The world is partitioned into regions by dividing the target area as a grid with 10 rows and 10 columns. The hive is placed at the center of this area. A circular area representing the presence of interesting features is modeled with its origin at $(15, 15)$ and a radius of 10 meters. For simplicity, no weather or hardware failure is modeled. A greedy dispatch policy is used on the hive. Each drone is given enough energy to complete a 40-50 second sortie, with a subsequent charge time of about 2 minutes.

Figure 4.5 depicts an execution trace for this application. The top panel shows the amount of remaining work over time. The bottom panel shows the number of drones that are allocated to each process over time. This data illustrates two points about our scheduling algorithm. First, the drones appear to be allocated in waves. For the most part, this is true. The peaks in the bottom panel represent times when drones were dispatched (opportunistically using the greedy policy) and the valleys represent charging cycles. Since the drones are returning from different regions in the target area and have slightly different battery capacities upon returning, they are not available for scheduling at exactly the same time. This creates the jagged appearance of each "step" in the allocation plot.

Second, notice that the `Survey` process is initially idle and remains idle until the first

`Search` sortie returns to the hive and the drones deposit the collected information into the central Datastore, demonstrating the implicit dependency between the two processes. Third, the bootstrapping sortie, as described in Section 4.2.2, is required when there is a lack of information about the progress rate for a process in a given region. In this example the number of drones allocated to the first sortie of each process is limited to one per region (100 total) despite the fact that more drones are available at the outset. Finally, the number of drones allocated to each process is proportional to the amount of work remaining and is a function of the estimated progress rate. For most of the execution, the amount of work remaining for `Survey` is far below that of `Search`, so `Search` is allocated most of the resources.

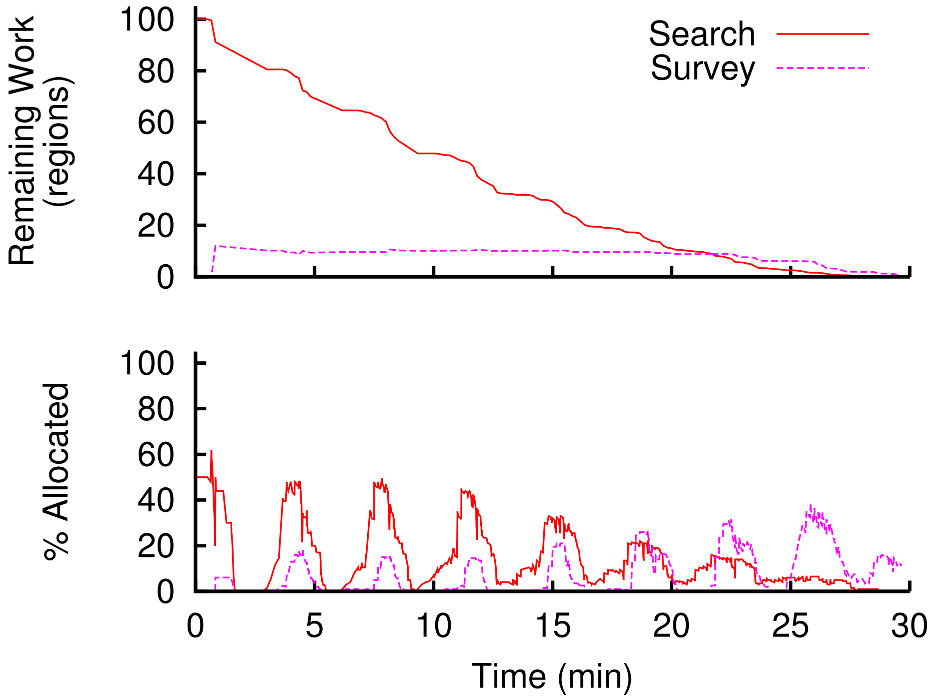


Figure 4.5: Karma behavior allocation. Karma allocates drones to behaviors according to the estimated amount of work to be done and the measured progress rate of each behavior per region. Remaining work is the sum of remaining progress across all regions.

Figure 4.6 illustrates how the dependency between processes results in selective allocation. The lefthand panel shows a drawing of the world in which the application is running. The hive is at the center of the world and an area containing interesting features is set in the top right as shown. The `Search` process records observations of these features when it executes in this area. The scheduler evaluates the activation predicate for the `Survey` process in each region of the discretized world and allocates drones accordingly. The righthand panel depicts the resulting cumulative allocation of drones to regions for the `Survey` process over the course of the execution. Without prior knowledge, our system correctly allocates drones to execute the `Survey` behavior only in regions where this behavior is useful. This allocation falls out of a single data dependency in the activation predicate.

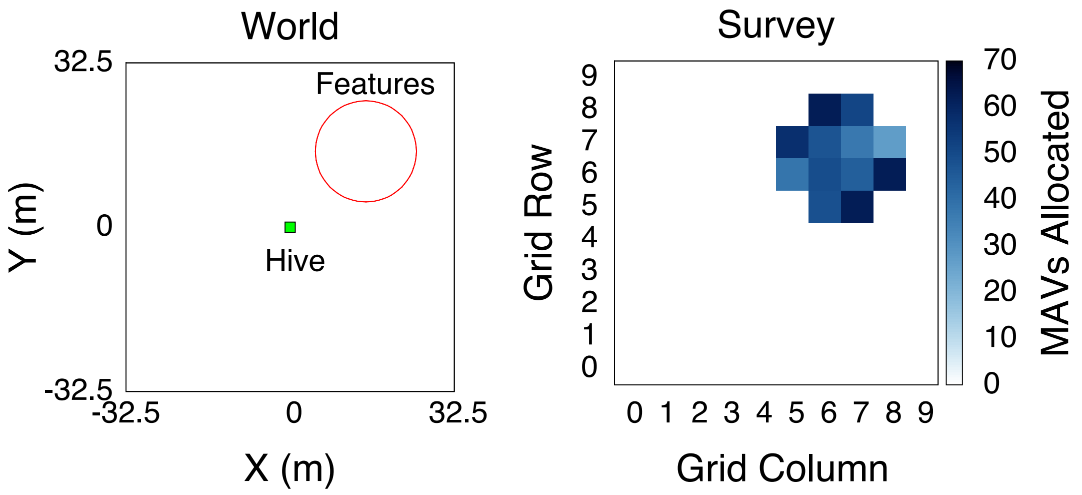


Figure 4.6: Selective allocation in Karma. Processes are selectively activated by the presence or absence of information. The righthand panel shows the regions in which the `Survey` process is activated by the prior detection of environmental features.

This example demonstrates how the key features of Karma, progress rate estimation, proportional scheduling, and selective activation of processes in regions can be combined to execute a swarm application.

4.3 Evaluation

In this section we demonstrate that Karma can be used to manage a swarm of MAVs and effectively execute applications inspired by real world workloads. We also show how applications in Karma can run on limited resources, are robust to individual MAV failure, and adapt to changes in the environment. We characterize the effectiveness of our system by evaluating its performance with respect to three metrics; *execution time, energy cost*, and *information latency*. Completion time and energy cost are useful metrics of efficiency. For instance, a farmer may want to minimize the total execution time so that a single hive can be shared among a number of fields on a fixed schedule. However, minimizing completion time may result in more resources being consumed than is strictly necessary, forcing a tradeoff to be made. We evaluate the scheduling decisions made by the system in the context of this tradeoff. Increased information latency is a direct result of the hive-drone paradigm. It is especially problematic for applications that track features of the environment that change frequently or continuously. To this end, we use this metric to evaluate how the selection of a dispatch policy can mitigate the negative effects of the sortie model.

4.3.1 Simulation Setup

For our experiments we use our Simbeeotic simulator, described in Chapter 3. The Karma implementation consists of two runtimes: one that executes on the hive, and one that is embedded on the drones. The hive runtime is responsible for monitoring the progress of applications, scheduling and dispatching sorties, and charging drones. The drone runtime provides location and data storage services to running behaviors. The Karma hive runtime is implemented as a standalone Java application and is decoupled from the drone runtimes by a custom *dispatch driver*. This arrangement allows the Karma hive to execute the same application on multiple deployments. Our evaluation is based on experiments carried out in Simbeeotic and on a testbed of toy helicopters using this mechanism. All of the results shown are obtained in simulation, with the exception of the testbed experiment in Section 4.3.6.

We evaluate our system by using the alfalfa crop monitoring and pollination application introduced in Section 4.1.1 and depicted in Figure 4.3. In the following experiments, we run this application in a model alfalfa field that is one acre in area (63.63 by 63.63 meters) discretized into a 6 x 6 region grid (with each grid being 10.61 by 10.61 meters). The origin of the world is at the center of the field, and the hive is placed outside the field at $(-35, 0)$. The application is designed to execute for an entire growing season, periodically re-executing the three monitoring behaviors and activating the pollination behavior when the crop is in bloom. As written, the application expects the field to be monitored for pests, disease, and flower blooms *once per day*. Our experiments consist of single-day snapshots of the application rather than the entire season. The battery and energy model of the drones are parameterized to provide approximate sortie and charge times of 5 minutes and 20 minutes respectively, which represent the flight time and recharge time of the helicopters used in our testbed. The drones have a top speed of 2 meters per second when cruising to and from the hive, but operate at speeds between 0.25 and 0.5 meters per second when executing behaviors. These numbers were also obtained empirically from the helicopters in our testbed. The Scheduler is set to allocate drones (if available) every 10 seconds. Statistics for experimental results are gathered by repeating simulated experiments five times with varying random seeds. Unless otherwise noted, a swarm size of 800 drones is used in combination with a greedy dispatch policy for all experiments.

