This is to certify that the
dissertation entitled

RAPID PROTOTYPING AND QUICK DEPLOYMENT OF SENSOR NETWORKS

presented by

UMAMAHESWARAN ARUMUGAM

has been accepted towards fulfillment
of the requirements for the

Ph.D. degree in Computer Science and Engineering

Major Professor's Signature

Sep 22, 2006

Date

MSU is an Affirmative Action/Equal Opportunity Institution

DATE DUE	DATE DUE	DATE DUE

Rapid Prototyping and Quick Deployment of Sensor Networks

By

Umamaheswaran Arumugam

A Dissertation

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

Doctor of Philosophy

Department of Computer Science and Engineering

2006

Abstract

Rapid Prototyping and Quick Deployment of Sensor Networks

By

Umamaheswaran Arumugam

Event-driven programming platforms for sensor networks require the programmers to deal with several challenges including buffer management, stack management, and flow control. To simplify the design of sensor network protocols, several high-level primitives are proposed. However, these primitives have to be implemented in an existing event-driven programming platform and most of them still require the programmers to use the same platform (though some intricate details of the platform are hidden).

In this dissertation, we develop tools and protocols that enable the programmers to rapidly prototype and quickly deploy sensor network protocols. We propose to reuse existing abstract models from distributed computing literature (e.g., read/write model, shared-memory model). Since these models hide several low-level challenges of the target system, programs written in these models are simple, easy to understand, and concise. These abstract programs must then be transformed into a model consistent with sensor networks. The main contributions of this dissertation are as follows.

- We consider a new computational model for sensor networks, namely, *write all with collision* (WAC) model and develop algorithms that transform programs written in traditional abstract models into sensor networks. We show that the transformation algorithms preserve the self-stabilization property of the original programs.

- Based on this theoretical foundation, we propose *ProSe*, a programming tool for sensor networks. ProSe enables the programmers to (1) specify protocols in simple abstract models, (2) reuse existing fault-tolerant/self-stabilizing protocols from the literature in the context of sensor networks, and (3) automatically generate and deploy code.

- To quickly deploy the generated programs, we propose *Infuse*, a bulk data dissemination protocol for reprogramming the sensors in-place. To deal with arbitrary channel errors, we extend the sliding window mechanisms of TCP/IP. We show that Infuse provides a reliable and energy-efficient data dissemination service through simulations and real-world experiments.

Based on our experience in developing sensor network applications, we expect that these results would simplify the protocol design, implementation, evaluation, and deployment. As a result, it enables the transition where protocols are designed by *domain experts* rather than *experts in sensor networks.*

To my mom A. Nalini and my dad Dr. P. S. Arumugam for their love and affection

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my advisor, Dr. Sandeep Kulkarni, for all his support and guidance during my graduate studies. He has been a wonderful guide and I appreciate him for the time and effort he spent in discussions that were extremely helpful in my research, writings, and presentations. Also, I would like to thank the members of my Ph.D. committee, Dr. Philip K. McKinley, Dr. Abdol-Hossein Esfahanian, and Dr. Rajesh Kulkarni, for their valuable comments and suggestions regarding my dissertation. In particular, I sincerely appreciate Dr. McKinley's efforts in taking a special interest in my research work and my career.

I would also like to thank the faculty of the Department of Computer Science and Engineering at the Michigan State University. I would like to express my thanks to Dr. Anil K. Jain, Dr. George C. Stockman, Dr. Betty H.C. Cheng, Dr. Laura Dillon, Dr. Eric Torng, and Dr. Jon Sticklen for providing an excellent learning, research, and teaching experience. In particular, I would like to thank Dr. Jain for his interest in knowing what I am doing and Dr. Stockman for his help in TA assignments and dissertation completion fellowship. I want to thank Ms. Linda Moore, Ms. Debbie Kruch, Ms. Cathy Davison, Ms. Starr Portice, and Ms. Norma Teague for their assistance in administrative tasks. Especially, I want to express my gratitude to Ms. Moore for her patience, care, and cheerfulness.

I want to express my sincere thanks to Dr. Anish Arora (at the Ohio State University), Dr. Rajiv Ramnath (at the Ohio State University), Dr. Mohamed Gouda (at the University of Texas at Austin), Dr. Ted Herman (at the University of Iowa), and Dr. Mikhail Nesterenko (at the Kent State University) for their valuable comments regarding my work. I also thank Vinod Kulathumani, Sandip Bapat, Dr. Vinayak Naik, Prabal Datta, Dr. Hongwei Zhang, Young-ri Choi, and Dr. Santosh Kumar for sharing the sensor devices for my experiments, helping me schedule experiments in Kansei testbed, and sharing some valuable time with me. We had a wonderful time during the intense two-week testing, debugging, and deployment for the DARPA demonstration at Avon Park, FL in December 2004.

I want to thank all my colleagues at the Software Engineering and Network Systems (SENS) Lab. Specifically, I would like to thank Dr. Ali Ebnenasir, Dr. Bru Bezawada, Karun Biyani, Limin Wang, Borzoo Bonakdarpour, Dr. Sascha Konrad, Farshad Samimi, Min Deng, Ji Zhang, Karli Lopez, and Fuad Suleiman Abujarad. I appreciate the efforts of Ali, Bru, Karun, Limin, and Borzoo for their help in proof reading my papers and providing constructive comments about the presentations. I also express my gratitude to all the SENS faculty and students for providing an excellent atmosphere to work, organizing SENS meetings every week, and planning SENS parties every semester.

I also would like to thank my friends at the Michigan State University, Karthik Nandakumar, Arun Prabakaran, Sunil Unikkat, Shankarshna Madhavan, Narasimhan

Swaminathan, Aravindhan Ravisekar, Prasanna Balasundaram, Loga Anjaneyulu, Raja Ganjikunta, and Srikanth Sridhar. I also thank my friends, Thangavelu Arumugam, Sridevi Srinivasan, and Janny Henry Rodriguez, for all the fun-time we had together during my visits to San Jose, CA. I appreciate S. Suresh Babu, G.R. Arun, and R. Naveen Kumar for always believing in me and being there for me. Thanks also goes to all my Anna University friends.

Last but not the least, I would like to thank my parents for their love and affection. I thank them for their support in whatever efforts I take. I am always grateful to them for allowing me to switch from Medicine to Computer Science. I am always indebted to my mom for her strong belief in me. She is the symbol of courage and caring. I was highly delighted to know that my dad is really proud of my doctorate degree and seeing me become a "Dr" eventually. I owe all my achievements to him. I thank my parents for their constant words of encouragement and for their prayers.

Thank you.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

In the recent years, sensor networks have become popular due to their wide variety of applications, including, unattended monitoring of undesirable events (e.g., border patrolling, critical infrastructure protection), hazard detection (e.g., landslide detection), habitat monitoring (e.g., studying the micro-climates of storm petrels), and structural monitoring (e.g., monitoring the structural integrity of bridges). In these applications, small low-power embedded devices (called sensors) are used to: (1) *sample* the physical phenomenon, (2) *process* the sampled values, (3) *communicate* the sampled values or their semantic interpretations to other sensors over the radio, and (4) in certain applications *respond* to the physical phenomenon (using actuators). Since the recent development in micro-electrical-mechanical systems (MEMS) technology enables the production of these sensors and actuators in large numbers and at low cost, large-scale deployment of these devices are now possible. For example, in [9, 10], sensor networks demonstrated the potential for monitoring a large field for detection and tracking of undesirable objects.

Table 1.1: Resource limitations of sensor network platforms

Platform	Processor	RAM	Communication bandwidth	Estimated lifetime (< 1% duty cycle) [112]
Mica 2 [31] and XSM [41]	8-bit, 7.37 MHz ATmega 128L	4 KB	38.4 Kbps	453 days (Mica 2)
Mica Z [32]	8-bit, 7.37 MHz Atmega 128L	4 KB	250 Kbps	328 days
Telos [113]	16-bit, 8 MHz TI MSP430	10 KB	250 Kbps	945 days

1.1 Challenges in Sensor Networks

During the design, implementation and deployment of sensor network applications, most existing sensor network platforms (e.g., Mica [61], XSM [41]) pose several low-level challenges to the designers. First, due to small form-factor, the sensors are limited by the amount of resources they have. Second, since the sensors communicate over a broadcast medium (e.g., radio), message collision (e.g., due to hidden terminal effect) affect the communication among sensors. And, third, faults such as message corruption, sensor failures and malicious sensors affect the state of a sensor network protocol. We discuss these challenges in more detail, next.

Resource limitations. The sensors are often constrained by limited computation cycles, limited communication bandwidth, limited memory, and limited power. Hence, they need to collaborate with each other in order to perform a certain task. Additionally, due to limited power, to sustain the network for longer duration, methods for power management need to be used so that a sensor is put in sleeping mode while its services are not necessary.

Table 1.1 highlights the constraints imposed by some of the existing sensor network platforms. Specifically, the sensors have memory of 4-10 KB and the bandwidth is at most 250 Kbps. Furthermore, for 1% duty cycle (i.e., sensors report data once every 3 minutes), the estimated lifetime is between 1-3 years. However, the duty cycle in many applications is usually higher as the sensors are required to sample the physical

phenomenon at higher rates. Therefore, the designer of a sensor network protocol needs to address low-level concerns such as when to forward the data to other sensors and when to put a sensor to sleep state in order to conserve energy.

Nature of communication. Since the sensors communicate using a shared wireless medium (e.g., radio), the basic mode of communication is *local broadcast with collision.* Specifically, whenever a sensor communicates all its neighbors receive the message. However, if two or more messages are sent to a sensor simultaneously then due to collision it receives none. Such message collision is undesirable in sensor networks as it results in wastage of power to transmit the messages resulted in the collision. Moreover, due to hidden terminal effect, a given message may collide at one sensor and be correctly received at another sensor. In addition, the communication range of a sensor is limited and, hence, the network is multi-hop in nature. Therefore, the sensors need to collaborate with each other in order to forward (e.g., route) data to its destination.

To deal with these problems, the designer needs to ensure reliable message communication among sensors. Towards this end, the designer has to address the issue of how to forward messages (e.g., using a proactive/reactive acknowledgment based communication or scheduled communication). Also, the designer needs to identify the tradeoff between energy and latency during multi-hop communication in order to choose the appropriate operating point for an application.

Faults. The execution of a sensor network protocol is often hindered by (1) simple *crash faults* such as failure of sensors and (2) *transient faults* such as message corruption (due to varying link properties, e.g., signal strength), message collision, and arbitrary state corruption. Since the sensors monitor and report real-time events, it is important that these faults do not restrict the functionality of a sensor network protocol. Therefore, the designer has to ensure that in the presence of faults, the protocol provides the specified functionality. In other words, the designer has to

guarantee that the protocol is fault-tolerant.

1.2 Challenges in Existing Programming Platforms for Sensor Networks

One of the important challenges in deploying sensor network applications is programming. Most of the existing platforms (e.g., nesC/TinyOS [47]) for developing sensor network programs use *event-driven programming model* [4]. As identified in [4, 70], while an event-driven programming platform has the potential to simplify concurrency by reducing race conditions and deadlocks, the programmer is responsible for stack management and flow control. For example, in nesC/TinyOS platform, the state of an operation does not persist over the duration of entire operation. As a result, programmers need to manually maintain the stack for the operation (through the use of global variables). This also suggests that the state of the operation is shared across several functions. Hence, the designer has to manually control the flow through the use of global state variables. As the program size grows, such manual management becomes complex and is often the source of programming errors.

In addition, typical sensor network platforms require the programmers to manage buffers, contend for access to radio channel, and deal with faults. Hence, as mentioned in [88], programming in nesC/TinyOS platform is "somewhat tricky" for domain experts (e.g., civil engineers in structural health monitoring applications, biologists in habitat monitoring applications, geologists in volcanic eruption monitoring applications). Moreover, in [88], authors motivate the need for a simpler programming model that would allow domain experts to specify their applications in terms of event-driven programming where they do not need to worry about several programming level issues.

Limitations of existing primitives for programming sensor networks. To

simplify programming sensor networks, several *macroprogramming* primitives are proposed (e.g., [55, 88, 96, 104, 105, 131–133]). These primitives allow the programmers to specify the functional aspects of the protocol while hiding most of the low-level details of the network. In other words, these primitives enable the programmers to specify abstract programs.

While macroprogramming primitives simplify the specification and implementation of sensor network protocols, they have the following important drawbacks: (1) the designer of a macroprogramming primitive has to still implement the primitive in a typical sensor network platform (e.g., nesC/TinyOS), (2) if existing primitives are insufficient or need to be extended to deal with hardware/software developments then domain experts have to rely on experts in sensor network platforms, and (3) most of these primitives still require the programmer to specify protocols in nesC/TinyOS platform (though some intricate details of the platform are hidden).

1.3 Thesis

Since sensor networks are often deployed in large numbers and in hostile/inaccessible fields, it is necessary to provide a reliable, efficient, and robust networking of sensors. To accomplish this, the designer of a sensor network protocol needs to deal with the challenges identified in Section 1.1 in addition to the functionality of the protocol. The designer also has to deal with programming level challenges (e.g., manual stack management, manual buffer management, flow control) as identified in Section 1.2. Moreover, in order to ensure reliable, efficient, and robust networking of sensors, the designer needs to solve several problems that are already considered in distributed systems and traditional networking. These include (variations of) consensus, agreement in presence of faulty/malicious sensors, leader election, reliable broadcast, routing, synchronization, and tracking. Furthermore, many *self-stabilizing* [38, 40] algorithms

are developed for these problems in the literature. A system is self-stabilizing, if starting from arbitrary initial states the system recovers (in finite time) to states from where the computation proceeds in accordance with its specification. Thus, a self-stabilizing system ensures that after the occurrence of faults, the system recovers to states from where it satisfies its specification. Therefore, it is advantageous to enable the designer to reuse existing self-stabilizing algorithms to the extent possible.

The abstract models considered in distributed systems literature to develop solutions to these problems hide low-level details such as message collision, race conditions, and synchronization. Also, the abstract models facilitate the designer to verify the correctness of the deployed programs and allow manipulation of existing programs to meet new properties. Therefore, reusing abstract models and the distributed programs developed using them will enable the designer to rapidly prototype protocols and quickly evaluate their performance.

Based on this discussion, in this dissertation, we defend the following thesis:

> **Rapid prototyping and quick deployment of fault-tolerant sensor networks can be achieved by hiding low-level details from the designer of a sensor network protocol.**

To enable this, we make the following contributions: (1) *foundational contributions* that identify models and provide algorithms for enabling rapid prototyping and (2) *experimental contributions* that enable the designer to simplify the construction and deployment of sensor network protocols.

1.3.1 Foundational Contributions

In this work, we develop the theoretical foundation that enables the designer to reuse existing algorithms and abstract models considered in distributed systems literature.

1. **Computational model and transformations for sensor networks.** One challenge in reusing existing algorithms is that the abstract models considered in these algorithms does not account for the difficulties and opportunities provided by sensor networks. Since the basic mode of communication in sensor networks is local broadcast with collision in nature, the computations in sensor networks can be thought of as a *write all with collision* (WAC) model. Intuitively, in this model, whenever a sensor executes, it can update the state of all its neighbors. However, when two sensors try to update the state of a common neighbor (say, k), due to collision, state of k remains unchanged. On the contrary, the abstract models do not have the notion of message collision and existing programs assume a point-to-point communication. In order to facilitate reuse of abstract models and existing programs, we present transformation algorithms that allow one to transform programs written in abstract models into programs in WAC model. Thus, the transformations enable the designer to quickly evaluate existing algorithms in the context of sensor networks. Also, the transformations preserve the self-stabilization property of the original program. In other words, if the original program is self-stabilizing then the transformed program preserves this property.

2. **Enabling stabilization-preserving deterministic transformations.** While designing transformation algorithms for WAC model, we show that a time division multiple access (TDMA) algorithm can be effectively used to transform programs written in abstract models into programs in WAC model. If a deterministic and self-stabilizing TDMA algorithm in WAC model is used with the transformation then it is possible to provide deterministic guarantees about the transformed program in WAC model. To facilitate this, we present a self-stabilizing deterministic TDMA algorithm for sensor networks. To the best of our knowledge, this is the first such algorithm that achieves these prop-

erties. Also, this is the first such algorithm that demonstrates the feasibility of stabilization-preserving deterministic transformation of programs written in abstract models into programs in WAC model.

1.3.2 Experimental Contributions

In this work, we design and implement protocols and tools for the designer to rapidly prototype and quickly deploy sensor network protocols.

