12-1-2012

Fault-tolerant wireless sensor networks using evolutionary games

Ricardo Villalon

Ricardo Villalon

Candidate

Computer Science

Department

This dissertation is approved, and it is acceptable in quality and form for publication:

Approved by the Dissertation Committee:

Patrick G. Bridges, Chairperson

Melanie Moses

David Ackley

Thomas Caudell

Fault-Tolerant Wireless Sensor Networks using Evolutionary Games

by

Ricardo Villalón-Fonseca

B.S., Computer Science, Universidad de Costa Rica, 1989
M.S., Computer Science, Universidad de Costa Rica, 2002

DISSERTATION

Submitted in Partial Fulfillment of the
Requirements for the Degree of

Doctor of Philosophy
Computer Science

The University of New Mexico

Albuquerque, New Mexico

December, 2012

Fault-Tolerant Wireless Sensor Networks using Evolutionary Games

by

Ricardo Villalón-Fonseca

B.S., Computer Science, Universidad de Costa Rica, 1989
M.S., Computer Science, Universidad de Costa Rica, 2002

Ph.D., Computer Science, University of New Mexico, 2012

Abstract

This dissertation proposes an approach to creating robust communication systems in wireless sensor networks, inspired by biological and ecological systems, particularly by evolutionary game theory. In this approach, a virtual community of agents live inside the network nodes and carry out network functions. The agents use different strategies to execute their functions, and these strategies are tested and selected by playing evolutionary games. Over time, agents with the best strategies survive, while others die. The strategies and the game rules provide the network with an adaptive behavior that allows it to react to changes in environmental conditions by adapting and improving network behavior.

To evaluate the viability of this approach, this dissertation also describes a micro-component framework for implementing agent-based wireless sensor network services, an evolutionary data collection protocol built using this framework, ECP, and experiments evaluating the performance of this protocol in a faulty environment. The framework addresses many of the programming challenges in writing network software for wireless sensor networks, while the protocol built using the framework provides a means of evaluating the general viability of the agent-based approach.

The results of this evaluation show that an evolutionary approach to designing wireless sensor networks can improve the performance of wireless sensor network protocols in the presense of node failures. In particular, we compared the performance of ECP with a non-evolutionary rule-based variant of ECP. While the purely-evolutionary version of ECP has more routing timeouts than the rule-based approach in failure-free networks, it sends significantly fewer beacon packets and incurs statistically fewer routing timeouts in both simple fault and periodic fault scenarios.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Wireless Sensor Networks (WSNs) [21, 43] are networks composed of small independent electronic devices, with environmental sensing capabilities and wireless networking to share collected information. They provide data in a broad range of fields using sensors such as temperature, humidity, visible light, infrared light, acoustic, vibration, pressure, chemical, mechanical stress, magnetic, and more. WSNs are used in a wide range of applications, including disaster relief applications [8, 9, 84, 78], environmental control [62], biodiversity mapping [54, 74, 18], and structural health monitoring [44, 63].

WSN applications have complex communication demands [25], requiring information processing inside the network and detailed control over the sensor nodes. Applications typically require periodic sensing of environmental events [13], and dynamically adjust sampling frequency. The collected data is usually sent to a main location, and many applications process data inside the network to produce summarized values before sending them to a destination [65].

Maintaining node communication in WSNs is challenging [70]. Most WSN applications require long-term operation with high levels of survivability. WSNs are deployed in dynamic failure-prone environments with harsh environmental conditions and physical failures of the devices [22, 75]. The network degrades over time because of device failures, environmental changes, or other external factors, and faults arise at different times and locations, affecting groups of nodes, single nodes, or the link between two nodes.

This dissertation proposes an approach to creating robust communication systems in WSNs, inspired by biological and ecological systems [55] particularly by evolutionary game theory [56]. In this approach, a virtual community of agents live inside the network nodes and carry out network functions. The agents use different strategies to execute their functions, and these strategies are tested and selected by playing evolutionary games. Over time, agents with the best strategies survive, while others die. The strategies and game rules provide the network with an adaptive behavior that allows it to react to changes in environmental conditions by adapting and improving network behavior.

1.1 Challenges in Constructing WSNs

This section describes the general challenges faced by sensor network software, and also describes the specific impact they have on network communication and software implementation. The broad range of applications for WSNs can not be implemented with a single network topology or software system [3]. However many of the applications share a common set of challenges, and realizing new ways to overcome the challenges is an important step for the development of WSNs.

1.1.1 General Challenges

Power management and fault tolerance are the two most significant challenges faced by WSNs. In addition, the diverse set of WSN deployments makes flexibility an important challenge. Details about these general challenges are provided below.

Power management. Power management is important in WSNs because nodes are generally battery powered, and this restricts the lifetime of the nodes and the whole network. Replacing batteries in the field is frequently either infeasible or very expensive. Furthermore, there is a trade-off between communication quality and battery lifetime because increasing communication quality usually requires more energy, at the cost of decreased sensor lifetime [46, 1]. New techniques to balance these two quantities are an important issue studied in this dissertation.

Fault tolerance. Failures are generally common in WSNs: nodes may run out of battery, suffer a hardware fault, or environmental conditions can block a communication link. As a result, some nodes can be disconnected temporarily or permanently. In addition, repairing failures can be challenging because it can be very expensive, or replacement of sensors in the field can be infeasible.

Flexibility. Applications can be different in the type of sensors they use to sample the environment, the frequency of sampling, the number of nodes in the network, the environmental conditions where the nodes are deployed, the communication scheme between the nodes, and more. In addition, operational requirements of the nodes can change over time. As a result, sensor network software must be flexible enough to adapt to different deployment scenarios and changing network conditions.

1.1.2 Network Communication Challenges

Network communication challenges are directly related to the general challenges in the previous section. Maintaining communication requires up-to-date routing tables, and the packet transmissions needed for this consume power, which impacts the network lifetime. On the other hand, frequent transmissions are required to detect changes in network topology or communication conditions. This makes balancing the power cost of packet transmissions with the potential improved communication quality important and challenging.

Fault tolerance is also important in WSN communication systems because data delivery in WSNs is inherently faulty [81]. Node failures and lost packets can cause previously working communication routes to fail. Detecting and recovering from these failures requires additional communication, consuming additional node power. As a result, faults and failures make balancing communication quality and power consumption very challenging.

1.1.3 Programming Challenges

Software and hardware capabilities of WSNs are different from conventional networked system. WSNs have constrained hardware resources on the sensor devices and this imposes important restrictions on software design.

The IRIS sensor mote from MEMSIC Inc.[57] is a good example of a low-cost WSN. It has a 16MHz processor with 128 Kbytes of flash memory to store the operating system and application software, 8 Kbytes of RAM for program data, and 512 Kbytes of serial low-speed memory to collect sensor samples. The operating system platform is TinyOS, developed by UC Berkeley [48].

With this hardware configuration, program instructions and the operating system must fit in 128 Kbytes as opposed to the 4 Gbytes of RAM available in a regular desktop computer. There is no support in the language for dynamic linking, and no support for function pointers. Also, some dynamic behaviors available in conventional programming environments, such as virtual functions, are not available in a sensor network platform. Similarly, memory for data is only 8 Kbytes, and there is no support in the operating system for allocating memory dynamically.

1.2 A WSN Communication Example

To better illustrate the communication challenges in WSNs, consider a WSN application that collects data from the environment and communicates it to the outside world. In this application, some nodes gather data from the environment using the sensors, and the collected data is sent to other nodes for export. We refer to nodes gathering information as sources, and nodes collecting gathered information as sinks.

To send data from sources to sinks, each node collects and maintains information about other nodes in the neighborhood, specifically a record of the number of successful and failed communications with them and how far each node is from a sink. The successful/failed values provide a quality measurement on the communication, and this, along with how close each node is to a sink, is used to decide to which neighbor to route gathered data.

The neighborhood information is maintained using a very simple communication scheme. Each node periodically advertises its location to the network by sending a beacon packet to all of its neighbors; the beacon contains information about its quality to communicate with the neighbors and its distance to a sink. This simple communication example can be used to demonstrate the challenges and trade-offs in WSNs.

A higher frequency of beacon packets keeps the neighborhood information updated, but at the same time may increase the number of packet collisions because more packets are sent simultaneously per unit time. A lower frequency of beacon packets may decrease packet collisions, but it may also increase the time to receive new information about costs between neighbors because costs are calculated from quality information, and a good quality value will take more time to show because less packets are sent.

Sending more beacons when the network starts running or when the network topology changes is preferable because stable quality values are calculated faster, but the quality value obtained could be lower than expected because of more packet collisions when trying to calculate the quality faster. On the other hand, sending fewer beacons when quality values are stable can help to reduce energy usage, but also makes it hard to quickly detect quality changes or node failures.

1.3 Optimizing WSN Communication

This dissertation describes an approach to optimizing WSN communication in failure-prone environments using an agent-based model. This agent-based approach addresses the challenges described in Section 1.1 and the trade-offs described in the communication example of Section 1.2.

1.3.1 Agent-based Approach

The approach to optimizing WSN communication proposed in this dissertation is based on using *agents* residing in WSN nodes to execute network functions. Each agent contains one or more parameters related to network communication that needs to be optimized, for example the rate at which to send a beacon. When a network function needs to be executed, the node selects an agent to perform that function, which the agent executes based on its parameters. In addition, agents may move between nodes or replicate on other nodes as part of executing this network functionality. Finally, agents use their parameters to compete with other agents to survive in the network. Agents with parameters that consistently perform better compared to other agents survive, changing and optimizing network performance.

The first step to optimizing a WSN application with an agent-based approach consists of identifying the parameters and function to be improved or optimized. These parameters are then assigned to agents. For example, using the example in Section 1.2, the beacon time parameter and beacon transmission function can be assigned to an agent. The second step in this process is determining how and when these agents are created, moved, and replicated in the system; for example, beacon time agents may move to or replicate on other nodes when those nodes receive beacon packets.

The final step in this process is to construct a game between agents that compares how well they perform their assigned function based on their parameter values. The structure of this game is specific to the parameter being optimized, but its outcome may result in the destruction of either of the agents in the competition based on their performance in the game. In our example, the beacon agent can be evaluated when it is received at some receiving node. This competition can be, for example, with a randomly selected agent already at the node, and could be based on the quality of the information the new beacon contains and how quickly it arrived.

1.3.2 Evolutionary Games

Our agent-based approach comprises an *evolutionary game* as defined in [56]*p.10-27. An evolutionary game consists of *players*, *strategy sets*, *strategies*, and *payoffs*.

It also assumes an infinite random-mixing population, asexual reproduction, and symmetric and asymmetric pairwise contests.

In the agent-based system the dissertation describes, agents are the players of the evolutionary game, strategy sets and strategies correspond to the network parameters being optimized and their specific values, and payoffs are defined by the rules of the competitions between agents. Our system provides an infinite random-mixing population over time through random agent creation, reproduction of agents, and movement of agents between network nodes. The pairwise competition between agents directly correspond to the competitions in evolutionary games. A more complete discussion of this agent-based approach in the evolutionary game context is provided in Chapter 3.

1.3.3 Micro-component Framework

The agent-based model and the evolutionary games are supported by a software framework that we call the micro-component framework. Functions executed by agents and the interactions for the evolutionary game require software support because agents execute their functions in several different ways, and the nodes must dynamically execute strategies for agents while at the same time respecting a correct execution according to the application requirements.

TinyOS does not have support for this dynamic behavior. It does not provide a convenient mechanism to add and remove new implementations for agent functions or new interaction rules for games between agents to improve the network. To address this, Chapter 4 presents a micro-component framework supporting an agent-based evolutionary game approach.

1.4 Thesis Statement

My thesis is that power consumption and node connectivity in WSNs in the presence of failures can be improved by implementing routing protocols as evolutionary games.

To evaluate this thesis, I examine the performance of both evolutionary and non-evolutionary variants of a wireless sensor network routing protocol in the presence of node reboot faults. The performance of these protocols is evaluated based on their ability to maintain communication with neighbors and on the amount of power they consume over the course of a test.

The remainder of this dissertation describes the general approach to implementing wireless sensor network protocols as agent-based evolutionary games, a framework for implementing these protocols, a routing protocol with both evolutionary and non-evolutionary variants implemented using this framework, and an evaluation of the thesis stated above using these protocols.

1.5 Contributions

The major contributions of my dissertation are:

- An agent-based approach to optimizing routing protocols for wireless sensor networks, where the behavior of the system is determined by the phenotypic makeup of the population of agents in the system.
- An approach to optimizing the composition of the population of agents in the system based on the use of biologically-inspired approaches, primarily evolutionary games.
- A software framework for implementing wireless network protocols based on

this approach in the TinyOS software environment.

- The design and implementation of a network routing protocol for TinyOS using this agent-based approach to create WSN applications.
- A simulation-based evaluation of the effectiveness of different approaches to controlling agent creation, selection, and survival in the context of a network routing protocol in simple and periodic faulty network configurations.
- A discussion with directions for future work using this approach.

1.6 Dissertation Outline

The rest of this dissertation is organized as follows. Chapter 2 summarizes the main ideas from other research projects related to this research. Chapter 3 explains the agent-based approach, and it relationship with evolutionary game theory. Chapter 4 describes the components of the micro-component framework to support the agent-based design, and evolutionary games to evaluate the performance of the network. Chapter 5 describes the agent-based implementation of a network protocol called Evolutionary Collection Protocol (ECP). Chapter 6 study the structure of the game in ECP and its general impact on WSN communication behavior. Chapter 7 then compares the obtained game with various heuristic-based routing variants of ECP, including a rule-based non-evolutionary game, to evaluate the thesis statement. Finally, Chapter 8 provides a summary of the dissertation and a discussion of future work.

Chapter 2

Related Work

Extensive research has been done in WSNs on power management and fault tolerance. Many existing routing protocols and algorithms [73, 66, 58] attempt to satisfy different application requirements, optimize energy usage, and improve fault tolerance, and this shows the importance of these elements when designing and optimizing a communication system for WSNs. There is also some research on using agent-based systems to optimize protocols and WSN operation.

This chapter describes such previous work. Section 2.1 describes the general features of WSN communication protocols, focusing primarily on the CTP protocol to which our agent-based routing protocol is most similar. Section 2.2 presents routing protocols for WSNs focusing primarily on fault tolerance. Section 2.3 describes solutions for WSNs using agent-based approaches or inspired by biological systems. Section 2.4 then describes previous research on evolutionary games related to routing protocols and WSNs. Finally, Section 2.5 describes general concepts related to programming in WSNs, and Section 2.6 summarizes related work.

2.1 WSN Routing Protocols

Routing protocols for WSN have been extensively researched. Traditional routing protocols have several shortcomings when applied to WSNs, mainly due to energy constrained operation. For example, techniques such as flooding, in which nodes broadcast received packets to the rest of nodes until a destination node is reached, produce undesirable effects in WSNs such as implosion and overlap, with multiple duplicated copies of data being delivered to the destination point [30, 2].

Major routing protocols available for WSNs are divided in several categories [73], namely location-based, data-centric, hierarchical-based, mobility-based, QoS-based, multipath-based, and heterogeneity-based. The example protocol we developed in this research is a data-centric protocol based on the Colletion Tree Protocol (CTP) provided by the TinyOS [48] operating system. In the remainder of this section we provide an overview of the CTP protocol used as the reference protocol to implement and test our agent-based approach using evolutionary games, and briefly discuss other related WSN protocols.

2.1.1 Collection Tree Protocol

CTP [33] is a tree-based collection protocol where some nodes advertise themselves as tree roots, and other nodes form routing trees towards the roots. Packets are sent to any root, and a routing decision is made at each node by selecting the next hop to the nearest root.

CTP assumes that the data-link layer provides four elements: a) an efficient local broadcast address; b) synchronous acknowledgments for unicast packets; c) a protocol dispatch field to support multiple higher-level protocols; and d) single source and destination fields. Other assumptions are that a link quality estimator, i.e. a measurement of the quality for wireless transmissions between each pair of communicating nodes, is available for some number of nearby neighbor nodes. The protocol does not guarantee 100% reliability.

Routes in CTP are generated using a routing gradient, a calculated value based on the quality of the wireless link between each pair of nodes. The routing metric is called ETX for expected transmissions. The ETX for one node is the sum of the individual ETX for all links in the path between the node and the nearest root node. The ETX for a root node is always 0.

A CTP network can have routing loops. They may occur when a node chooses a new route with significantly higher ETX than the previous one, or in response to losing connectivity with a candidate parent. A loop occurs when the new route includes a node that was a descendant. CTP detects loops by including the source ETX in the data packets when moving to the next hop. If a data packet comes from a node with lower ETX, then a loop is detected and the node must request an update of its routing tables.

Routing tables are updated when a node receives a beacon packet. ETX information coming from neighbor nodes is used to change the routing table accordingly, and then communicate any relevant changes to other nodes in the neighborhood. CTP uses a variant of the Trickle algorithm [47] for beacon timing [32]. Routing validation and failed nodes are detected with data packet acknowledgments, and by relying on the mechanism to detect routing loops described above.

Our agent-based protocol inherited the described features available in CTP, but we changed the conventional procedural implementation into an agent-based approach and added evolutionary behavior to test the approach in a dynamic and faulty environment. Furthermore, we replaced the trickle algorithm with our agent-based optimization approach, and failed nodes are detected using beacon packets and timeouts, not data/ack packets.

2.1.2 Other Routing Protocols

WSNs can be used in a large number of different applications, and most of them work with many hardware and software restrictions in the sensor devices and are deployed in dynamic environments with unreliable communications. This has generated the creation of a vast set of routing protocols [73] to satisfy different application requirements. In this section we briefly describe additional routing protocols that are commonly used in WSN applications.

GEAR [83] is a location-based energy-efficient routing protocol to route queries to specific regions of a sensor network. In this protocol, sensors need localization hardware, for example a GPS unit, so they know their position. Furthermore, sensors are aware of their residual energy, and also the residual energy and localization of the neighbors. This protocol uses energy-aware heuristics based on geographical information to select the route to send a packet towards its destination.

Directed diffusion [41, 42] is a data-centric routing protocol to disseminate and process queries. This protocol has several key elements, namely data naming, interests and gradients, data propagation, and reinforcement. A sensing task is defined by a list of attribute-value pairs. At the beginning of the routing process, the sink specifies a low data rate for incoming events. Then, the sink can reinforce one particular sensor to send events with a higher rate using a smaller time interval, and the reinforcement is also applied to the neighbor nodes receiving the message.

Rumor routing [14] is another data-centric routing protocol that makes a compromise between query flooding and event flooding application schemes. The protocol is based on the concept of an agent, which is a long-lived packet that traverses a network and informs the sensors about events it has learned while traveling. The agent travels for a certain number of hops and then dies. Each sensor and the agent keep an event list with event-distance pairs to provide the actual distance in hops to the corresponding event, and the agent synchronizes its event list with the visited sensors.