4.3.2 Efficiency

We evaluate the efficiency of our system by comparing the overall completion time and energy cost of the alfalfa crop monitoring and pollination application as executed by Karma in Simbeeotic to a restricted version of an oracle offline scheduling model. For a fair comparison, the offline solution is required to use the gridded spatial decomposition and search each region for flower blooms despite having perfect knowledge of the bloom density. For each drone scheduled, the oracle model has full knowledge of the performance of all previously deployed drones. Since drones are scheduled concurrently, this implies that the oracle model has *foreknowledge* of all the activity of the drones that are currently deployed. As such, the Scheduler can allocate the minimum number of drones required to complete the application without having to estimate drone performance. We use this oracle model as a theoretical lower bound for comparison with Karma, even though it is not attainable in practice.

Figure 4.7 shows the scenario completion time and expended energy as a function of swarm size. We do not expect the performance (w.r.t. completion time) of Karma to improve linearly with swarm size in short scenarios because recharging is the dominant contributor to swarm overhead. Unless there is enough swarm growth to reduce the number of sorties executed by each drone, the gains will be limited. As expected, the oracle solution scales nearly linearly to the point where there are enough drones to complete all of the required work without recharging (about 850 drones). Karma scales sublinearly with swarm size. The performance gap between Karma and the offline solution is mainly attributed to the initial test sortie (with the accompanying charge period) and the information latency caused by muling data. The first sortie that Karma allocated is a bootstrapping sortie, where only a few MAVs are deployed to each region. This ramp up of allocation ensures Karma will not initially overcommit resources.

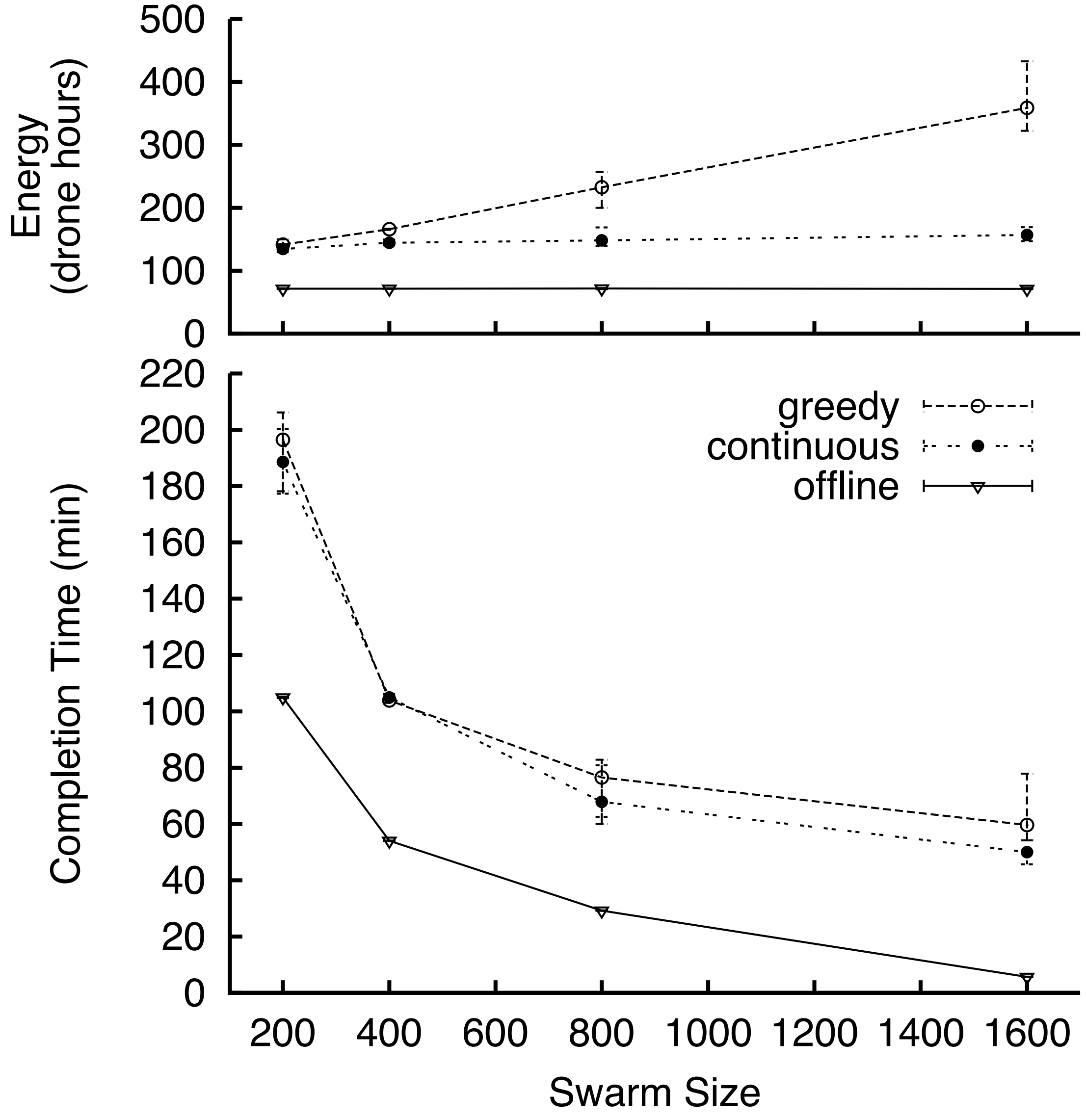


Figure 4.7: Karma efficiency. Karma scales sub-linearly (w.r.t. completion time) as swarm size increases. Gains are offset by relatively long charging periods in short scenarios.

In the case where 1,600 drones were deployed, the large performance gap between Karma and the oracle was because the oracle had enough drones to complete all of the behaviors in one sortie. First the oracle deployed drones to complete the three monitoring behaviors. As soon as those drones were deployed, the oracle then immediately knew how many drones should be allocated to complete the pollination behavior. The oracle then was able to immediately deploy enough drones to complete the pollination behavior, which resulted in being able to complete the entire application in about the time of one sortie. Since Karma ramps up allocation of resources with test sorties before it commits significant resources to a region, it took several more sorties to complete the application than the oracle. Under realistic application workloads, the amount of work far exceeds the number of drones available, so this situation in the experiment represents a corner case rather than a typical occurrence.

We evaluate Karma using the greedy and continuous dispatch policies described in Section 4.2.3. Since the workload is fixed and the evaluation metric is total completion time, the performance of the two dispatch policies is roughly equivalent. However, there is a disparity between the two dispatch policies with respect to energy consumption. As the size of the swarm increases, the system using the greedy dispatch policy consumes 50% more resources. This is due to a combination of poor allocation estimates and opportunistic dispatching. The greedy dispatch policy tends to release drones in batches. We have seen when early allocation estimates are inaccurate, it leads to an overcommitment of resources. Since the continuous policy staggers drone dispatching over time, the erroneous resource allocations can be gradually corrected with limited overhead.

4.3.3 Resilience to Failure

We demonstrate Karma's resilience to individual drone failure. Our system is designed to provide a graceful degradation in performance (as measured by overall completion time) that is proportional to the degree of failure that occurs. The approach we take follows naturally from the hive-drone model and the use of progress rates to estimate resource allocation needs. When a drone fails in the field, it never returns to update the Datastore and, from the perspective of the hive, no progress is made. The Dispatcher will detect the failure with a timeout and inform the Scheduler that it has one less resource in the field and that the swarm size has decreased. The Scheduler will take this into account during the next allocation cycle and update its estimates accordingly. This mechanism works well when failure is uniformly distributed across all regions of the target area. However, there is a corner case in which a disturbance (e.g., strong wind) is localized to a subset of the area. In this case we would prefer that the system detect the anomaly and discontinue sending drones to the hazardous area. Because this disturbance may also be time-varying, an attempt could be made to resume allocation to the problem region after a period of time. Handling these situations in Karma is left as future work.

We evaluate the effectiveness of our system design with respect to failure by executing the alfalfa crop monitoring and pollination application with a swarm of drones that have a constant probability of failure. With this model we expect to see an increase in the total execution time of the application with an increase in failure. As shown in Figure 4.8, Karma handles the unexpected failures with a predicted graceful degradation in performance. The majority of the extra time is due to the swarm's inability to immediately react to a detected failure (all of the drones are deployed or charging). This issue can be mitigated by reserving drones or increasing the swarm size. In addition to the time overhead, there is a small energy penalty (5% in the worst case scenario) that is caused by the additional sorties required to make up for the lost drones. The swarm size may dwindle to the point that no progress can be made, but that point is reached through graceful degradation, not hard failure.