1. **Programming tool for sensor networks.** Based on the theoretical foundation for reusing abstract models and distributed programs, we develop *ProSe*, a tool for programming sensor networks. Towards simplifying the construction and deployment of sensor network protocols and applications, ProSe enables the following: (i) specify protocols in simple abstract models while hiding low-level concerns from the designer, (ii) reuse existing algorithms, (iii) automatically transform the specified programs into WAC model, and (iv) automate code generation and deployment. Furthermore, we develop abstractions to deal with failed sensors, arbitrary message loss (due to random channel errors), and arbitrary state corruption. Also, we show how we used ProSe to generate programs for routing tree maintenance, distributed tracking service, and power management. We expect that the tool enables the transition where protocols are designed by *domain experts* rather than *experts in sensor networks.*

2. **Data dissemination protocol for sensor networks.** In order to quickly deploy sensor network protocols, we develop *Infuse*, a TDMA based reliable data dissemination protocol for sensor networks. Reliable data dissemination service is essential, especially, in network reprogramming, where the sensors are reprogrammed with a new program image *in place.* One of the important requirements in data dissemination is 100% reliability, both in terms of the

number of sensors receiving the data and the data being delivered. Towards providing a reliable service, Infuse uses sliding window based flow control mechanisms and implicit acknowledgments to deal with the problem of arbitrary message loss (due to channel errors). Furthermore, we propose optimizations to reduce energy usage during the dissemination process. Also, we show that Infuse tolerates failure of sensors and ensures that the data is delivered to all the active sensors.

3. **Tradeoffs in sensor communication.** In order to assist the designer in fine-tuning the parameters of the protocols, we identify the tradeoffs in sensor communication. First, we identify the tradeoff between energy and latency. We show that a simple TDMA service can be effectively used to conserve energy when the network is idle. Also, we identify the improvement in the lifetime of the network with respect to the probability of occurrence of an event. Second, we identify the tradeoff between causality and timely delivery. We show that a simple logical timestamp algorithm can be used to achieve causal delivery of messages at the base station in order to observe the computations in sensor networks. Also, we provide guidelines for the designer to determine the time a sensor has to buffer messages depending on the level of causality violations the application can tolerate.

1.4 Organization of the Dissertation

The rest of the dissertation is organized into 2 parts. In the first part, we present the foundational aspects of the dissertation work. In Chapter 2, we formally introduce the models of computation considered in distributed systems literature. We identify the assumptions made in this work and define terminologies and notations. In Chapter 3, we present a transformation algorithm that allow the designer to transform existing

programs in distributed systems literature into a model consistent with sensor networks. We show how a TDMA algorithm is useful in obtaining such transformation. In Chapter 4, we present a self-stabilizing deterministic TDMA algorithm for sensor networks. This algorithm demonstrates the feasibility of stabilization-preserving deterministic transformation.

In the second part, we present the experimental aspects of the dissertation work. In Chapter 5, we present a programming tool for sensor networks that allows the designer to specify programs in simple abstract models considered in distributed systems literature. This tool is based on the theoretical foundations established in Chapter 3. Then, in Chapter 6, we present case studies on rapid prototyping of sensor network protocols with this tool. Specifically, we consider a network routing tree maintenance program, a distributed target tracking program, and a power management program. Subsequently, in Chapter 7, we present a bulk data dissemination protocol for sensor networks that allows one to reprogram the sensors in-place in a reliable and energy-efficient manner. Thus, using the tool developed in Chapter 5 and the protocol developed in Chapter 7, the designer can rapidly prototype sensor network protocols, evaluate existing algorithms in the context of sensor networks, and quickly deploy them in a sensor network. Additionally, in Chapter 8, we identify the tradeoffs in sensor communication. This helps the designer to tune the protocols to operate at the desired level while deploying them. Finally, in Chapter 9, we make the concluding remarks and identify the scope for future research.

Part I

Foundational Aspects

Chapter 2

Preliminaries

In this chapter, we precisely define what a program is, specify the structure of the programs written in abstract models considered in distributed systems literature and in write all with collision (WAC) model, and discuss the different semantics of distributed programs. We also briefly discuss existing transformation algorithms that allow one to transform programs written in one model into another model.

2.1 Program

We define a program based on the work of Chandy and Misra [23].

Definition. A program is a set of variables and a finite set of actions. Each variable has a predefined nonempty domain. In this dissertation, the programs are specified in terms of guarded commands [39]; each guarded command is of the form:

$$\langle guard \rangle \quad \longrightarrow \quad \langle statement \rangle$$

The guard of each action is a predicate over the program variables. The statement of each action atomically updates zero or more program variables. □

For a program p, we define the following.

Definition (State). A state of p is defined by a value of each variable of p, chosen from predefined domain of the variable. □

Definition (State Predicate). A state predicate is a boolean expression over the variables of p. □

Definition (Enabled). An action of p is enabled in a state iff its guard is true in that state. □

Definition (Computation). A computation of p is fair, maximal sequence of states $s_0, s_1, \ldots$, such that for each j, $j > 0$, s_j is obtained from s_{j-1} by executing (one or more) actions of p. □

2.2 Computational Models in Distributed Systems

A computational model limits the variables that an action can read and write. Towards this end, we split the program actions into a set of processes. Each action is associated with one of the processes in the program. Some of the commonly encountered models include *message passing model*, *read/write model* and *shared-memory model*. In message passing model, processes share no memory and they communicate by sending and receiving messages. Thus, in each action, a process can perform one of the following tasks: send a message, receive a message, or perform some internal computation. A read/write model reduces the complexity in modeling message passing programs. In read/write model, the variables of a process are classified as *public* variables and *private* variables. In each action, the process can either (1) read the state of one of its neighbors (and update its private variables), or (2) write its own variables (public and private). Thus, this model hides the complexities of message queues, message delays, etc. The shared-memory model simplifies the read/write model further in that in one action it allows a process to atomically read its state as well as the state of its neighbors and write its own state. Thus, this model hides the

intermediate states, where a process has read the state of a subset of its neighbors, that occur in read/write model. We now formally describe the restrictions imposed by read/write model and shared-memory model.

2.2.1 Read/Write Model

In read/write model, a process consists of a set of public variables and a set of private variables. In the read action, a process reads (one or more) public variables of one of its neighbors. For simplicity of presentation, we assume that each process j has only one public variable $v.j$ that captures the values of all variables that any neighbor of j can read.

Furthermore, in a read action, a process could read the public variables of its neighbor and write a different value in its private variable. For example, consider a case where each process has a variable x and j wants to compute the sum of the x values of its neighbors. In this case, j could read the x values of its neighbors in sequence. Whenever j reads $x.k$, it can update a private variable $sum.j$ to be $sum.j + x.k$. Once again, for simplicity, we assume that in the read action where process j reads the state of k, j simply copies the public variables of k. In other words, in the above case, we require j to copy the x values of all its neighbors and then use them to compute the sum.

Based on the above discussion, we assume that each process j has one public variable, $v.j$. It also maintains $copy.j.k$ for each neighbor k of j; $copy.j.k$ captures the value of $v.k$ when j read it last. Now, a read action by which process j reads the state of k is represented as follows:

$$\boxed{true \quad \longrightarrow \quad copy.j.k = v.k;}$$

And, the write action at j uses $v.j$ and $copy.j$ (i.e., copy variables for each neighbor) and any other private variables that j maintains to update $v.j$. Thus, the write

action at j is as follows:

$$\boxed{\begin{array}{l} predicate(v.j, copy.j, other\_private\_variables.j) \\ \qquad\qquad\qquad\qquad \longrightarrow \quad \text{update } v.j,\ other\_private\_variables.j; \end{array}}$$

2.2.2 Shared-Memory Model

In shared-memory model, similar to read/write model, for simplicity of presentation, we assume that each sensor (say, j) maintains one public variable, $v.j$. It also maintains $copy.j.k$ for each neighbor k of j that captures the value of $v.k$ when j read it last.

Since, in each action, a process can read the state of all its neighbors and write its own state, we model the restrictions imposed by shared-memory model as follows:

$$\boxed{\begin{array}{lll} true & \longrightarrow & \forall k : k \text{ is a neighbor of } j : copy.j.k = v.k; \\ & & \text{if } (predicate(\ v.j, copy.j, other\_private\_variables.j)\ ) \\ & & \qquad \text{update } v.j,\ other\_private\_variables.j; \end{array}}$$

Remark. We use the term abstract models to denote the computational models such as read/write model and shared-memory model developed for specifying distributed programs.

2.3 Computational Model of Sensor Networks

As mentioned in Introduction, in most existing sensor platforms, the basic mode of communication is local broadcast with collision. More specifically, due to the shared wireless medium, if a sensor simultaneously receives two or more messages then they collide and, hence, the messages become incomprehensible. If a message communication does not suffer from a collision then the message is written in the memory of all neighbors of the sender. Based on this description, we can view the

model of computation in sensor networks as *write all with collision* (WAC) model. Intuitively, in this model, in one atomic action, a sensor (process) can update its own state and the state of all its neighbors. However, if two sensors (processes) simultaneously try to update the state of a sensor (say, k) then due to collision, the state of k is unchanged. We now formally describe the restrictions imposed by this model.

2.3.1 Write All with Collision (WAC) Model

In the WAC model, each process consists of write actions (to be precise, write-all actions). Each write action at j writes the state of j and the state of its neighbors. Similar to the case in read/write model, we assume that each process j has a variable $v.j$ that captures all the variables that j can potentially write to any of its neighbors. Likewise, process j maintains $l.j.k$ for each neighbor k; $l.j.k$ denotes the value of $v.k$ when k wrote it last. Thus, an action in WAC model is as follows:

$$
\begin{array}{|l|}
\hline
predicate(v.j, l.j, other\_private\_variables.j) \\
\qquad\qquad\qquad\qquad \longrightarrow \quad \text{update } v.j,\ other\_private\_variables.j; \\
\qquad\qquad\qquad\qquad\quad\ \ \forall k : k \text{ is a neighbor of } j : l.k.j = v.j; \\
\hline
\end{array}
$$

We model the collision as follows. If two neighbors, say j and k, of process q were to execute simultaneously then their write actions collide at q. Hence, neither $l.q.j$ nor $l.q.k$ is updated. Thus, if several processes execute simultaneously then only a subset of their neighbors may be updated.

2.4 Semantics of Distributed Programs

Semantics of a distributed program specify how the enabled actions of the processes are executed. For shared-memory model (and also for the WAC model), there are different types of semantics that are often used. These include *interleaving semantics*

(also known as *central daemon*), *maximum parallelism*, and *powerset semantics* (also known as *distributed daemon*).

Interleaving semantics. In interleaving semantics, a central scheduler (called central daemon) non-deterministically selects a process, say j, among all the processes that have one or more enabled actions. Process j then non-deterministically chooses one action out of all the enabled actions and executes it atomically.

Maximum parallelism. In maximum parallelism, all the processes (with one or more enabled actions) non-deterministically choose an enabled action and execute concurrently with other processes.

Powerset semantics. In powerset semantics, a distributed scheduler (called distributed daemon) selects any non-empty subset of processes (with one or more enabled actions). Each selected process then non-deterministically chooses an enabled action and executes it concurrently with other selected processes.

2.5 Transformation Algorithms

The ability to model programs in simple, abstract models (e.g., shared-memory model under interleaving semantics) and later transform them into concrete models (e.g., message passing model, WAC model under powerset semantics) is one of the important problems in distributed systems. Several algorithms (e.g., [6, 48, 49, 65, 67, 101, 103] have been proposed for enabling such transformations.

Transformations for read/write model and message passing model. In [103], a stabilization preserving transformation is proposed to transform programs written in shared-memory model into programs in read/write model. The main idea in this paper is to split the action by which a process reads the state of all its neighbors in shared-memory model into a sequence of individual read actions in read/write model.

To achieve this transformation, *local mutual exclusion* is necessary to ensure that neighboring processes are not reading the state of one of its neighbors simultaneously. In [103], the authors extend this solution to transform programs written in read/write model into message passing model. Also, in [40], algorithm for transforming programs in shared-memory model into programs in message passing model is proposed.

Semantics transformation. In [6, 48, 49, 67, 101], programs written under the assumption of interleaving semantics are transformed to execute under the assumption of powerset semantics. Also, these transformations preserve the stabilization property of the original program. In other words, if the original program is self-stabilizing then the transformed program under powerset semantics is also self-stabilizing. The solutions developed in [6, 48, 49, 67, 101] ensure local mutual exclusion. In other words, if a process has an enabled action then they ensure that no neighbor of that process has an enabled action. In [48, 49, 101], the programs are transformed into shared-memory model under a distributed daemon. In [6], the programs are transformed into message passing model under a distributed daemon.

Chapter 3

Transformations for Write All With Collision Model

The abstract models introduced in Chapter 2 hide low-level details such as message collision, race conditions, and synchronization from the designer of a sensor network protocol. Hence, it is desirable to reuse these models and the distributed programs developed using them in the context of sensor networks. Also, it enables the designer to verify the correctness of the protocols (e.g., using model checkers) as well as to manipulate existing programs in order to meet new properties (e.g., fault-tolerance, self-stabilization, etc). In short, the reuse of abstract models will aid the designer to (1) rapidly prototype sensor network protocols, (2) quickly evaluate their performance, and (3) gain assurance about the deployed protocols.

One challenge in reusing abstract models is that it does not account for the difficulties and opportunities provided by sensor networks. More specifically, as mentioned in Chapter 2, the computational model in sensor networks is *write all with collision* (WAC) model. Hence, to enable reuse of abstract models, it is necessary to transform the programs written in abstract models into programs in WAC model.

While previous literature has focused on transformations among other models of

computation (e.g., [6, 48, 49, 65, 67, 101, 103]), the issue of transformation to WAC model from other models has not been addressed. To redress this deficiency, in this chapter, we present a transformation algorithm that allow us to correctly execute programs written in read/write model and shared-memory model in the context of sensor networks. Also, we show that the transformation preserves the self-stabilization property of the original program. In other words, if the original program is self-stabilizing then the transformed program in WAC model is also self-stabilizing.

The rest of the chapter is organized as follows. First, in Section 3.1, we introduce the system assumptions. Then, in Section 3.2, we present the transformation algorithm. Subsequently, in Section 3.3, we illustrate our transformation algorithms using the grid routing protocol designed in [25]. In Section 3.4, we discuss the efficiency of our transformation algorithms.

3.1 System Model and Proving Correctness of Transformations

We assume that the set of processes in the system are connected. Furthermore, we assume that in the given program in read/write model, for any pair of neighbors j and k, j can never conclude that k does not need to read the state of k. In other words, we require that the transformation should be correct even if each process executes infinitely often. We also assume that the transformation should work correctly even if the communication among neighbors is bidirectional, i.e., the transformation algorithm should not assume that communication between some neighbors be only unidirectional. Additionally, we assume that the topology remains fixed during the program execution, i.e., failure or repair of processes does not occur. Thus, while proving stabilization, we disallow corruption of topology related information. This assumption is similar to assumptions in previous stabilizing algorithms where the

process IDs are considered to be incorruptible.

We assume that each process has a clock variable and the rate of increase of clock is same for all the processes. We assume that there exists a distinguished process (or a base station) in the network that is responsible for initiating the computation of the program in WAC model. Furthermore, we assume that the execution of an action is instantaneous. The time is consumed *between* successive actions.

Proving correctness of transformation. We now discuss how we prove that our algorithms are correct and preserve stabilization property during transformation. Towards this end, we define the notion of *equivalence* between a computation of the given program and computation of the transformed program. This notion is based on the definition of refinement [1, 95] and simulation [95].

Consider the transformation of program p in read/write model into program p' in WAC model. Note that in WAC model, multiple writes can occur at once whereas in read/write model, at most one read/write can occur at a time. Hence, each step of the program in WAC model would be simulated in read/write model by multiple steps. If $c' = \langle s_0, s_1, \ldots \rangle$ is a computation of p' and c is a computation of p, we say that c and c' are equivalent if c is of the form $\langle t_{00}, t_{01}, \ldots, t_{0f_0}(= t_{10}), t_{11}, \ldots, t_{1f_1}(= t_{20}), \ldots \rangle$, where $\forall j : s_j$ and t_{j0} are identical (subject to renaming of variables) as far as the variables of p are concerned.

To show that our transformations are stabilization preserving, we proceed as follows. We show that given any computation of p', there exists a suffix of that computation such that there is a computation of p that is equivalent to that suffix. If the given program, p, is stabilizing fault-tolerant then any computation of p is guaranteed to reach legitimate states and satisfy the specification after reaching legitimate states. It follows that eventually, a computation of p' will reach legitimate states from where it satisfies its specification.

3.2 Read/Write Model to WAC Model in Timed Systems

In this section, we present the transformation algorithm to transform a program in read/write model into a program in WAC model for timed systems. From Chapter 2, we note that in the WAC model there is no equivalent of a read action (as in read/write model or shared-memory model). Hence, an action by which process j reads the state of k in read/write model needs to be modeled in the WAC model by requiring k to write the appropriate value at j. When k writes the state of j in this manner, it is necessary that no other neighbor of j is writing the state of j at the same time. Finally, a write action (of read/write model) can be executed in WAC model as is.