LEACH [35, 36] the Low-Energy Adaptive Clustering Hierarchy routing protocol, uses clusters to extend the life time of the network, and it also does aggregation of data inside the network. In LEACH, clusters are created using localized coordination and control to reduce the amount of data transmitted to the sink. The cluster head is rotated based on its energy level to avoid battery depletion of individual devices. Protocol operation is divided into rounds having two phases. First, a setup phase creates the clusters, performs cluster head advertisement, and creates a transmission schedule. Then, there is a steady-state phase for data aggregation, compression, and transmission to the sink. LEACH uses single-hop routing where each node can transmit directly to the cluster-head and the sink, and is not suitable for deployment in large regions.

2.2 Fault tolerance in WSN Protocols

Fault tolerance in WSN protocols is a hot topic because of the inherent complexity of the environment where the networks are deployed and the restrictions that nodes have. In this section we describe theoretical and practical research projects proposed for fault tolerance in WSNs to adapt to network conditions.

2.2.1 WEAR and SCORE

In [72], authors propose WEAR, a routing protocol for fault tolerance in WSNs that considers four factors affecting the routing policy, namely the distance to the destination, the energy level of the sensor, global location information, and local hole information. To handle holes, large spaces without active sensors caused by faulty sensors, they propose a size-oblivious hole identification and maintenance protocol. Complimentary to this protocol, [4] proposes a framework named SCORE that provides basic pieces of information such as neighborhood information and node operational state that are used by WEAR network components to base their actions and promote protocol optimization.

The primary difference of this project with our approach is that they do not address optimization for the timing of the actions, and they instead check for the physical state of the nodes to distribute the load and deal with faults appropriately.

2.2.2 ENFAT-AODV

ENFAT-AODV [20] is a fault-tolerant routing protocol based on the AODV [67] routing protocol. It uses a backup route to improve reliability for packet delivery and keep the system running even under presence of failures such as link breaks and node failures. Backup routes are used when the main route is not available, and this improves throughput, reduces the delay to deliver packets, and reduces the number of packets dropped in the network. This solution satisfies the trade-off between fault tolerance and low transmission delay, but at the same time increases the load of control packets to create the backup route.

2.2.3 Fault Management Architecture for WSNs

[6] proposes a fault management architecture for WSNs. This system partitions the network into a virtual grid of cells to perform fault detection, execute recovery actions locally with minimum energy consumption, and support scalability. The grid

architecture [82] detects faults in a distributed way and reports them across the cells.

A cell manager handles management tasks at individual cells, and coordinates with a gateway node to detect faults and perform recovery inside the cell. The cells combine to form groups, and each group promotes one of its cell managers to a group manager. Group managers detect faulty cells and avoid future faults.

While this architecture can detect multiple and distributed faults in the network, it has significant management overhead imposed by creation of the groups, the cell manager, and gateway maintenance. In contrast, we chose a simpler fault detection system that detects individual node failures to test our agent-based approach.

2.2.4 Dynamic hybrid fault-models

In [52, 53], the authors introduce the term dynamic hybrid fault models to add time and covariate dependence to hybrid fault models, and make real-time predictions of fault tolerance in WSNs. The authors propose a theoretical layered architecture to create fault-tolerant sensor networks. In the approach, sensor nodes are players of an evolutionary game, and they propose extensions to the classical failure models to represent real-time and dynamic hybrid models.

Some of the theoretical ideas about fault tolerance handling proposed in this work inspired the approach described in this dissertation. However, the approach they describe is largely theoretical, and does not clearly define how strategies evolve nor how population dynamics happen. Our approach, on the other hand, clearly defines these features to make a concrete system that can be evaluated and tested.

2.3 Agent-based Systems in WSNs

There are currently research projects for WSNs using different agent-based approaches, and several use biological ideas because the way nature optimizes processes has been been useful for solving engineering problems. In this section, we briefly describe the main features of a few projects in this area. An important difference between our approach and all these projects is that they do not have the competition environment provided by the evolutionary games to compare strategies when the network is operating.

2.3.1 BIONETS

BIONETS [17] is an agent-based bio-inspired architecture. It proposes mobile sensor-enabled networks with self-organizing and self-optimizing services to enable operation in low-cost pervasive environments. The approach is proposed in the context of the communications requirements placed by pervasive communication environments on low-cost sensor nodes.

In this approach, network services are modeled as living organisms. The goal of the network is to optimize entire network services. The network is proposed as the habitat where services move from device to device, and genetic information encodes their behavior and goals. Services evolve and adapt to the environment constantly and autonomously using what appears to be a basic genetic algorithm. In contrast, we focus on optimizing parameters in individual services and use an evolutionary game approach to parameter optimization.

2.3.2 BiSNET

BiSNET [11, 12, 10] is an agent-based bio-inspired sensor network architecture that seeks to address issues in WSNs such as autonomy, adaptability, self-healing, and simplicity. It is implemented as a middle-ware platform on top of TinyOS, where agents follow biological principles such as decentralization, food gathering and storage, and natural selection.

BiSNET uses a bee analogy to structure the system, where the platform corresponds to a hive and agents to bees. Agents read sensor data, and discard or report it to a base station using biological behaviors such as replication, death, and migration. BisNET designs agent behavior based on virtual energy exchange—agents acquire energy by sensing data, split energy with their children with they replicate, report results when their energy is high, and die due to energy starvation when they cannot balance energy gain and expenditure.

As a result of this biological design, BiSNET allows sensor nodes to autonomously adapt their duty cycles for battery efficiency, to draw inference on potential environmental changes from sensing activities of neighboring sensor nodes, to collectively detect and eliminate false positives in sensor readings, and to be simple and lightweight.

2.3.3 kOS

kOS [16] is an operating system designed to support the operation of distributed biologically-inspired algorithms by defining biological agents which interact with their neighbors via simple rules, and cooperate with a large number of individuals to perform some complex global task. kOS provides a single-task run-to-completion execution model designed to run on a cheap wallet-sized devices. This model is simpler than the one we selected when using TinyOS because the target sensor devices are larger than the devices typically used by TinyOS.

2.3.4 Agent-based architecture for fault tolerance in WSN

In [71], the authors propose an agent-based architecture for fault tolerance in WSNs based on a federation of mobile agents that diagnose and repair the network. Agents are classified as local, metropolitan, and global, to provide fault tolerance at node, network, and functional levels. Agents play two roles, namely sniffers and correctors. Sniffers observe the behavior of the network at different levels, and correctors repair the network.

The authors state that interactions between agents are inspired by honey bee dance language. An error database contains detailed information about errors, the faults causing the errors, and the resulting failures. The database is present partially at some nodes identified as cluster heads in a hierarchical structure of nodes that define the system.

The system has a mechanism to capture the statistics of the network elements in the form of attributes for fault detection. Later, a protocol announces the presence of faults to the relevant entities in order to initiate fault repair. The scope of communication is set according to the severity of the fault.

This system is modeled mathematically to analyze the overhead imposed by the fault tolerance architecture, and the authors concluded that the overhead generated is need-based, leading to an attractive cost-benefit relation.

2.4 Evolutionary Approaches in WSN protocols

This section provides a brief description of other theoretical research using evolutionary approaches to model and design routing protocols for WSNs.

2.4.1 Evolutionary congestion control protocol for WSN

The authors in [5] describe a theoretical approach to congestion control in WSNs where they apply evolutionary game theory to non-cooperative networks containing a large number of sensors. They show how the characteristics of the wireless channel influences evolution and the evolutionarily stable strategy by defining two populations of connections using a TCP protocol implementation based on the technique additive increase and multiplicative decrease (AIMD). The approach proposes an iterative application of the Hawk and Dove classical game that uses the parameters to increase and decrease the window size of the protocol, and they evaluate the performance for throughput and congestion control for the selected strategies.

2.4.2 Evolution of Cooperation in Multi-class WSNs

In [23], the authors propose evolution of cooperation for reliable routing in a finite large WSN with a static population of nodes that can be stationary or mobile. They define multi-class network nodes as players in the context of an evolutionary game motivated by the iterated prisoners dilemma game with strategies and fitness functions.

The approach determines conditions under which spatially dispersed multi-class WSNs exhibit tendencies to cooperate, and also proposes a localized distributed and scalable algorithm called the Patient Grim Strategy that enforces cooperation in

WSNs. The solution focuses on packet forwarding with random static topologies; they do not analyze network flow because of the complexity of the problem.

2.4.3 Routing Protocol with Hybrid Genetic Algorithm in WSNs

The routing protocol described in [34] uses a genetic algorithm in the design of a high performance multi-path routing protocol for WSNs to improve energy usage. The algorithm has two stages, namely single-parent evolution and population evolution. In single-parent evolution, only a single individual is evaluated, and the speed of the evolution that produces a good individual is very fast; a global optimal path is generated at the same time. For the second stage, population evolution is introduced to improve the solution quality. Results show that their genetic operators avoid premature convergence, balances energy consumption, and extends network lifetime.

2.5 Programming in WSN

[61] and [60] provide a reference to fundamental concepts for WSN programming. The authors present a taxonomy for WSN applications that considers distributed processing occurring inside the WSN and focuses on solutions that allow programmers to express communication and coordination among the nodes.

The taxonomy includes the following aspects of a WSN application: goal, interaction pattern, mobility, space, and time. Goal refers to sense-only or sense-and-react applications. Interaction patterns can be one-to-many, many-to-many, or many-to-one on the communication between nodes. Mobility has to do with static nodes, mobile nodes, or mobile sinks in the network topology. Space relates to global or localized processing of information in the network. Finally, time aspects refer to periodic execution of operations or event-triggered functions.

Besides the taxonomy for classifying WSNs applications, they also present a reference hardware and software architecture. The boundaries between programming abstractions and the rest of the software are often unclear in WSN, mainly because of the restricted resources, and application are usually intertwined with system-level services.

[61] also classifies aspects and features required by the programming language to implement WSN applications. It specifically describes aspects of the programming paradigms, data access models, computation scope, and some components of communication such as scope and addressing that can be considered from the programming language perspective. Even though this publication appeared after we selected the hardware and software platform for this dissertation, it provides a detailed explanation and good support of our decision to use TinyOS as the selected operating system, and many decisions about locality of the implemented solutions are supported in the taxonomy.

2.6 Summary

In this chapter we described previous work about WSN communication protocols, including the CTP protocol we used as the reference routing protocol to implement our agent-based approach, and other routing protocols for WSNs with fault-tolerance capabilities. We found that, in most cases the timing issues and real-time optimizations are not addressed in detail, or that systems implement complex schemes for fault tolerance with high costs on resource usage for WSNs. We also described previous theoretical research addressing fault-tolerance problems in WSNs that suggests biological ideas as the approach to improve WSN operation, and we considered some of these ideas when designing our agent-based approach with evolutionary games.

We also described some agent-based and biologically-inspired projects, such as BIONETS, BiSNET, and kOS. None of these consider evolutionary games as a mechanism to evolve agent strategies when network conditions change, and to evaluate and optimized network operation. Finally, we presented previous theoretical research applying evolutionary games to a routing protocol in WSNs, and also research about concepts and challenges when creating software for WSNs.

Chapter 3

Agent-based WSNs Optimization

3.1 Overview

As discussed in Section 1.3, this dissertation describes an agent-based approach to optimizing routing protocols for WSNs. Agents are virtual organisms living in the network, and they represent parameters requiring optimization. Agents execute network functions when they interact with other agents and with the environment according to their parameter values.

Repeated agent interactions optimize the composition of the population, and changes in the population optimize the network behavior. Agents interact in the context of an evolutionary game, and the game provides measurements of performance of the actions executed by the agents.

We selected an agent-base model because of its ability to cope with the dynamic behavior [39] and irregular failure rates generally present in WSNs [64]. In our model, agents are selected to execute specific network functions inside the nodes, and performance optimization is done locally because agents are assigned to localized network functions. In addition, an agent-based approach naturally optimizes in a given network locality. Agents move between nodes, and optimization decisions at one node are automatically shared in the network neighborhood. Interactions between agents result in an evolutionary game [56], as mentioned briefly in Chapter 1.

In the remainder of this chapter, we describe the proposed agent-based model in more detail, including illustrative examples in the context of the routing example described in Chapter 1. In addition, we fully describe the relationship between the agent-based model and evolutionary games as defined in the literature.

3.2 Agent-Based Model

This section describes the components of the agent-based model, namely the agents representing parameters to be optimized, the environment representing the network where agents live, and the interactions between agents to execute the network optimization process.

3.2.1 Process Overview

The agent-based model is designed to optimize power management and improve routing performance in WSNs. Every time a network operation involving some kind of optimization is executed, agents participate in the operation to improve the process.

Participation of agents is as follows:

1. **Initialization**. Each network node has a pre-allocated memory space to store a group of agents to execute/optimize network functions, termed the *selection room*. When a node starts, a randomly generated set of agents is placed in the

selection room to provide an initial population available to execute network operations.

2. **Selection**. Before executing each network operation, the system randomly selects an agent from the pool of available agents residing in the selection room, and the selected agent is assigned to execute the operation. If the selection room is empty, additional agents are randomly created as necessary.

3. **Operation execution**. The selected agent executes the operation. As part of this operation, the agent may replicate or move to a different network node. For example, an additional identical or modified replica of the agent can be added to the selection room, the agent may send itself to a single remote node, or may replicate by sending copies of itself to multiple remote nodes (e.g. by broadcast).

4. **Competition**. After executing the operation, the agent, if it remains on the local node, may enter a competition with another agent if another agent is also involved in the network operation. The goal of the game is to evaluate how well the agents performed their jobs under the rules defined in the game. Either or both agents may win the game and survive this competition, but if an agent loses this game, it is discarded.

5. **Return**. Any agents that remain on the node and survived any resulting competition return to that node's selection room, where they may later again be selected to perform network operations.

The described process is executed repeatedly over time. Iterated agent competition produces changes in the composition of the population of agents, and the composition of the population optimizes the network parameters that optimize network operation.

3.2.2 The Agents

An agent encapsulates a set of parameters to execute a network function. A function assigned to an agent is executed according to the value of its parameters. Parameters of agents are similar to phenotypes of organisms in an ecological system. They are the visible features or behaviors that interact with the environment and with other organisms. In the context of a WSN, parameters are network values to be optimized for a network function.

An agent is represented by an agent type and a set of parameters. For an individual agent, the type and parameters are constant during its life, but different agents can have different parameter values. The agent type enables different parameter sets, similar to the species in real world representing organisms with different phenotypes [77]*p.16. Note that while there can be in general be multiple agent types, this dissertation focuses on cases with only a single agent type.

3.2.3 The Environment

The environment comprises all the network nodes and the real environment where the network is deployed. All variables in RAM memory are considered part of the environment, except for the memory locations containing the agents.

This definition of the environment is convenient because it enables agents to obtain useful information from network state variables. For example, information from the real environment can be collected periodically using sensor devices. Data collected from the sensors can be used by agents to respond to changes in the real environment. Information can also be obtained from the environment in RAM memory, for example data about actions executed by agents such as the current sampling rate of a sensor device, or the difference between two values read from a sensor can be used to adjust the time for sampling the sensor.

Statistics collected from some components of the node are also an important source of information obtained from the environment. For example the number of packets sent, received, or lost by the wireless transmitter can help adjust the timing of transmissions.

3.2.4 Agent Interactions

Interactions are the relations between an agent and the environment, or between two agents. Agent interactions modify the environment, and changes in the environment can also change the composition of the population of agents. Interactions measure the successfulness of the parameter values represented by the agents; a survivor agent represents an instance of a successful parameter value that can be re-selected over time because it was successful when executing a network function under the existing environmental conditions. We consider three different types of agent interactions:

1. **Selection**. The system selects an agent from the existing population in the selection room and assign it to execute a network function.
2. **Creation**. The system creates a new agent because there are no available agents in the selection room, or an agent is replicated after another agent executes a network function.
3. **Competition**. Two agents participate in a one-to-one competition to compare their parameter values, and determine how well are they performing in the network.

Interactions requiring selection of agents generally randomly select them from the existing population, resulting in an evolutionary system where the composition of the population determines the behavior of the system. In addition, the node can also select the agent preferentially by considering the influence of some environmental variables. With preferential selection, picking an agent from the population is not necessarily a random function; it can also be a function mapping a value of the environment to a parameter value of the agent. However, if agent selection does not consider the entire population of available agents and the composition of the agent population does not impact system behavior, the resulting system cannot be considered evolutionary.

Similar to selection, creation of agents can be random by selecting a random value for each its parameters, or environmentally influenced. In the environmentally influenced case, some parameters of the new agent are based on environmental conditions instead of a random function. This approach can bias the population from which agents are selected; if this biased creation in some way dominates the entire population, the resulting system again may not be considered evolutionary.

Competitions are interactions requiring two agents. The details about the interaction are specific to the network function being implemented, but the goal in all cases is to evaluate how well the agents are doing according to network requirements. After a competition, one of the agents is declared the winner, where the winner agent survives and the other agent dies. In some cases, a tie is also a possible result for the game and both agents survive.

3.2.5 Illustrative Example

To illustrate our agent-based approach, consider the network process of keeping the neighborhood information to decide the route for gathered data updated, as described in Section 1.2. For this process, we want to optimize the frequency at which to send beacon packets by adjusting the time between beacons. In this case, each agent contains a beacon time parameter, the interval between successive beacons. There are two operations in this process which involve agents, the *beacon send operation* and the *beacon receive operation.*

For the beacon send operation, the node selects an agent from the available agents in the selection room, creating an agent with a random beacon time if the selection room is empty. The selected agent then starts a timer to send the beacon. The length of the timer is selected according to the strategy for beacon timing in the agent. When the timer expires, the node transmits a beacon containing information about the communication status of this node, and the selected agent is also transmitted inside the beacon. Any node that receives the beacon will receive a replica of this agent. At this point, the beacon send operation is completed at the node, no agents remain, no competition occurs, no agents return the local selection room, and a new beacon send operation can be started.

For the beacon receive operation, a complementary process is executed to evaluate and monitor the incoming beacon packets. First, the node again selects an agent from its selection room which listens for incoming beacons using a timeout of the beacon time in this agent. When a beacon is received, this agent processes the incoming beacon, updating neighborhood information appropriately. Because the incoming beacon also contains an agent, the agent performing the receive operation and the incoming agent conduct a competition based on the quality of the information received and the speed at which it was sent to evaluate both agents. Agents that survive this competition are placed in this node's selection room, where they can later be selected for both beacon send and receive operations.