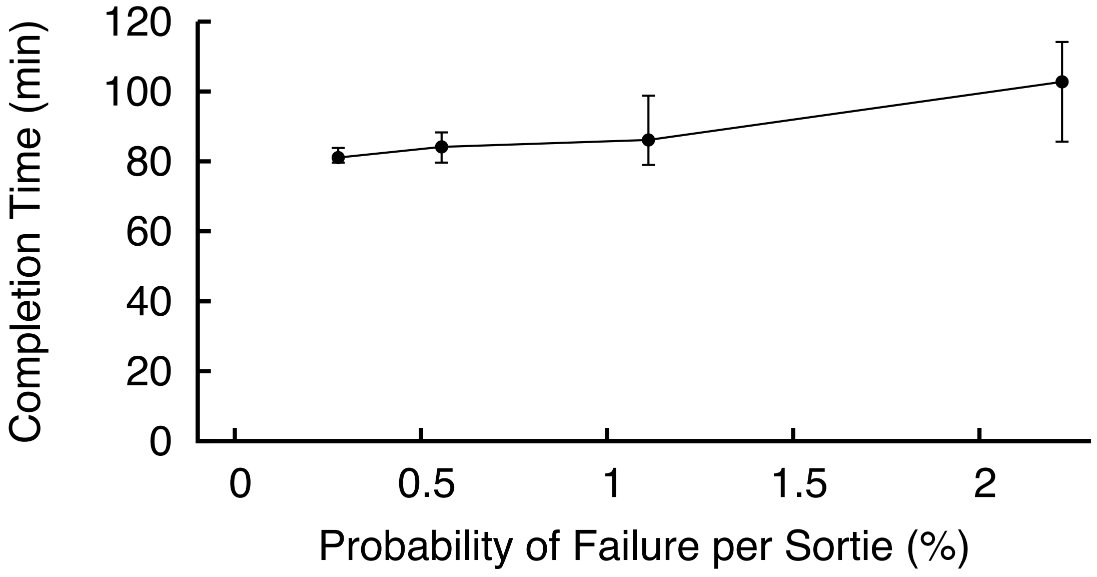


Figure 4.8: Drone failure in Karma. Karma is resilient to individual drone failures, exhibiting a graceful degradation in performance as the probability of failure increases.

4.3.4 Adaptability

Consider the impact on a drone of wind blowing at a constant speed and direction against the flight path. The drone must work harder to compensate for the additional force and avoid being blown off course. As a result, it will expend more energy to fight the wind, resulting in a shorter sortie time. In turn, the progress rate for the behavior it is executing will be reduced and the system will produce higher resource estimates.

In the following experiment we modify the alfalfa crop monitoring and pollination application to introduce a constant wind over the bottom third of the field. Drones dispatched to the bottom third of the field experienced a 32% reduction in sortie time. Because the workload is constant for the four behaviors in this application, shorter sortie times result in less work accomplished per sortie, which reduces the progress rate for these regions. In this case, the Scheduler responds by allocating 12% more drones to the windy regions than the wind-free regions to accomplish the same amount of work. Correspondingly, the energy cost for executing in the presence of regional wind is 7% higher than that of the windless scenario.

The environmental dynamics are implicitly captured in the regional behavior progress rates. Even though the Karma system did not explicitly measure the wind speed, the swarm was able to adapt to its presence. By representing the dynamics in a single variable (the progress rate), the hive is able to adapt to external influences, but it cannot disambiguate the causes or apply specific solutions. Though this experiment demonstrates spatial variation, the dynamics of the environment can also change over time. The Scheduler's use of the regional progress rates to capture these dynamics also accounts for temporal fluctuation. The accuracy with which the scheduler can track the progress rate (and implicitly the environmental dynamics influencing it) depends on how frequently that rate is sampled (how often a drone returns from that region). This is defined in Section 4.1.2 as the information latency problem, and is addressed in the next set of experiments.

4.3.5 Information Latency

The previous experiment focused on evaluating our system in the presence of environmental dynamics. However, the features of interest in the environment were essentially static (or did not change in any detectable way) over the period of execution. How does the system behave when applications track phenomena that change continuously?

We demonstrate that the hive-drone paradigm can be used to continuously measure time-varying phenomena. To this end, we define a chemical plume tracking application that consists of three behaviors: one to perform a uniform search of the target area, one to detect a level set (contour) of the plume, and one to find the center. Figure 4.9 defines this application in pseudocode.

```
process DetectPlume
  priority = 2
  runs RandomWalkPlumeSearch
  uses ()
  yields ('plume_obs', 'plume_det')
  activated when ('plume_obs':5m < 400)
  progress := ('plume_obs':5m / 400)

process FindLevelSet
  priority = 1
  runs RandomWalkLevelSet
  uses ('plume_det')
  yields ('level_obs', 'level_det')
  activated when ('plume_det':5m > 0)
  progress := (if (isNull('plume_det':5m)
               then 1
               else ('level_obs':5m / 500)))

process FindCenter
  priority = 1
  ...
```

Figure 4.9: Psuedocode for the definition of processes that make up the plume tracking application.

The application is executed in a target area that is 100 meters by 100 meters, discretized into 10 meter by 10 meter regions. The plume is centered at $(-25, 25)$ and expands in a hemispherical pattern at a rate of one centimeter per second. Unless otherwise noted, a swarm size of 800 drones is used along with the continuous dispatch policy.

All three processes defined by the plume tracking application are unbounded, meaning that they are not meant to terminate after some fixed amount of work is done. Rather, they define sliding windows in which the activation predicates and progress functions are evaluated as time moves forward. Using these windows, drones are allocated to regions that have the most recent information, allowing the frontier of swarm activity to follow the plume as it expands. Figure 4.10 depicts a series of snapshots taken from the Datastore as the plume tracking application is executed. Using a windowed query, we are able define the regions in which the *level_det* feature was most recently observed. Older observations of this feature are stored in the Datastore but are not included in the view defined by the windowed query.

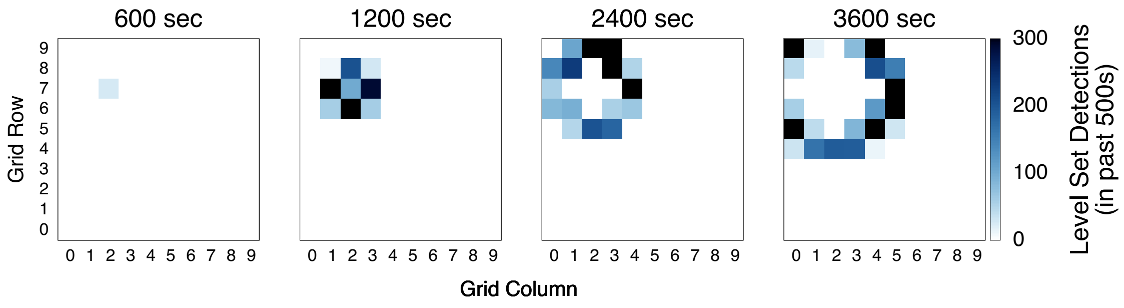


Figure 4.10: A series of snapshots from the hive Datastore depicting the measured contour of an expanding chemical plume.

In the next experiment, we quantify the effect the dispatching policy has on information latency. We execute the plume tracking scenario with the two dispatch policies (greedy and continuous). As a proxy for information latency (as defined in Section 4.1.2), we measure the *process-region return period*, the amount of time between consecutive drones returning from each region for each process. This allows us to measure how frequently we receive information from a region, which is directly related to the information latency on events that occur in that region. Because the drones that are dispatched concurrently do not return at exactly the same time, we cluster return events that occur within a 30-second period as one event.

Figure 4.11 shows the results of the experiment running for 6 virtual hours. Depicted is the mean and standard deviation of information latency measurements for a single process-region pair. As expected, the continuous policy outperforms the greedy policy with respect to minimizing information latency. As the swarm size increases, there are more drones available when the Scheduler requests resources, and the charge time plays a smaller role in defining the information latency. On average, switching policies reduces the measured information latency by 63%, with an order of magnitude improvement (97%) when the swarm size is increased to 1,600 drones.

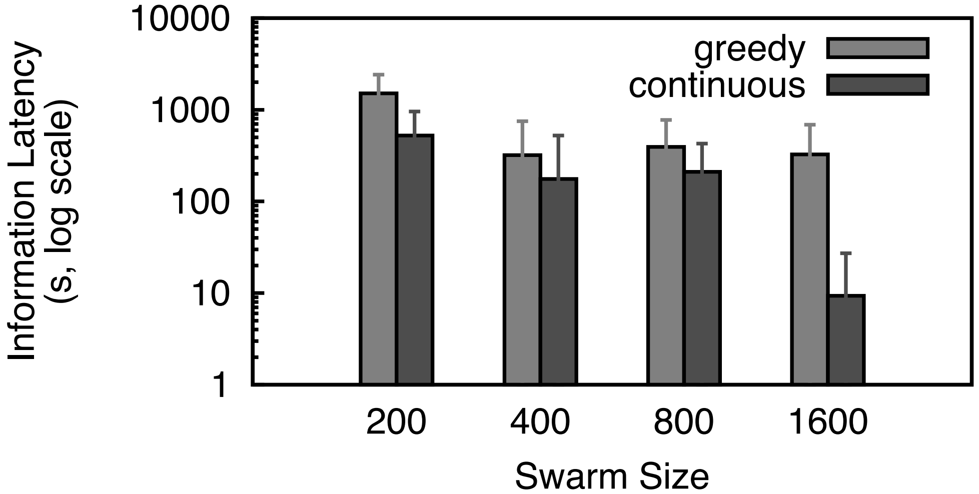


Figure 4.11: Information latency measurements for one process-region pair in the plume tracking application. The continuous dispatch policy consistently outperforms the greedy policy.

The takeaway is that information latency is inherent in the centralized design of Karma. In general the continuous scheduler works better for applications that are monitoring the environment, because the scheduler staggers the allocation of MAV resources to a region. How tolerant an application is to information latency depends on how fast the state of the environment of interest is changing. That understanding is needed to plan how many total MAVs should be used for a given application.