To obtain the transformed program that is correct in WAC model, we can use a collision-free time-slot based protocol like time-division multiple access (TDMA) [12, 14, 60, 76, 80]. TDMA ensures that when process k writes the state of j no other neighbor of j is executing simultaneously. More specifically, when k executes, TDMA guarantees that the neighbors within distance 2 of k do not execute simultaneously.

We initially assume that the clocks of the processes are initialized to 0 and the rate of increase of the clocks is same for all processes. Subsequently, we also present an approach to deal with uninitialized clocks; this approach enables us to ensure that if the given program in read/write model is self-stabilizing then the transformed program in WAC model is also self-stabilizing.

3.2.1 Transformation Algorithm

The main idea behind this algorithm is graph coloring. Let the communication graph in the given program in read/write model be $G=(V, E)$. We transform this graph into $G'=(V, E')$ such that $E'=\{(x, y)|(x \neq y) \wedge ((x, y) \in E \vee (\exists z :: (x, z) \in E \wedge (z, y) \in$

$E))\}$ (cf. Figure 3.1). In other words, two distinct vertices x and y are connected in G' if the distance between x and y in G is at most 2. Let $f : V \rightarrow [0 \ldots K-1]$ be the color assignments such that $(\forall j, k : k$ is a neighbor of j in $G' : f(j) \neq f(k))$, where K is any number that is sufficient for coloring G'.

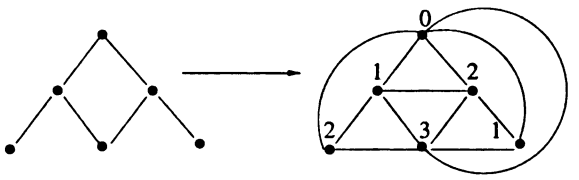


Figure 3.1: Transformation using graph coloring. The number associated with each process denotes the color of the process.

Let $f.j$ denote the color of process j. Now, process j can execute at $clock.j = f.j$. Moreover, process j can execute at time slots $f.j + c * K$, where c is an integer. When j executes, it writes its own state and the state of its neighbors in G. Based on the color assignment, it follows that two write actions do not collide. Figure 3.2 shows the actions of process j. As discussed in Chapter 2, for simplicity of presentation, the variable $v.j$ at j (in Figure 3.2) captures the values of all variables that any neighbor of j can read in read/write model.

process j
const $f.j$, K;
var $v.j$, $l.j$, $clock.j$;
initially
set $v.j$ according to the initial value of $v.j$ in read/write model;
set $l.j$ according to the initial value of $copy.j$ in read/write model;
$clock.j = 0$;
begin
$(\exists c : clock.j = f.j + c * K) \longrightarrow$ if($predicate(v.j, l.j)$) update $v.j$;
$\forall k : k$ is a neighbor of j in the original graph :
$l.k.j = v.j$;
end

Figure 3.2: Transformation for arbitrary topology

Theorem 3.1 *For every computation of the transformed program in WAC model*

under powerset semantics there is an equivalent computation of the given program in read/write model.

Proof. Consider the program in the WAC model (cf. Figure 3.2). Based on the initial values of the clocks, the process with $color = 0$ starts the computation. The initial values of the variables of the program are assigned according to the program in the read/write model.

A write action by process j in the transformed program is equivalent to the following computation of the original program in read/write model: a write by process j, followed by the read actions by the neighbors of j. Now, consider the case where multiple processes execute simultaneously in WAC model. Based on the slot assignment, there are no collisions and, hence, these multiple write-all actions can be serialized. If $\langle s_0, s_1 \rangle$ is a transition of the transformed program in WAC model under powerset semantics then there is a corresponding computation $\langle u_0, u_1, \ldots, u_m \rangle$ of the program in interleaving semantics, where m is the number of processes that execute simultaneously in the transition $\langle s_0, s_1 \rangle$. Moreover, for each transition $\langle u_i, u_{i+1} \rangle$ in interleaving semantics, there is a corresponding computation of the program in read/write model. Consider the transition $\langle u_0, u_1 \rangle$ of the program in interleaving semantics. For the transition $\langle u_0, u_1 \rangle$ (say, of process p), we can construct a computation in read/write model, $\langle t_{00}, t_{01}, \ldots, t_{0f_0} \rangle$, where $x_1, x_2, \ldots, x_{f_0-1}$ are neighbors of p. Since t_{ab} is a state of the program in read/write model, we identify the v and *copy* values for t_{ab}.

- For state t_{00} : $v.p(t_{00}) = v.p(u_0), \forall x : x$ is a neighbor of p : $copy.x.p(t_{00}) = l.x.p(u_0)$. This is due to the fact that the variables of the program in WAC model are initialized according to the initial values in the program in read/write model.

- State t_{01} is obtained from t_{00} by executing the write action at process p.

- State $t_{0w}, 1 < w < f_0$, is obtained from $t_{0(w-1)}$ where process x_{w-1} reads the

value of $v.p$.

Now, in $t_{10} = t_{0f_0}$, we have $v.p(t_{10}) = v.p(u_1), \forall x : x$ is a neighbor of p : $copy.x.p(t_{10}) = l.x.p(u_1)$. Furthermore, by induction, if $u_0(= s_0), u_1, \ldots u_m(= s_1)$ is a computation of the program in interleaving semantics then there exists a sequence of transitions $t_{00}, t_{01}, \ldots, t_{0f_0}(= t_{10}), t_{11}, \ldots, t_{1f_1}(= t_{20}), \ldots, t_{(m-1)f_{m-1}}(= t_{m0})$ that is a computation of the given program in read/write model.

Thus, for each computation of the transformed program in WAC model in interleaving semantics, there is an equivalent computation of the given program in read/write model. It follows that, for a computation of the transformed program in the WAC model under powerset semantics, there is an equivalent computation of the given program in the read/write model. □

3.2.2 Preserving Stabilization in Timed Systems

To show that stabilization is preserved during transformation, we first present how we deal with the case where the clocks are not initialized or clocks of processes are corrupted. Note that we have assumed that the rate of increase of the clocks is still the same for all processes. To recover from uninitialized clocks, we proceed as follows: Initially, we construct a spanning tree of processes rooted at the base station. Let $p.j$ denote the parent of process j in this spanning tree. Process j is initialized with a constant $c.j$ which captures the difference between the initial slot assignment of j and $p.j$.

If clocks are not synchronized, the action by which $p.j$ writes the state of j may collide with other write actions in the system. Process j uses the absence of this write to stop and wait for synchronizing its clock. Process j waits for $K * n$ slots, where K is the period between successive slots and n is the number of processes in the system. This ensures that process j starts executing only when all its descendants

have stopped in order to synchronize their clocks. When j later observes the write action performed by $p.j$, it can use $c.j$ to determine the next slot in which it should execute. Since the root process or the base station continues to execute in the slots assigned to it, eventually, its children will synchronize with the root. Subsequently, the grandchildren of the root will synchronize, and so on. Continuing thus, the clocks of all processes will be synchronized and hence, further computation will be collision-free.

In the absence of topology changes, the spanning tree constructed above and the values of $f.j$ and $c.j$ are constants. Hence, for a fixed topology, we assume that these values are not corrupted. (This assumption is similar to assumptions elsewhere where process ID cannot be corrupted.) Once the clocks are synchronized, from Theorem 3.1, for the subsequent computation of the transformed program, there is an equivalent computation of the given program in read/write model. Also, if the given program is self-stabilizing then every computation of that program reaches legitimate states. Combining these two results, it follows that every computation of the transformed program eventually reaches legitimate states. Thus, we have

Theorem 3.2 *If the given program in read/write model is self-stabilizing then transformed program (with the modifications discussed above) in WAC model is also self-stabilizing.* □

3.2.3 Transformation of Programs in Shared-Memory Model

From Chapter 2, we note that in shared-memory model, a process can read the state of all its neighbors and write its own state. In the transformation proposed in Section 3.2.1, a read action of process j in read/write mode is simulated in WAC model by requiring the neighbors of j to write their state at j. Now, to transform a program in shared-memory model into a program in WAC model, we proceed as follows.

In shared memory model, whenever a process executes, it has the *fresh* state

information of all its neighbors. Therefore, in our transformation, we need to ensure the following: if process j executes its actions at time t then, before j can execute its actions again (at time $t' > t$), it should allow all its neighbors to execute at least once. In other words, when process j executes again, it should have the new state of all its neighbors. Thus, read all action (i.e., read the state of all its neighbors) by process j in shared-memory model is simulated in WAC model. To implement this transformation, we need a TDMA service that ensures the following property; between every two slots assigned to a process (say, j), each neighbor of j is assigned at least one slot. The transformation algorithm proposed in Section 3.2.1 ensures this property. We also note that the TDMA algorithm from [80] guarantees this property. Hence, a program in shared-memory model can be transformed into a program in WAC model using these algorithms.

3.3 Illustration: Transformation of a Routing Program

In this section, we illustrate the transformation algorithm from read/write model into WAC model. Specifically, we consider the logical grid routing protocol (LGRP) [25] designed for A Line in the Sand (LITeS) experimental project [9], where a sensor network is used for target detection, classification and tracking. We note that the LGRP program is a variation of the balanced routing program proposed in [27], modified to work in the context of grid based network topology. We transform the LGRP program in read/write model into a program in WAC model. The resulting program is a variation of the program used in LITeS project [9]. (There are small differences between the transformed program and the program in [9]. We identify them at the end of this section.)

In LGRP, sensors are organized into a logical grid, either offline or using a local-

ization algorithm. A routing tree is dynamically constructed with the base station as the root. The base station is located at $\langle 0, 0 \rangle$ of the logical grid. A sensor (say, j) classifies its neighbors within H hops as low neighbors or high neighbors. Specifically, sensor k, located within H hops of j, is classified as j's low neighbor if k's distance to the base station in the logical grid is less than that of j's distance. Otherwise, it is classified as j's high neighbor. The set of low neighbors and high neighbors identify the potential parents of j in the routing tree. The low and high neighbors are computed statically (or at initialization) and are assumed to be constant as far as LGRP is concerned.

Sensor j maintains a variable, *inversion count*. The inversion count of the base station is 0. If j chooses one of its low neighbors as its parent then it sets its inversion count to that of its parent. Otherwise, j sets its inversion count to inversion count of its parent + 1. If j finds a neighbor which gives a smaller inversion count, then it replaces its parent and updates its inversion count. Furthermore, when the inversion count exceeds a certain threshold, it indicates that the tree may be corrupted (i.e., contain cycles). To deal with this problem, when the inversion count exceeds a predetermined threshold, j sets its parent to *null*. Sensor j will later rejoin the routing tree when it finds a neighbor which provides a better inversion count. Once the routing tree is established, whenever j receives a data message, it forwards the message to its parent.

Next, we specify the program in read/write model and then transform it into WAC model. In this illustration, we consider $H = 1$, to simplify the presentation of the program. We note that our transformations can be used for other values as well.

3.3.1 LGRP Program in Read/Write Model

Figure 3.3 shows the LGRP program in read/write model. In this program, process j $(1..N-1)$ maintains 2 public variables, $inv.j$, inversion count of j, and $up.j$, status

of j (indicates whether j has failed or not).[1] Also, process j maintains a private variable, $p.j$, the parent of j and 2 copy variables, $copy.j.inv.k$, the inversion count values of neighbors of j when j read last, and $copy.j.up.k$, the status of neighbors of j. The action of the base station (i.e., process bs) is as follows:

$$\boxed{true \longrightarrow p.bs, inv.bs, up.bs = bs, 0, true;}$$

In this program, process j reads the state of its neighbors and updates its copy variable, $copy.j.inv.k$ and $copy.j.up.k$. When j finds a low neighbor or a high neighbor that gives a better inversion count value, it replaces its parent and updates the inversion count accordingly. If $p.j$ has failed, inversion count of $p.j$ has reached the maximum allowed value, or inversion count of j is not consistent with its parent, j removes itself from the routing tree by setting $p.j$ to $null$ and $inv.j$ to $cmax$. It will rejoin the routing tree when it finds a neighbor that provides a better inversion count. Thus, the LGRP program in read/write model is as shown in Figure 3.3.

3.3.2 Transformed LGRP Program in WAC Model

Figure 3.4 shows the transformed routing program in WAC model. We obtained this program using the transformation algorithm proposed in Section 3.2.1. In this program, process j $(1..N-1)$ knows the color assigned to it $(f.j)$ and the maximum number of colors used in the system (K). Furthermore, j maintains 2 copy variables, $l.j.inv.k$ and $l.j.up.k$ similar to the copy variables in the program in read/write model. However, $l.j.inv.k$ and $l.j.inv.k$ indicates the value of $inv.k$ and $up.k$ respectively when process k last wrote j. The action of the base station (i.e., process bs) is as follows, where $f.bs$ is the color assigned to the base station:

[1] In this program, whenever a sensor/process fails, it notifies its neighbors. This action can be implemented as follows; whenever a sensor fails to read its neighbor for a threshold number of consecutive attempts, it declares that neighbor as failed. Similarly, in WAC model, whenever a sensor fails to receive update from its neighbor, it declares that neighbor as failed.

```
process j : (1..N−1)
const
  cmax;                                          // maximum inversion count
  ln.j;                                          // low neighbors of j
  hn.j;                                          // high neighbors of j
var
  public    inv.j : {0..cmax};                   // inversion count of j
  public    up.j : {true, false};                // status of j
  private   p.j : {ln.j ∪ hn.j ∪ null};          // parent of j
  copy      ∀k ∈ {ln.j ∪ hn.j} : copy.j.inv.k;   // inversion count of neighbors of j
  copy      ∀k ∈ {ln.j ∪ hn.j} : copy.j.up.k;    // status of neighbors of j
begin
  true  ⟶  copy.j.inv.k = inv.k;                 // read inv.k
           copy.j.up.k = up.k;                   // read up.k
  k ∈ ln.j ∧ copy.j.up.k ∧ (copy.j.inv.k < cmax) ∧ (copy.j.inv.k < inv.j)
        ⟶  p.j, inv.j = k, copy.j.inv.k;         // low neighbor which gives a better
                                                 // path to the base station (i.e., root)
  k ∈ hn.j ∧ copy.j.up.k ∧ (copy.j.inv.k < cmax) ∧ (copy.j.inv.k + 1 < inv.j)
        ⟶  p.j, inv.j = k, copy.j.inv.k + 1;     // high neighbor which gives a better
                                                 // path to the base station (i.e., root)
  p.j ≠ null∧
    (¬copy.j.up.(p.j)∨                           // parent is dead, or
    (copy.j.inv.(p.j) >= cmax)∨                  // parent's inversion count >= cmax, or
    (p.j ∈ ln.j ∧ inv.j ≠ copy.j.inv.(p.j))∨     // j's inversion count is not
    (p.j ∈ hn.j ∧ inv.j ≠ copy.j.inv.(p.j) + 1)) //   consistent with its parent
        ⟶  p.j, inv.j = null, cmax;
  p.j = null ∧ inv.j < cmax                      // j has no parent and its
        ⟶  inv.j = cmax;                         // inversion count is less than cmax
end
```

Figure 3.3: Grid routing program in read/write model

```
∃c : clock.bs = f.bs + c * K ⟶

        p.bs, inv.bs, up.bs = bs, 0, true;

        ∀k : k is a neighbor of bs : l.k.inv.bs, l.k.up.bs = inv.bs, up.bs;
```

Also, in the time slot assigned to j, process j executes the write actions consistent with the write actions in the program in read/write model. Specifically, j replaces its parent and updates its inversion count if it finds a neighbor that gives a better inversion count value. Also, it sets $p.j$ to *null* and $inv.j$ to $cmax$ if its current parent has failed, inversion count of its current parent exceeds the threshold, or j's inversion

count is not consistent with that of its current parent. Once j updates its local variables, it executes its write-all action. In other words, j updates $inv.j$ and $up.j$ at its neighbors.

```
process j : (1..N−1)
const
  cmax, ln.j, hn.j;
  f.j;                                          // color of j
  K;                                            // maximum colors used in the network
var
  public    inv.j, up.j;
  private   p.j, clock.j (initially, clock.j = 0);
  copy      ∀k ∈ {ln.j ∪ hn.j} : l.j.inv.k;     // inversion count of neighbors of j
  copy      ∀k ∈ {ln.j ∪ hn.j} : l.j.up.k;      // status of neighbors of j
begin
  (∃c : clock.j = f.j + c * K)
      ⟶
      // execute write action consistent with the read/write program
      if (k ∈ ln.j ∧ l.j.up.k ∧ (l.j.inv.k < cmax) ∧ (l.j.inv.k < inv.j))
          p.j, inv.j = k, l.j.inv.k;
      if (k ∈ hn.j ∧ l.j.up.k ∧ (l.j.inv.k < cmax) ∧ (l.j.inv.k + 1 < inv.j))
          p.j, inv.j = k, l.j.inv.k + 1;
      if (p.j ≠ null∧
        (¬l.j.up.(p.j)∨
        (l.j.inv.(p.j) >= cmax)∨
        (p.j ∈ ln.j ∧ inv.j ≠ l.j.inv.(p.j))∨
        (p.j ∈ hn.j ∧ inv.j ≠ l.j.inv.(p.j) + 1)))
          p.j, inv.j = null, cmax;
      if (p.j = null ∧ inv.j < cmax)
          inv.j = cmax;
      // execute write-all actions, updating the state of all the neighbors of j
      ∀k : k ∈ {ln.j ∪ hn.j} : l.k.inv.j, l.k.up.j = inv.j, up.j;
end
```

Figure 3.4: Grid routing program in WAC model

Thus, the program written in read/write model is transformed into a program in WAC model. The transformed program shown in Figure 3.4 differs from the program in [25]. First, unlike the program in [25], the transformed program uses TDMA to execute the write-all actions. This ensures collision-free update of the state of a sensor at its neighbors. Next, the transformed program abstracts the failure of sensors. In

[25], the authors use "heartbeat" messages; the lack of a threshold number of such messages indicates failure of sensors.