3.3 Evolutionary Games

Interactions between agents comprise an evolutionary game in the network that determines the composition of the population of agents. In our model, we used evolutionary games as defined in [56] and [77]:

> "The players are individual organisms. Strategies are heritable phenotypes. A player's strategy set is the set of all evolutionarily feasible strategies. Payoffs in the evolutionary game are expressed in term of fitness, where fitness is defined as the expected per capita growth rate for a given strategy and ecological circumstance. The fitness of an individual directly influences changes in the strategy's frequency within the population as that strategy is passed from generation to generation. Evolution, then, has to do with the survival of a given strategy within a population of individuals using potentially many different strategies." [77]*p.16.

Evolutionary games provide a mechanism to measure the performance of the agents, and also provide a simple but powerful tool to evaluate and optimize network behavior based on agent actions.

Following the notation in [77]*p.29-30, the dynamics of an evolutionary game with n_s types of players (species) and n_y environmental resources is determined by the number of players of each species, $\mathbf{x}$, the strategies of these players, $\mathbf{u}$, the resources available to these players, $\mathbf{y}$, and the fitness function of the strategies, $\mathbf{H}(\mathbf{u}, \mathbf{x}, \mathbf{y})$. Note that the fitness function of the strategies, which corresponds directly to the population dynamics of the system in the notation of [77], is partially determined by the game rules that evaluate the relative fitness of individual agents' strategies. The rules of the game define the relative fitness of pairs of agents, but are usually considered part of the formal definition of an evolutionary game in the above notation.

In the remainder of this section, we describe how the elements of our agent-based system correspond to the strategies and resources, the role of game rules that in our system and evolutionary game theory, and the resulting fitness function and population dynamics.

3.3.1 Strategies

Strategies in the evolutionary game setting are the parameters of the agents in our agent-based model. A strategy represents a network parameter to be optimized. Strategies are used by the community of agents to interact and select the best strategy values.

Strategies are the equivalent of phenotypes in real organisms. They represent visible features or behaviors of the organism. Strategies are fixed for an individual agent during its life, but they evolve over time when individuals with winning strategy values make progress in the population by replicating as a result of the interactions.

Using our example on Section 1.2, the beacon time parameter is a strategy of the agents advertising a node to the neighborhood. Agents have different beacon times and they compete according to some rules to survive the network.

3.3.2 Resources

A resource is any element of a node that is not an agent. Resources are an important component of evolutionary game theory because they influence the selection and creation processes described in Section 3.2.4. Resources are used by agents to modify the environment, and changes in the resources modify the population of agents.

Considering again our beacon broadcasting example, the quality of the communication with a neighbor node is an environmental resource that can be changed by agents, but at the same time the population could be affected by changes in the communication quality.

3.3.3 Game Rules

An evolutionary game has an inner game and an outer game:

> The inner game involves only ecological processes and can be considered as a classical game. For the *inner game*, players interact with others and receive payoffs in accordance with their own and others' strategies. Evolution takes place in the *outer game*. It is the dynamical link, via inheritance and fitness, whereby the players' payoffs become translated into changes in strategy frequencies [77]*p.17.

In our agent-based system, the rules of the game produce the inner game between strategies that results in the destruction of agents of the population. The repeated execution of this game with agents changes the composition of the entire population, resulting in the overall evolutionary game between different strategies.

In our system, the individual games between strategies correspond to the interactions between agents. The rules of these games produce pressure in the population of agents according to the goals of the network. This pressure generates population dynamics because some agents live and others die as a consequence of the interactions.

Considering the example in Section 1.2 one more time, the rules for this game define that an advertising agent survives if the link quality value it brings inside a beacon is different from previous information received from the same node at some neighbor node, or if the information is not different but the beacon is not coming too quickly. On the other side of the game, a monitoring agent located at a destination node dies if the incoming quality value is different from a previous value because the network is changing and the monitoring agent was going to wait a longer time for the next beacon because it was not expecting a change in the quality.

3.3.4 Population Dynamics

As mentioned above, the available resources and game rules produce a fitness function for strategies in the population, and this fitness function corresponds directly to the population dynamics of the strategies. This fitness function determines the change in the population of each strategy over time. When resources and game rules are known, this fitness function can be explicitly defined, and applying it repeatedly to an initial population simulates the population dynamics of the system.

In our system, the resources available in the network are dynamic, and not generally explicitly known at a given time; as a result, we cannot define an explicit complete fitness function. Instead, we explicitly simulate individual games on agents across the entire system, implicitly defining this fitness function.

Despite the lack of an explicit complete fitness function, analyzing the structure of the function provides insight into the dynamics of the system. To do this, we use the definition of fitness function as defined in [77]*p.40, where fitness is defined as "the per-capita change in population density (the finite growth rate) from one time period to the next, for discrete time periods."

In the example from Section 1.2, the fitness function shows how the population of different agents changes over time. Table 3.1 contains the functions comprising the fitness function for this example. In all these functions, t represents the current time, $\mathbf{u}$ represents the strategies of the agents, and $\mathbf{y}$ represents the environmental

Variable	Description
$\mathbf{u}, \mathbf{x}, \mathbf{y}$	Vectors of strategies, population of agents, and other resources respectively, at time t.
$e(\mathbf{u}, \mathbf{x}, \mathbf{y})$	Agents created during time interval [t,t+1]
$v(\mathbf{u}, \mathbf{x}, \mathbf{y})$	Agents replicated by arriving at some neighbor after beacon transmissions during time interval [t,t+1]
$k(\mathbf{u}, \mathbf{x}, \mathbf{y})$	Agents killed because of the game rules during time interval [t,t+1]
$r(\mathbf{u}, \mathbf{x}, \mathbf{y})$	Agents killed because of the system capacity (i.e., system was full) during time interval [t,t+1]

Table 3.1: Fitness Function Components for Advertising Agents

resources used in the game.

In this case, our population model for advertising agents at time t+1, as in [77]*p.40, eq. 2.6, and is given by the expression:

$$x(t+1) = x(t)[1 + H(\mathbf{u}, \mathbf{x}, \mathbf{y})]$$

where $H(\mathbf{u}, \mathbf{x}, \mathbf{y})$ is the fitness function given by

$$H(\mathbf{u}, \mathbf{x}, \mathbf{y}) = [[e(\mathbf{u}, \mathbf{x}, \mathbf{y}) + v(\mathbf{u}, \mathbf{x}, \mathbf{y})] - [k(\mathbf{u}, \mathbf{x}, \mathbf{y}) + r(\mathbf{u}, \mathbf{x}, \mathbf{y})]]/x(t)$$

This fitness function reflects the new or replicated agents with the terms $e(\mathbf{u}, \mathbf{x}, \mathbf{y})$ and $v(\mathbf{u}, \mathbf{x}, \mathbf{y})$, and dead agents with $k(\mathbf{u}, \mathbf{x}, \mathbf{y})$ and $r(\mathbf{u}, \mathbf{x}, \mathbf{y})$ during the last time interval. How the frequency of the strategies in the population changes over time is shown with $x(t+1)$. Note that agents can die because of the game rules, and because the maximum capacity of the node to host agents is reached.

3.4 Summary

This dissertation proposes an agent-based approach to optimizing WSNs. A population of virtual agents living in the nodes optimize network operation by playing evolutionary games, and agents with the best strategies replicate over time.

The resulting population represents the best adapted agents according to the game and environmental conditions of the network, and that population determines the network behavior. Evolutionary games provide a mechanism to measure population dynamics, and consequently a measurement of the network performance. The agent-based approach combined with evolutionary games produces an adaptive system to improve performance and fault-tolerance locally at the nodes, and allow optimization and adaptation of the network according to changes on operating conditions.

Chapter 4

Implementing Agent-Based WSN Software

In this chapter, we describe a component-based micro-protocol architecture designed to build WSN applications that support the agent-based approach described in Chapter 3. The micro-protocol architecture is based on ideas proposed in [15] of designing and implementing network protocols using a set of fine-grained composable components.

To build a WSN application, this architecture has a high-level abstraction called a *micro-protocol*. A micro-protocol represents a network function, such as sending or receiving a beacon packet in the example application described in Section 1.2. A micro-protocol is constructed from one or more smaller units called *actions*. For example, a micro-protocol to send a beacon packet might be composed of three actions, namely pick a time to send the beacon, setup the beacon packet, and transmit the beacon.

To implement micro-protocols, we constructed a *micro-component framework* with fine-grained composable components. Simple micro-components define micro-protocol actions, while more complex micro-component types handle component specialization for different types of agents and sequencing of actions in full micro-protocols.

In the rest of this chapter, Section 4.1 explains the basic micro-protocol architecture, and Section 4.2 describes the micro-component framework for implementing micro-protocols and actions. Section 4.3 follows with an example of execution a micro-component-based implementation of a simple micro-protocol. Finally, Section 4.4 describes key details of the implementation of this software framework in TinyOS.

4.1 Micro-Protocol Architecture

Micro-protocols are sequences of actions executed to perform a specific network function. They may be invoked at any point in the system, and they are executed as deferred procedure calls. Micro-protocols can be passed arguments; when network functions are optimized using agents, an agent may be passed to the micro-protocol for optimization purposes, for example.

We represent micro-protocols in this dissertation using conventional control flow diagrams. Boxes in the flow are actions executed by agents, and arrows represent the execution ordering. As an example, Figure 4.1 shows a micro-protocol implementing the receive packet function for the Evolutionary Collection Protocol described in more detail in Chapter 5.

The names inside the boxes in Figure 4.1 provide a basic description of the purpose of each action. Rectangular boxes represent normal actions containing a sequence of instructions executed as an atomic block of code. Actions with a diamond shape represent decisions to change the normal execution sequence inside the micro-protocol. Cross-hatched boxes are actions that use an agent for optimization pur-

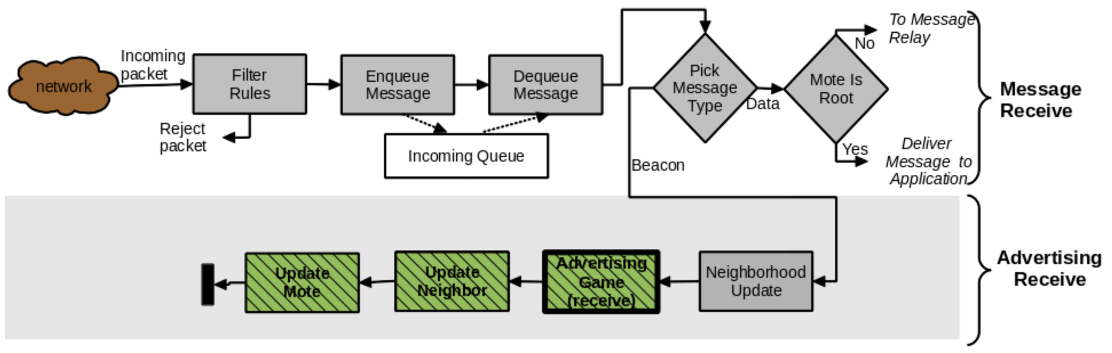


Figure 4.1: Micro-protocol example

poses. Boxes with no pattern represent functions not assigned to any agent because they are always executed as part of the micro-protocol, and they do not optimize any network parameter.

Each WSN application generally requires several micro-protocols to implement a complete system. These micro-protocols optimize one or more network parameters, and the parameters are optimized by running the actions composing the corresponding micro-protocols using appropriate agents.

We define two types of actions within micro-protocols:

- **Simple actions**, which require no optimization, and consequently do not use an agent. For example, preparing a data packet for transmission requires information available in the node but no optimization.

- **Virtual actions**, which are actions executed by agents that optimize network operation when executed using agents with different strategies. For example, picking the time to send a beacon packet can be optimized by agents to improve power management in the node. They allow different action implementations to be selected at runtime based on the specific agent executing the action.

4.2 Micro-component Framework

To implement the micro-protocol architecture, we created a micro-component software framework that we describe in this section. A micro-component is a simple event-driven execution element that can be composed with other micro-components to implement complex, modular network services, including agent-based execution to optimize system performance. Micro-components provide a simple `run()` interface for external code to schedule their future execution.

The micro-component framework provides the abstractions required to optimize a WSN application using the proposed agent-based approach. A block diagram of the framework is shown in Figure 4.2.

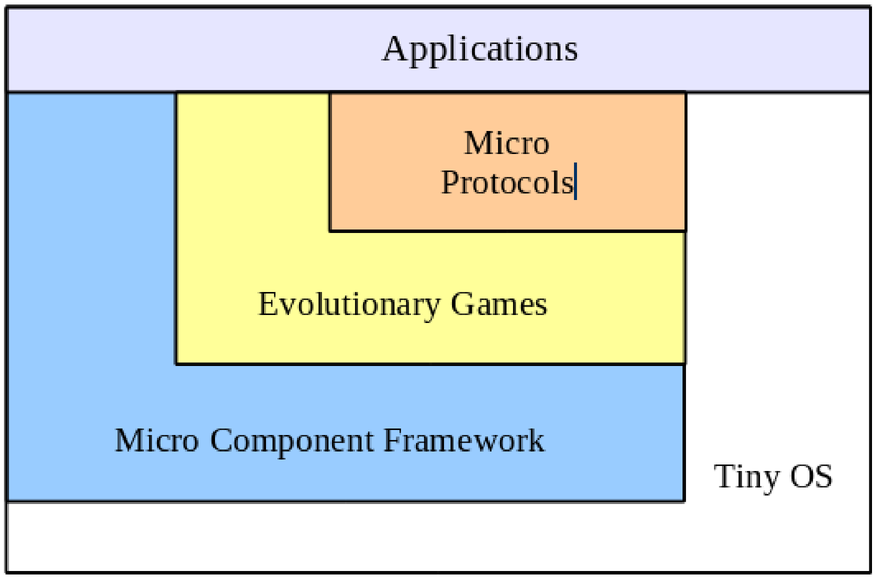


Figure 4.2: Micro-component Framework Block Diagram

The micro-component framework addresses the hardware and software restrictions inherent to a WSN platform and the programming challenges mentioned in Section 1.1.3. The framework uses a variety of techniques to address these challenges, particularly by providing higher-level features than provided by the sensor network OS, as well as new programming abstractions to support agent-based execution. For example, the micro-component framework supports virtual actions using virtual micro-components, providing functionality similar to virtual functions in a conventional object-oriented programming language.

This section describes the execution model and key design features of the micro-component framework. This includes a description of the three types of micro-components: simple micro-components to execute regular blocks of instructions and to interact with slow hardware devices, virtual micro-components to allow for dynamic selection of the simple micro-components based on agent information, and group micro-components to control the execution order of other micro-components.

4.2.1 Simple Micro-components

Simple micro-components implement straightforward code execution inside micro-protocols. They provide actions containing sequences of instructions that can then be composed using the virtual and group micro-components described later.

There are two types of simple micro-components: single (one-phase) and split-phase (two-phase) micro-components. Single micro-components are executed atomically by calling the user-provided function `run()`. Note that as part of the execution of this function, a single-phase micro-component may also request later execution of other micro-components.

Split micro-components, in contrast, have one execution stage that is atomic with respect to other micro-components and one that runs at a later time asynchronously (e.g. from inside an interrupt handler). They are used mainly for interacting with slow hardware devices, similar to the split-phase event abstraction provided by TinyOS [48]. The first phase of a split-phase micro-component atomically runs the user-provided `run()` function. The micro-component ties exectuion of its second phase, which runs the user-provided `runDone()` function, to the firing of a system event, for example an interrupt generated by a hardware device. Note that this function may run in the middle of the execution of another micro-component, and can schedule the later execution of another micro-component.

A split-phase component that sends a data packet wirelessly to a neighbor node, for example, starts transmission when the first phase of the micro-component is executed. It also ties its a second phase to the event associated with the hardware interrupt at the end of the transmission. A different but useful split-phase event occurs when a timer is fired to execute a timed action. The timer request is the first phase, and the second phase is launched automatically when the timer expires to execute the desired action.

4.2.2 Virtual Micro-components

Virtual micro-components provide micro-component functionality that is specialized by the agent executing the micro-component. In particular, virtual micro-components include user-provided functions to select the agent to run the micro-component, and to choose between one or more simple micro-components to execute based on the agent selected. Virtual micro-components are used to implement virtual actions from the micro-protocol architecture in the micro-component framework.

The structure used to select a micro-component implementation from an agent is called the Virtual Action Structure (VAS). The VAS is a hierarchical structure with information about an evolutionary game that integrates with the micro-component framework. It provides a search mechanism to select simple micro-components dynamically, while the simple micro-components provide the actual executable code for particular agent running in the virtual micro-component.

Every agent in the micro-component framework has four values associated with it: game, species, strategies, and strategy value, that specify the specific agent, strategy, and parameter value of the agent. Each game has a unique id to differentiate itself from other games running in the same application. Similarly, each agent is a member of some species or type, allowing multiple agent types in a game, as discussed in Chapter 3. Finally, the strategy and strategy value express the details of the network parameters represented by the agent.

Each of these values correspond to four levels in the VAS hierarchy so that agent execution can be specialized based on each these values. Figure 4.3 illustrates the basic VAS structure.

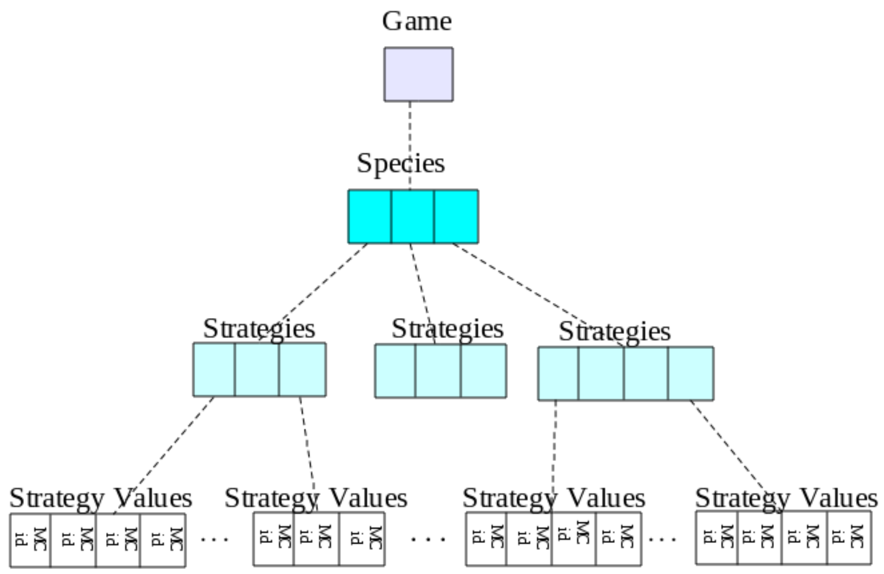


Figure 4.3: Structure of the micro-component framework Virtual Action Structure for selecting the micro-component to use based on the agent associated with the micro-component. The lowest level of the tree contains the micro-component ID of the appropriate micro-component to execute.