4.3.6 Helicopter Testbed

In addition to the experiments in simulation, we evaluate Karma on our MAV testbed described in Section 3.3. We evaluate a simple tracking application using three helicopters and one ground vehicle, which is shown in Figure 4.12.

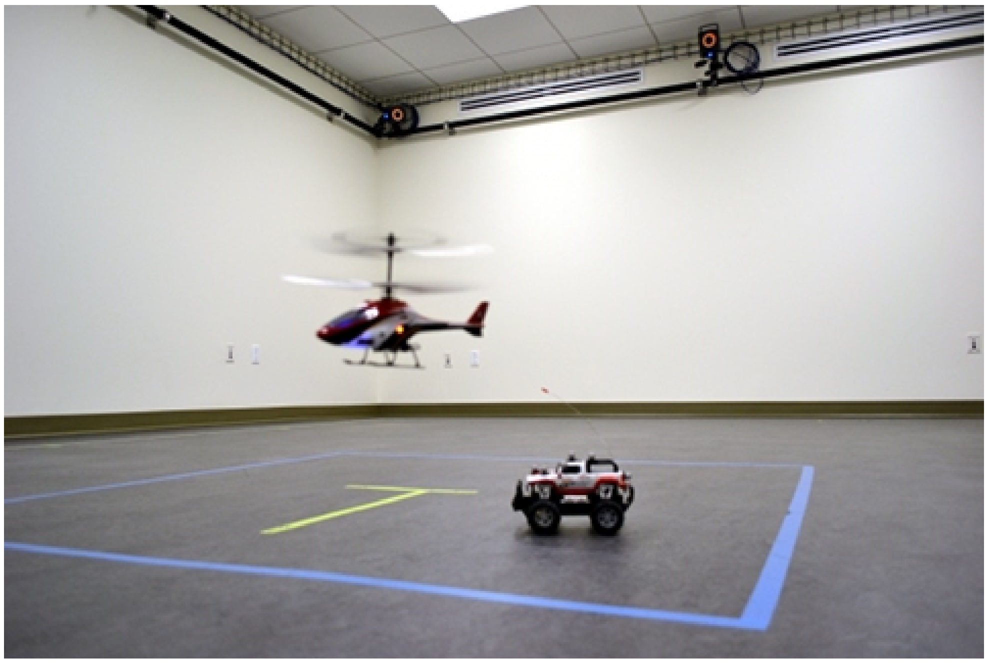


Figure 4.12: The ground vehicle and helicopters operating in the indoor testbed.

Our testbed space is divided into four equally sized regions. Initially, Karma dispatches helicopters with the `Search` behavior, causing the helicopters to look for ground vehicles. When a helicopter has located a vehicle, it writes the location of the vehicle on its local scratchpad. When the drone returns to the hive and uploads its data, the tracking process is activated. Karma then dispatches a helicopter with the `Track` behavior to the last known vehicle location to resume observation.

The ground vehicle starts off stationary in the center of one of the regions. After approximately 30 seconds, the vehicle moves to an adjacent region and waits another 30 seconds before moving on. This process repeats for the duration of the experiment. The helicopters fly 20-second sorties. This is short of the seven-minute flight time of the helicopter but allows us to see Karma perform several sorties during the course of the experiment. We equip the helicopters with a virtual sensor that allows them to locate the ground vehicle if they are within 90 cm in the x-y plane.

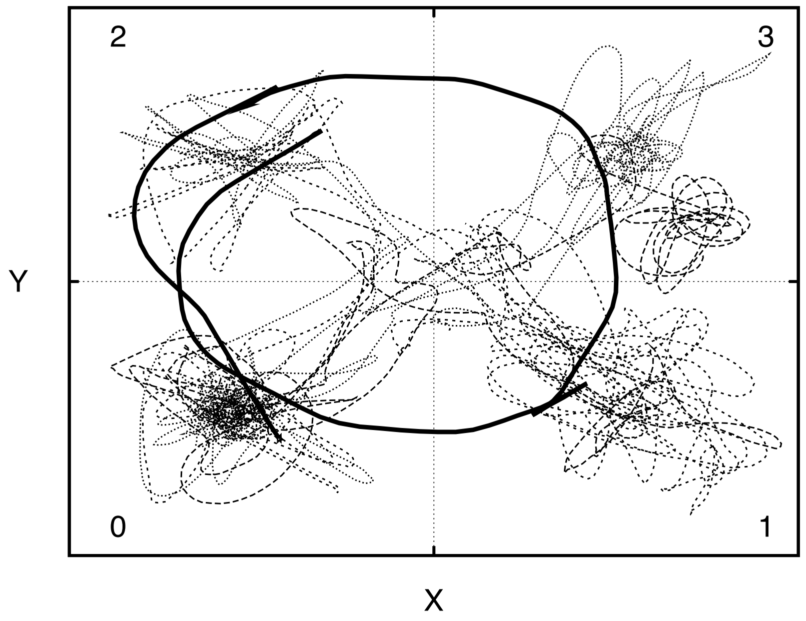


Figure 4.13: Ground truth target location and helicopter flight paths as recorded by the motion capture system during the testbed experiment. The target is solid and the helicopters are dotted.

Figure 4.13 shows a ground truth trace of the tracking experiment. At the start, the vehicle begins in Region 1 and moves in a clockwise fashion. Figure 4.14 shows a region-level trace of the target location during the experiment. The markers indicate when a helicopter records an observation into its scratchpad. The hive-drone model introduces information latency, which can be seen here as a gap (on the time axis) between the solid and dotted lines, which represent the truth and perceived target location, respectively.

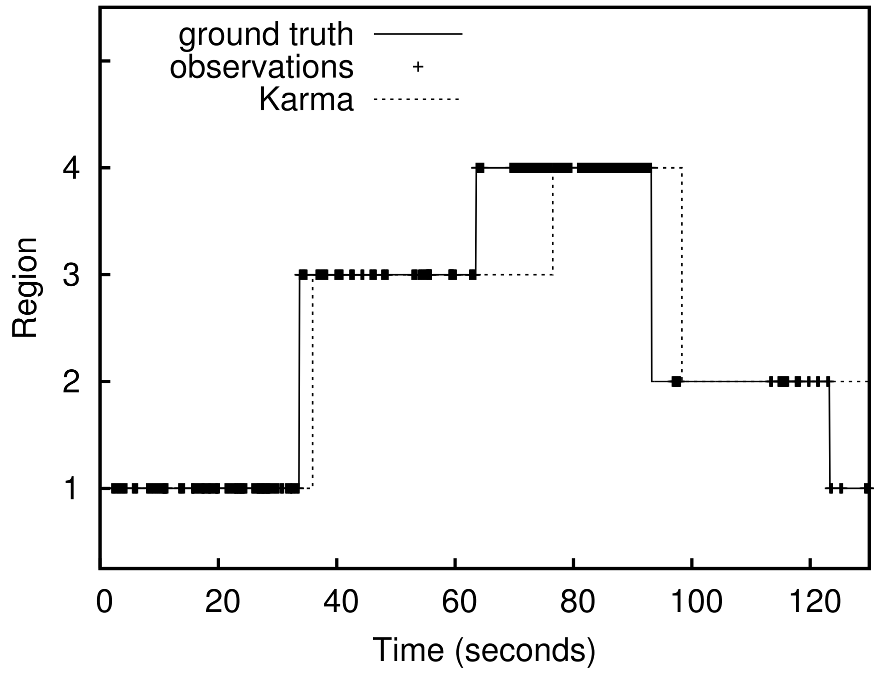


Figure 4.14: The region occupied by the target throughout the vehicle tracking experiment. The perceived location (dotted line) lags behind the truth location due to information latency.

4.4 Summary

Insect-sized MAV swarms are an emerging class of mobile sensing systems. These MAVs can perform tasks that are challenging for larger platforms, such as landing on a flower to collect or deposit pollen, and their small size means they can be deployed in large numbers in a small area. However, as the size of the swarm grows, so too does the complexity of resource coordination. We propose a novel system architecture based on the hive-drone model that decouples the behaviors performed by the MAV from where and when those behaviors are performed. This shifts the coordination complexity from the application to a central hive, which maintains the current global state of world to efficiently allocate MAV resources. We implement this model in our prototype system called Karma and show that it is efficient, adaptive, and resilient to failure.

Chapter 5

Related Work

The research in this dissertation draws from three different areas: coordinated energy management in embedded systems, distributed execution engines for programming clusters, and resource coordination for swarms. Energy is a precious resource in embedded systems and needs to be carefully managed. However, much of the work on coordinating energy usage has either focused on point solutions, not general frameworks, or optimizing energy at the node-level without offering network-wide solutions. Section 5.1 discusses these systems and how IDEA builds on and extends ideas from these ideas.

Cluster-programming execution engines share goals similar to our research. They coordinate work that needs to be done with the resources to do the work by exposing a simple programming model while hiding the complexities of fault tolerance, scheduling, synchronization and communication. However, coordinating resources for embedded systems presents new challenges due to the different workloads, severe constraints on resources, and the need for low-overhead coordination. Section 5.2 discusses how our work extends these ideas to work in embedded systems.

Section 5.3 concludes the chapter with a discussion of the additional challenges mobility adds to embedded systems and how Karma relates to swarm programming languages and other swarm systems. One difference is that Karma favors explicit coordination over the emergent behavior of typical swarm systems. This allows for easier programming and reasoning about applications. Users can think about applications in terms of explicit tasks to be completed and measuring progress on those tasks, instead of having the desired behavior emerge somehow by the choice of purely local actions.