3.4 Efficiency of the Transformation

In the transformation from read/write model (and shared-memory model) to WAC model, the lower bound on the time required for an enabled process to execute again is $O(d)$, where d is the maximum degree of the communication graph. This is due to the fact that once a process executes, it needs to allow its neighbors to execute before executing again.

Now, we compute the efficiency of the transformations proposed in this chapter and in [81]. For timed systems, a process executes once in K slots where K is the number of colors used to color the extended communication graph. Thus, in timed systems, the transformation can slow down the given algorithm by a factor of K that is bounded by $d^2 + 1$, where d is the maximum degree of any node in the graph. This slow down is reasonable in sensor networks where topology is typically geometric and value of K is small (e.g., $K = 5$ for grid-based topology). Table 3.1 shows the performance of our transformation algorithms.

Table 3.1: Performance of different transformation algorithms

Systems assumptions	Time complexity
Untimed (asynchronous) systems [81]	$O(N)$
Timed systems, grid topology [81]	$O(1)$
Timed systems, arbitrary topology	$O(K)$, $K \leq d^2 + 1$

where N is the number of processes in the system, K is the number of colors required to obtain distance-2 vertex coloring, and d is the maximum degree.

In [58], Herman introduce *local broadcast with collision* model for sensor networks. They propose *cached sensor transform* (CST) that allows one to correctly simulate a

program written for interleaving semantics in sensor networks. Unlike the transformation algorithms proposed in this chapter, CST uses carrier sense multiple access (CSMA) to broadcast the state of a sensor. Furthermore, CST uses randomization while allowing concurrent execution of multiple processes.

3.5 Chapter Summary

In this chapter, we showed that a TDMA algorithm can be effectively used to transform a program in read/write model or shared-memory model into a program in WAC model. If the TDMA algorithm is self-stabilizing then the transformation preserves the stabilization property of the given program in read/write model or shared-memory model.

We illustrated the transformation algorithm using the logical grid routing protocol (LGRP) [25] designed for the Line in the Sand project [9]. We transformed the LGRP program in read/write model into a program in WAC model. We showed that the transformed program is similar to the LGRP implementation in [9], where the write-all action is implemented using a broadcast message. Furthermore, the transformed program is the TDMA version of the current LGRP implementation and, hence, the problem of message collision is avoided.

Chapter 4

Stabilization-Preserving Deterministic Transformations for WAC Model

In Chapter 3, we showed that a time division multiple access (TDMA) algorithm is applicable in transforming existing programs written in read/write model or shared-memory model into write all with collision (WAC) model. Specifically, the transformation proposed in Chapter 3 takes any TDMA algorithm (or distance 2 coloring algorithm) in WAC model (e.g., [22, 60, 80]) as input. If the algorithm in [80], which is self-stabilizing, deterministic and designed for grid based topologies, is used with the transformation algorithm in Chapter 3 then the transformed program in WAC model is self-stabilizing and deterministically correct for grid based topologies. And, if the algorithms in [22, 60], which are randomized, are used then the transformed program in WAC model is probabilistically correct. Thus, if a self-stabilizing deterministic TDMA algorithm in WAC model for an arbitrary topology were available then it would enable us to provide deterministic guarantees about the transformed program in WAC model. With this motivation, in this chapter, we propose a self-

stabilizing deterministic TDMA algorithm. This algorithm can be used to transform existing self-stabilizing abstract programs into programs in WAC model that are deterministically self-stabilizing.

The rest of the chapter is organized as follows. In Section 4.1, we present our self-stabilizing TDMA algorithm in shared-memory model. Programs written in this model are easy to understand and, hence, we discuss our algorithm first in this model. In this algorithm, we reuse existing graph traversal algorithms (e.g., [36, 66, 108, 109]). Subsequently, in Section 4.2, we transform this algorithm into WAC model. Then, in Section 4.3, we show how stabilization can be added to the TDMA algorithm in WAC model. In Section 4.4, we present extensions to improve the bandwidth utilization of the sensors. Then, in Sections 4.5 and 4.6, we present optimizations to deal with corruption and topology changes. In Section 4.7, we discuss some of the questions raised by this work and in Section 4.8, we discuss the related work.

4.1 Self-Stabilizing TDMA in Shared-Memory Model

TDMA is the problem of assigning time slots to each sensor. Two sensors j and k can transmit in the same time slot if j does not interfere with the communication of k and k does not interfere with the communication of j. In other words, j and k can transmit in the same slot if the communication distance between j and k is greater than 2. Towards this end, we model the sensor network as a graph $G = (V, E)$, where V is the set of all sensors deployed in the field and E is the communication topology of the network. Specifically, if sensors j and k can communicate with each other then the edge $(j, k) \in E$. The function $distance_G(j, k)$ denotes the distance between j and k in G. Thus, the problem statement of TDMA is shown in Figure 4.1.

In this algorithm, we split the system architecture into 3 layers: (1) token circu-

Problem statement: TDMA
Given a communication graph $G=(V,E)$; assign time slots to V such that the following condition is satisfied:
If $j,k \in V$ are allowed to transmit at the same time, then $distance_G(j,k) > 2$

Figure 4.1: Problem statement of TDMA

lation layer, (2) TDMA layer, and (3) application layer. The token circulation layer circulates a token in such a way that every sensor is visited at least once in every circulation. The TDMA layer is responsible for assigning timeslots to all the sensors. And, finally, the application layer is where the actual sensor network application resides. All application message communication goes through the TDMA layer. Now, we explain the functions of the first two layers in detail.

4.1.1 Token Circulation Layer

The token circulation layer is responsible for maintaining a spanning tree in the network and traversing the graph infinitely often. In this chapter, we do not present a new algorithm for token circulation. Rather, we only identify the constraints that this layer needs to satisfy. The token circulation protocol should recover from token losses and presence of multiple tokens in the network. In other words, we require that the token circulation protocol be self-stabilizing. We note that graph traversal algorithms such as [36, 66, 108, 109] satisfy these constraints. Hence, any of these algorithms can be used.

Remark. Although TDMA slot assignment in shared-memory model is (expected to be) possible without a token circulation layer, we have used it to simplify the transformation to WAC model.

4.1.2 TDMA Layer

The TDMA layer uses a distance 2 coloring algorithm for determining the initial slots of the sensors. Hence, we present our algorithm in two parts: (1) distance 2 coloring and (2) TDMA slot assignment.

Distance 2 coloring. Given a communication graph $G = (V, E)$ for a sensor network, we compute E' such that two distinct sensors x and y in V are connected if the distance between them in G is at most 2. To obtain distance 2 coloring, we require that $(\forall (i,j) \in E' :: color.i \neq color.j)$, where $color.i$ is the color assigned to sensor i. Thus, the problem statement is defined in Figure 4.2.

Problem statement: Distance 2 coloring
Given a communication graph $G=(V,E)$; assign colors to V such that the following condition is satisfied:
$(\forall (i,j) \in E' :: color.i \neq color.j)$
where, $E' = \{(x,y) | (x \neq y) \wedge ((x,y) \in E \vee (\exists z \in V :: (x,z) \in E \wedge (z,y) \in E))\}$

Figure 4.2: Problem statement of distance 2 coloring

We use the token circulation protocol in designing a distance 2 coloring algorithm. In our algorithm, each sensor maintains two public variables: *color*, the color of the sensor and *nbrClr*, a vector consisting of $\langle id, c \rangle$ elements, where id is a neighbor of the sensor and c is the color assigned to corresponding sensor. Initially, *nbrClr* variable contains entries for all distance 1 neighbors of the sensor, where the corresponding color assignments are undefined. A sensor can choose its color from K, the set of colors. To obtain a distance 2 coloring, $d^2 + 1$ colors are sufficient, where d is the maximum degree in the graph (cf. Lemma 4.1). Hence, K contains $d^2 + 1$ colors.

Figure 4.3 shows the algorithm for distance 2 coloring. In this algorithm, whenever a sensor (say, j) receives the token from the token circulation layer, it executes actions A1 and A2 (in that order). Action A1 determines the colors used in the distance 2 neighborhood of j and chooses a non-conflicting color. Action A2 ensures that $color.j$

is properly updated at its neighbors and subsequently forwards the token. We note that for simplicity of presentation, we represent action A2 separately from action A1. Whenever j receives the token, we require that action A2 is executed only after action A1 is executed at least once.

Action A1. First, j reads *nbrClr* of all its neighbors and updates its private variable *dist2Clr.j*. The variable *dist2Clr.j* is a vector similar to *nbrClr.j* and contains the colors assigned to the sensors at distance 2 of j. Next, j computes the set *used.j* which denotes the colors used in its distance 2 neighborhood. If *color.j* $\in$ *used.j*, j chooses a color from K that is not used in its distance 2 neighborhood. Otherwise, j keeps its current color.

Action A2. Once j chooses its color, it requires that its neighbors read its current color. Specifically, j waits until all its distance 1 neighbors have copied *color.j*. Towards this end, sensor l will update *nbrClr.l* with $\langle j, color.j \rangle$ (using action A3) if j is a neighbor of l and *color.j* has changed. Once all the neighbors of j have updated *nbrClr* with *color.j*, j forwards the token (using the token circulation layer).

Now, we illustrate our distance 2 coloring algorithm with an example (cf. Figure 4.4). Let us assume that the token circulation layer maintains a depth first search (DFS) tree rooted at sensor r. Whenever a sensor receives a token, the TDMA layer computes the colors used in the distance 2 neighborhood and decides the color of the sensor. In Figure 4.4, let the colors assigned to sensors r, a, c and d be $0, 1, 2$ and 3 respectively. When sensor b receives the token, *nbrClr.b* contains $\{\langle c, 2\rangle\}$ and *dist2Clr.b* contains $\{\langle a, 1\rangle, \langle d, 3\rangle\}$. Thus, *used.b* contains $\{1, 2, 3\}$. Once this information is known, b determines its color. In this example, b sets its color to 0, the minimum color not used in its distance 2 neighborhood. Similarly, other sensors determine their colors.

Lemma 4.1 *If d is the maximum degree of a graph then $d^2 + 1$ colors are sufficient*

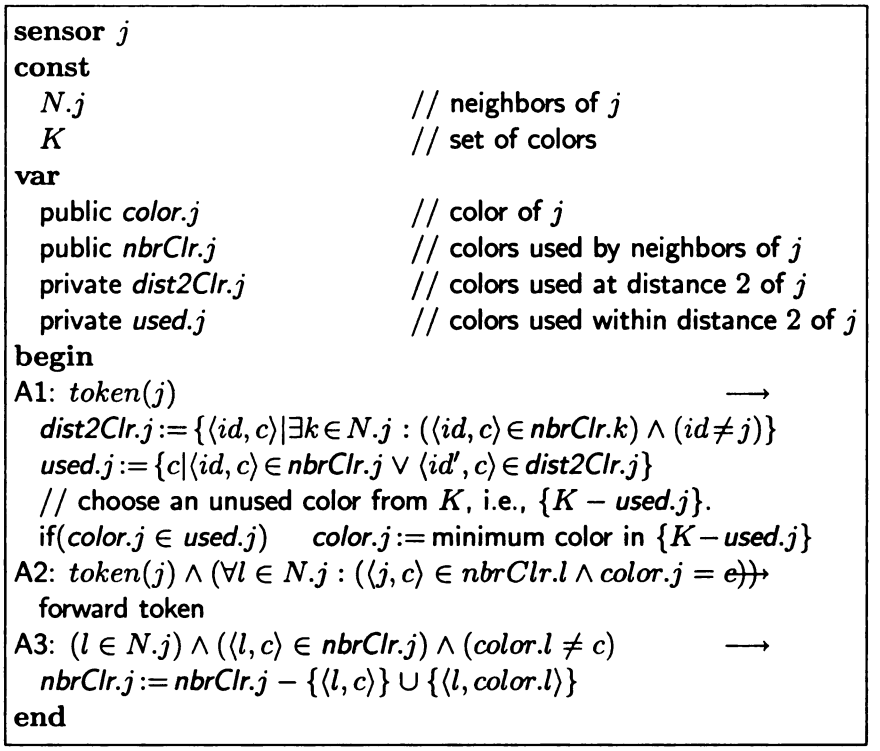


Figure 4.3: Algorithm for distance 2 coloring in shared-memory model

to obtain distance 2 coloring.

Proof. Based on the assumption about degree, given any vertex v, there exists at most d distance 1 neighbors, $d(d-1)$ distance 2 neighbors. Thus, at most d^2 vertices are within distance 2 of v. Now, we can arrange the vertices in some order and allow them to choose a color in such a way that the choice does not conflict with vertices that are considered earlier and within distance 2. When a vertex is about to choose a color, at most d^2 colors could be in its distance 2 neighborhood. Thus, a vertex can choose a color such that it does not overlap with the colors assigned to vertices in its distance 2 neighborhood. □

Corollary 4.2 *For any sensor j, used.j contains at most d^2 colors.* □

Theorem 4.3 *The above algorithm satisfies the problem specification of distance 2*

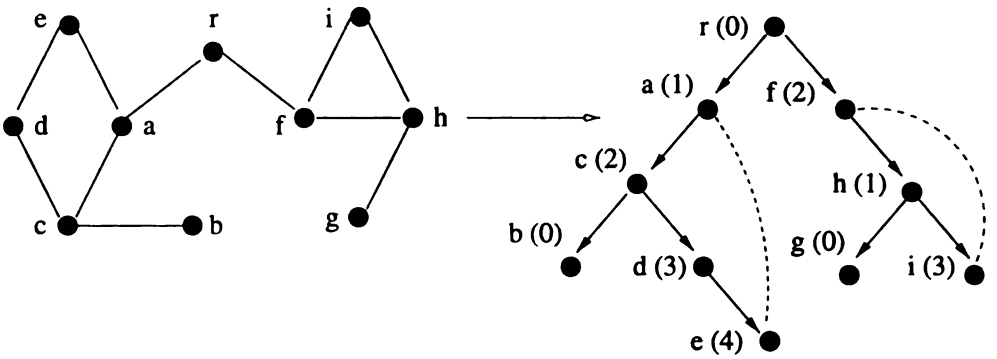


Figure 4.4: Color assignments using depth first search token circulation. The number associated with each sensor denotes the color assigned to that sensor. The dashed edges denote the back edges in the depth first search tree.

coloring. □

Theorem 4.4 *Starting from arbitrary initial states, the above algorithm recovers to states from where the problem specification of distance 2 coloring is satisfied.*

Proof. Based on the assumption in Section 4.1.1, the token circulation layer is self-stabilizing. The TDMA layer preserves the stabilization property of the token circulation layer since it eventually allows a sensor to forward the token. Thus, starting from arbitrary initial states, the token circulation algorithm self-stabilizes to states where only one token is present in the network. In the circulation of the token after stabilization, we show that the following conditions are satisfied.

- Given any sensor v_a that is visited by the token, color of v_a does not conflict with sensors that are within distance 2 of v_a and have been visited.
- Given any sensor v_a that is visited by the token, color of v_a is correctly captured in all its neighbors.