4.2.3 Group Micro-components

Group micro-components aggregate and control the execution and sequencing of other micro-components, and are used to implement micro-protocols in the micro-component framework. A group micro-component contains a sequence of micro-components. By default, these micro-components are executed sequentially. However, micro-components in a group micro-component can modify group execution order by invoking a provided control flow function that sets the next micro-component in the group to run. This allows group micro-components to implement complex control flow between micro-components, including conditional execution and loops.

4.3 Micro-component Framework Example

To understand how the control flow of a micro-protocol works in this framework, consider the example in Figure 4.4. This micro-protocol is a simplified version of the receive process in the WSN communication example described in Section 1.2.

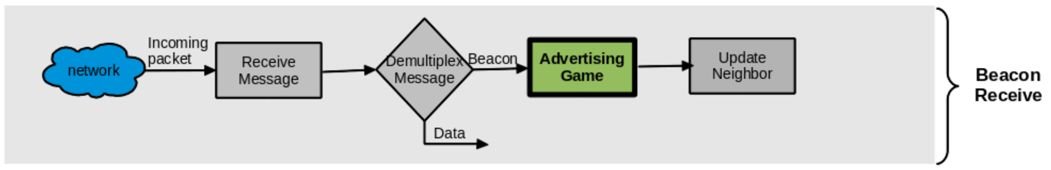


Figure 4.4: Micro-protocol Execution Example

When the wireless receiver of a node receives a packet from the network, the operating system fires an interrupt to process it, and the corresponding interrupt handler schedules the execution of the group micro-component that implements the Beacon Receive micro-protocol, passing the packet to process as an argument. This micro-component then executes a sequence of micro-components to process this packet, starting with the first micro-component, Receive Message in Figure 4.4. The Receive Message micro-component executes packet format verifications and enqueues the packet into the protocol queue for later processing.

Next, the Demultiplex Message micro-component in Figure 4.4 runs. It examines the type of message received, data or beacon, and changes the control flow of the group micro-component based on this value by making function calls to the group micro-component that contains it.

Assume that the next micro-component executed is the Advertising Game virtual micro-component. Before running this micro-component, the system calls a user-provided function to obtain the location and information about the agents participating in the game, and then consults the VAS to pick the appropriate micro-component to run based on the agent. Note that this micro-component may change the set of available agents in the node as part of its implementation of the advertising game. After running the correct micro-component, the system calls the Update Neighbor micro-component to update the routing table with information coming in the packet.

4.4 Framework Implementation Details

While the micro-component framework is conceptually relatively simple, implementing it in the NesC language in TinyOS was non-trivial because of limitations of this environment. This section describes key internal features and implementation details of the micro-component framework. This includes how micro-components are implemented in TinyOS, and details of how individual micro-components are scheduled to handle load balancing and congestion control.

4.4.1 Micro-component Implementation

All micro-components implement the NesC interface `MicroComponent` shown in Figure 4.5 for simple micro-components, with virtual and group micro-components having extended versions of this interface for additional functionality. In addition, each micro-component has a queue of requests for it to handle, and each micro-component is associated with a schedulable TinyOS task.

```
interface MicroComponent {

    command error_t run(position,element);

    command id_t    getMicroComponentId();
    command type_t  getMicroComponentType();
    command id_t    getRunningId(element);
    command id_t    getParentId();

    command error_t enqueueTask(runStack, element);
    command void    setParentId(parentId);
    command void    changeParentDecision(runStack, decision);
}
```

Figure 4.5: MicroComponent Interface source code

In Figure 4.5, the function `run(...)` receives two parameters. `Position` is used by group micro-components to indicate the initial micro-component to execute (see Section 4.2.3), and `element` is a parameter passed to the component. This function is run by the TinyOS task associated with the component when it is scheduled. The functions of the form `getXXX()` get information about the state of the component, and the function `enqueueTask(...)` enqueues more elements to be processed by the micro-component. The functions `setParentId(...)`, and `changeParentDecision(...)` enable explicit control flow changes when the micro-component is executed inside a group micro-component.

Split micro-components implement the same basic interface, but execute the user provided functions in a slightly different way. Specifically, the function `runDone` provided by the user is executed when the operating system signals the second-phase of the event.

Virtual micro-components behave like virtual functions in languages such as C++, but they inspect the strategies of the acting agents at run-time to select the correct function according to the strategy value of the agents, as opposed to using a Virtual Method Table. Virtual micro-components use the `VirtualTrait` interface shown in Figure 4.6 to obtain information about the relevant agent at run-time. The parameter `element` in all these functions contains the agent to execute the action, and the functions `getXXX(...)` extract the required strategy values from the agent to select the corresponding micro-component.

```
interface VirtualTrait{

    event evo_game_unique_t getGameUniqueId(element);
    event species_id_t      getSpeciesId(element);
    event trait_id_t        getTraitId(element);
    event trait_value_t     getTraitValueId(element);
}
```

Figure 4.6: VirtualTrait Interface source code

Group micro-components extend the basic micro-component functionality with the `MicroComponentGroup` interface shown in Figure 4.7. This interface provides a function `add(...)` to add elements to the group, and a few `getXXX(...)` functions to retrieve information about the members.

```
interface MicroComponentGroup{

    command error_t        add(microComponentId);

    command group_data_t * getData();
    command id_t           getIdFromPosition(position);
    command position_t     getPositionFromId(microComponentId);
}
```

Figure 4.7: MicroComponentGroup Interface source code

4.4.2 Micro-component Scheduling and Execution

Micro-components are scheduled and executed using a modified version of the task construct provided by TinyOS. The basic TinyOS task abstraction handles generic scheduling, but enhancements to the task abstraction support additional framework functionality.

TinyOS Task Model . The TinyOS execution model is based on deferred run-to-completion tasks, split-phase operations, and interrupt handlers. Tasks are deferred lightweight procedure calls, and they are the basic concurrency mechanism in TinyOS [48]. Tasks do not receive parameters, and can be posted at any time. Posted tasks are executed later, one at a time, by the operating system scheduler. A TinyOS component can not post multiple copies of the same task to run, but an already started task may re-post itself. Tasks are declared with the `task` keyword and posted for later execution with the `post` keyword, as shown in the following example:

```
task void myTask() {
    // task code
}
```

```
event void Boot.booted() {
    call Timer.startPeriodic(1024);
    post myTask();
}
```

TinyOS Task Enhancements . The `MicroComponentTask` interface is our extended version of TinyOS tasks supporting micro-components. The extensions include:

- Support to receive a generic parameter when calling the task.
- Support to enqueue several requests to one task, each request with a possibly different parameter.
- Extended support for split-phase events.
- Automatic re-posting of the task when there are additional elements in the request queue.
- Automatic scheduling of the next task when the current one is running inside a group micro-component.
- Support to change the normal control flow of a group micro-component from user-provided code.
- Provision to release resources in case of errors if the task is running inside a group micro-component.
- A back-off mechanism to deal with internal congestion problems when the request queues are full.

Figure 4.8 shows the two interfaces supporting the extended task features. The Interface `MicroComponentTask` enables posting of extended tasks, and the interface `MicroComponentTaskEvents` provides the slots for user-defined functionality through five event functions. `run()` and `runDone(...)` implement the first and second phase of the micro-components, respectively. For one-phase micro-components, the `runDone(...)` function is executed before leaving the component, while for split-phase micro-components the function is executed when the system fires the second phase. The functions `getElement()`, `setElement(...)`, and `freeElement(...)` change of the generic parameter sent to the task.

```
interface MicroComponentTask{

    event bool isRunning();
    event void postTask();
}

interface MicroComponentTaskEvents{

    command void      postNextTask(error);
    command stack_t * getRunStack();

    event   error_t   run();
    event   void      runDone(error, element);
    event   void      setElement(element);
    event   void *    getElement();
    event   void      freeElement(element);
}
```

Figure 4.8: MicroComponentTask Interface source code

Micro-component Congestion Control . Micro-components can suffer congestion or overload problems if many requests are posted to them at once. This can happen when the queue of a slow micro-component is full and its predecessor micro-component is trying to post an additional request. For example, the micro-component to transmit a wireless message is inherently slow, while the previous micro-component that prepares a message for transmission is comparatively faster.

To deal with this situation, we implemented a back-off timing mechanism available at each micro-component. If a micro-component tries to post a request to one that whose queue of requests is full, the current micro-component enters a timer to delay the next post. At that timeout, the next task is re-posted. If the post fails again, the process is repeated with an exponential back-off that multiplies the previous timer length by two. This process is repeated until a maximum wait of 512 milliseconds is reached. After that, the micro-component keeps trying using the same wait time value until a successful post happens.

4.5 Summary

The micro-protocol architecture explained in this chapter enables creation and optimization of agent-based WSN applications. A WSN application is divided in several network functions represented by micro-protocols. Each micro-protocol is composed of a sequence of actions that can be optimized with the help of agents, and each action is implemented with a micro-component. The micro-component framework also described in this chapter supports the implementation of this architecture on modern sensor network nodes running TinyOS.

Chapter 5

Evolutionary Collection Protocol

This chapter describes the Evolutionary Collection Protocol (ECP), a routing protocol for WSNs we designed to test our agent-based approach and implemented with the micro-component framework. ECP is a collection protocol for WSNs. As described in Section 2.1.1, a collection protocol takes information collected from the physical world using sensor devices and relays the sensor readings towards a central base station or server using multi-hop wireless communication [33].

ECP is an agent-based version of the Collection Tree Protocol (CTP) [28], a protocol used to collect information in WSNs and send it towards one or more main locations. It is a best-effort, multi-hop delivery protocol where some nodes advertise themselves as root nodes, and the other nodes collect information and send it to some root node. CTP is described in more detail in Chapter 2.

The rest of this chapter is organized as follows: Section 5.1 provides an overview of the protocol. Section 5.2 describes the micro-protocols and actions of the routing engine in ECP. Section 5.3 describes a simple example ECP executioun. Following this, Section 5.4 describes the structure of the agents used to implement the routing and Section 5.5 describes the agent interactions and structure of the evolutionary games in the protocol.

5.1 ECP Overview

ECP is a collection protocol for WSNs with features to improve power management and node connectivity in the presence of faults. It does this using an adaptive system for communicating and monitoring the network nodes built with the agent-based approach described in this dissertation.

The protocol is composed of two parts, a routing engine and a forwarding engine. The routing engine creates and updates the routing table for each node, and periodically advertises connection information to the neighborhood to find a path towards some root node. The forwarding engine sends data packets towards a root node using the routes maintained by the routing engine.

We selected the routing engine to test the agent-based implementation using the micro-protocol architecture because it has several interesting elements to optimize. In addition, routing is challenging in a faulty environment. Because of this, this chapter focuses on describing the routing engine of ECP.

5.1.1 Basic Features

ECP is a protocol used to collect information from the environment. Some nodes sample data and send it to locations called root nodes that receive and summarize the collected values in some way.

Routes to root nodes are created by sharing local information between neighbor nodes, specifically the link quality between each pair of nodes. The link quality is defined as the number of received packets divided by the total number of packets transmitted between two nodes during a short time period.

Nodes also share a cumulative value or cost to reach a root node called ETX, like in CTP described in Chapter 2. The ETX for a root node is always zero, and the ETX between two directly communicating neighbor nodes is a value derived from their link quality. The cumulative ETX for a node B is the sum of the ETX from B to its best neighbor A, i.e. the neighbor with smallest cumulative ETX, plus the cumulative ETX of A to the root. ETX is used to make routing decisions for data packets in the network.

5.1.2 ECP Enhancements

In the current implementation, most ECP enhancements over CTP focus on the routing engine. The new features aim to optimize the number beacons sent to minimize power consumption and improve node connectivity in the presence of faults.

ECP optimizes the number of beacons sent with an adaptive beacon mechanism. That mechanism works by assigning an agent to send the next beacon, with the agent parameter controlling the time when the beacon is sent. In general, more beacons per unit time are required when the link between two neighbor nodes is being calculated, but the number of beacons can be reduced after the network link costs are already calculated. The original CTP uses trickle-timers [47] with exponential increase as the adaptation strategy. ECP relies on the game rules and the random selection of agents to execute the advertising function and adapt to changes in network conditions.

ECP improves node connectivity by having agents to monitor connections with neighbor nodes. ECP assign agents to monitor individual routing tables entries, and the node advertising operations get partially synchronized when agents coming with beacon packets and agents monitoring failures agree on the timing and the timeouts of the process.

5.2 ECP Routing Engine

The routing engine executes the advertising process in ECP. This module updates the routing tables to find a path towards a root node. The engine is implemented using the micro-protocol architecture, and is composed of three micro-protocols:

- **Beacon Advertising**. This micro-protocol periodically advertises a node to the network using beacon packets with information about the cost (ETX) to reach some root node.
- **Beacon Receive**. This micro-protocol is executed each time a beacon packet is received. It updates the routing table with information in the beacon packet.
- **Neighbor Check Fault**. This micro-protocol periodically monitors connected neighbors, waiting for beacons from the neighbors and detecting timeouts of routes in the routing tables.

5.2.1 Beacon Advertising

The Beacon Advertising micro-protocol starts the advertising process at each node. It is composed of the three actions shown in Figure 5.1.

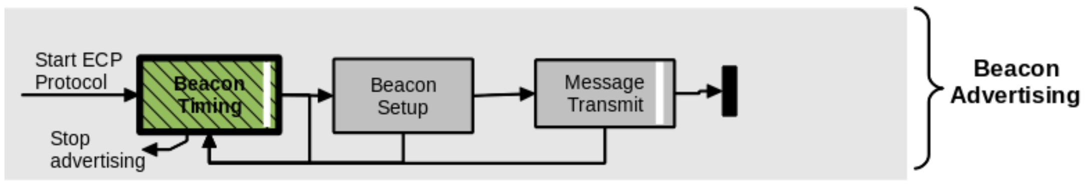


Figure 5.1: ECP beacon advertising micro-protocol

The Beacon Timing action waits some time before sending the next beacon packet. It is implemented with a virtual and split-phase micro-component executed by an agent. The agent executing the action is located in a structure called ecp_mote_t. This structure contains information about the node and its location in the network.

```
typedef struct {
    state_t       state;
    statistics_t  statistics;
    shell_t       playerShell;
} ecp_mote_t;
```

Figure 5.2: ECP Mote structure

Figure 5.2 shows the ecp_mote_t structure. The field state has information about the node state used to fill the beacon packets. The field statistics contains information about the cost of the best located neighbor, and this information is used to advertise the node to other neighbors. The playerShell field contains the agent preparing the next beacon packet.

The Beacon Setup and Message Transmit actions in Figure 5.1 are not subject to optimization, and there is no agent involved in their execution. Beacon Setup is implemented as a single micro-component to fill the beacon packet data structure, and Message Transmit is implemented with a split-phase micro-component that actually transmits the message using the wireless transmitter.

5.2.2 Beacon Receive

The Beacon Receive micro-protocol updates the routing table of the node every time a beacon packet is received. A routing table in ECP is an array containing information about directly-connected or 1-hop-distance neighbor nodes. The maximum table size is 15 entries for the current implementation of the protocol. The information for each entry is represented by the structure ecp_neighbor_t in Figure 5.3.

```
typedef struct {
    mote_id_t    id;
    state_t      state;
    time_info_t  timeInfo;
    statistics_t statistics;
    shell_t      playerShell;
} ecp_neighbor_t;
```

Figure 5.3: Routing table entry structure in ECP

The structure provides an `id` for each neighbor node, some bits with `state` information about the neighbor, a `timeInfo` timer to monitor the connection with the neighbor, `statistics` with information to support agent decisions, and `playerShell` to store the agent monitoring the corresponding neighbor node.

The Beacon Receive micro-protocol is shown in Figure 5.4. It is executed every time a beacon packet is received from a new or current neighbor node with information to update the routing table. The Filter Rules action executes sanity checks for the received packet, and can also be used to implement filtering and firewalling operations on received messages. Enqueue Message and Dequeue Message implement conventional message queue processing to speed up the reception of packets at the receiver. The Pick Message Type action demultiplexes the processing flow for messages. The Neighbor Update action inspects the routing table to add an entry if the beacon is coming from a new neighbor node. None of these actions are associated with agents for optimization.

The Advertising Game action evaluates the performance of the agents. It defines the game between the agent coming in the beacon and the agent monitoring the corresponding neighbor node in the routing table. This action defines how agents survive or die, according to the specific rules of the game. Section 5.5.3 describes the general structure of the game, and Chapters 6 and 7 describe different game rule

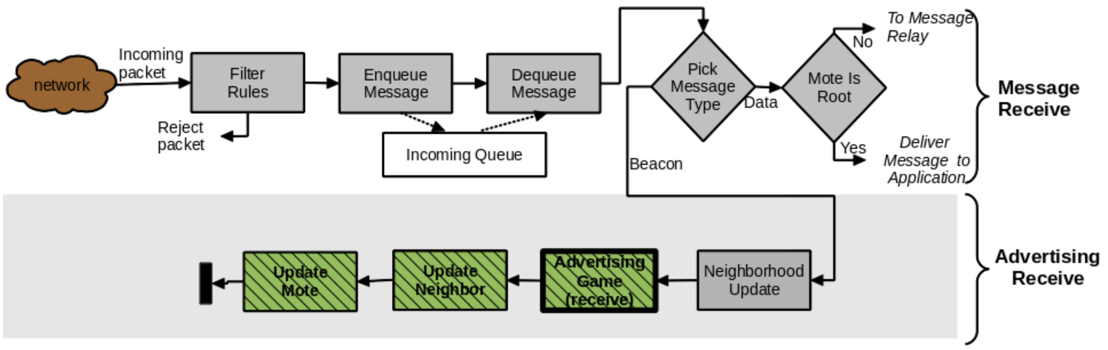


Figure 5.4: ECP beacon receive micro-protocol

implementations and results for those games.

Finally, actions Update Neighbor and Update Mote update the routing table entries and the state of the node according to the information inside the beacon packet.

5.2.3 Neighbor Check Fault

The Neighbor Check Fault micro-protocol periodically checks for timeouts with neighbors and disables the corresponding entries from the routing table when required. Figure 5.5 shows the micro-protocol components.