5.1 Coordinated Energy Management

Energy is the critical limited resource in networked embedded systems and the need for coordination in embedded systems to achieve good energy efficiency has been recognized in the literature [42, 74, 91]. However, previous approaches have been *ad hoc* in nature, focusing on point cases of specific problems, such as routing, tracking, or sensor coverage. Many of the proposed algorithms have been studied only in simulations, and the few implementations would have required substantial effort to build, given only low-level messaging support provided by the OS. As a result, general-purpose abstractions for coordinated energy management have yet to emerge.

Eon [74] , Levels [42], and ECOSystem [91] provide programming models for adapting to energy availability. Eon provides a dataflow model and automatically tunes timer rates and dataflow paths based on energy availability; application code is not involved in resource management decisions at runtime. Eon attempts to automatically chose states that match the energy consumption to the energy production. While IDEA performs similar energy tracking and forward energy projection, IDEA takes a more explicit approach, allowing the application to decide when it is best to choose a new state. Levels allows application components to define a stack of multiple fidelity levels, which are configured in response to energy availability. Levels is targeted explicitly to systems where no node should fail early, similar to IDEA's "maximize time to first node death" energy objective function. However, Levels is a point solution for a specific problem domain, whereas IDEA is a more general framework. ECOSystem [91] introduces the concept of *Currentcy* to allocate energy resources to tasks to achieve a target lifetime. ECOSystem does not require application code changes, instead tuning OS scheduling parameters automatically based on energy availability, enforcing fairness between applications and tying the application to a specific set of policies. IDEA, on the other hand, allows the user to specify the policy through the energy and utility objective functions. However, the biggest difference between IDEA and these systems is that IDEA takes a distributed, network-wide approach to allocating energy resources while all three of these systems are focused on single-node, not network-wide adaptations.

SORA [54] focuses on decentralized resource allocation based on an economic model in which nodes respond to incentives to produce data or perform specific tasks, with each node trying to maximize its profit for taking a series of actions. While similar to the utility-based decision making in IDEA, nodes in SORA perform purely local, decentralized tuning of their actions, but without any explicit coordination across nodes. IDEA simplifies the problem of global network control through the energy objective function, which directly expresses the application's goal.

Nano-RK [18] provides real-time guarantees through *static* resource reservations based on offline estimates of CPU time, packet rates, and sampling intervals used by an application. However, this approach fails to address dynamically varying load or fluctuations in resource availability that arise at runtime.

Systems such as Odyssey [20], PowerScope [21] and more recently Cinder [70] have addressed measuring or adapting to energy variations on battery-powered devices, primarily to support mobile applications. Odyssey is a framework for adaptive mobile applications that permits applications to adapt to changing energy [22, 49] and other resources. PowerScope maps energy consumption to program structure, producing a prole of energy usage by process and procedure. Cinder is an energy-aware system for mobile computing devices that features a capacitor abstraction associated with tasks. Each capacitor represents a task's right to request energy from the system to perform its operations.

IDEA's approach is different from these systems, targeting networks consisting of multiple nodes being treated as a single entity. Since nodes are collaborating, we can enable more sharing and ask nodes to sacrifice for each other, whereas mobile device users would likely be upset if they discovered that their phone was running low on power because it was trying to improve the lifetime of a stranger's phone located nearby.

Work on energy-aware routing [72, 89] has addressed equitable energy distribution within the network by probabilistically choosing between multiple good paths between each source and sink pair. LEACH [31] and other similar approaches attempt to distribute energy in an entirely decentralized way, using local heuristics to do so. Lexicographically maximum rate allocation [19] uses a decentralized algorithm to tune optimum data collection rates in perpetual networks when static routes are used, all nodes route to a single sink, and the recharging profiles of the nodes are known ahead of time. These are application specific solutions, while IDEA is a more general framework. For example, it would be possible for rate allocation to be implemented in IDEA.

The IDEA architecture emerged from our own prior work on energy management for wireless sensor networks, including Pixie [52], and Peloton [81]. Pixie proposes an operating system and programming framework for sensor network nodes that promotes resources to a first-class primitive, using tickets to manage resource consumption and brokers to enable specialized management policies. Pixie does not consider the energy impact of a node on other nodes.

Peloton proposes an architecture for distributed resource management in sensor networks combining state-sharing, vector tickets to represent distributed resource consumption and a decentralized architecture in which nodes serve as ticket agents managing the resource consumption of themselves and on behalf of nearby nodes. IDEA shares many features with Peloton and can be viewed as the beginnings of an implementation of the Peloton design, with data sharing to enable energy decision making and every node serving as a ticket agent for itself but considering the distributed impact of its own local state.

5.2 Cluster Programming

Resource coordination problems arise in many distributed systems, in particular, distributed execution engines [5, 34, 60]. These systems attempt to handle the difficult aspects of distributed programming – fault tolerance, scheduling, synchronisation and communication – while exposing a simple programming model. Networked embedded systems present new challenges from programming clusters. While sharing the same overall goal as cluster programming – matching work requests to the resources that are able to perform them – embedded systems have vastly different workloads from clusters, typically running a single program throughout the life of the application, with the application being tightly integrated with the physical environment instead of with human operators. In addition there are large differences in communication bandwidth, cost, and latency between clusters and embedded systems. Instead of fast-wired connections for fast low-latency communication within a cluster, embedded systems typically communicate via lossy low-bandwidth wireless channels, where multihop routing can introduce signifiant latency. Further, the communication costs can dominate the energy budgets of these devices requiring new methods for resource coordination to meet these challenges.

River [5] provides adaptive mechanisms that allow database query-processing applications to cope with performance variations in cluster platforms. They propose a dataflow programming model and two new constructs – a *distributed queue* to decouple the dependence of producers of data from consumers and a *graduated declustering mechanism*, which decouples the consumers from producers. River handles massively parallel operations by estimating the performance variations and doing intelligent scheduling to achieve the best performance. However, the workloads are inherently parallel in River, whereas the parallelism in networked embedded systems is achieved by partitioning space. The Karma scheduler is designed around allocating and coordinating resources spatially, relieving this burden from the application programmer.

Applications in Dryad [34] and CIEL [60] are expressed as dataflow graphs. Dryad schedules jobs while optimizing for data locality, network usage, and resource availability. However, the inputs and outputs of these systems are mostly deterministic (a static partitioning of existing data). As embedded systems are tightly coupled with the physical world, applications can experience vastly different workloads in response to changes in the environment, so they must be flexible and adaptable during the course of their execution. This may be hard to express in the simple programming model of systems like Dryad. The Karma programming model avoids the explicit parallelization used in Dryad graphs, allowing processes to be scaled with the size of the number of nodes. Although the process interdependencies in Karma applications can be similarly expressed with dataflow graphs, Karma takes a reactive approach to coordinating system resources, basing decisions on feedback.

CIEL overcomes some of the limitations of Dryad by dynamically building the dataflow graph as tasks execute. CIEL provides support for iterative and recursive computations by allowing the dataflow graph to be modified at runtime using its own scripting language called Skywriting. A single master dispatches tasks to many workers. CIEL it designed for course-grained parallelism across large datasets and may not be suited for the finer grained task execution as nodes respond to changes in the physical environment. CIEL assumes fast and low-cost communication, where the master can easily communicate with all the workers and relies on workers sending periodic heartbeats to demonstrate availability. On a typical sensor node, it costs an order of magnitude more power to turn on the node's radio, so supporting this model on resource constrained devices with lossy wireless links may be energy cost prohibitive. For this reason Karmauses a centralized coordination model where coordination and communication occur at a centralized hive, where the nodes must periodically return to recharge anyway. IDEA also manages this issue with careful energy monitoring and efficient state sharing among nodes.

5.3 Resource Coordination for Swarms

Adding mobility to networked embedded systems is an exciting new frontier for research. Typically sensor nodes must be preinstalled in the area of interest. If the area of interest is uncertain at the time of the deployment, nodes must be over provisioned at a greater cost to ensure adequate coverage. Having the nodes be capable of controlled movement, i.e., able to move based on the needs of the network application, adds adaptability and robustness to the system but requires new ways of thinking about coordination.

Spatially oriented computing offers an alternative approach to swarm coordination. In this paradigm, space is used as a first-class computing abstraction, and individual nodes typically act according to some spatially oriented conditions. Protoswarm [7] is a language that presents the swarm as a single continuous spatial computer. Meld [6] is a declarative-logic programming language to program robotic ensembles. Meld was designed for modular robots where the inter-robot communication is limited to immediate neighbors. Locally distributed predicates [12] are distributed conditions that hold for a connected ensemble of the robotic system. Programs in this paradigm are collections of LDPs with actions that are triggered when subensembles match a particular predicate. Karma partitions space to achieve data-parallel operation.

Swarm robotics algorithms and systems typically focus on emergent behavior arising from local decisions made by large numbers of simple agents. These often biologically inspired algorithms have proven successful in diverse tasks such as collective construction [78] and multiparameter optimization [37]. The large body of work in this area is directly applicable to MAV swarm programming. In designing Karma, we have eschewed the principles of emergent collective behavior in favor of explicit, global coordination based on the hive-drone model. This makes our tasks easier to implement on our MAVs and allows for easier reasoning about application behavior. We employ simple agent behaviors and move most of the complexity to the central hive. Since resource allocation is an iterative, centralized problem, we are able to make a reasoned assessment of swarm progress – something that is often difficult with emergent algorithms.