Let $v_1, v_2, \ldots, v_x$ be the path taken by the token after stabilization. It is straightforward to see that the above conditions are satisfied when the token is sent by v_1. Furthermore, based on the algorithm, these conditions are preserved when the token

is passed. When the token circulation is complete, based on the above conditions, it follows that the specification of distance 2 coloring is satisfied and the colors will be unchanged in subsequent token circulations. □

TDMA slot assignment. Once a sensor (say, j) decides its color, it can compute its TDMA slots. Specifically, *color.j* determines the initial TDMA slot of j. And, future slots are computed using the knowledge about the period between successive TDMA slots. Since the maximum number of colors used in any distance 2 neighborhood is $d^2 + 1$ (cf. Lemma 4.1), the period between successive TDMA slots, $P = d^2 + 1$, suffices. Once the TDMA slots are determined, the sensor forwards the token in its TDMA slot. And, the sensor can start transmitting application messages in its TDMA slots. Thus, the algorithm for TDMA slot assignment is shown in Figure 4.5.

```
const P = (d^2 + 1);
when j decides its color
   j can transmit at slots color.j+c * P, where c ≥ 0
```

Figure 4.5: TDMA slot assignment algorithm in shared-memory model

In Figure 4.4, the maximum degree of the graph is 3. Hence, the TDMA period is 10. However, since the number of colors assigned to sensors is 5, the desired TDMA period is 5. We note that while the number of colors used by our algorithm is small as the value of the d is expected to be small in sensor networks, identifying an optimal assignment is not possible. This is due to the fact that the problem of distance 2 coloring is NP-complete even in an offline setup [92]. In [73, 114], approximation algorithms for offline distance 2 coloring in specific graphs (e.g., planar graphs) are proposed. However, in this chapter, we consider the problem of distributed distance 2 coloring where each sensor is only aware of its local neighborhood. In this case, given a sensor with degree d, the slots assigned to this sensor and its neighbors must be

disjoint. Hence, at least $d+1$ colors are required. Thus, the number of colors used in our algorithm is within d times the optimal. We present an algorithm for computing the TDMA period depending on the local knowledge of the maximum difference in colors assigned to distance 2 neighborhood of each sensor in Section 4.4.1.

Theorem 4.5 *The above algorithm ensures collision-free communication.*

Proof. Consider two distinct sensors j and k such that the distance between j and k in the communication graph G is at most 2. The timeslots assigned to j and k are $color.j + c * P$ and $color.k + c * P$ respectively, where c is an integer and $P = (d^2 + 1)$. Suppose a collision occurs when j and k transmit a message at slots $color.j + c_1 * P$ and $color.k + c_2 * P$ respectively, where $c_1, c_2 > 0$. In other words, $color.j + c_1 * P = color.k + c_2 * P$. From Theorem 4.3, we know that $color.j \neq color.k$. Therefore, collision will occur iff $|color.j - color.k|$ is a multiple of P. However, since the distance between j and k is at most 2, $|color.j - color.k|$ is at most d^2 (less than P). In other words, $|color.j - color.k| \leq d^2 < P$. Hence, if j and k transmit at the same time, then the distance between them is greater than 2. This is a contradiction. Thus, collisions cannot occur in this algorithm. □

Since the distance 2 coloring algorithm is self-stabilizing (cf. Theorem 4.4), starting from arbitrary initial states, the algorithm recovers to states from where the initial TDMA slots assigned to the sensors are collision-free. Once the initial TDMA slots are recovered, the sensors can determine the future TDMA slots. Thus, we have

Theorem 4.6 *Starting from arbitrary initial states, the above algorithm recovers to states from where collision-free communication is restored.* □

4.2 TDMA Algorithm in WAC Model

In this section, we transform the algorithm presented in Section 4.1 into a program in WAC model that achieves token circulation and distance 2 coloring upon appropriate

initialization. (The issue of self-stabilization is handled in Section 4.3.) As discussed earlier, in shared-memory model, in each action, a sensor reads the state of its neighbors as well as writes its own state. However, in WAC model, there is no equivalent of a read action. Hence, the action by which sensor j reads the state of sensor k in shared-memory model is simulated by requiring k to write the appropriate value at j. Since simultaneous write actions by two or more sensors may result in a collision, we allow sensors to execute in such a way that simultaneous executions do not result in collisions.

Observe that if collision-freedom is provided then the actions of a program in shared-memory model can be trivially executed in WAC model. Specifically, the write all action of a sensor (say, j) in WAC model can be thought of as simultaneous read action by all neighbors of j. Our algorithm in this section uses this feature and ensures that collision-freedom is guaranteed. Thus, the effect of execution of a token circulation program in WAC model is similar to the case where it is executed in shared-memory model.

To obtain a program in WAC model, we proceed as follows. In this program, in the initial state, (a) sensors do not communicate among each other and (b) *nbrClr* and *dist2Clr* variables contain entries such that the color assignments are undefined. We present our algorithm in two parts: (1) distance 2 coloring, and (2) TDMA slot assignment.

Distance 2 coloring. Initially, the base station (i.e., sensor r) circulates the token for obtaining distance 2 coloring. Whenever a sensor (say, j) receives the token (from the token circulation layer), it chooses its color. Towards this end, j first computes the set *used.j* which denotes the colors used in its distance 2 neighborhood. If *nbrClr.j* (or *dist2Clr.j*) contains $\langle l, undefined \rangle$, l did not receive the token yet and, hence, *color.l* is not assigned. Therefore, j ignores such neighbors. Afterwards, j chooses a color such that $color.j \notin used.j$. Subsequently, j reports its color to its neighbors

within distance 2 using the primitive *report_distance_2_nbrs* (discussed later in this section) and forwards the token. Thus, the action by which k reads the colors used in its distance 2 neighborhood (in shared-memory model) is modeled as a write action where j reports its color to the sensors in its distance 2 neighborhood using the primitive *report_distance_2_nbrs*.

Note that the order in which the token is circulated is determined by the token circulation algorithm used in Section 4.1, which is correct under the shared-memory model (e.g., [36, 66, 108, 109]). Since token circulation is the only activity in the initial state, it is straightforward to ensure collision-freedom. Specifically, to achieve collision-freedom, if j forwards the token to k in the algorithm used in Section 4.1, we require that the program variables corresponding to the token are updated at j and k without collision in WAC model. This can be achieved using the primitive *report_distance_2_nbrs*. Hence, the effect of executing the actions in WAC model will be one that is permitted in shared-memory model. Figure 4.6 shows the transformed algorithm in WAC model.

```
sensor j
const  N.j, K
var    color.j, nbrClr.j, dist2Clr.j, used.j
begin
token(j)  ⟶ used.j := {c | ⟨id, c⟩ ∈ nbrClr.j ∨ ⟨id', c⟩ ∈ dist2Clr.j}
            color.j := minimum color in K − used.j
            execute report_distance_2_nbrs
            forward token
end
```

Figure 4.6: Algorithm for distance 2 coloring in WAC model

Theorem 4.7 *The above algorithm satisfies the problem specification of distance 2 coloring.*

Proof. Observe that, the action by which a sensor (say, j) reads the colors assigned to sensors in its distance 2 neighborhood is simulated in this algorithm by requiring

j to write its color at the sensors within distance 2 of j. Since there is no other communication before color assignment, token circulation will succeed. Hence, from Theorem 4.3, it follows that the above algorithm satisfies the problem specification of distance 2 coloring. □

TDMA slot assignment. Once a sensor determines its color, it can compute its TDMA slots. Similar to the discussion in Section 4.1, the color of the sensor determines the initial TDMA slot. Subsequent slots can be computed using the knowledge about the period between successive slots. If d is the maximum degree of the communication graph G, the TDMA period, $P = d^2 + 1$ suffices.

However, unlike the algorithm in Section 4.1 in shared-memory model, sensors do not start transmitting messages immediately. Otherwise, the token circulation may be interrupted due to collisions. Once the TDMA slots are determined, a sensor forwards the token in its TDMA slot. Hence, the token circulation does not collide with other TDMA slots. Next, a sensor waits until all the sensors in its distance 2 neighborhood have determined their TDMA slots before transmitting application messages in its TDMA slots. Thus, when a sensor starts transmitting application messages, all sensors in its distance 2 neighborhood have determined their TDMA slots and, hence, does not interfere with other TDMA slots and token circulation. Figure 4.7 shows the TDMA slot assignment algorithm.

const $P = (d^2 + 1)$;
if (j has decided its color) $\wedge$ (all sensors within distance 2 of j are colored)
  j can transmit application messages at slots: $color.j + c * P$, where $c \geq 0$

Figure 4.7: TDMA slot assignment algorithm in WAC model

Theorem 4.8 *The above algorithm ensures collision-free communication.* □

Implementation of *report_distance_2_nbrs*. In the above algorithm, we use the primitive *report_distance_2_nbrs*. In particular, whenever a sensor (say, j) decides its

color, this primitive reports the color to its distance 2 neighborhood. Specifically, it updates the *nbrClr* value of its distance 1 neighbors and *dist2Clr* value of its distance 2 neighbors. We discuss its implementation, next.

Sensor j sends a broadcast message with its color and a schedule for its distance 1 neighbors. The sensors at distance 1 of j update their *nbrClr* values. Based on the schedule in the report message, each of the neighbors broadcast their *nbrClr* vectors. Specifically, if a distance 1 neighbor (say, l) of j is already colored, the schedule requires l to broadcast *nbrClr.l* in its TDMA slot. Otherwise, the schedule specifies the slot that l should use such that it does not interfere with the slots already assigned to j's distance 2 neighborhood. If there exists a sensor k such that $distance_G(l, k) \leq 2$, then k will not transmit in its TDMA slots, as l is not yet colored. (Recall that a sensor transmits application messages only if all its distance 2 neighbors have determined their TDMA slots.) Now, a sensor (say, m) updates *dist2Clr.m* with $\langle j, color.j \rangle$ iff $(m \neq j) \wedge (j \notin N.m)$. Thus, this schedule guarantees collision-free update of *color.j* at sensors within distance 2 of j. Furthermore, this primitive requires at most $d+1$ update messages.

4.3 Adding Stabilization in WAC Model

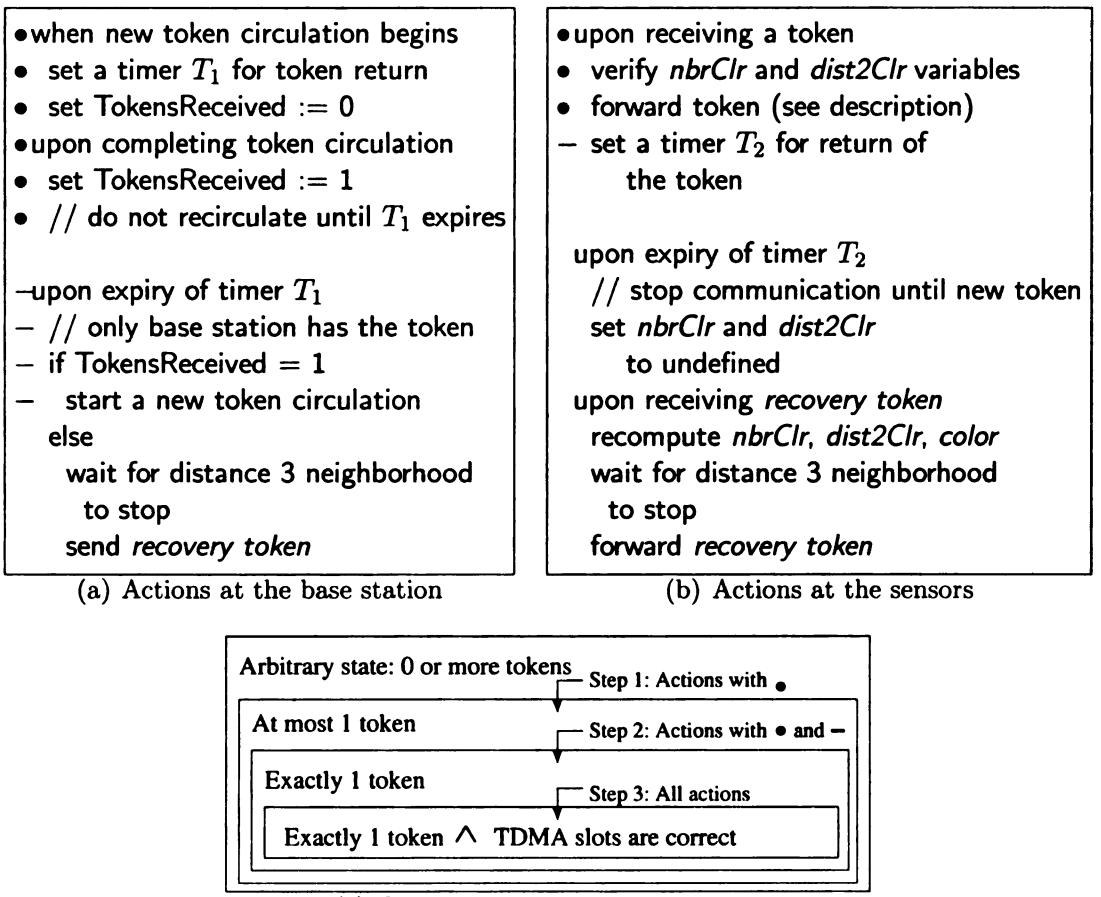
In the algorithm presented in Section 4.2, if the sensors are assigned correct slots then validating the slots is straightforward. Towards this end, we can use a simple diffusing computation to allow sensors to report their colors to distance 2 neighborhood and ensure that the slots are consistent. For simplicity of presentation, we assume that token circulation is used for revalidating TDMA slots. Now, in the algorithm presented in Section 4.2, we observe that in the absence of any faults, the token circulates the network successfully and, hence, slots are revalidated. However, in the presence of faults, the token may be lost due to a variety of reasons, such as, (1) slots

assigned to sensors are not collision-free, (2) *nbrClr* values are corrupted, and/or (3) token message is corrupted. Or, due to transient faults, the token may circulate in a cycle or there may be several tokens.

To obtain self-stabilization, we use the *convergence-stair* approach proposed in [52]. First, we ensure that the token does not circulate in a cycle and if the system contains multiple tokens then it recovers to states where there is at most one token. Then, we ensure that the system recovers to states where there is a unique token (cf. Figure 4.8).

Step 1: Dealing with multiple tokens. During token circulation, there may be multiple tokens in the network or the tokens may circulate in a cycle. To deal with these problems, we add a time-to-live (TTL) field to the token message. Whenever the base station initiates a token circulation, it sets TTL to the number of hops the token traverses during one circulation. Since the token traverses an edge twice (once during visiting a sensor and once during backtracking), the base station sets TTL to $2 * |E_t|$, where $|E_t|$ is number of edges traversed by the token in one circulation. At each hop, the token decrements its TTL value. If this value is zero, the token circulation is terminated. Thus, this ensures that the token returns to the base station within $2 * |E_t|$ hops or it is lost.

To deal with the case of multiple tokens, we ensure that any token in the network either returns to the base station within a predetermined time or it is lost. Towards this end, we ensure that a sensor forwards the token as soon as possible. To achieve this, whenever a sensor, say j, receives the token, j updates its color at its neighbors in its TDMA slot. (This can be achieved within P slots, where P is the TDMA period.) Furthermore, in the subsequent slots, (a) the neighbors relay this information to distance 2 neighbors of j and (b) j forwards the token. (Both of these can be achieved within P slots.) Observe that if the TDMA slots are valid then any token will return in $2 * P * |E_t|$ slots to the base station. Otherwise, it may be lost.



(a) Actions at the base station

(b) Actions at the sensors

(c) Convergence to legitimate states

Figure 4.8: Adding stabilization

In order to revalidate the slots assigned to the sensors, the base station initiates a token circulation once every *token circulation period*, P_{tc} slots. The value of P_{tc} is chosen such that it is at least equal to the time taken for token circulation (i.e., $P_{tc} \geq 2 * P * |E_t|$). Thus, when the base station (i.e., r) initiates a token circulation, it expects to receive the token back within P_{tc} slots. Towards this end, the base station sets a timeout for P_{tc} duration whenever it forwards the token. Now, if the base station sends a token at time t and it does not send any additional token before time $t + P_{tc}$ then all tokens in the network at time t will return to the base station before time $t + P_{tc}$ or they will be lost. Hence, when the timeout expires, there is no token in the network. If the base station does not receive any token before the

timeout expires, it concludes that the token is lost. Similarly, whenever a sensor (say, $j \neq r$) forwards the token, it expects to receive the token in the subsequent round within P_{tc}. Otherwise, it sets the color values in *nbrClr.j* and *dist2Clr.j* to undefined. And, stops transmitting until it recomputes *color.j* and the sensors in its distance 2 neighborhood report their colors. Therefore, at most one token resides in the network at any instant. Thus, we have

Lemma 4.9 *For any system configuration, if the base station initiates a token circulation at time t and does not initiate additional token circulation before time* $t + P_{tc}$ *then no sensor other than the base station may have a token at time* $t + P_{tc}$. □

Steps 2 and 3: Recovery from lost token. Now, if the token is lost in the network, the base station initiates a recovery by sending a *recovery token*. Before the base station sends the *recovery token*, it waits until the sensors in its distance 3 neighborhood have stopped transmitting. This is to ensure that the primitive *report_distance_2_nbrs* can update the distance 2 neighbors of the base station successfully. Let T_{rt} be the time required for sensors in the distance 3 neighborhood of the base station to stop transmitting. Specifically, the value of T_{rt} should be chosen such that the sensors within distance 3 of the base station can detect the loss of the token within this interval. Although, the actual value of T_{rt} depends on the algorithm used for token circulation, it is bounded by P_{tc}. After waiting for T_{rt} amount of time, the base station recomputes its color. Furthermore, it reports its color to the sensors within distance 2 of it. As mentioned in Section 4.2, the primitive *report_distance_2_nbrs* ensures collision-free update since the sensors within distance 3 have stopped. Then, it forwards the *recovery token*.