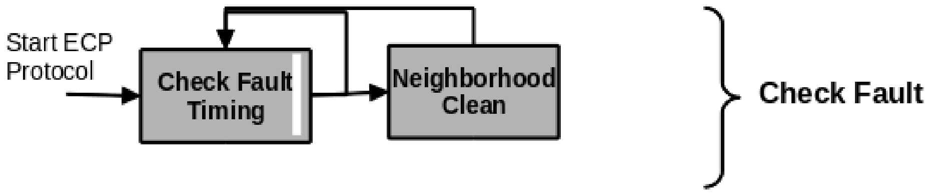


Figure 5.5: ECP check fault micro-protocol

The Check Fault Timing action sets a timer with the period of the check. The current implementation of the protocol uses a fixed value of half the length of the smallest time for beacon advertising strategies for this timer. The Neighborhood Clean action does the actual check and disables the neighbors with due timeouts. After disabling a neighbor entry, the agent monitoring that entry is returned to the selection room of the node.

5.3 ECP Execution Example

To understand the high-level processing of packets using the micro-protocol architecture in ECP, consider a basic test application that simply starts ECP running. That application defines an EcpC component in the NesC language to start running the protocol.

The steps executed by the EcpC component during the boot process of a sensor node to start the protocol operation are:

1. Initialize the wireless transmitter.
2. Create an initial population of agents.
3. Initalize an empty routing table for the protocol.
4. Initialize protocol state variables.
5. Run the Beacon Advertising Micro-protocol.
6. Run the Neighbor Check Fault Micro-protocol.

After the Beacon Advertising and Neighbor Check Fault micro-protocols start running, no other additional function calls are required to keep the routing engine running; the automatic run features provided by the micro-component framework periodically executes all the actions contained in the micro-protocols to update the routing tables and check for connectivity.

5.4 Agents in ECP

ECP uses one type of agent to optimize the routing engine processes. Agents of the routing engine are called *advertisers* because they are in charge the advertising operations of the node

5.4.1 Structure of Agents

Agents in ECP have a compact and efficient definition to keep the amount of memory they use small. The current implementation of the protocol allocates a maximum number of 100 agents per node with a total size per agent of 5 bytes. This value is small enough to store on agent inside a beacon or data packet, and a fully occupied node requires only 500 bytes for its population.

Advertisers agents have a data structure containing the following fields:

- **Id**: a numeric identifier for the type of agent; this field is 3 bits in size, and provides the capability to extend the system with more types of agents in the future.
- **Reserved**: a field for future use, this field is 5 bits size.
- **Strategy set**: a 32-bit array representing the strategy set of the agent. This 32-bit variable is split in several bit-fields to represent individual strategies.

5.4.2 Advertiser Strategies and Interactions

Advertisers optimize the routing engine. They also check for node connectivity by monitoring the links with neighbors of the routing table. Advertisers have the following strategies:

- **Beacon Timing Value** is the strategy defining how much time the agent will wait before sending a beacon packet. The current implementation maps fixed time intervals to a corresponding binary value for the strategy, with time intervals ranging from 200 milliseconds to 5.3 minutes.
- **Energy Saving Time Threshold** is the strategy defining the randomization to be applied to the beacon time value. This strategy helps to prevent collisions of packets sent simultaneously. The current implementation assigns a range between 10% to 80% of the beacon time value of the agent.

Advertisers participate in the following actions of the routing engine:

- **Beacon Timing**: The advertiser located at the advertising location of the node generates the time to send the next beacon packet.
- **Advertising game**: The advertiser arriving at a node with a beacon packet plays a game with the advertiser monitoring the neighbor to determine who survive and who die.
- **Update Neighbor**: The advertiser arriving at a node updates the corresponding entry of the routing table.
- **Update Mote**: The advertiser arriving at a node with a beacon packet updates the node information.

5.5 Interactions in ECP

Agent interactions in ECP are defined according to the types of interactions described for the agent-based approach in Chapter 3. This section describes general characteristics of interactions in ECP, and later chapters describe specific interactions implemented and evaluated in game variants.

5.5.1 Agent Creation

Agent creation occurs every time a new agent or replica of an existing agent is generated as a consequence of other interactions. It can also happen when there are no available agents to execute a network function. A special case happens when a node is booted and an initial population of agents is created to start node operations. In all cases, the function used to generate the strategies of the new agent depends on the rules defined for the specific game. The following is a description of the cases when a new agent is created in ECP.

No available agents in the selection room. A new agent is generated when one is requested to execute a network function, such as monitoring a routing table entry or sending a beacon packet, but there are no available agents in the node to execute the function. This case includes the generation of an initial population of agents to start node operation, and according to the game rules. The specific function used to generate the strategies of the new agent is game dependent, for example it can be a function generating random values for the strategies.

Replication by wireless transmission. This is a natural replication mechanism in ECP, and it is possible because of the wireless communication available between nodes. Every time a beacon packet is transmitted using broadcast communication, the neighbor nodes in the range of transmission receive a copy of the beacon, and consequently a replica of the agent contained inside the original packet.

Replication in place. In some cases, a game can define rules to create a new agent following the selection of another agent from the existing population. We present an example of this replication in Chapter 7. The specific function used to generate the strategies of the new agent is again game dependent.

5.5.2 Agent Selection

Agent selection in ECP happens before executing a micro-component which is subject to optimization and requires an agent to execute the action. Examples are the selection of an agent to send a beacon packet, or the selection of an agent to monitor an entry of the routing table. Selection criteria may vary, for example, selecting an agent randomly from the existing population, or using some criteria to preferentially select an agent from the existing population. The specific selection process used is also game dependent.

5.5.3 Agent Competitions

Agent competitions in ECP enable evaluation of how well different agent strategies are doing. Agent competitions have specific definitions depending on the game implemented. In the ECP routing engine, the advertising game requires two parameters, namely the time interval between consecutive beacon packets sent from one node, and the link quality between two communicating nodes.

We divide the time between successive beacon packets into two time intervals, the deny (D) and the accept (A) interval. These intervals classify how fast or slow the packets are coming from the source node. The deny interval represents a period in which beacons may be arriving too quickly. The accept interval represents a period in which the node desires new information from the neighbor. Figure 5.6 contains a representation of the timing parameters used in the games.

Agent competitions also consider how fresh (F) or stale (S) the link quality of an update between two nodes is by defining a minimum threshold value for the change in this link quality compared to previous information collected for the communicating nodes. If the link quality changes more than the threshold, the information is

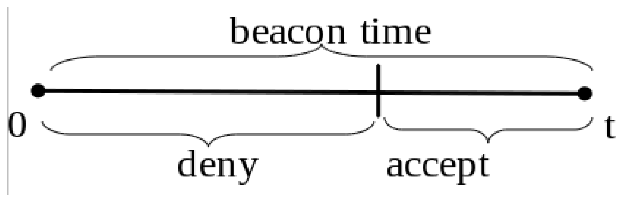


Figure 5.6: Beacon Time Parameters of ECP Games

considered fresh, but if the change in link quality is smaller than the threshold, the incoming information is considered stale.

ECP competition rules state which agents survive a competition considering the combinations of Deny/Accept interval and Fresh/Stale information. For each of the parameter combinations shown in Table 5.1, we have three possible kill actions for the agents participating in the game, namely kill the agent already existing (E) in the neighbor node receiving a beacon, kill the agent incoming (I) into the neighbor node, and kill none (N) of the agents by returning them to the selection room. Chapter 6 describes the specific values for the competition rules we evaluated.

	Fresh	Stale
Accept	FA	SA
Deny	FD	SD

Table 5.1: Parameters for Advertising Game in ECP

We identified the games by the value of the two parameters and by their kill rules, for example SA:N means if the beacon comes with Stale information during the Accept time interval, kill None of the agents. We use this notation in Chapters 6 and 7 to provide an easier way to identify the games.

5.6 Summary

This chapter presented ECP, a collection protocol for WSNs implemented using the agent-based approach described in Chapter 3. ECP inherited its main features from CTP, another network protocol designed for WSNs that is part of the TinyOS operating system. ECP includes enhancements to improve power management and node connectivity in the presence of faults using an agent-based approach supported by evolutionary games.

ECP is composed of a routing engine to create the network communication paths, and a forwarding engine to send the data collected inside the network towards main locations or root nodes. We selected the routing engine to implement and test our agent-based approach.

ECP uses advertiser agents to optimize the routing engine. The game interactions in ECP define operations for creation, selection, and competition between agents that evaluate the performance of different strategies.

Chapter 6

ECP Game Structure

This chapter examines the behavior of an evolutionary version of the game structure and rules in the ECP routing engine described in Chapter 5. First, we study a set of *basic games* to understand the impact of different rules on the behavior of ECP games. This evaluation focuses on power consumption and node connectivity in the presence of failures. We tested the basic games and analyzed their behavior running primarily in faulty environments; results for the behavior of these games in failure-free environments are presented in Appendix A.

Our analysis of these results identifies a small subset of rules that are viable for ECP as well as minor flaw in the design of the basic setup of the original game. Based on this, we study an improved game structure, the *aligned games*, and evaluate the remaining subset of rules in the context of this game to find the rules most appropriate for ECP in our test environment.

In the remainder of this chapter, Section 6.1 explain the methodology we used to design the tests and Sections 6.2 and 6.3 follow with a description of the basic games and the results of their evaluation. Sections 6.4 and 6.5 then describe the aligned games and the results of our evaluation of these games. Section 6.6 summarizes our results.

6.1 Methodology

This section describes the methodology used to evaluate ECP games. The ECP application we created to test the games was executed in a simulated environment covering multiple game configurations, a network topology, and environmental conditions.

6.1.1 Evaluation Metrics

We considered two metrics to measure power consumption and node connectivity in the presence of faults:

- **Total number of beacons sent**. We use the total number of beacons sent for for an entire simulation to evaluate power consumption of a game, as packet transmission is extremely costly in WSNs. Lower values correspond to improved power consumption.
- **Number of timeouts**. We use how many times an entry of the routing table timed out to measure node connectivity. Lower values correspond to improved connectivity.

In addition, we also collected the average beacon time and population dynamics of the agents for analysis purposes. Examining the average beacon time per node allows us to analyze how beacon time adapts to changes in network conditions. The population dynamics show how the different strategies make progress over time for different game rules and network conditions.

To measure statistical significance of the results we used one-way Analysis of Variance (ANOVA). We compared when different games behave statistically similarly; our null hypothesis is that *the behavior of the games is the same*, meaning that changing the rules of the games make no difference in the results. We reject the null hypothesis based on a 95% confidence interval ($p \leq 0.05$).

6.1.2 Network Scenarios

We considered several network scenarios for the test application. Two elements make up the scenarios, the network topology and the environmental conditions.

- **Network topology** defines the number of nodes in the network and their spatial distribution.
- **Environmental conditions** specifies the network parameters, particularly relating to communication and node failures.

We defined a network topology with 9 nodes distributed in a 3x3 mesh and 4 meters between neighbor nodes to test our approach. In addition to the network topology, we selected three environmental conditions in which to run the tests:

- **Failure-free environment**, an environment with no faults in the network.
- **Simple-fault environment**, an environment with two node crash/reboot faults, each at different locations of the network and at different times.
- **Periodic-fault environment**, an environment with a node crash/reboot fault at the center node in the mesh every 10 minutes.

The failure-free environment provided an initial reference environment to test and compare all the games, and results for the failure-free environment are presented for reference in Appendix A. This chapter focuses on the faulty environments because our thesis is specifically related to network performance in the presence of faults.

6.1.3 Simulation Configuration

Tests were run using the TinyOS Simulator (TOSSIM) [49] for TinyOS 2.1. We simulated each game/environment for a 180 minute simulation 50 times, with a new random number seed for each simulation and node for each run. We use full logs of the events in the run to collect data for evaluation and analysis.

The wireless environment for simulation was defined using the features available in TOSSIM. Table 6.1 shows the values required to define the channel and the radio. Channel parameters define the gain at which other nodes receive a signal when a node transmits. Radio parameters produce variations in communication. The channel model is based on the Log-Normal Shadowing Path Loss Model and the parameters are Path Loss Exponent, Shadowing Standard Deviation, DO and PL(D0). Radio parameters are Noise Floor, White Gaussian Noise, S11, S12, S21, S22. The last four values are hardware specific to the Micaz hardware platform used for the tests.

Topology	**Environment definition (channel params)**	**Asymmetry Level (radio params)**

Mesh	Path Loss Exp=4.7
	Shadowing Std Dev=3.2
	D0=1.0, PLD0=55.4
		Noise Floor=-105.0
		S11=0, S12=-1,
		S21=-1, S22=0
		WGN=4

Table 6.1: Wireless Environment Setup for Simulation

6.2 Basic Game Description

The basic games are a fully evolutionary approach to optimizing the routing engine of ECP. These games aim to improve power consumption and node connectivity in the presence of failures by adjusting the time between beacon packets, and monitoring the communication between neighbor nodes. To adjust the timing of beacon packets, each node advertises periodically some information to reach a root node, and also exchanges information about the link quality with all its neighbors.

The basic games create and update the routing tables for all the network nodes using a population of agents that interact using the game rules defined later in this section, and the two parameters described in Section 5.5.3:

- **Beacon Time Interval**: the total wait time before sending the next beacon.
- **Accept Interval Size**: the size of the accept interval. This value is given as a percentage of the beacon time interval.

The rest of this section describes agent creation, selection, and competition for basic games.

6.2.1 Agent Creation and Selection

The creation of agents in basic games is done by randomly assigning values to the strategies of new agents. When a node starts running, an initial random population of agents is generated. After that, new agents are created randomly if there are no agents available in the population.

The selection process of agents from the selection room to execute a network function is also random. When a node needs to send a beacon packet, it selects an agent randomly, and the time to send the beacon is set according to the beacon time strategy of the selected agent. A similar process is executed to assign agents for monitoring incoming beacon packets in the routing table.

6.2.2 Agent Competitions

Competitions occur at the receiver node when a beacon packet is received. The agent arriving with the beacon and the agent monitoring the incoming beacon play the game. The rules for the competition are defined in terms of the timing of the receiver agent, and the change in the quality of the link between the nodes, as explained in Section 5.5.3.

We defined nine sets of rules for basic games that consider different options for the survival of the agents. We defined a fixed rule of kill incoming (I) for the Stale/Deny (SD) combination of parameters and a fixed rule of kill none (N) for the combination Fresh/Accept for all basic games.

Game	**Stale/Accept**	**Fresh/Deny**
B/SA:N/FD:E	Kill None	Kill Existing
B/SA:I/FD:E	Kill Incoming	Kill Existing
B/SA:E/FD:E	Kill Existing	Kill Existing
B/SA:N/FD:I	Kill None	Kill Incoming
B/SA:I/FD:I	Kill Incoming	Kill Incoming
B/SA:E/FD:I	Kill Existing	Kill Incoming
B/SA:N/FD:N	Kill None	Kill None
B/SA:I/FD:N	Kill Incoming	Kill None
B/SA:E/FD:N	Kill Existing	Kill None

Table 6.2: ECP Basic Games Rules

Table 6.2 presents the rule variants we evaluated. We used the same notation explained in Section 5.5.3 to identify the games, for example the name B/SA:N/FD:E represents the **B**asic game where the **S**tale/**A**ccept rule is Kill **N**one, and the rule for **F**resh/**D**eny is Kill **E**xisting. The first row of the table defines the game rules according to ECP parameters excluding the parameter values we assigned a fixed rule. The internal table entries show the actual rule applied to the agents.

6.3 Basic Game Results

This section presents the results of the basic games in the simple-fault environment. We discuss the results based one-way analysis of variance, and we also analyze the network behavior and how agent interactions produce it. Appendix A includes additional data on the behavior of the basic games in the failure-free environment.

6.3.1 Results

The results for the simple-fault environment are presented in Table 6.3. This table shows the average total number of beacon packets sent during a simulation for each basic game, and the total number of timeouts detected according to the agents monitoring the routing tables.

The number of timeouts for all the basic games is high, about 2 timeouts per minute for the best case. This problem is not well addressed in these games, because their is no clear mechanism defined to synchronize the timing of the agents at both sides of the connection, and this generates high rates of timeouts.

ANOVA results over the nine basic games show that the games are not the same either for the number of beacons sent (p=0.0000) or the number of timeouts (p=0.00000). We then discarded the games that never kill an agent for Stale/Accept (B/SA:N/FD:*) because they show a high number of beacons sent and high number

Game	**Beacons Sent**	**Timeouts**
	(Avg ± StdDev)	(Avg ± StdDev)
B/SA:N/FD:E	13418 ± 19084	4168 ± 4462
B/SA:I/FD:E	510 ± 59	336 ± 47
B/SA:E/FD:E	513 ± 57	332 ± 47
B/SA:N/FD:I	18422 ± 21909	4655 ± 4588
B/SA:I/FD:I	518 ± 35	339 ± 43
B/SA:E/FD:I	517 ± 36	336 ± 47
B/SA:N/FD:N	7051 ± 3294	1303 ± 723
B/SA:I/FD:N	604 ± 142	319 ± 45
B/SA:E/FD:N	601 ± 172	321 ± 53

Table 6.3: Basic Game Results - Simple-fault Environment

of timeouts.

We then applied ANOVA to the six remaining games, namely B/SA:I/FD:* and B/SA:E/FD:*, that kill some agent in Stale/Accept. These games are also different for the number of beacons sent (p=0.00000), but we cannot reject the null hypothesis that the games are identical for the number of timeouts (p=0.17050).

Next, we compared these six games in pairs, specifically games differing on the agent they kill for Stale/Accept (for example, B/SA:I/FD:E with B/SA:E/FD:E). In all cases, we could not reject the null hypothesis for beacons sent ($p \geq 0.76258$) and number of timeouts ($p \geq 0.69159$). Based on this, we chose to kept one game of each pair, the three games killing the Incoming agent for Stale/Accept (B/SA:I/FD:*).

6.3.2 Analysis

Results for basic games running in the simple-fault environment can be better understood when we consider the population dynamics produced by the interactions of the agents. Figure 6.1 shows the average population dynamics generated by a game not killing any agent for Stale/Accept (B/SA:N/FD:E) and a game killing the Incoming agent (B/SA:I/FD:E). These two plots are representative of the population dynamics found in basic games.