SensorFly [67] is a MAV sensing platform designed for indoor emergency response applications. The SensorFly nodes are lightweight (29 grams) and low-cost ($200), based on a coaxial helicopter design, with two main rotors, one going clockwise and the other going counter clockwise for stability. These nodes are similar to the helicopters we use in our testbed described in Section 3.3. One important difference is that our testbed helicopters are controlled by a central computer while the SensorFly nodes are designed to be autonomous. For flight control, the SensorFly nodes have several sensors including a three-axis accelerometer, two-axis gyroscope, compass, and ultrasonic rangefinder. A low-power radio is used for broadcasting sensor readings to a basestation and is also used for internode range estimates.

SensorFly applications rely on implicit coordination, using internode ranging estimates to try to disperse themselves throughout an area. Currently, no explicit coordination is done among the nodes. For the simple sense-and-send applications they have implemented, this explicit coordination is enough, but they have yet to tackle more sophisticated applications such as alfalfa monitoring as described in Section 4.1.1. The SensorFly nodes would be an interesting target platform for both IDEA and Karma.

Chapter 6

Future Work and Conclusion

In this dissertation we examined two programming frameworks, IDEA and Karma, and showed with simple programming abstractions, network-wide resource coordination is efficient and useful for programming embedded sensor networks. In support of our research in MAV swarms, we developed Simbeeotic, a simulator and testbed for MAV coordination algorithms. We have been using the testbed for over two years, and it has allowed us to evaluate prototype hardware and explore new concepts with real flight dynamics while also simulating coordination algorithms at scale. This chapter concludes the dissertation by discussing future work for these systems.

6.1 Coordinated Energy Management

In Chapter 2, we described the IDEA architecture in detail, motivated its use through two examples, and demonstrated that IDEA can improve performance by better managing distributed energy resources. We also discussed the process of developing an application-specific energy objective function and showed how this can improve the performance of a localization application while maintaining application fidelity.

For future work we are interested in addressing the problem of cross-component interaction to optimize several IDEA components running simultaneously. This is complicated by the fact that there are likely to be dependencies between components that cause decisions made by one to affect others. As an example, the LPL intervals used by a node would affect the power cost to use the link seen by the routing protocol. In addition we are investigating ways to model the impact of node failure on other nodes. Many sensor network protocols will try to work around nodes leaving the network or going offline, but this repair process is costly and causes load within the network to shift.

With IDEA, node energy tracking is done entirely by software that we developed for the Pixie project. Quanto [23] provides a framework for tracking and understanding energy consumption in embedded sensor systems. The existence of systems such as Quanto was a primary motivation for IDEA, since the visibility distributed resource tracking provides creates an opportunity to adapt to changes in availability across the network. A software-only approach may be difficult for components with complex behavior. An interesting area of future work would be integrating Quanto into IDEA to provide more precise tracking of energy at runtime, which could eliminate the need for component-specific modeling and ease the process of integrating applications with IDEA.

6.2 Resource Coordination for Swarms

As our research moved to the even more resource constrained world of insect-sized MAV swarms, we found that in order to simulate the swarm at scale, a new class of simulator was needed. Chapter 3 described Simbeeotic, our simulator and testbed for testing MAV coordination algorithms.

We demonstrated, through our own experiments and using Simbeeotic for our own research, that Simbeeotic provides the appropriate level of fidelity to evaluate prototype hardware while maintaining the ability to test at scale.

There are three main directions for future work on Simbeeotic – scalability, fidelity, and autonomy. From the results in Section 3.4.1 it is clear that the the physics engine can be a bottleneck. We rely heavily on JBullet for modeling actuation (dynamics, collision detection) and sensing (ray tracing). Though it has satisfied our needs thus far, we may consider replacing JBullet with Bullet [8] as we move toward modeling swarms with tens of thousands of MAVs. JBullet is a pure Java port of Bullet, which is written in C++. In addition to being written in a native language, newer versions of Bullet support hardware acceleration on the GPU. The potential performance improvement may be worth the modest engineering effort to create Java wrappers for the subset of the Bullet interfaces used by Simbeeotic.

Though we model the breadth of the MAV swarm domain, the fidelity of the networking models in Simbeeotic could be improved. To date, our work on MAV swarms has not focused on communication. It is likely that the networking interfaces will need to evolve beyond the simple physical layer implementation. We will look to leverage community standard tools and models such as ns-3 as our needs develop. In addition, we may expand our HWIL capabilities to include real radios in a mote testbed, much like in EmStar [26]. On first inspection, it appears that the ghost model approach will work well with a radio interface. Packets sent on a ghost interface would be transmitted on the physical radio in addition to the virtual radio, and packets received on the physical radio would be captured and injected as a virtual packet reception. Some care must be taken to prevent duplicate transmission and reception events by ghost radio models participating in both domains.

As we develop the software stack that will execute on the autonomous MAVs, it may be possible to leverage the Robot Operating System (ROS), described in Section 3.1. This presents an opportunity for Simbeeotic to be used as a virtual input to software that will be embedded on a vehicle. We view this TOSSIM-like approach as another (purely simulated) intermediate step toward MAV autonomy that is orthogonal to HWIL operation.

A natural extension of the testbed would be to add a third stage to the staged deployment, where MAV control is still done through Simbeeotic, but using real sensor input instead of data from the virtual sensors. With such a deployment we can no longer rely on a purely simulated environment. Obstacles must be represented in both worlds, a feat that can be accomplished by adding Vicon markers to physical objects and creating a corresponding ghost representation in the simulation. In this case we would feed physical sensor information wirelessly Simbeeotic, replacing the input from the virtual sensor. To test the feasibility of this idea, we have attached a Texas Instruments EZ430-RF2500 [2] development board to one of our MAVs. The EZ430-RF2500 is a small and lightweight development board with MSP430 microcontroller and low-power 2.4 GHz radio. The MAV was able to fly with this board attached to it without any noticeable control issues; however the overall flight time was reduced by 15-20%. We were able to send data from the onboard radio to our testbed computer without difficultly, giving us confidence that a sensor (e.g., digital compass) with an update rate of 50 Hz or less would be possible to implement in this fashion.

Our testbed currently depends on having an accurate motion capture system. On our path toward fully autonomous MAVs, we may relax the requirement that physical objects and virtual objects are covisible. Instead, we could construct a virtual world to match the physical world and ignore interactions between MAVs. This would allow us to experiment outside of the testbed and obviate the need for accurate tracking once the MAVs are fully autonomous (other than for ground truth during experimentation). As discussed in Section 3.3.3, we have used a Microsoft Kinect sensor that enables one of our helicopters to hover in a small area. Eventually, we hope this system can be expanded with more Kinect sensors and be used for collecting ground truth positional information of indoor exploration experiments. Simbeeotic would remain as a useful tool, allowing physical MAVs to coordinate with simulated MAVs in the communication layer.

We demonstrated in Chapter 4 that Karma is capable of executing swarm applications with reasonable efficiency and resiliency for the emerging class of MAV swarm mobile sensing systems. Our novel system architecture based on the hive-drone model simplifies programming individual MAVs and shifts the coordination complexity to a central hive, and we show that Karma is efficient, adaptive, and resilient to failure.

However, some aspects of the system design warrant further discussion and study. In our design each drone is assigned a behavior to execute per sortie. This policy is a direct result of the short flight times of the current MAV prototype. If the drones had longer flight times, multitasking on individual drones might allow for more efficient operation. Further, we only consider sorties that keep the drones deployed for the maximum amount of time. It may be possible to reduce information latency and gain more flexibility in scheduling by considering variable-length sorties. In addition, a larger area could be covered if a multihive solution were adopted (with interhive communication), allowing for one-way sorties that redistribute the swarm's resources at runtime. For latency-sensitive applications, such as target tracking, in-field coordination might allow the application to react more quickly to changes in the environment. These features would require significant modifications to the

Scheduler to incorporate resource planning

The Karma scheduler relies on estimating the progress rates of drones executing behaviors in the field. Though this allows the system to adapt to varying conditions, it can be problematic when the estimate is inaccurate. It may be possible to produce more robust estimates by incorporating values from neighboring regions or using a priori models. An overprovisioning strategy could mitigate the impact of overestimation (or underperforming drones) on completion time at an additional cost in resources.

Bibliography

[1] http://robobees.seas.harvard.edu.

[2] http://www.ti.com/tool/ez430-rf2500.

[3] Andreas M. Ali, Kung Yao, Travis C. Collier, Charles E. Taylor, Daniel T. Blumstein, and Lewis Girod. An empirical study of collaborative acoustic source localization. In *IPSN '07: Proceedings of the 6th international conference on Information processing in sensor networks*, Cambridge, MA, 2007.

[4] Kiam Heong Ang, Gregory Chong, and Yun Li. PID control system analysis, design, and technology. *IEEE Transactions on Control Systems Technology*, 13(4):559–576, July 2005.

[5] Remzi H. Arpaci-Dusseau. Run-time adaptation in River. *ACM Transactions on Computer Systems (TOCS)*, 21(1):36–86, February 2003.

[6] Michael P. Ashley-Rollman, Seth Copen Goldstein, Peter Lee, Todd C. Mowry, and Padmanabhan Pillai. Meld: A declarative approach to programming ensembles. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, October 2007.

[7] Jonathan Bachrach, James McLurkin, and Anthony Grue. Protoswarm: a language for programming multi-robot systems using the amorphous medium abstraction. In *Proceedings of the 7th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pages 1175–1178, 2008.