Now, when a sensor (say, j) receives the *recovery token*, similar to the base station, it waits until the sensors in the distance 3 neighborhood of j have stopped transmitting. Then, j follows the algorithm in Section 4.2 to recompute its color. Once j decides its color, it uses the primitive *report_distance_2_nbrs* to update the sensors

within distance 2 of j with *color.j*. Thus, we have

Lemma 4.10 *Whenever a sensor (say, j) forwards the* recovery token, *sensors within distance* 2 *of j are updated with color.j without collision.* □

The pseudo-code for stabilization and the illustration of how sensors converge to legitimate states are shown in Figure 4.8. Once a sensor recomputes its color, it can determine its TDMA slots using the algorithm in Section 4.2. Thus, we have

Theorem 4.11 *With the above modification, starting from arbitrary initial states, the TDMA algorithm in WAC model recovers to states from where collision-free communication is restored.* □

Time complexity for recovery. Based on the above discussion, the value of T_{rt} depends on the algorithm used for token circulation. Suppose $T_{rt} = P_{tc}$, i.e., the base station waits for one token circulation period before forwarding the *recovery token.* Now, when the base station forwards the *recovery token*, all the sensors in the network would have stopped transmitting. Furthermore, whenever a sensor receives the token, it can report its color without waiting for additional time. To compute the time for recovery, observe that it takes (a) at most one token circulation time (i.e., P_{tc}) for the base station to detect token loss, (b) one token circulation for the sensors to stop and wait for recovery, and (c) at most one token circulation for the network to resume normal operation. Thus, the time required for the network to self-stabilize is at most $2 * P_{tc}+$ time taken for resuming normal operation. Since the time taken for resuming normal operation is bounded by P_{tc}, the time required for recovery is bounded by $3 * P_{tc}$.

We expect that depending on the token circulation algorithm, the recovery time can be reduced. Since this chapter does not focus on a specific token circulation algorithm, we do not consider the issue of optimizing the recovery time. We refer the reader to Section 4.7 for a discussion on local recovery for small perturbations.

4.4 Improving Bandwidth Utilization

In this section, we discuss mechanisms for improving the bandwidth utilization of sensors. First, we show how the TDMA period can be updated. Next, we show how the sensors can locally negotiate to request for additional bandwidth.

4.4.1 Dynamic Update of TDMA Period

In this extension, we focus on the problem of reducing the period between successive slots. This solution is based on the approach presented in [80]. Our solution involves three tasks: (1) allowing each sensor to compute the maximum difference in colors assigned to sensors within distance 2, (2) communicating the difference in the network, and (3) updating the TDMA period.

Task 1: Computing the desired local TDMA period. Regarding the first task, when a sensor (say, j) starts transmitting application messages, j has the knowledge about the colors assigned to sensors within distance 2 of j. Hence, j can compute the maximum difference ($LP.j$) among the colors assigned to the sensors in its distance 2 neighborhood. Specifically, $LP.j = max(\{\forall l, k : distance_G(l, j) \leq 2 \wedge distance_G(k, j) \leq 2 : |color.l - color.k|\}) + 1$. Since j maintains the colors assigned to sensors within distance 2 of j in $used.j$, $LP.j = max(\{\forall c_1, c_2 \in used.j \cup \{color.j\} : |c_1 - c_2|\}) + 1$. The variable $LP.j$ denotes the desired TDMA period for sensor j, since it reflects the maximum number of slots occupied in its distance 2 neighborhood.

Remark. In order to improve the TDMA period, we can ensure that a sensor chooses its color by locally minimizing the maximum difference in colors assigned to its distance 2 neighborhood. Specifically, whenever sensor j receives the token, it sets $color.j = c_j$ where c_j minimizes the quantity $max(\{\forall c_i \in used.j : |c_j - c_i|\})$. In other words, j chooses a color such that the maximum difference between its color and the colors assigned to its distance 2 neighborhood is minimized. Thus, this greedy

approach minimizes the desired local TDMA period value of j.

Task 2: Computing the maximum local TDMA period. Regarding the second part, we use the token circulation algorithm to compute the maximum local TDMA period. Let $token.LP$ denote the maximum desired TDMA period determined so far. When the base station initiates token circulation, it sets $token.LP = LP.r$, where $LP.r$ denotes the maximum difference among the colors assigned to the sensors in the distance 2 neighborhood of the base station. Now, whenever a sensor (say, j) forwards the token, it sets $token.LP = max(token.LP, LP.j)$. It follows that when the base station receives the token back, it will obtain the maximum value of the desired TDMA period of all sensors in the network.

Task 3: Updating the TDMA period. Finally, regarding the third part, once the base station learns the new TDMA period value, it can include this when it initiates the next token circulation. Now, the sensors will learn the new TDMA period value. When the base station initiates the subsequent token circulation, the new TDMA period is used to determine the slots at which a sensor can send a message.

The above extension is intended to show that it is possible to dynamically update the TDMA period based on the colors assigned in the distance 2 neighborhood of all the sensors. However, we note that this approach may not improve the bandwidth utilization of the sensors if the number of colors used in a distance 2 neighborhood is equal to $|K|$ (i.e, all $d^2 + 1$ colors). In Section 4.4.2, we show how sensors can improve their bandwidth utilization by requesting for unused slots in its distance 2 neighborhood.

4.4.2 Local Negotiation Based Bandwidth Reservation

The algorithm in Section 4.3 allocates uniform bandwidth to all sensors. In this section, we consider an extension where a sensor can request for additional bandwidth,

if available. This extension is based on the traditional mutual exclusion algorithms and it utilizes the fact that there is time synchronization and reliable timely delivery provided by TDMA.

In our TDMA algorithm, each sensor is aware of the slots used by the sensors in its distance 2 neighborhood. Hence, a sensor can determine the unused slots and if necessary request for the same. Whenever a sensor (say, j) requires additional bandwidth, it broadcasts a *request_slot* message in its TDMA slot. The message includes the slot requested by j and the time when j made the request. Since the message is broadcast, all distance 1 neighbors of j will receive the message. The distance 1 neighbors of j rebroadcast the message immediately to their neighbors in their earliest TDMA slots. If two or more *request_slot* messages are received before the communication slot assigned to a sensor, these messages are grouped and sent as a single request message.

Now, we show that if j transmitted its request in timeslot x_j and it did not receive any other request with timestamp ts such that $ts < x_j + P$ then j can access the requested timeslot without collisions. Towards this end, observe that if j transmits at slot x_j, all distance 1 neighbors of j can transmit at least once before j's next slot, $x_j + P$, where P is the TDMA period. Thus, if x_j is the slot when j requests unused slots, this request message is received by sensors in its distance 2 neighborhood within slot $x_j + P$. Likewise, if sensor l requests for the same slot such that $distance_G(j, l) \leq 2$, j will learn about l's request within time P of the request. Hence, if j does not receive a request with earlier timestamp before $x_j + P$ then j can use its requested slot without collisions. Furthermore, if j and l request for same slot then only one of them would succeed as the slots in which they request are different (due to collision-freedom of TDMA slots). Additionally, lease mechanisms [125] could be used to avoid starvation, where a sensor is required to renew the additional slots within a certain period of time.

Thus, sensors can request for unused slots when necessary using a simple local negotiation protocol. Furthermore, when a sensor requests unused slots, at most $d + 1$ request messages are transmitted, where d is the maximum degree of the communication graph. And, the sensor can determine whether or not it is allowed to use the requested slots within P slots.

4.5 Optimizations for Token Circulation and Recovery

In this section, we propose mechanisms that allows sensors to improve the reliability of token circulation. First, we note that in the algorithm in Section 4.3, whenever the token is lost, recovery is initiated by the base station. However, it is possible that the slots are still collision-free. This could happen if the token is lost due to message corruption or synchronization errors. To deal with this problem, the base station can choose to initiate recovery only if it misses the token for a threshold number of consecutive attempts.

Additionally, to ensure that the token is not lost due to message corruption, whenever a sensor (say, j) forwards the token, it expects its successor (say, $k \in N.j$) to forward the token within a certain interval. If j fails to receive such *implicit acknowledgment* from k, j retransmits the token (in its TDMA slots) a threshold number of times. If a sensor receives duplicate tokens, it ignores such messages. In [79], we have used implicit acknowledgments in the context of data dissemination across a large scale sensor network. Such data dissemination service is similar to a token circulation algorithm as a given message (respectively, token) is required to be disseminated reliably across the network. In [79], we show that the implicit acknowledgments improved the reliability of dissemination of messages by detecting message losses (for example, due to corruption) and recovered quickly from them,

with the help of simulations and real-world experiments. Based on our experiences with the use of implicit acknowledgments, we expect that the reliability of token circulation can be improved with the help of such implicit acknowledgments.

4.6 Optimizations for Controlled Topology Changes

In our algorithm, controlled addition and removal of sensors do not affect the normal operation of the network. Let us first consider the removal/failure of sensors. Whenever a sensor is removed or fails, the TDMA slots assigned to other sensors are still collision-free and, hence, normal operation of the network is not interrupted. However, the slots assigned to the removed/failed sensor are wasted. We refer the reader to Section 4.4 for approaches on how to reclaim the wasted slots.

Suppose a sensor (say, q) is added to the network such that the assumption about the maximum degree is not violated. Towards this end, we require that whenever a sensor forwards the token, it includes its color and the colors assigned to its distance 1 neighbors. Before q joins the network and starts transmitting application messages, we require q to learn the colors assigned to the sensors within its distance 2 neighborhood. One way to achieve this is by listening to token circulation of its distance 1 neighbors. Once q learns the colors assigned to sensors within distance 2, it can choose its color. Thus, q can determine the TDMA slots that it can use. Now, when q sends a message, its neighbors learn the presence of q and include it in the subsequent token circulations.

With this approach, if two or more sensors are added simultaneously then these new sensors may choose conflicting colors and, hence, collisions may occur. Since our algorithm is self-stabilizing, the network self-stabilizes to states where the colors assigned to all sensors are collision-free. Thus, new sensors can be added to the

network. However, if adding new sensors violates the assumption about the maximum degree of the communication graph, slots may not be assigned to the sensors and/or collisions may occur.

4.7 Discussion

In this section, we discuss some of the questions raised by this work.

Scalability. One of the questions about the transformation is scalability. While the algorithm uses a token circulation approach for assigning initial colors (or recalculating them in the context of stabilization), we note that the algorithm provides acceptable performance in a typical scenario where sensor networks are deployed.

To illustrate the issue of scalability, we consider a network with 100 Mica-2 sensors. (Typically, networks with more than 100 sensors will be organized in sections to ensure that the path to a base station is within acceptable limits [10]. Then, our algorithm can be used independently for each section.) If such sensors are arranged in a 10x10 communication grid then five colors suffice [80]. The token circulation time (P_{tc}) in such networks is 0.99 minutes (where the timeslot interval is 30 ms = the time required to transmit one message in Mica-2 motes) and, hence the recovery time is $3 * P_{tc} = 2.97$ minutes.

Thus, the time required for the network to self-stabilize is small. Additionally, we expect that the number of colors required to obtain distance 2 coloring is small for random deployments. Therefore, our algorithm provides acceptable performance in such deployments.

Local recovery. It is possible to extend our algorithm so that sensors can locally correct the corrupted slots when only a small number of sensors is corrupted. For example, if a sensor learns that its color overlaps with its neighbor within distance 2, it can change its color locally. Alternatively, if only a small set of sensors are corrupted

then we could combine our algorithm with that in [60]. Specifically, whenever a sensor detects that the slots are corrupted, initially, it could use the algorithm in [60] to locally correct the slots. Thus, for the case where only a small subset of sensors are corrupted, the slots will be quickly restored. However, if it fails to assign slots in a fixed interval then the *recovery token* from the base station will restore the slots. Local recovery is especially useful if the base station tries multiple tokens before initiating recovery. Specifically, in this case, small perturbations are corrected locally. However, if the corruption is excessive then our algorithm will ensure recovery in a deterministic interval.

Edge coloring vs. vertex coloring. Our solution is based on vertex coloring where timeslots are assigned to each sensor. An alternative approach is edge coloring where timeslots are assigned to each edge. Formally, the problem of edge coloring is stated as follows: Let $f(a,b)$ be the color assigned to edge (a,b); then $\forall(x,y) \in E$: $(f(x,y) \notin (\{f(j,x)|j$ is a neighbor of $x\} \cup \{f(l,y), f(y,l)|(x \neq l) \wedge (l$ is a neighbor of $y)\})$. Now, a sensor (say, x) can send messages at slots $\exists y$: y is a neighbor of x : $f(x,y)$. Moreover, x can send messages at slots $f(x,y) + c * K$, where $c \geq 0$ and K (the TDMA period) is the number of colors used in the network. Based on the color assignments, whenever x sends a message in the slot $f(x,y) + c * K$, sensor y receives the message successfully (although it may cause collision elsewhere).

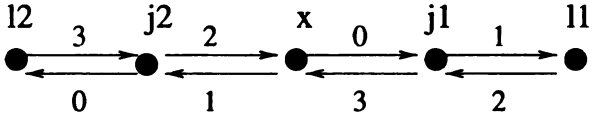


Figure 4.9: Sample time slots assigned for edges in a network. The number associated with each edge indicates the slot at which a process can write to the other process.

Figure 4.9 shows a sample computation slot assignment using edge coloring. In this example, process x can execute in slots $0, 1$ and write the state of its neighbors

$j1, j2$ respectively. When process x executes in slot 0, process $j2$ can execute and write the state of $l2$. We note that in this case, there will be a collision at processes $j2$ and x. However, this does not interfere with the write actions at processes $l2$ and $j1$. This is due to the fact that collisions do not occur at the receivers. Thus, edge coloring can be used to design transformations for the WAC model.

One of the main drawbacks with edge coloring is energy-efficiency. More specifically, with edge coloring, a process has to execute up to d times in order to update the state of all its neighbors, where d is the number of its neighbors. By contrast, with vertex coloring, a process has to execute only once in order to update the state of all its neighbors. Thus, slot assignment using edge coloring is not energy-efficient when compared to vertex coloring.

Time synchronization. We assume that all the sensors have identical clocks. Time synchronization can be achieved as follows: whenever a sensor receives the token, it synchronizes its clock with respect to its parent (i.e., the sensor from which it receives the token for the first time). Thus, sensors can deal with clock drifts and ensure that the slots are collision-free. Furthermore, in the case where TDMA slots are consistent, we can use time synchronization algorithms proposed in the literature for sensor networks. For example, we can integrate a time synchronization service (e.g., [45, 59, 127]) with the TDMA algorithm proposed in this chapter. The time synchronization service synchronizes the clocks of the sensors within a few microseconds. Moreover, we expect that the performance of the time synchronization service will be improved as TDMA will ensure that the time synchronization messages are transmitted successfully.

Violation of maximum degree assumption. As discussed in Section 4.3, whenever a sensor is added to the network, as long as the assumption about the maximum degree, d, of the communication graph is not violated, the normal operation of the network is not affected. However, if this assumption is violated then slots

may not be assigned to the new sensors and/or collisions may occur. To deal with this problem, whenever a sensor is added such that the maximum degree of the communication graph is increased, we can use the approach proposed in Section 4.4.1 to increase the period between successive TDMA slots of the sensors. Towards this end, the base station can update the TDMA period while circulating the token.

Additionally, if the base station is not aware of the violation of the maximum degree, during stabilization, the sensors adjacent to the added sensors learn that the maximum degree has changed. Now, the sensors can use the algorithm in Section 4.4.1 to change the TDMA period accordingly. Thus, if sensors are added to the network in small numbers and in a controlled fashion, normal operation of the network will not be affected.