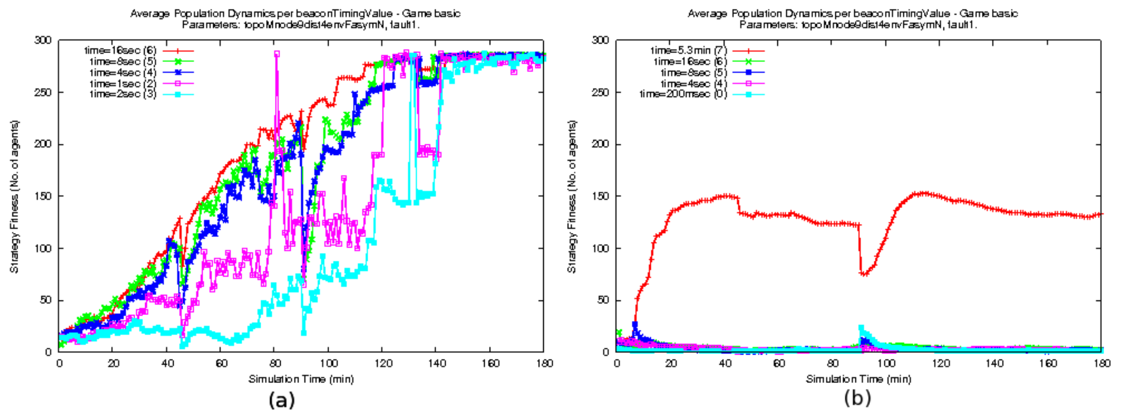


Figure 6.1: Basic Game Population Dynamics - Faulty Environment

From Figure 6.1 (a) we can see the population dynamics for the game not killing any agent is not stable between runs because several different strategies evolve in different runs, most of them with small beacon time values. In practice, not killing any agent with Stale/Accept favors agents with small beacon times, because there is no penalty for sending quickly. Such strategies reproduce very quickly, and slower agents are killed while faster agents replicate and populate the node.

Figure 6.1 (b) shows a game killing the Incoming agent. In this case, the population dynamics are stable between runs because the game penalizes fast agents bringing stale information. Fast strategies make progress at the beginning of the simulation when setting up the routing tables with fresh information. After that, they start being killed by the slower agents that make progress if the network conditions stay the same. They can also make progress again if network conditions change like in the faults shown in the Figure 6.1 (b).

Figure 6.2 shows the behavior for average beacon time. After the first crash at minute 45, the beacon time decreases a few minutes after the event. In this case, the crashed node is not in a central position in the network, so fast agents coming in the replacement node take some time to spread over the network. For the second crash at minute 90, the reaction is faster. This happens because the node is at a central position in the network, and fast agents in the replacement node have more options to spread and update the network quickly.

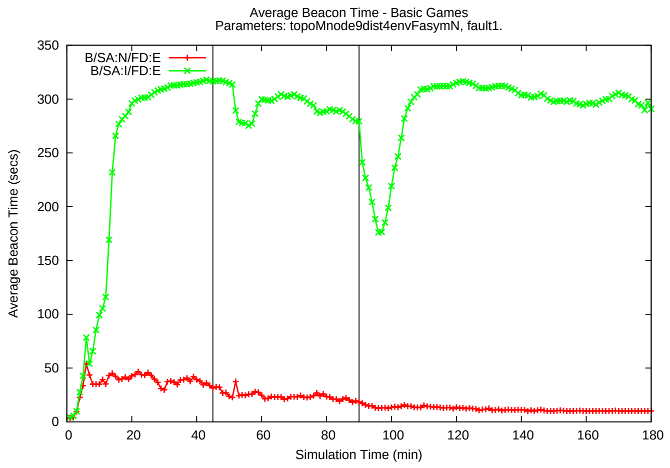


Figure 6.2: Average Beacon Time for B/SA:N/FD:E and B/SA:I/FD:E - Simple-fault Environment

6.4 Aligned Game Description

We designed the aligned games to address the connectivity issues found in basic games, specifically to reduce the number of timeouts when monitoring neighbor nodes. Aligned games have similar components to the basic games, namely random creation and selection, the same strategies with deny and accept intervals, and the fresh/stale parameter to evaluate changes in the link quality between nodes. However they modify the rules to attempt to synchronize agent actions and to reduce the number of timeouts in the routing tables.

Changes compared to the basic game. The aligned game examines how long an agent waited at the sender before transmitting the beacon, instead of simply considering when the beacon arrived at the receiver as in the basic games. This process also attempts to removes random timing introduced by the network from the game's comparison. It also favors agents when both the sender and receiver agents agree on the appropriate beacon time.

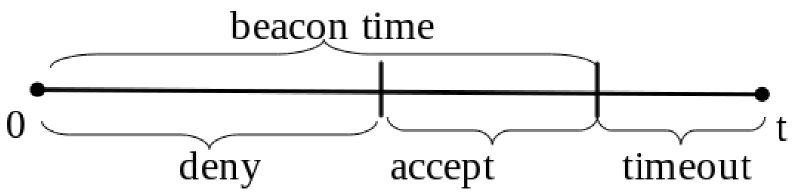


Figure 6.3: Aligned Game Beacon Time Strategy for ECP

Compared to the basic games, the aligned games have three different time intervals, *deny*, *accept*, and *timeout*. When an incoming beacon arrives, it is placed into one of these intervals by using the beacon time in the arriving beacon as an offset from time 0 in the receiving beacon. As in the basic game, beacons sent very quickly are placed into the *deny* interval and beacons sent less quickly are placed into the *accept* interval. Beacons that were sent very slowly, however, are placed into a new *timeout* interval.

Rules in the aligned games. For the aligned games, we considered the three remaining basic games from Section 6.3, those that killing the Incoming agent during the Stale/Accept interval, as shown on Table 6.4. We also added fixed rules for the timeout interval. In particular, Stale/Timeout always kills the incoming agent and Fresh/Timeout always kills the existing agent.

Game	Stale/Accept	Fresh/Deny
A/SA:I/FD:E	Kill Incoming	Kill Existing
A/SA:I/FD:I	Kill Incoming	Kill Incoming
A/SA:I/FD:N	Kill Incoming	Kill None

Table 6.4: ECP Aligned Games Rules

6.5 Aligned Game Results

6.5.1 Results

The results for the aligned game running in a faulty environment are shown in Tables 6.5 and 6.6 for simple-fault and periodic-fault environments respectively. ANOVA tests for these three games show that they are statistically different (p=0.00000) for the number of beacons sent and the number of timeouts. Comparing the games in pairs, the game that does not kill any agent for Fresh/Deny (A/SA:I/FD:N) is different (p=0.0000) from the other two in the number of beacons sent and the timeouts. However, we can not reject the null hypothesis ($p \geq 0.45509$) when comparing the games that kill either agent for Fresh/Deny (A/SA:I/FD:I and A/SA:I/FD:E).

From the table in terms of power, we can also see the game not killing any agent for Fresh/Deny generated between 35% and 40% less packets than the other two games that kill some of the agents, but there are more packets sent with the aligned games when compared with the basic games in Section 6.3.

Node connectivity improves in terms of the number of timeouts in the game not killing any agent for Fresh/Deny (A/SA:I/FD:N). It is about 35% fewer timeouts than basic games. The game not killing agents for Fresh/Deny also generated 37% less timeouts than the other tested aligned games.

Game	**Beacons Sent** (Avg ± StdDev)	**Timeouts** (Avg ± StdDev)
A/SA:I/FD:E	1071 ± 92	251 ± 33
A/SA:I/FD:I	1068 ± 62	256 ± 30
A/SA:I/FD:N	785 ± 83	212 ± 25
B/SA:I/FD:E	510 ± 59	336 ± 47
B/SA:I/FD:I	518 ± 35	339 ± 43
B/SA:I/FD:N	604 ± 142	319 ± 45

Table 6.5: Aligned Game Results - Simple-fault Environment

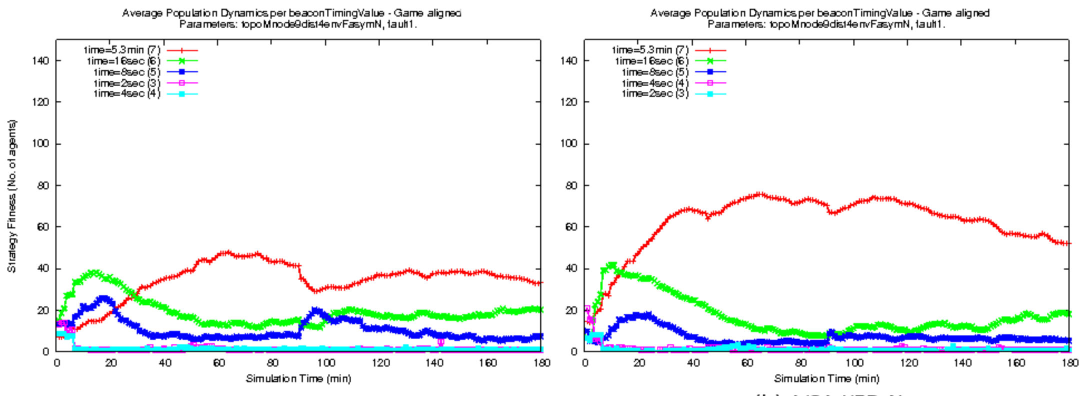
Game	**Beacons Sent** (Avg ± StdDev)	**Timeouts** (Avg ± StdDev)
A/SA:I/FD:E	1162 ± 66	489 ± 47
A/SA:I/FD:I	1160 ± 75	491 ± 49
A/SA:I/FD:N	700 ± 63	310 ± 42

Table 6.6: Aligned Game Results - Periodic-fault Environment

6.5.2 Analysis

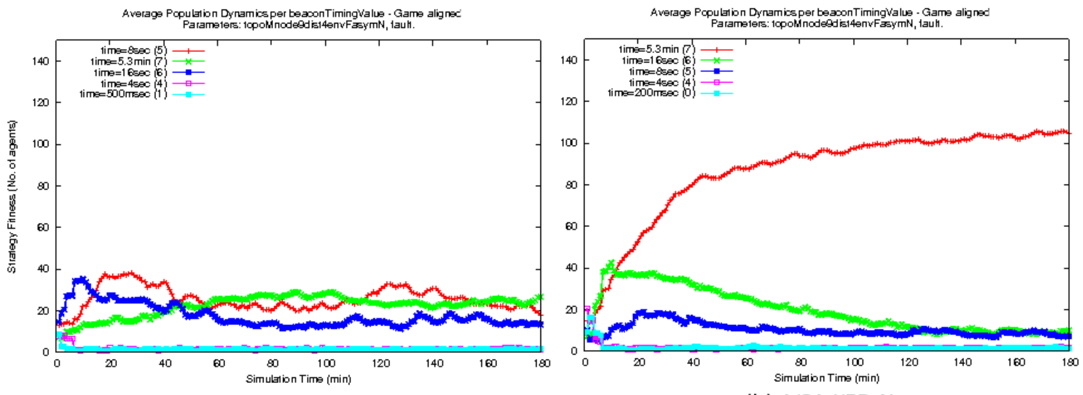
Figure 6.4 shows the population dynamics for game A/SA:I/FD:E and A/SA:I/FD:N running in the simple-fault environment. Figure 6.4 (b) shows that the game that does not kill agents bringing fresh information (A/SA:I/FD:N) favors faster changes in link quality, and the slow agents make progress faster after link qualities have been updated. This produces larger beacon times, and results in a population with larger beacon times compared to the other aligned games, such as the population dynamics shown in Figure 6.4 (a) for game A/SA:I/FD:E.

The population dynamics for the game killing no agents for Fresh/Deny (game A/SA:I/FD:N) in an environment with periodic faults, as shown in Figure 6.5 are similar. The fast agents make progress after node reboots for a short time, but the slow agents dominate after that.



(a) A/SA:I/FD:E (b) A/SA:I/FD:N

Figure 6.4: Aligned Game Population Dynamics - Simple-fault Environment



(a) A/SA:I/FD:E (b) A/SA:I/FD:N

Figure 6.5: Aligned Game Population Dynamics - Periodic-fault Environment

The described population dynamics produces average beacon times shown in Figures 6.6 and 6.7 for simple-fault and periodic-fault environments, respectively.

Overall, the aligned game that does not kill any agent that brings fresh information (A/SA:I/FD:N) has a larger population of agents containing longer beacon times, but retains a smaller population of agents with shorter beacon times to update link quality during network changes.

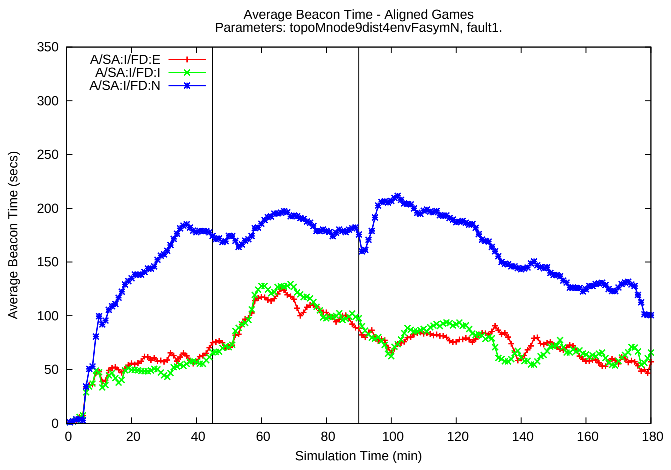


Figure 6.6: Average Beacon Time for Aligned Games - Simple-fault Environment

6.6 Summary

This chapter evaluated the performance of different variants of an evolutionary game that optimizes beacon transmission time in ECP. The results show that different game rules can result in dramatically different routing protocol behavior, including ones that behave very poorly and others that correctly control beacon transmission speeds in the presense of faults to maintain connectivity and minimize power consumption. In addition, our results demonstrate the importance of taking into account asynchrony introduced by the network in game design, and that games that do not penalize overly aggressive beacon transmission behave poorly.

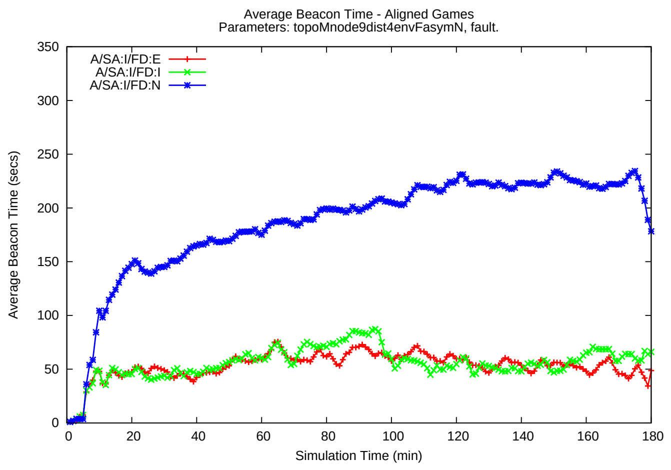


Figure 6.7: Average Beacon Time for Aligned Games - Periodic-fault Environment

Chapter 7

ECP Rule-based Game Comparison

In this chapter we compare the best evolutionary game from Chapter 6 with a game that use heuristics to set beacon time and detect timeouts. In addition, we also compare these games with games that use hybrid approaches that attempt to use designed heuristics to bias the behavior of evolutionary approaches. Together, these approaches allow us to evaluate the thesis statement put forth in Chapter 1.

In the remainder of this chapter, Section 7.1 presents an ECP variant that uses a fixed rule that examines network state to generate the specific agent to use to control beacon sending and timeout detection. While this approach is implemented in the context of ECP and our agent-based approach, it is a non-evolutionary approach because it does not use the existing population when selecting an agent. We then compare the performance of this rule-based approach with the best evolutionary game from Chapter 6. Section 7.2 then presents *hybrid games* that attempt to combine the advantages of the rule-based approach with those of the evolutionary approach by biasing the creation or selection of agents from the overall population, and present the results of these games. Finally, Section 7.3 summarizes the results from this chapter with a specific focus on the overall thesis of the dissertation.

7.1 A Rule-based ECP Variant

To compare the evolutionary approach in Chapter 6 with a non-evolutionary approach, we created a simple rule-based variant of ECP that uses a fixed rule to generate agent strategies when agents are needed, instead of creating and selecting agents randomly. As a result, the population of agents in the system does not affect system behavior. We compare the performance of the ECP variant with that of the best evolutionary ECP variant from Chapter 6 to test the thesis statement.

7.1.1 Agent Generation

The goal of the rule-based game is to speed up the response to changes in network operating conditions. We used a rule that picks beacon intervals based on the change in link quality. In particular, it uses small beacon intervals if the change in link quality is large, and large beacon intervals if link quality is stable. The specific values used are shown in Table 7.1.

These rules provide high-quality routing on a failure-free network, as shown in Table 7.2. They do this by sending many more beacon packets than the evolutionary variants of ECP presented in Chapter 6, and therefore use much more power.

Finally, we note that the rules chosen in this case are only one possible set of rules. It would be possible to design a different set of rules specialized to a particular topology or failure environment. In this case, we chose rules based on our experience with wireless sensor networks that we believed would work well for comparison with

Link Quality Change	**Beacon Time (secs)**
≤ 3	320
4–7	160
8–13	80
14–21	4
22–31	2
32–43	1
44–57	0.5
> 57	0.2

Table 7.1: Beacon times for given link quality changes in the rule-based ECP variant

Game	**Beacons Sent** (Avg $\pm$ StdDev)	**Timeouts** (Avg $\pm$ StdDev)
Rule-based	2956 ± 4942	38 ± 14
Aligned	902 ± 96	248 ± 37

Table 7.2: Rule-based ECP Results - Failure-free Environment

evolutionary approaches.

7.1.2 Comparison with Evolutionary ECP

To evaluate our thesis, we compare the performance of the rule-based ECP variant with the game with the best evolutionary version of ECP from Chapter 6 in the presence of failures. In particular, we compare versus the ECP variant that uses an aligned game, that kills the incoming agent in the Stale/Accept case and keeps both agents in Fresh/Deny case (game A/SA:I/FD:N).

We evaluated this game and the rule-based ECP variant in simple-fault and periodic-fault environments using the same methodology as the previous chapter. Tables 7.3 and 7.4 present the results of this comparison.

Game	Beacons Sent	Timeouts
	(Avg ± StdDev)	(Avg ± StdDev)
Rule-based	8345 ± 10081	232 ± 37
Evolutionary	785 ± 83	212 ± 25

Table 7.3: Rule-based ECP vs. Evolutionary ECP - Simple-fault Environment

Game	Beacons Sent	Timeouts
	(Avg ± StdDev)	(Avg ± StdDev)
Rule-based	28061 ± 46168	398 ± 73
Evolutionary	700 ± 63	310 ± 42

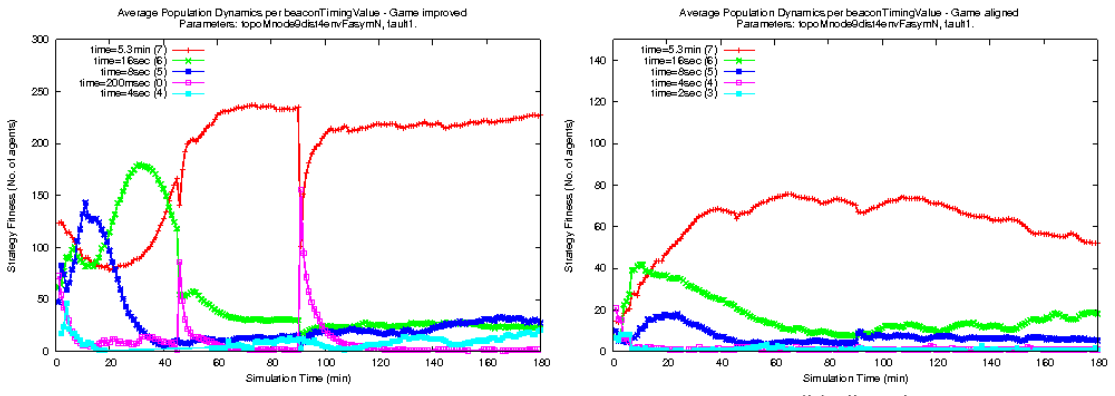
Table 7.4: Rule-based ECP vs. Evolutionary ECP - Periodic Fault Results

In the simple fault environment, the rule-based ECP variant sends more than an order of magnitude more beacons than the evolutionary version of ECP ($p = 0.00000$), but still results in more routing timeouts ($p = 0.00242$). In the periodic fault case, the rule-based system performs even worse, sending 40 times as many beacon packets and a greater number of routing timeouts ($p = 0.00000$).