[8] Bullet Physics Library. http://bulletphysics.org/wordpress.

[9] Nicolas Burri, Pascal von Rickenbach, and Roger Wattenhofer. Dozer: ultra-low power data gathering in sensor networks. In *IPSN '07: Proceedings of the 6th international conference on Information processing in sensor networks*, pages 450–459, New York, NY, USA, 2007. ACM.

[10] Geoffrey Werner Challen. *Data fidelity and resource management for data-rich sensor networks*. PhD thesis, Harvard University, Cambridge, MA, USA, 2010. AAI3414646.

[11] Karthik Dantu, Bryan Kate, Jason Waterman, Peter Bailis, and Matt Welsh. Programming micro-aerial vehicle swarms with karma. In *Proceedings of the 9th ACM Conference on Embedded Networked Sensor Systems*, SenSys '11, pages 121–134, New York, NY, USA, 2011. ACM.

[12] Michael De Rosa, Seth C. Goldstein, Peter Lee, Padmanabhan Pillai, and Jason Campbell. Programming modular robots with locally distributed predicates. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pages 3156–3162, May 2008.

[13] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proceedings of 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 137–150, 2004.

[14] Keith S. Delaplane and Daniel F. Mayer. *Crop Pollination by Bees*. CABI Publishing, New York, NY, 2000.

[15] Amit Dhariwal, Gaurav S. Sukhatme, and Aristides A.G. Requicha. Bacterium-inspired robots for environmental monitoring. In *Proceedings of the 2004 IEEE International Conference on Robotics and Automation (ICRA '04)*, volume 2, pages 1436–1443, May 2004.

[16] Prabal Dutta, Jonathan Hui, Jaein Jeong, Sukun Kim, Cory Sharp, Jay Taneja, Gilman Tolle, Kamin Whitehouse, and David Culler. Trio: Enabling sustainable and scalable outdoor wireless sensor network deployments. In *Proceedings of the 5th international conference on Information processing in sensor networks*, IPSN '06, pages 407–415, New York, NY, USA, 2006. ACM.

[17] E-flite Blade MCX2. `http://www.e-fliterc.com`.

[18] Anand Eswaran, Anthony Rowe, and Raj Rajkumar. Nano-RK: An energy-aware resource-centric RTOS for sensor networks. In *Proc. IEEE Real-Time Systems Symposium*, December 2005.

[19] Kai-Wei Fan, Zizhan Zheng, and Prasun Sinha. Steady and fair rate allocation for rechargeable sensors in perpetual sensor networks. In *SenSys '08: Proceedings of the 6th ACM conference on Embedded network sensor systems*, pages 239–252, New York, NY, USA, 2008. ACM.

[20] Jason Flinn and M. Satyanarayanan. Energy-aware adaptation for mobile applications. *SIGOPS Oper. Syst. Rev.*, 33(5):48–63, 1999.

[21] Jason Flinn and M. Satyanarayanan. Powerscope: A tool for profiling the energy usage of mobile applications. In *WMCSA '99: Proceedings of the Second IEEE Workshop on Mobile Computer Systems and Applications*, page 2, Washington, DC, USA, 1999. IEEE Computer Society.

[22] Jason Flinn and M. Satyanarayanan. Managing battery lifetime with energy-aware adaptation. *ACM Transactions on Computer Systems (TOCS)*, 22(2), May 2004.

[23] Rodrigo Fonseca, Prabal Dutta, Philip Levis, and Ion Stoica. Quanto: Tracking energy in networked embedded systems. In *OSDI*, pages 323–338, 2008.

[24] Rodrigo Fonseca, Omprakash Gnawali, Kyle Jamieson, Sukun Kim, Philip Levis, and Alec Woo. The collection tree protocol (ctp). TinyOS Extension Proposal TEP-123, `http://www.tinyos.net/tinyos-2.x/doc/txt/tep123.txt`, 2006.

[25] Brian P. Gerkey, Richard T. Vaughan, and Andrew Howard. The player/stage project: Tools for multi-robot and distributed sensor systems. In *Proceedings of the International Conference on Advanced Robotics (ICAR 2003)*, pages 317–323, 2003.

[26] Lewis Girod, Jeremy Elson, Alberto Cerpa, Thanos Stathopoulos, Nithya Ramanathan, and Deborah Estrin. EmStar: A software environment for developing and deploying wireless sensor networks. In *Proceedings of the annual conference on USENIX Annual Technical Conference (ATEC '04)*. USENIX Association, 2004.

[27] Google Guice. `http://code.google.com/p/google-guice`.

[28] Google Protocol Buffers. `http://code.google.com/apis/protocolbuffers`.

[29] Chih-Chieh Han, Ram Kumar Rengaswamy, Roy Shea, Eddie Kohler, and Mani Srivastava. SOS: A dynamic operating system for sensor networks. In *Proc. Third International Conference on Mobile Systems, Applications, And Services (Mobisys)*, 2005.

[30] Sabine Hauert, Laurent Winkler, Jean-Christophy Zufferey, and Dario Floreano. Ant-based swarming with positionless micro air vehicles for communication relay. *Swarm Intelligence*, 2(2-4):167–188, 2008.

[31] Wendi Heinzelman, Anantha Chandrakasan, and Hari Balakrishnan. Energy-efficient communication protocol for wireless microsensor networks. In *Proc. The 33rd Hawaii International Conference on System Sciences (HICSS)*, January 2000.

[32] David Hiebeler. The Swarm simulation system and individual-based modeling. In *Proceedings of Decision Support 2001: Advanced Technologies for Natural Resource Management*, Toronto, September 1994.

[33] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David E. Culler, and Kristofer S. J. Pister. System architecture directions for networked sensors. In *Proc. the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 93–104, Boston, MA, USA, November 2000.

[34] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems*, 2007.

[35] JBullet. `http://jbullet.advel.cz`.

[36] Bryan Kate, Jason Waterman, Karthik Dantu, and Matt Welsh. Simbeeotic: a simulator and testbed for micro-aerial vehicle swarm experiments. In *Proceedings of the 11th international conference on Information Processing in Sensor Networks*, IPSN '12, pages 49–60, New York, NY, USA, 2012. ACM.

[37] James Kennedy and Russell Eberhart. Particle swarm optimization. In *Proceedings of IEEE International Conference on Neural Networks*, 1995.

[38] Oussama Khatib. Real-time obstacle avoidance for manipulators and mobile robots. *International Journal of Robotics Research*, 5(1):90–98, April 1986.

[39] Sukun Kim, Rodrigo Fonseca, Prabal Dutta, Arsalan Tavakoli, David Culler, Philip Levis, Scott Shenker, and Ion Stoica. Flush: A Reliable Bulk Transport Protocol for Multihop Wireless Networks. In *Proc. SenSys'07*, 2007.

[40] Sukun Kim, Shamim Pakzad, David Culler, James Demmel, Gregory Fenves, Steve Glaser, and Martin Turon. Wireless sensor networks for structural health monitoring. In *Proceedings of the 4th international conference on Embedded networked sensor systems*, SenSys '06, pages 427–428, New York, NY, USA, 2006. ACM.

[41] Jon Klein. BREVE: A 3d simulation environment for the simulation of decentralized systems and artificial life. In *Proceedings of Artificial Life VIII, the 8th International Conference on the Simulation and Synthesis of Living Systems*, 2002.

[42] Andreas Lachenmann, Pedro Jose Marron, Daniel Minder, and Kurt Rothermer. Meeting lifetime goals with energy levels. In *Proc. ACM SenSys*, November 2007.

[43] Dimitrios Lambrinos, Ralf Moller, Thomas Labhart, Rolf Pfeifer, and Rudiger Wehner. A mobile robot employing insect strategies for navigation. *Robotics and Autonomous Systems*, 30(1-2):39–64, 2000.

[44] HyungJune Lee, Alberto Cerpa, and Philip Levis. Improving wireless simulation through noise modeling. In *IPSN '07: Proceedings of the 6th international conference on Information processing in sensor networks*, pages 21–30, New York, NY, USA, 2007. ACM.

[45] Philip Levis, Nelson Lee, Matt Welsh, and David Culler. TOSSIM: Accurate and scalable simulation of entire TinyOS applications. In *Proc. the First ACM Conference on Embedded Networked Sensor Systems (SenSys 2003)*, November 2003.

[46] Philip Levis, Nelson Lee, Matt Welsh, and David Culler. TOSSIM: accurate and scalable simulation of entire tinyos applications. In *Proceedings of the 1st international conference on Embedded networked sensor systems (SenSys '03)*, November 2003.

[47] Philip Levis, Neil Patel, Scott Shenker, and David Culler. Trickle: A self-regulating algorithm for code propagation and maintenance in wireless sensor networks. In *Proc. the First USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI)*, 2004.

[48] Ming Li, Deepak Ganesan, and Prashant Shenoy. Presto: feedback-driven data management in sensor networks. *IEEE/ACM Trans. Netw.*, 17(4):1256–1269, 2009.

[49] Xiaotao Liu, Prashant Shenoy, and Mark D. Corner. Chameleon: Application level power management. *IEEE Transactions on Mobile Computing*, 2008.

[50] Konrad Lorincz, Bor-rong Chen, Geoffrey Werner Challen, Atanu Roy Chowdhury, Shyamal Patel, Paolo Bonato, and Matt Welsh. Mercury: a wearable sensor network platform for high-fidelity motion analysis. In *SenSys '09: Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*, pages 183–196, New York, NY, USA, 2009. ACM.