Variability in degree. If the communication topology of the network is such that the degree of sensors varies considerably in different parts of the network then bandwidth is underutilized in some parts of the network. To address this problem, in Section 4.4.1, we proposed a mechanism by which a sensor can calculate the ideal TDMA period. Specifically, during token circulation, we can compute the maximum difference in colors assigned in distance 2 neighborhood of all sensors, and update the period accordingly. If a sensor requires additional bandwidth, it can request for more slots using the local negotiation protocol proposed in Section 4.4.2.

4.8 Related Work

In this section, we compare and contrast the proposed algorithm with the related work [7, 22, 26, 35, 57, 60, 80, 117, 121].

Self-stabilizing deterministic TDMA algorithms. Related work that deals with self-stabilizing deterministic TDMA algorithms include [35, 80, 117].

SS-TDMA. In [80], Kulkarni and Arumugam propose a self-stabilizing TDMA (SS-

TDMA) algorithm where the topology is known and cannot change. However, in our algorithm, we allow addition/removal of sensors. Additionally, in our solution, we require that the sensors are only aware of their local neighborhood.

Self-stabilizing philosophers. In [35], Danturi et al proposed a self-stabilizing solution to dining philosophers problem where a process cannot share the critical section (CS) with non-neighboring processes also. Such generalized dining philosophers problem has application in distance-k coloring, where k is the distance up to which a process cannot share CS. In [35], each process p is assumed to maintain a tree (rooted at p) that spans the processes with whom p cannot share CS using algorithms from the literature. However, existing tree construction and maintenance algorithms are not written for WAC model. On the contrary, in our algorithm, we show how a token circulation algorithm can be used in WAC model in order to obtain distance 2 coloring. And, on the other hand, unlike our algorithm, the approach in [35] allows concurrent coloring of processes.

BitMAC. In [117], the authors propose BitMAC, a deterministic, collision-free MAC protocol for sensor networks. One of the important assumptions in this paper is that when two writes collide the result is an OR operation between them. Moreover, the algorithm in [117] is not self-stabilizing. Unlike [117], our algorithm is written for WAC model and is also self-stabilizing.

Self-stabilizing randomized TDMA algorithms. In [60], Herman and Tixeuil propose a randomized TDMA slot assignment algorithm where a probabilistic fast clustering technique is used. In their algorithm, first, a maximal independent set is computed. This set identifies the leaders that are responsible for obtaining distance 2 coloring. Further, addition/removal of nodes in their algorithm can cause local collisions (and the effects are contained within distance 3 neighborhood). By contrast, our approach uses a deterministic algorithm to assign timeslots.

In [22], Busch et al propose a randomized TDMA algorithm for sensor networks.

In their approach, initially, a randomized algorithm is used to determine the slots. Later, the sensors enter another phase where the TDMA period is reduced. Both these phases are self-stabilizing and are interleaved. By contrast, we propose a deterministic TDMA solution, where the sensors identify their timeslots without any collisions.

Other TDMA algorithms. Other TDMA algorithms include [7, 26, 57, 121]. In [26], whenever a collision occurs during startup (synchronization phase), exponential backoff is used for determining the time to transmit next. One of the important assumption in [26] is that each node has a unique message length. By contrast, we do not make any such assumption in our TDMA algorithm.

In [121], Sohrabi and Pottie propose a network self-organization protocol, where nodes identify the presence of other nodes and form a multi-hop network. In [7], Arisha et al propose a clustering scheme to allot timeslots to different sensors. Each cluster has a gateway node that informs the sensors in its cluster about the timeslots in which the sensors can transmit. And, in [57], Heinzelman et al propose a clustering algorithm. In these papers, initially, nodes are in random access mode and TDMA slots are assigned during network organization. By contrast, in our solution, we use a deterministic algorithm to assign timeslots. Unlike the algorithms proposed in [7, 26, 57, 121], our algorithm is self-stabilizing.

4.9 Chapter Summary

In this chapter, we presented a self-stabilizing, deterministic time division multiple access (TDMA) algorithm for sensor networks. As discussed in [81], such algorithms suffice in transforming existing programs in shared-memory model into programs in write all with collision (WAC) model. Thus, this algorithm can allow us to transform existing distributed programs and evaluate them in sensor networks. It follows that we can quickly prototype a sensor network protocol with such transformations.

In addition, we presented approaches that allow one to reduce the TDMA period and improve the bandwidth utilization of sensors. Specifically, we discussed an approach where the sensors are allowed to reduce the TDMA period depending on the maximum difference in colors assigned in the distance 2 neighborhood of all sensors. Furthermore, we presented another approach where sensors can request for additional bandwidth whenever necessary. With this scheme, whenever a sensor requires additional bandwidth, we showed that at most $d + 1$ messages are transmitted and it learns whether or not it is allowed to use the additional bandwidth within $d^2 + 1$ slots.

Part II

Experimental Aspects

Chapter 5

ProSe: Programming Tool for Sensor Networks

In this chapter, we propose *ProSe*, a programming tool for sensor networks that will allow the designers to concisely specify sensor network protocols. ProSe is based on the theoretical foundation on computational model in sensor networks established in Chapter 3. ProSe enables the designer to (1) specify sensor network protocols and macroprogramming primitives in simple, abstract models considered in distributed computing literature (cf. Chapter 2 for a brief introduction to these models), (2) transform the programs into WAC model while preserving properties such as fault-tolerance and self-stabilization of the original programs, (3) reuse existing fault-tolerance/self-stabilizing from the literature in the context of sensor networks, and (4) automatically generate and deploy code. An advantage of ProSe is that it will facilitate the designer to use existing algorithms for automating the addition of fault-tolerance to existing programs. Moreover, since abstract models are used to specify protocols, ProSe allows the designer to gain assurance about the programs deployed in the network using tools such as model checkers.

The rest of the chapter is organized as follows. In Section 5.1, we present an

overview of ProSe. Then, in Section 5.2, we discuss the features of ProSe that allow one to rapidly prototype fault-tolerant/self-stabilizing sensor network protocols. And, in Section 5.3, we present additional features available with ProSe. In Section 5.4, we compare and contrast ProSe with the related work. We present case studies on rapid prototyping of sensor network protocols with ProSe in Chapter 6.

5.1 ProSe: Overview

In this section, we present an overview of ProSe. Specifically, we discuss (1) the programming architecture of ProSe, (2) the input to ProSe, (3) the output of ProSe, and (4) how ProSe generates the nesC/TinyOS program.

5.1.1 Programming Architecture of ProSe

The programming architecture of ProSe is shown in Figure 5.1. Programs are specified in ProSe using guarded commands [39] (cf. Chapter 2 for more details about the structure of programs). ProSe transforms the input guarded commands program into a program in WAC model (if the input program is specified in shared-memory or read/write model), as discussed in Chapter 3. Once the input guarded commands program is transformed into WAC model, ProSe generates the corresponding nesC code (targeted for TinyOS). Furthermore, ProSe *wires* the generated code with a MAC layer (e.g., [60, 80, 81, 134]) to implement the write-all action in the WAC model. The MAC layer provides an interface for broadcasting (i.e., writing all neighbors) and receiving WAC messages.

Now, the designer can use the TinyOS platform to build the binary of the nesC code. This binary image can then be disseminated across the network using a network programming service (e.g., [63, 79, 86]). Thus, sensor network protocols can be prototyped and deployed across the network.

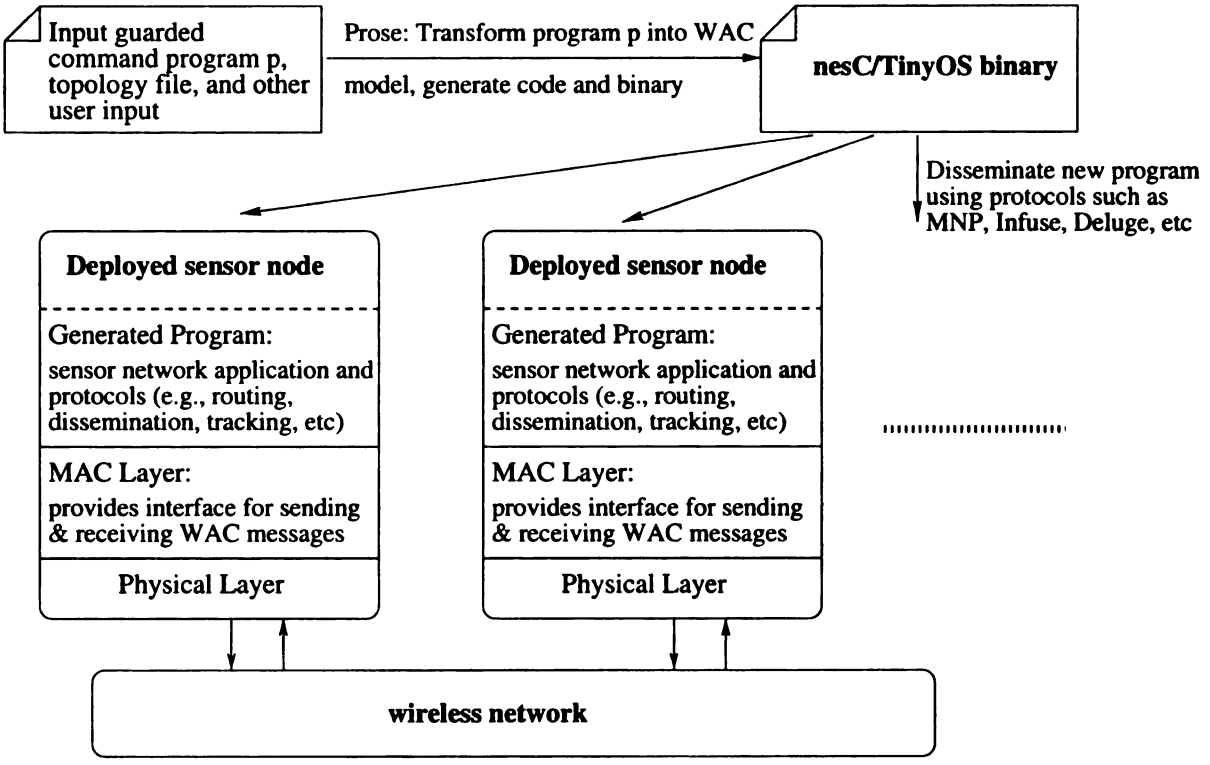


Figure 5.1: ProSe programming architecture

5.1.2 ProSe: Input

The input to ProSe consists of the guarded commands program in shared-memory, read/write, or WAC model, its initial states and (optionally) topology information. For simplicity, we discuss the input guarded command program in shared-memory model.

In the input guarded command program, the designer has to specify whether a variable is *public* or *private*. In shared-memory or read/write model, a sensor can read the public variables of its neighbors. Also, the designer has to identify the sensor to which the variable belongs. For example, if sensor j accesses its local variable x, the designer has to specify it as $x.j$. Now, we discuss the input/output of ProSe in the context of an example.

Input guarded commands program. Consider the *MAX* program, where each

sensor maintains a public variable x. The goal of MAX is to eventually identify the maximum value of this variable across the network. Whenever $x.j$ is less than $x.k$ (i.e., variable x at sensor k), j copies $x.k$ to $x.j$. This action allows j to update $x.j$ whenever its guard holds and, thus, j eventually computes the maximum value of x across the network. In the input file of ProSe, we specify the actions of j as shown in Figure 5.2 (keywords are shown in bold font):

```
1 program max
2 sensor j
3 var public int x.j;
4 begin
5         (x.k > x.j) -> x.j = x.k;
6 end
7 init state x.j = j;
```

Figure 5.2: MAX program in ProSe

Initial state. The designer also specifies zero or more initial states in the program. If the program has zero initial states then ProSe initializes the program variables to arbitrary values. In case of self-stabilizing programs, during execution, the sensors recover to legitimate states. And, if the program contains more than one initial state then ProSe initializes the variables to a randomly selected state from the input. In the above example, $x.j$ is initialized to j (i.e., the ID of the sensor).

Auxiliary variables. ProSe provides abstractions to deal with failure of sensors and presence of Byzantine sensors. To determine whether a neighbor (say, k) is alive or failed, sensor j can just access the public variable $up.k$; if $up.k$ is $TRUE$ (respectively, *FALSE*) then k is alive (respectively, failed). ProSe provides implementation of this variable using heartbeat protocol (e.g., [51]). For example, if j fails to receive update messages (i.e., WAC messages) for a pre-determined time interval from its neighbor k, then j declares k as failed. Thus, designers can use this abstract variable to simplify the design of sensor network protocols. Similarly, ProSe also allows designers to model

Byzantine sensors through abstract variables ($b.k$).

Topology information. ProSe *wires* a component (*NeighborStateM*) that maintains the state information of the neighbors at each sensor, with the generated code. Towards this end, each sensor should identify its neighborhood. ProSe allows the designers to integrate a neighborhood abstraction layer (e.g., [132]) with the generated code. Such an abstraction layer allows a sensor to learn its neighborhood dynamically. Optionally, the designers can specify the static topology of the network as input to ProSe using the *topology file*. This file includes the ID of the base station, size of the network, and the communication topology. Based on the neighborhood information, ProSe configures the MAC layer and NeighborStateM component.

5.1.3 ProSe: Output

ProSe generates the code for the input guarded commands program in nesC as follows. As mentioned in Chapter 3, the read action in shared-memory or read/write model is simulated in WAC model using the write-all action. Towards this end, each sensor maintains a copy vector for each public variable of its neighbor (in NeighborStateM, the module that implements NeighborState interface to get/set public variables of the neighbors). This copy vector captures the value of the corresponding variable at its neighbors. The number of elements in this vector is determined using the information on the neighborhood of each sensor.

The actions of the input program are executed whenever a timer fires. Once the sensor executes each action for which the corresponding guard is enabled, it marshals all the public variables as a message *wacMsg* and schedules it for transmission (broadcast). Depending on the transformation algorithm and the MAC layer selected by the user, it configures when the timer fires and how *wacMsg* is transmitted. In case of a TDMA based transformation (e.g., [81]), ProSe configures the timer to fire in every TDMA slot assigned to the sensor and broadcasts *wacMsg* using a TDMA

service (e.g., [60, 80, 81]). In case of a CSMA based transformation (e.g., [58]), it configures the timer to fire in a random interval whenever the sensor receives a message containing values of public variables at the sender. And, it uses a CSMA service (e.g., [134]) to broadcast *wacMsg*.

ProSe also generates code to (1) initialize all the program variables, (2) configure network services (e.g., TDMA, CSMA), and (3) configure and start middleware services (e.g., Timer).

5.1.4 Execution of ProSe

In this section, we discuss (1) how the input guarded commands are translated into nesC code and (2) how the sensors maintain the state of their neighbors.

ProSe generates five files: a *configuration* file, an *interface* file, 2 *module* files (one for implementing the actions of the input program and another for maintaining the state of the neighbors of a sensor), and a *makefile*. Figure 5.3 shows how ProSe generates different files from the input program and topology information; these files are required to generate the TinyOS binary.

Configuration file. Configurations wire components together, connecting interfaces used by components to interfaces provided by others [47]. ProSe generates *pC.nc*, given the input guarded commands program p. Specifically, *pC.nc* wires the module file, *pM.nc*, NeighborStateM.nc (i.e., the module that implements the interface NeighborState), network services (e.g., TDMA, CSMA, etc), and other interfaces required by the module.

Interface and module files. Modules provide the application code and implement one or more interfaces [47]. ProSe generates: (1) the implementation file *pM.nc*, given the input guarded command program p and (2) NeighborStateM.nc, as follows (cf.