7.1.3 Analysis

Figures 7.1 and 7.2 show the population dynamics for the rule-based and evolutionary variants of ECP in both faulty environments tested. Similarly, Figures 7.3 and 7.4 show how average beacon times change for these two ECP variants.

In general, the rule-based approach uses shorter beacon times as the system starts up, resulting in more beacons sent but fewer timeouts than the evolutionary approach prior to the occurrence of failures in the simple failure case. On the other hand, the rule-based approach appears to be much more aggressive than the evolutionary approach in sending beacons when faults occur. However, these added beacons do



(a) rule-based (b) aligned

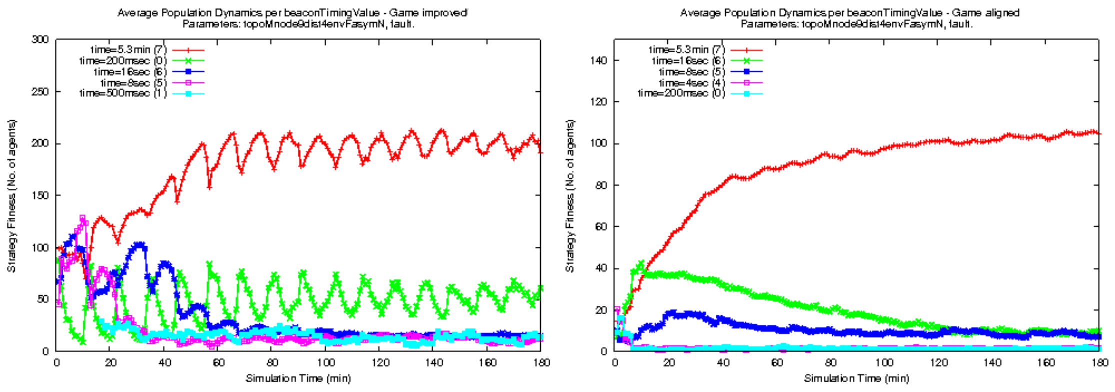
Figure 7.1: Rule-based ECP Variant Population Dynamics - Simple-fault Environment

not appear to reduce the number of timeouts that occur and only increase the power consumption of the protocol.

7.2 Hybrid Game Description

In addition to purely evolutionary and purely rule-based approaches, we also examined two hybrid approaches that combine aspects of each. In particular, these approaches use the rules of the rule-based game to influence the behavior of the evolutionary ECP variant by biasing agent creation and selection while still allowing the agent population to control network behavior. They do this by injecting new rule-derived agents into the general node population whenever an agent leaves the node. This biases the population with agents the rules believe will be helpful based on network conditions. Note that these agents still compete with existing agents in the system, including those created randomly at system startup.

For these variants, we used the optimized aligned game for agent competitions that was evaluated in Chapter 6 and used for comparison in the previous section



(a) rule-based (b) aligned

Figure 7.2: Rule-based ECP Variant Dynamics - Periodic-fault Environment

(A/SA:I/FD:N). The two hybrid approaches differ from the purely evolutionary approach in the following way:

Rule-based creation. This approach uses the rules defined in Section 7.1 to create new agents that are added to the node selection room when an agent leaves the node as part of packet transmission. Agents created at node startup are still created randomly.

Randomized Rule-based Creation. This approach uses the rules defined in Section 7.1 to create new agents, but selects randomly from the categories near those chosen by rule to bias the created agent. For values in the middle of the range, the value chosen by rule is used 50% of the time, and the values one category above or below are used 25% of the time. For values at the end of the range (maximum or minimum categories), the value chosen by rule is used 50% of the time, the next category over is used 35% of the time, and the category two away from the maximum is used 15% of the time.

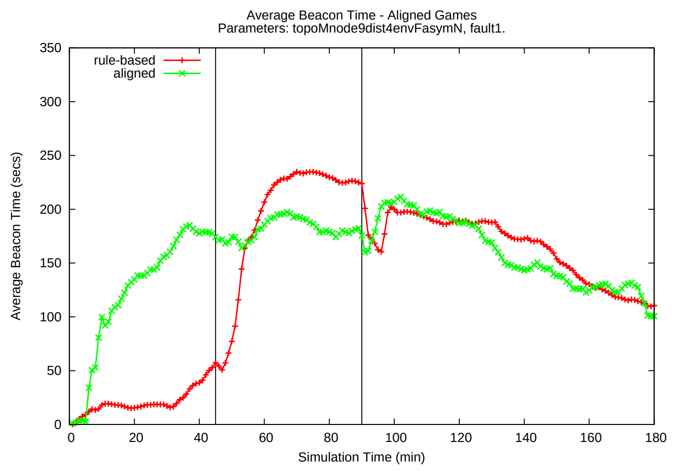


Figure 7.3: Rule-based Game Beacon Time - Simple-fault Environment

7.2.1 Results

Tables 7.5 and 7.6 show the results for hybrid games running in simple-fault and periodic-fault environments. In general, the hybrid games send fewer beacons than the rule-based ECP variant but still significantly more than the purely evolutionary version of ECP. In addition, the hybrid games have significantly more routing timeouts than either the purely rule-based or evolutionary approaches, and are outperformed in all cases by those approaches.

Game	**Beacons Sent** (Avg ± StdDev)	**Timeouts** (Avg ± StdDev)
Rule-based Creation	6983 ± 5733	588 ± 129
Randomized Rule-based Creation	4571 ± 4375	1297 ± 977
Rule-based ECP	8345 ± 10081	232 ± 37
Evolutionary ECP	785 ± 83	212 ± 25

Table 7.5: Hybrid Game Results - Simple-fault Environment

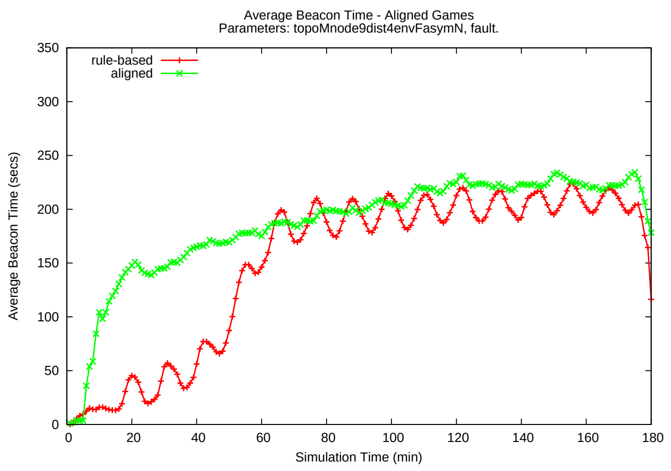


Figure 7.4: Rule-based Game Beacon Time - Periodic-fault Environment

Game	**Beacons Sent** (Avg ± StdDev)	**Timeouts** (Avg ± StdDev)
Rule-based Creation	7636 ± 4673	1076 ± 163
Randomized Rule-based Creation	4062 ± 4110	1775 ± 929
Rule-based ECP	28061 ± 46168	398 ± 73
Evolutionary ECP	700 ± 63	310 ± 42

Table 7.6: Hybrid Game Results - Periodic Fault Results

7.3 Summary

The results in this chapter demonstrate that the power consumption and node connectivity in WSNs in the presence of failures can be improved by implementing routing protocols as evolutionary games. Specifically, they show that the evolutionary approach outperforms a specific non-evolutionary approach that works well in failure-free networks because the evolutionary approach can adapt successfully to the faulty environment. While other rule-based approaches could be specially optimized to work well in the presence of failure, our results demonstrate the viability of an evolutionary approach to constructing WSN routing protocols. Finally, additional results examining the viability of hybrid evolutionary/rule-based approaches were not promising.

Chapter 8

Conclusions and Future Work

8.1 Conclusions

This dissertation presented an agent-based approach to building and optimizing WSN communication in dynamic and faulty environments. In this approach, a population of agents executes the network functions, and interacts with other agents and the environment. Repeated agent interactions produce an evolutionary game where agents having the fittest strategies survive over time to optimize network parameters. This dissertation also presented a software framework to create WSN applications using the agent-based approach and overcome important programming challenges in WSNs.

The specific thesis evaluated in this dissertation was

> Power consumption and node connectivity of WSN can be improved in the presence of failures by implementing routing protocols as evolutionary games

To evaluate this thesis, we designed and implemented an agent-based communication protocol for wireless sensor networks, ECP. ECP is a collection protocol for WSN that uses an agent-based evolutionary game approach to optimize the rate at which beacons are sent. We evaluated the efficacy of ECP in the presence of failures using a WSN simulator and measuring the number of beacon packets sent and beacon timeouts between nodes. We compared different variants of ECP with a strictly rule-based non-evolutionary approach.

Our results confirm the hypothesis we proposed. After experiments to determine the appropriate game structure, the resulting evolutionary game performs better in the presense of failures than a similar routing protocol whose behavior is controlled by predetermined rules. While other rule-based approaches could be customized to perform as well or potentially better than this evolutionary approach in this or other specific scenarios, this demonstrates the general viability of this approach.

8.2 Future Work

This dissertation describes a solution to create and optimize WSN applications that can be expanded in different directions. The design of the solution considered several perspectives, i.e., the agent-based model, the evolutionary games, and the biologically-inspired ideas, and provides multiple directions for improvement. For example, future work could consider additional integration with the operating system, improvements to the micro-component framework, definition of games with several types of agents, or creation of agents with more complex structures. This section contains discusses discussion some of these directions.

8.2.1 Operating System Integration

Some operating system modules such as the timer manager, the link estimator, and low-level drivers for sensor devices and the wireless transmitter are good candidates for optimization using an agent-based approach. For example, the wireless transmitter could directly optimize transmission power to provide additional control over communication quality, fault tolerance, and power consumption.

8.2.2 Micro-component Framework Extensions

The micro-component framework provides the tools to execute agent actions concurrently inside the nodes. Two future extensions related to this functionality, concurrent micro-components and dynamic micro-protocols, are good directions for future work.

Concurrent micro-components. The current version of the framework allows for concurrency between micro-components. Having concurrency inside some micro-components, specifically concurrency for split-phase timers, would allow for more efficient processing of timed events, and new ways to implement agent actions. For example, ECP could check for disconnected neighbors using independent timers for agents located in the routing table, instead running a duty cycle to periodically check for disconnections.

Dynamic micro-protocols. A group micro-component is composed of an array of micro-components that can be modified at run-time. The current implementation provides functions to modify the group of micro-components composing a micro-protocol, but this can only be done if there are no running instances of the micro-protocol. Enabling additions and deletions of micro-components while the group micro-component is executing would enable more dynamic micro-protocols designs and implementations.

8.2.3 Complex Evolutionary Games

This dissertation optimized the ECP routing engine, and other components such as the forwarding engine and some internal components of the protocol are directions for future work. For example, a full implementation of ECP could include multiple types of agents that optimize other portions of ECP, for example agents and evolutionary games to optimize the forwarding engine. The new games could also consider interactions between several types of agents having different sets of strategies.

8.2.4 Extensions Based on Biological Concepts

Biological and ecological systems inspired some decisions shown in this dissertation. An idea presented as future work is the creation of more complex agent structures, such as defining higher-level agents composed of two or more of the agents we presented here. This idea could enable more complex agents and agent behavior, and support for more optimizing more complex network systems.

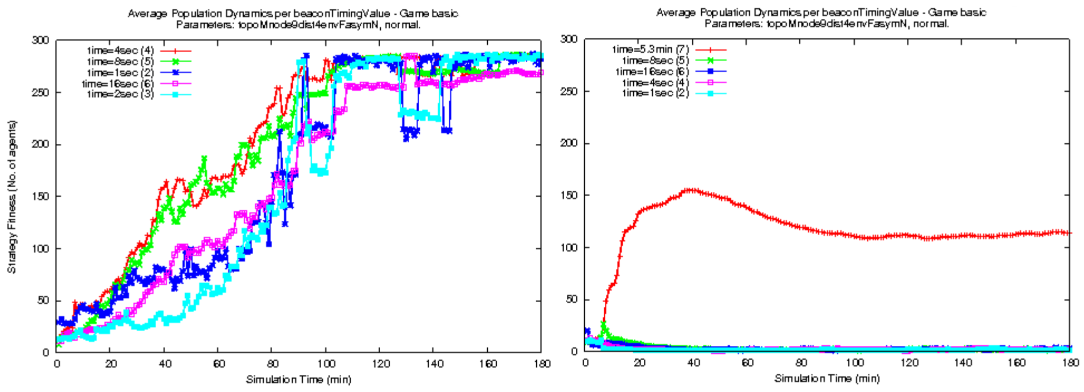
Appendix A

ECP Variant Behavior in Failure-Free Environments

In addition to an evaluation of the various evolutionary games in faulty environment, described in Chapter 6, we also evaluated their performance in a failure-free environment for completeness. This appendix presents includes the results of this evaluation for reference purposes. All tests were run using the methodology described in Chapter 6.

Game	**Beacons Sent** (Avg ± StdDev)	**Timeouts** (Avg ± StdDev)
B/SA:N/FD:E	18351 ± 21374	4379 ± 4820
B/SA:I/FD:E	488 ± 43	296 ± 40
B/SA:E/FD:E	496 ± 35	293 ± 36
B/SA:N/FD:I	17416 ± 19324	3883 ± 3224
B/SA:I/FD:I	488 ± 41	291 ± 45
B/SA:E/FD:I	483 ± 45	292 ± 33
B/SA:N/FD:N	6223 ± 1827	934 ± 583
B/SA:I/FD:N	559 ± 141	280 ± 37
B/SA:E/FD:N	588 ± 194	287 ± 53

Table A.1: Basic Game Results - Failure-free Environment



(a) B/SA:N/FD:E (b) B/SA:I/FD:E

Figure A.1: Basic Game Population Dynamics - Failure-free Environment

Game	**Beacons Sent** (Avg ± StdDev)	**Timeouts** (Avg ± StdDev)
A/SA:I/FD:E	1123 ± 65	289 ± 32
A/SA:I/FD:I	1123 ± 80	301 ± 46
A/SA:I/FD:N	902 ± 96	248 ± 37

Table A.2: Aligned Game Results - Failure-free Environment

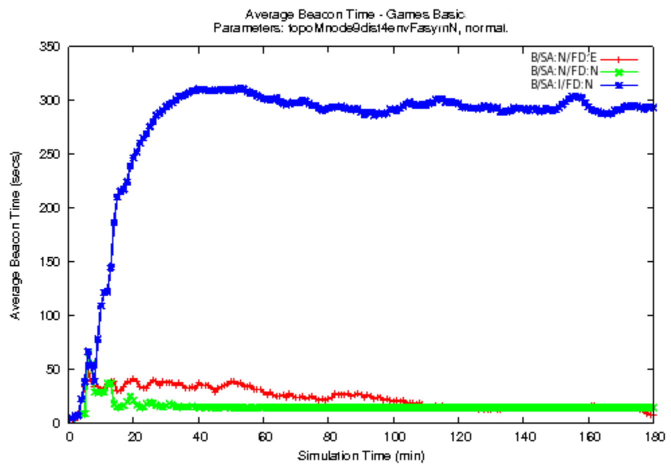


Figure A.2: Average Beacon Time for Games B/SA:N/FD:E, B/SA:N/FD:N and B/SA:I/FD:N - Failure-free Environment

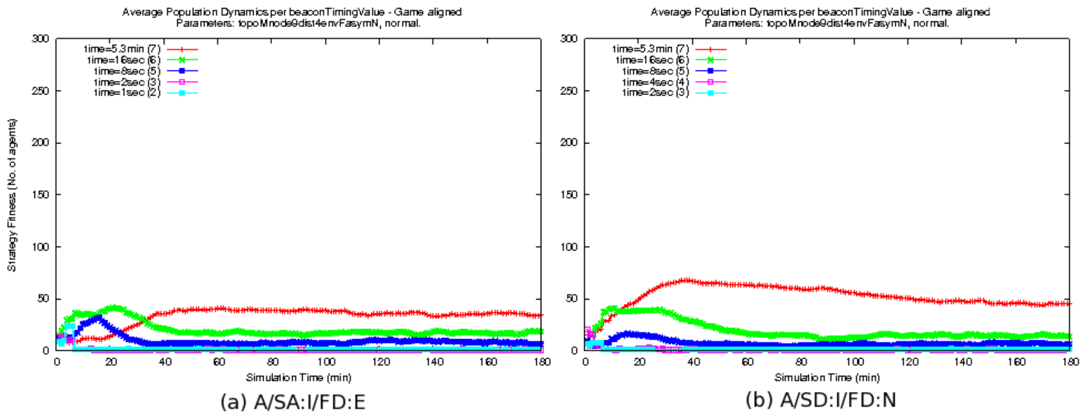


Figure A.3: Aligned Game Population Dynamics - Failure-free Environment

Game	**Beacons Sent** (Avg ± StdDev)	**Timeouts** (Avg ± StdDev)
Rule-based	2956 ± 4942	38 ± 14
Rule-based Creation	5614 ± 4418	492 ± 91
Randomized Rule-based Creation	3452 ± 3282	931 ± 802