[51] Konrad Lorincz, David Malan, Thaddeus R. F. Fulford-Jones, Alan Nawoj, Antony Clavel, Victor Shnayder, Geoff Mainland, Steve Moulton, and Matt Welsh. Sensor Networks for Emergency Response: Challenges and Opportunities. *IEEE Pervasive Computing*, Oct-Dec 2004.

[52] Konrad Lorincz, Bor rong Chen, Jason Waterman, Geoff Werner-Allen, and Matt Welsh. Resource aware programming in the pixie os. In *SenSys '08: Proceedings of the 6th ACM conference on Embedded network sensor systems*, pages 211–224, New York, NY, USA, 2008. ACM.

[53] Sean Luke, Claudio Cioffi-Revilla, Liviu Panait, and Keith Sullivan. MASON: A new multi-agent simulation toolkit. In *Proceedings of the 2004 SwarmFest Workshop*, 2004.

[54] Geoffrey Mainland, David C. Parkes, and Matt Welsh. Decentralized, adaptive resource allocation for sensor networks. In *NSDI'05: Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation*, pages 315–328, Berkeley, CA, USA, 2005. USENIX Association.

[55] Nathan Michael, Jonathan Fink, and Vijay Kumar. Cooperative manipulation and transportation with aerial robots. *Autonomous Robots*, 30(1):73–86, September 2010.

[56] Nathan Michael, Daniel Mellinger, Quentin Lindsey, and Vijay Kumar. The GRASP multiple micro UAV testbed. *Robotics & Automation Magazine, IEEE*, 17(3):56–65, 2010.

[57] Olivier Michel. WebotsTM: Professional mobile robot simulation. *International Journal of Advanced Robotic Systems*, 1(1):40–43, 2004.

[58] Microsoft Kinect. `http://www.xbox.com/kinect`.

[59] David Moss, Jonathan Hui, and Kevin Klues. Low power listening. TinyOS Extension Proposal TEP-105, `http://www.tinyos.net/tinyos-2.x/doc/txt/tep105.txt`, 2007.

[60] Derek G. Murray, Malte Schwarzkopf, Christopher Smowton, Steven Smith, Anil Madhavapeddy, and Steven Hand. Ciel: a universal execution engine for distributed data-flow computing. In *Proceedings of the 8th USENIX conference on Networked systems design and implementation*, NSDI'11, pages 9–9, Berkeley, CA, USA, 2011. USENIX Association.

[61] Dragos Niculescu and Badri Nath. Ad hoc positioning system (APS) using AOA. In *INFOCOM*, pages 1734–1743, 2003.

[62] ns-3. `http://www.nsnam.org`.

[63] American Society of Civil Engineers, American Congress on Surveying, Mapping, American Society for Photogrammetry, and Remote Sensing. *Glossary of the Mapping Sciences*. American Society of Civil Engineers, 1994.

[64] Joseph Polastre, Jason Hill, and David Culler. Versatile low power media access for wireless sensor networks. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 95–107, New York, NY, USA, 2004. ACM.

[65] Joseph Polastre, Robert Szewczyk, and David Culler. Telos: enabling ultra-low power wireless research. In *Proceedings of the 4th international symposium on Information processing in sensor networks*, IPSN '05, Piscataway, NJ, USA, 2005. IEEE Press.

[66] Dimitris Psychoudakis, William Moulder, Chi-Chih Chen, Heping Zhu, and John L. Volakis. A portable low-power harmonic radar system and conformal tag for insect tracking. *IEEE Antennas and Wireless Propagation Letters*, 7:444–447, 2008.

[67] Aveek Purohit, Zheng Sun, Memo Salas, and Pei Zhang. Sensorfly: Controlled-mobile sensing platform for indoor emergency response applications. In *Proceedings of the 10th International Conference on Information Processing in Sensor Networks (IPSN)*, 2011.

[68] Razvan, Chieh, and Andreas Terzis. Koala: Ultra-low power data retrieval in wireless sensor networks. In *IPSN '08: Proceedings of the 7th international conference on Information processing in sensor networks*, pages 421–432, Washington, DC, USA, 2008. IEEE Computer Society.

[69] Robot Operating System. `http://www.ros.org`.

[70] Stephen M. Rumble, Ryan Stutsman, Philip Levis, David Mazières, and Nickolai Zeldovich. Apprehending joule thieves with cinder. In *MobiHeld '09: Proceedings of the 1st ACM workshop on Networking, systems, and applications for mobile handhelds*, pages 49–54, New York, NY, USA, 2009. ACM.

[71] G. Salton, A. Wong, and C. S. Yang. A vector space model for automatic indexing. *Commun. ACM*, 18(11):613–620, November 1975.

[72] Rahul C. Shah and Jan M. Rabaey. Energy aware routing for low energy ad hoc sensor networks. In *IEEE Wireless Communications and Networking Conference (WCNC)*, March 2002.

[73] Gyula Simon, Miklós Maróti, Ákos Lédeczi, György Balogh, Branislav Kusy, András Nádas, Gábor Pap, János Sallai, and Ken Frampton. Sensor network-based countersniper system. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 1–12, New York, NY, USA, 2004. ACM.

[74] Jacob Sorber, Alex Kostadinov, Matt Brennan, Matt Garber, Mark Corner, and Emery D. Berger. Eon: A Language and Runtime System for Perpetual Systems. In *Proc. ACM SenSys*, November 2007.

[75] Erik Stokstad. The case of the empty hives. *Science*, 316(5827):970–2, 2007.

[76] Robert Szewczyk, Alan Mainwaring, Joseph Polastre, John Anderson, and David Culler. An analysis of a large scale habitat monitoring application. In *In Proceedings of the Second ACM Conference on Embedded Networked Sensor Systems (SenSys*, pages 214–226, 2004.

[77] Jay Taneja, Jaein Jeong, and David Culler. Design, modeling, and capacity planning for microsolar power sensor networks. In *IPSN '08: Proceedings of the 7th international conference on Information processing in sensor networks*, pages 407–418, Washington, DC, USA, 2008. IEEE Computer Society.

[78] Guy Theraulaz and Eric Bonabeau. Coordination in distributed building. *Science*, 269(5224):686–688, 1995.

[79] Richard T. Vaughan. Massively multi-robot simulations in stage. *Swarm Intelligence*, 2(2-4):189–208, 2008.

[80] Vicon Motion Capture Sytems. `http://www.vicon.com`.

[81] Jason Waterman, Geoffrey Werner Challen, and Matt Welsh. Peloton: Coordinated resource management for sensor networks. In *Proc. the 12th Workshop on Hot Topics in Operating Systems (HotOS-XII)*, May 2009.

[82] Jason Waterman, Bryan Kate, Karthik Dantu, and Matt Welsh. Simbeeotic: a simulation-emulation platform for large scale micro-aerial swarms. In *Proceedings of the 11th international conference on Information Processing in Sensor Networks*, IPSN '12, pages 139–140, New York, NY, USA, 2012. ACM.

[83] Matt Welsh and Geoff Mainland. Programming sensor networks using abstract regions. In *Proc. the First USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI '04)*, March 2004.

[84] Geoff Werner-Allen, Konrad Lorincz, Jeff Johnson, Jonathan Lees, and Matt Welsh. Fidelity and yield in a volcano monitoring sensor network. In *Proc. 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2006)*, Seattle, WA, November 2006.

[85] Geoff Werner-Allen, Pat Swieskowski, and Matt Welsh. MoteLab: A Wireless Sensor Network Testbed. In *Proc. the Fourth International Conference on Information Processing in Sensor Networks (IPSN'05)*, April 2005.

[86] Geoffrey Werner-Allen, Stephen Dawson-Haggerty, and Matt Welsh. Lance: Optimizing high-resolution signal collection in wireless sensor networks. In *Proc. ACM Conference on Embedded Networked Sensor Systems (Sensys)*, Raleigh, NC, USA, November 2008.

[87] Kamin Whitehouse, Cory Sharp, Eric Brewer, and David Culler. Hood: A neighborhood abstraction for sensor networks. In *Proc. the International Conference on Mobile Systems, Applications, and Services (MOBISYS '04)*, June 2004.

[88] Alec Woo, Terence Tong, and David Culler. Taming the underlying challenges of reliable multihop routing in sensor networks. In *Proc. the First ACM Conference on Embedded Networked Sensor Systems (SenSys 2003)*, November 2003.

[89] Ya Xu, John Heidemann, and Deborah Estrin. Geography-informed energy conservation for ad hoc routing. In *MobiCom '01: Proceedings of the 7th annual international conference on Mobile computing and networking*, pages 70–84, New York, NY, USA, 2001. ACM.

[90] Wei Ye, J. Heidemann, and D. Estrin. An energy-efficient mac protocol for wireless sensor networks. In *INFOCOM 2002. Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 3, pages 1567 – 1576 vol.3, 2002.

[91] Heng Zeng, Xiaobo Fan, Carla S. Ellis, Alvin Lebeck, and Amin Vahdat. ECOSystem: Managing Energy as a First Class Operating System Resource. In *Proc. Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, San Jose, CA, October 2002.

[92] Xiang Zeng, Rajive Bagrodia, and Mario Gerla. GloMoSim: a library for parallel simulation of large-scale wireless networks. In *Proceedings of the twelfth workshop on Parallel and distributed simulation (PADS '98)*, pages 154–161. ACM, July 1998.