Table A.3: Rule-based and Hybrid Game Results - Failure-free Environment

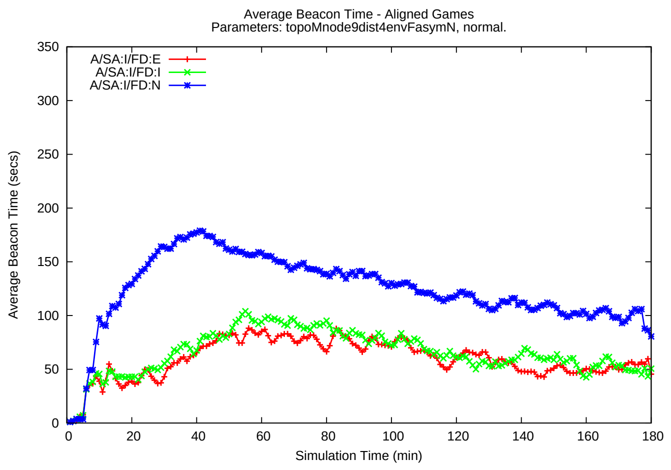
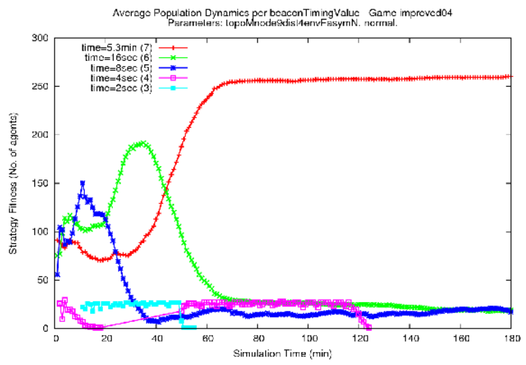
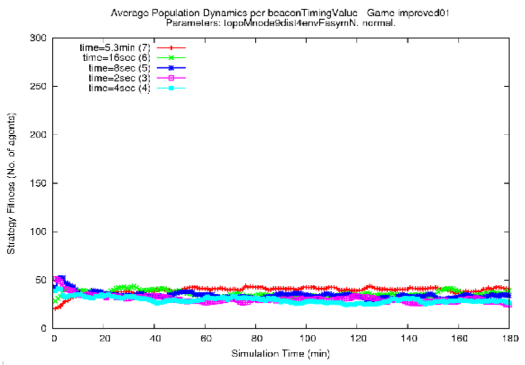


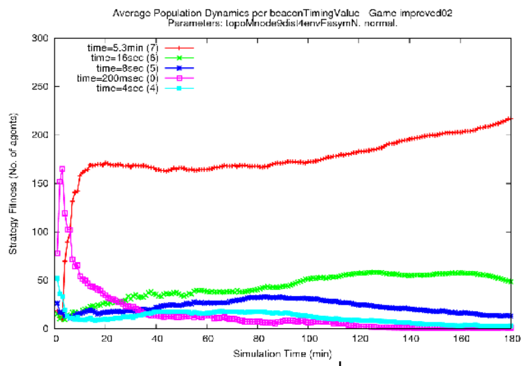
Figure A.4: Average Beacon Time for Aligned Games - Failure-free Environment



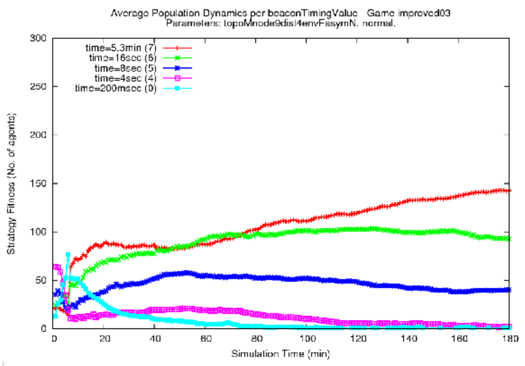
(a) rule-based



(b) hybrid random



(c) hybrid rule



(d) hybrid rule+rnd

Figure A.5: Rule-based and Hybrid Game Population Dynamics - Failure-free Environment

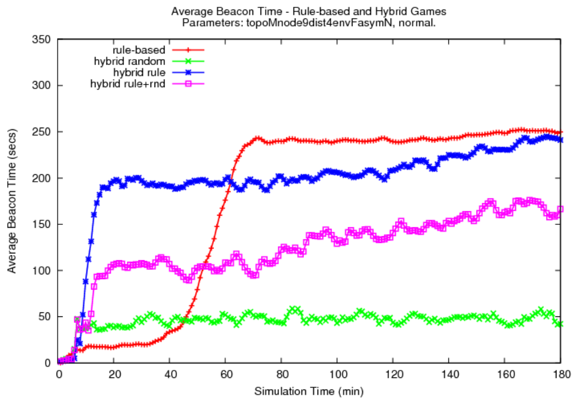


Figure A.6: Average Beacon Time for Rule-based and Hybrid Games - Failure-free Environment

References

[1] P. Agnihotri and P. Nuggehalli. Enhancing sensor network lifetime using interactive communication. *ISIT*, 2007.

[2] I. Akyildiz, Y. Sankarasubramaniam, W. Su, and E. Cayirci. A survey on sensor networks. *IEEE Communication Magazine*, 40(8):102–114, August 2002.

[3] I. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. Wireless sensor networks: A survey. *Computer Networks*, December 2001.

[4] S. Al-Omari, J. Du, and W. Shi. Score: A sensor core framework for cross-layer design. *QShine*, August 2006.

[5] E. Altman, R. ElAzouzi, Y. Hayel, and H. Tembine. An evolutionary game approach for the design of congestion control protocols in wireless networks. *Modeling and Optimization in Mobile, Ad Hoc, and Wireless Networks and Workshops*, April 2008.

[6] M. Asim, H. Mokhtar, and M. Merabti. A fault management architecture for wireless sensor networks. *Wireless Communication and Mobile Computing Conference*, August 2008.

[7] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1:11–33, January 2004.

[8] M. Bahrepour, N. Meratnia, and P. Havinga. Automatic fire detection: A survey from wireless sensor network perspective. Technical report, University of Twente, 2008.

[9] L. Bernardo, R. Oliveira, R. Tiago, and P. Pinto. A fire monitoring application for scattered wireless sensor networks. *Wireless Information Networks and Systems*, July 2007.

[10] P. Boonma, P. Champrasert, and J. Suzuki. A biologically inspired architecture for self-managing sensor networks. *Sensor and Ad Hoc Communications and Networks*, 2006.

[11] P. Boonma, P. Champrasert, and J. Suzuki. BiSNET: A biologically-inspired architecture for wireless sensor networks. *IEEE International Conference on Autonomic and Autonomous Systems*, 2006.

[12] P. Boonma and J. Suzuki. BiSNET: A biologically-inspired middleware architecture for self-managing wireless sensor networks. *Computer Networks*, 2007.

[13] P. Bourquin. Adaptive sampling for sensor networks. *Swiss Federal Institute of Technology Zurich*, 2005.

[14] D. Braginsky and D. Estrin. Rumor routing algorithm for sensor networks. *WSNA*, September 2002.

[15] P. G. Bridges, G. T. Wong, M. Hiltunen, R. D. Schlichting, and M. J. Barrick. A configurable and extensible transport protocol. *IEEE/ACM Transactions on Networking*, 15(6):1254–1265, 2007.

[16] M. Britton, V. Shum, L. Sacks, and H. Haddadi. A biologically-inspired approach to designing wireless sensor networks. *Proceedings of EWSN 2005: 2nd European Workshop on Wireless Sensor Networks*, February 2005.

[17] I. Carreras, I. Chlamtac, H. Woesner, and C. Kiraly. Bionets: Bio-inspired next generation networks. *IFIP International Federation for Information Processing*, 2005.

[18] A. Cerpa, E. Jeremy, D. Estrin, L. Girod, M. Hamilton, and J. Zhao. Habitat monitoring: Application driver for wireless communication technology. *SIGCOMM*, April 2001.

[19] C. Charalambous and S. Cui. A biologically inspired networking model for wireless sensor networks. *IEEE Network*, May/June 2010.

[20] Z. Che-Aron, W. Al-Khateeb, and F. Anwar. The enhanced fault-tolerant AODV routing protocol for wireless sensor network. *2nd International Conference on Computer Research and Development*, 2010.

[21] D. Cook and S. Das, editors. *Smart Environments: Technologies, Protocols, and Applications.* John Wiley & Sons Ltd, 2004.

[22] P. Corke, T. Wark, R. Jurdak, W. Hu, P. Valencia, and D. Moore. Environmental wireless sensor networks. *Proceedings of the IEEE*, 98(11), November 2010.

[23] G. Crosby and N. Pissinou. Evolution of cooperation in multi-class wireless sensor networks. *32nd IEEE Conference on Local Computer Networks*, 2007.

[24] C. Darwin. *On the Origin of Species.* John Murray, United Kingdom, 1859.

[25] A. Dunkels, F. Osterlind, and Z. He. An adaptive communication architecture for wireless sensor networks. *Proceedings of SenSys*, November 2007.

[26] A. E. Eiben and J. Smith. *Introduction to Evolutionary Computing.* Springer-Verlag, 2003.

[27] D. B. Fogel. *Evolutionary Computation: Toward a New Philosophy of Machine Intelligence.* John Wiley & Sons Ltd, 3rd edition, 2005.

[28] R. Fonseca, O. Gnawali, K. Jamieson, S. Kim, P. Levis, and A. Woo. The collection tree protocol (CTP), Feb 2007.

[29] C. W. Fox, D. A. Roff, and D. J. Fairbairn, editors. *Evolutionary Ecology: Concepts and Case Studies.* Oxford University Press, New York, NY, United States, 2001.

[30] L. J. Garcia, A. L. Sandoval, A. Trivino, and C. Barenco. Routing protocols in wireless sensor networks. *Sensors*, 98:8399–8421, 2009.

[31] O. Gnawali, R. Fonseca, K. Jamieson, and P. Levis. CTP: Robust and efficient collection through control and data plane integration. Technical Report SING-08-02, 2008.

[32] O. Gnawali, R. Fonseca, K. Jamieson, D. Moss, and P. Levis. Collection tree protocol. Technical Report SING-09-01, 2009.

[33] O. Gnawali, P. Levis, R. Fonseca, K. Jamieson, and D. Moss. CTP: Collection tree protocol, July 2011.

[34] L. Guo and Q. Tang. An improved routing protocol in WSN with hybrid genetic algorithm. *2nd International Conference on Network Security, Wireless Communication and Trusted Computing*, 2:289–292, April 2010.

[35] W. Heinzelman, A. Chandrakasan, and H. Balakrishnan. Energy-efficient communication protocol for wireless microsensor networks. *Proceeding of the Thirty Third Hawaii International Conference on System Sciences*, 8, January 2000.

[36] W. Heinzelman, A. Chandrakasan, and H. Balakrishnan. An application-specific protocol architecture for wireless microsensor networks. *Transactions on Wireless Communications*, 1(4), October 2002.

[37] M. Hiltunen, R. Schlichting, X. Han, M. Cardozo, and R. Das. Real-time dependable channels: Customizing QoS attributes for distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, 10(6):600–612, June 1999.

[38] J. Hofbauer and K. Sigmund. *Evolutionary Games and Population Dynamics.* Cambridge University Press, 2002.

[39] A. Husseini, A. Zeid, S. Onel, and S. Kamarthi. An agent-based modeling and control of wireless sensor networks. *Intelligent Engineering Systems through Artificial Neural Networks*, 20, 2010.

[40] N. Hutchinson and L. Peterson. The x-kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, 1991.

[41] C. Intanagonwiwat, G. Ramesh, and D. Estrin. Directed diffusion: A scalable and robust communication paradign for wireless sensor networks. *Proceedings ACM MobiCom*, August 2000.

[42] C. Intanagonwiwat, G. Ramesh, D. Estrin, J. Heidemann, and F. Silva. Directed diffusion for wireless sensor networking. *IEEE/ACM Transactions on Networking*, 11(1), February 2003.

[43] H. Karl and A. Willig. *Protocols and Architectures for Wireless Sensor Networks.* John Wiley & Sons Ltd, Hoboken, New Jersey, 2007.

[44] S. Kim, S. Pakzad, D. Culler, J. Demmel, G. Fenves, S. Glaser, and M. Turon. Health monitoring of civil infrastructures using wireless sensor networks. *IPSN*, April 2007.

[45] A. W. Krings and Z. S. Ma. Fault-models in wireless communication: Towards survivable ad hoc networks. *Idaho National Lab*, 2008.

[46] A. Lachenmann, P. J. Marron, D. Minder, and K. Rothermel. Meeting lifetime goals with energy levels. *Proceedings of SenSys*, 2007.

[47] P. Levis, T. Clausen, J. Hui, O. Gnawali, and J. Ko. The trickle algorithm. *Internet Engineering Task Force*, RFC6206, March 2011.

[48] P. Levis and D. Gay. *TinyOS Programming.* Cambridge University Press, New York, NY, United States, 2009.

[49] P. Levis and N. Lee. TOSSIM: A simulator for tinyos networks. September 2003.

[50] J. Lin, X. Xiong, A. Vasilakos, G. Chen, and W. Guo. Evolutionary game-based data aggregation model for wireless sensor networks. *IET Communication*, 5(12):1691–1697, 2011.

[51] Z. S. Ma and A. W. Krings. Bio-robustness and fault tolerance: A new perspective on reliable, survivable and evolvable network systems. *Aerospace Conference*, March 2008.

[52] Z. S. Ma and A. W. Krings. Dynamic hybrid fault models and the applications to wireless sensor networks (wsns). In *Proceedings of the 11th International Symposium on Modeling, Analysis and Simulation of Wireless and Mobile Systems*, MSWiM '08, pages 100–108, New York, NY, USA, 2008. ACM.

[53] Z. S. Ma and A. W. Krings. Dynamic hybrid fault modeling and extended evolutionary game theory for reliability, survivability and fault tolerance analyses. *IEEE Transactions on Reliability*, 60(1):180–196, 2011.

[54] A. Mainwaring, J. Polastre, R. Szewczyk, D. Culler, and J. Anderson. Wireless sensor networks for habitat monitoring. *Wireless Sensor Network Applications*, September 2002.

[55] P. J. Mayhew. *Discovering Evolutionary Ecology: Bringing Together Ecology and Evolution.* Oxford University Press, New York, NY, United States, 2009.

[56] J. Maynard Smith. *Evolution and the Theory of Games.* Cambridge University Press, Cambridge, New York, 1982.

[57] MEMSIC. Memsic web site, 2012.

[58] S. Misra et al., editors. *Guide to Wireless Sensor Networks.* Computer Communications. Springer, 2009.

[59] R. W. Morrison. *Designing Evolutionary Algorithms for Dynamic Environments.* Natural Computing Series. Springer, 2004.

[60] L. Mottola. *Programming Wireless Sensor Networks: From Physical to Logical Neighborhoods.* PhD thesis, Dipartamento di Elettronica e Informazione, Politecnico di Milano, 2008.

[61] L. Mottola and G. P. Picco. Programming wireless sensor networks: Fundamental concepts and state of the art. *ACM Computing Surveys*, 43(3), April 2011.

[62] M. Nakamura, A. Sakurai, and J. Nakamura. Distributed environment control using wireless sensor/actuator networks for lighting applications. *Sensors*, (9), 2009.

[63] J. Paek, K. Chintalapudi, G. Ramesh, J. Caffrey, and S. Masri. A wireless sensor network for structural health monitoring performance and experience. *Embeded Networked Sensors*, May 2005.

[64] L. Paradis and Q. Han. A survey of fault management in wireless sensor networks. *Journal Network System Management*, 15(2):171–190, 2007.

[65] N. S. Patil and P. Patil. Data aggregation in wireless sensor network. *Conference on Computational Intelligence and Computing Research*, 2010.

[66] M. A. Perillo and W. B. Heinzelman. Wireless sensor network protocols. Technical report, University of Rochester, Dept. of Electrical and Computer Engineering, 2006.

[67] C. Perkins, E. Belding-Royer, and S. Das. Ad-hoc on-demand distance vector routing (AODV). *RFC 3561*, July 2003.

[68] S. F. Pileggi, C. E. Palau, and M. Esteve. An adaptive and flexible fault tolerance mechanism designed on multi-behavior agents for wireless sensor/actuator network. *International Conference on Sensor Technologies and Applications*, 2007.

[69] V. Ponnusamy and A. Abdullah. Biologically-inspired (botany) mobile agent based self-healing wireless sensor network. *Sixth International Conference on Intelligent Environments*, 2010.

[70] D. Puccinelli and M. Haenggi. Wireless sensor networks: Applications and challenges of ubiquitous sensing. *Circuits and Systems*, 3rd quarter, 2005.

[71] S. Salim, M. Javed, and A. H. Akbar. A mobile agent-based architecture for fault-tolerance in wireless sensor networks. *Communication Networks and Services Research Conference*, 2010.

[72] K. Sha, J. Du, and W. Shi. WEAR: A balanced, fault-tolerant energy-aware routing protocol in wsns. *International Journal of Sensor Networks*, 1(3/4), 2006.

[73] S. K. Shing, M. Singh, and D. Singh. Routing protocols in wireless sensor networks - a survey. *International Journal of Computer Science & Engineering Survey*, 1(2), November 2010.

[74] R. Szewczyk, A. Mainwaring, J. Anderson, and D. Culler. An analysis of a large scale habitat monitoring application. *Proceedings of SenSys*, November 2004.

[75] J. K. Taneja. Design, deployment, and analysis of sustainable sensor networks for environmental monitoring. Master's thesis, University of California, Berkeley, December 2007.

[76] T. L. Vincent and J. S. Brown. Stability in an evolutionary game. *Theoretical Population Biology*, (26):408–427, 1984.

[77] T. L. Vincent and J. S. Brown. *Evolutionary Game Theory, Natural Selection and Darwinian Dynamics*. Cambridge University Press, New York, NY, United States, 2005.

[78] G. Werner-Allen, K. Lorincz, J. Johnson, J. Lees, and M. Welsh. Fidelity and yield in a volcano monitoring sensor network. *Operating Systems Design and Implementation*, 2006.

[79] D. Westneat and C. W. Fox, editors. *Evolutionary Behavioral Ecology*. Oxford University Press, 2010.

[80] Wikipedia. Evolution - http://en.wikipedia.org/wiki/evolution, April 2012.

[81] A. Woo, T. Tong, and D. Culler. Taming the underlying challenges of reliable multihop routing in sensor networks. In *Proceedings of SenSys*, pages 14–27. ACM Press, 2003.

[82] Y. Xu, J. Heidemann, and D. Estrin. Geography-informed energy conservation for ad hoc routing. *In Proceedings of the ACM/IEEE International Conference on Mobile Computing and Networking*, pages 70–84, July 2001.

[83] Y. Yu, G. Ramesh, and D. Estrin. Geographical and energy-aware routing: A recursive data dissemination protocol for wireless sensor networks. Technical Report UCLA/CSD-TR-01-0023, UCLA Computer Science Department, May 2001.

[84] J. Zhang, W. Li, Z. Yin, S. Liu, and X. Guo. Forest fire detection system based on wireless sensor network. *Industrial Electronics and Applications*, 2009.