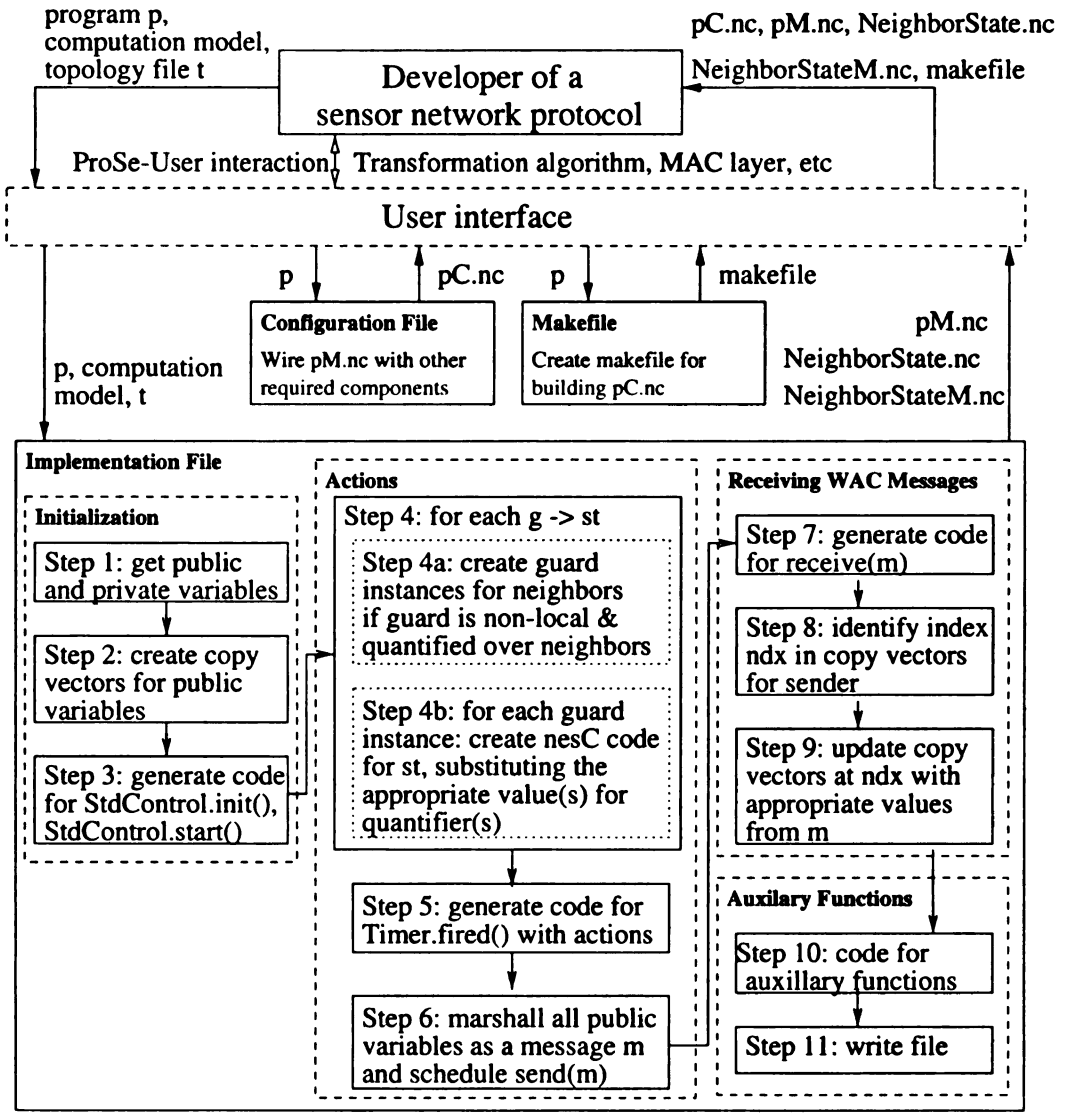


Figure 5.3: Execution sequence of ProSe

Figure 5.3).

Steps 1-3: Initialization. First, ProSe identifies the public and private variables of the input program. Next, for each public variable, in NeighborStateM.nc, it generates a copy vector (containing entries for all the neighbors of a sensor). ProSe generates the interface NeighborState.nc that provides functions to get/set the public variables of the input program. Subsequently, in NeighborStateM.nc, it generates the code for initializing these variables. NeighborStateM.nc implements the interface NeighborState.nc.

In pM.nc, ProSe generates code to (1) initialize components (e.g., TDMA, CSMA, Timer, NeighborStateM, etc) and (2) start network and middleware services (e.g., TDMA, CSMA, Timer). In case of a TDMA based transformation, in pM.nc, ProSe sets the timer to fire during the TDMA slots assigned to the sensor. A sensor executes each action for which the corresponding guard is enabled whenever this timer fires.

For example, Figure 5.4 show the generated program segment for MAX to initialize program variables (including the copy vectors in NeighborStateM) and initialize/start TDMA and Timer services.

```
1 command result_t StdControl.init() {
2  sensorID = TOS_LOCAL_ADDRESS;
3  x.j = sensorID;
4  call NeighborState.init();
5  call SSTDMA.setParameters(sensorID, BASE_ID, SIZE_X,
6         INTERFERENCE_RANGE, BROADCAST);
7  return SUCCESS;
8 }
9
10 command result_t StdControl.start() {
11  call SSTDMA.start();
12  call Timer.start(TIMER_REPEAT, SSTDMA_SLOT);
13  return SUCCESS;
14 }
```

Figure 5.4: Generated program segment in nesC for MAX: Initialization

Steps 4-6: Actions. ProSe generates the nesC code for the actions specified in the input program in *Timer.fired()* event. For each action $g \longrightarrow st$, first, it determines if the guard g has any non-local variables (e.g., process j accessing public variable $x.k$ of process k in the *MAX* program). If g contains no non-local variables, it generates the corresponding nesC code of the form *if(g){st;}*.

If g has non-local variables, ProSe proceeds as follows. For each non-local variable, it generates guard instances for each neighbor. For example, in *MAX*, it generates:

- *s_i:* *copy_x_i = call NeighborState.get_x(i);*

g'_i: copy_x_i > x_j

(quantified over neighbors i of j)

If the index to a non-local variable is a local variable, ProSe generates a single guard instance for the neighbor identified by the index. For example, if g is of the form $x.(p.j) > x.j$, it generates:

- *s_ndx:* *ndx = call NeighborState.getCopyVectorIndex(p_j);*

  copy_x_ndx = call NeighborState.get_x(ndx);

  g'_ndx: copy_x_ndx > x_j

  (where *getCopyVectorIndex(p_j)* returns the index of *p_j* to the copy vectors)

Thus, the guard g in the input program is transformed into a set of type $s; g'$. Now, for each $s; g'$ of g, ProSe generates the nesC code of the form s; *if(g'){st;}* (cf. Figure 5.5 for an example).

Once the code for all actions are generated, ProSe generates code for implementing the write-all action. Towards this end, first, it marshals all public variables into a message. Then, it uses the MAC layer selected by the user to schedule transmission of the message.

The nesC code segment for *Timer.fired()* event of MAX obtained using the TDMA based transformation from [81] and SS-TDMA [80], is shown in Figure 5.5. (The function *getNoOfNbrs()* in NeighborState returns the number of neighbors of the given sensor.)

Steps 7-9: Receiving WAC messages. ProSe generates code for updating NeighborStateM whenever it receives a message. Towards this end, ProSe generates code for determining the sender of the message and the index of the sender to the copy vectors (using *getCopyVectorIndex(sender)*). Once the index is identified, the values of the public variables of the sender are updated in the corresponding copy vectors (cf. Figure 5.6 for more details). Thus, each sensor maintains up-to-date values of

```
1 event result_t Timer.fired() {
2  uint8_t i, nbrs, msgSizeInBytes;
3  int copy_x_i;
4  if(sendDone == TRUE) {
5     nbrs = call NeighborState.getNoOfNbrs();
6     for(i = 0; i < nbrs; i++) {
7      copy_x_i = call NeighborState.get_x(i);
8      if((copy_x_i > x_j)) x_j = copy_x_i;
9     }
10    wacMsg.data[0] = x_j;
11    wacMsg.data[1] = (x_j >> 8);
12    sendDone = FALSE;
13    msgSizeInBytes = 2;
14    call SSTDMA.send(msgSizeInBytes, &wacMsg);
15 }
16 return SUCCESS;
17}
```

Figure 5.5: Generated program segment in nesC for MAX: Timer fired event

the public variables of the neighbors. Furthermore, in case of CSMA based transformation, ProSe sets the timer to fire in a random interval whenever it receives a WAC message. A sensor executes each action for which the corresponding guard is enabled whenever this timer fires.

For example, Figure 5.6 shows the how the generated program updates NeighborStateM.nc with the current state information of the neighbors for the MAX program.

Steps 10-11: Auxiliary functions. Finally, ProSe adds the code for all auxiliary interface functions of NeighborState (e.g., *getCopyVectorIndex(neighbor)*, *getNoOfNbrs()*, etc) in NeighborStateM.

Makefile. To facilitate quick compilation and deployment, ProSe generates the makefile for building the TinyOS binary of the generated code.

Thus, ProSe provides the designer with the different files required for building and deploying the new binary image.

```
1event SSTDMA_MsgPtr SSTDMA.receive(uint8_t size, SSTDMA_MsgPtr data) {
2 uint8_t type, nbr_index;
3 uint16_t senderID, *temp;
4 type = (uint8_t) data->type;
5 if(type == SSTDMA_MESSAGE) {
6   senderID = (uint16_t) data->senderID:
7   nbr_index = call NeighborState.getCopyVectorIndex(senderID);
8   temp = (uint16_t *) &data->data[0];
9   call NeighborState.set_x(nbr_index, *temp);
10 }
11 return data;
12}
```

Figure 5.6: Generated program segment in nesC for MAX: Receive event

5.2 Generating Fault-Tolerant/Self-Stabilizing Programs with ProSe

In this section, we discuss the properties and features of ProSe that enable the designers to prototype fault-tolerant/self-stabilizing sensor network protocols easily.

Preserving fault-tolerance/self-stabilization properties. Properties such as fault-tolerance and self-stabilization are important in sensor networks. Specifically, since sensor networks are deployed in large numbers and in inaccessible fields, the network should be able to *self-stabilize* [38, 40] after faults (e.g., message corruption, message losses, synchronization errors, etc) stopped occurring. Towards this end, ProSe preserves the fault-tolerance and self-stabilization properties of the program in WAC model. Additionally, if the algorithm used in transforming a program (in read/write or shared-memory model) into a program in WAC model preserves the properties of interest then ProSe also preserves such properties. The transformation algorithms presented in Chapter 3 and those in [58, 81] preserve the fault-tolerance and self-stabilization properties in the transformed programs. Since we have implemented these algorithms in ProSe, ProSe preserves fault-tolerance/self-stabilization

properties of the input programs in read/write or shared-memory model.

Dealing with faults in sensor networks. The normal operation of a typical sensor network is affected by (1) failure of sensors, (2) state corruption, and (3) message loss. Regarding failure of sensors, ProSe provides abstractions to the designer of a sensor network protocol. The designer can abstract sensor failures using the *up* variables (cf. Section 5.1.2 for more details). Similarly, ProSe can model intermittent faults, sensors sleeping to save energy, or Byzantine (malicious) sensors. Regarding state corruption, ProSe permits arbitrary initial states. This allows the designer to model systems that are perturbed to an arbitrary state. When used in the context of a self-stabilization preserving transformation (e.g., [58, 81]), this feature enables the design of self-stabilizing protocols for sensor networks. Finally, regarding message loss, ProSe allows the designer to provide probability of transmission on any given link. Thus, ProSe succinctly models failure of sensors, state corruption and message loss.

5.3 Additional Features in ProSe

In this section, we discuss the additional features of ProSe. Specifically, we discuss how the designer can (1) specify priorities for each action, (2) combine actions specified in WAC model with actions in other abstract models, and (3) invoke components in guarded commands.

5.3.1 Priorities of Actions

In sensor networks, it is expected that the frequency of execution of different actions may be different. For example, consider a routing protocol. The actions in a routing protocol can be classified as either *heartbeat* actions or *protocol* actions. Heartbeat actions are responsible for checking the status of the neighbors and protocol actions

are responsible for construction and maintenance of the routing structure. During execution, these two classes of actions may have different priorities. In other words, the frequency of execution of heartbeat actions may be different from protocol actions. Typically, in a network where failures are common, heartbeat actions have higher priority. To represent such actions, in ProSe, we allow the designer to specify priorities for each action. Priority characterizes the frequency with which an action would be executed. Priority of an action is listed along with the guarded command corresponding to the action.

5.3.2 WAC Actions

During the design of sensor networks, it should be possible for the designer to leverage on existing programs in WAC model. For example, consider the design of a routing protocol. Suppose a link estimation program is available in WAC model. It should be possible for the designer to combine this program with the routing program he/she is designing. Towards this end, in ProSe, we allow the designer to specify actions in WAC model together with other actions (in shared-memory or read/write model). In order to specify an action in WAC model, a note is added in the guarded command corresponding to the action. When presented with such a hybrid program, ProSe transforms the actions in shared-memory or read/write model into WAC model. The actions in WAC model (in the hybrid program) are implemented as is. Once the hybrid program is transformed into a program in WAC model, the corresponding nesC code is generated. Thus, the designer can leverage on existing programs in shared-memory, read/write, or WAC models.

5.3.3 *Local* Component Invocations in Guarded Commands

Since ProSe allows the designers to specify programs in guarded commands format, it makes protocol design highly intuitive and concise. However, it is not always desirable

to use guarded commands to specify protocols. For example, consider the design of a routing protocol for sensor networks, where the sensors maintain a spanning tree rooted at the base station. In this program, whenever the parent of a sensor fails, it chooses one of its active neighbors for which the link quality is greater than a certain threshold, as its parent. Towards this end, the sensor has to compute the link quality of each of its neighbors. Specifying this action in guarded commands is difficult. Moreover, nesC/TinyOS components may exist that provide the desired functionality.

To simplify the design of sensor network protocols, ProSe allows component invocations in guarded commands. In the design of routing protocol, in order to find a neighbor that has a better link quality, the designer can invoke the component *LinkEstimator* to compute the quality estimate of a given link. Thus, parent update action in the routing protocol can be specified in guarded commands as shown in Figure 5.7.

```
1// current parent (p.j) has failed and j-k link quality > threshold
2(up.(p.j) == FALSE) && (up.k == TRUE) &&
3                (LinkEstimator.getQuality(k) > LINK_THRESHOLD)
4  -> p.j = k;
5     quality.j = LinkEstimator.getQuality(k);
```

Figure 5.7: Invoking components in guarded commands program

In the above action, the *getQuality(k)* method of LinkEstimator component returns the quality of the link $j - k$. This component may need certain variables to compute the quality estimate. For example, it may need counters that maintain the number of messages successfully transmitted over each link. The action by which the counters are updated would be specified in guarded commands. The variables used in the guarded commands program and the copies of the public variables of the neighbors (maintained in NeighborStateM) are made available to the invoked component.

The designer has to implement *LinkEstimator* in nesC/TinyOS platform. This

component, however, uses only local data (i.e., it uses NeighborStateM). ProSe generates the code for NeighborStateM. And, it wires the component implemented by the designer with the generated code.

5.4 Related Work

Related work that deals with programming abstractions include [91, 105, 131, 133], tools for programming sensor networks include [2, 21, 44, 53, 55, 88, 96, 104, 132], and network programming protocols include [63, 79, 86, 123].

Programming abstractions. In [91], a state centric approach is proposed that captures algorithms such as sensor fusion, signal processing and control. This model views the network as a distributed platform for in-network processing. Furthermore, in this model, the abstraction of *collaboration groups* hides the designer from issues such as communication protocols, event handling, etc. In [105, 131], *macroprogramming* primitives that abstract communication, data sharing and gathering operations are proposed. These primitives are exposed in a high-level language. However, these primitives are application-specific (e.g., *abstract regions* for tracking and gathering [131] and *region streams* for aggregation [105]). And, in [133], *semantic services* programming model is proposed where each service provides semantic interpretation of the raw sensor data or data provided by other semantic services. In this model, users only specify the end goal on what semantic data to collect. Thus, users make less low-level decisions on which operations to run or which data to run them over.

While [91, 105, 131, 133] are designed for simplifying programming application services such as tracking, aggregation, etc, ProSe is designed to simplify programming both network services (e.g., routing, clustering, leader election, distributed reset, etc) and application services. Furthermore, ProSe allows the designer to evaluate existing algorithms in the context of sensor networks. Moreover, since the programs are writ-

ten in abstract models considered in distributed systems, ProSe permits the designer to verify the correctness of the programs as well as to manipulate the programs to meet new properties.

Programming tools. Techniques like virtual machine (e.g., *Maté* [88]), middleware (e.g., *EnviroTrack* [2]), library (e.g., *SNACK* [53], *TASK* [21]), database (e.g., *TinyDB* [96]), and mobile agent (e.g., [44]) are proposed for simplifying programming sensor network applications. However, these solutions are (i) application-specific, and/or (ii) restrict the designer to what is available in the virtual machine, middleware, library, or network. By contrast, ProSe provides a simple abstraction while allowing the designer to specify wide variety of protocols.

In [55], macroprogramming model, called *Kairos*, that hides the details of code-generation and instantiation, data management, and control is proposed. Kairos provides the programmer with three abstractions; (i) node-level abstraction that allows the programmer to manipulate nodes and list of nodes, (ii) one-hop neighbor list abstraction for performing operations on the neighbor list, and (iii) remote data access that allows a sensor to read the named sensors. While ProSe provides similar abstractions, it differs from Kairos. Specifically, unlike Kairos, ProSe hides low-level details such as message collisions, corruption, sensor failures, etc. Moreover, ProSe does not require any runtime support in the generated sensor network binary. Additionally, unlike Kairos, ProSe enables reuse of existing algorithms while preserving properties such as self-stabilization of the input program.

Network programming protocols. Complementary to the work on programming abstractions and programming tools, in [63, 79, 86, 123], protocols for reprogramming large scale networks *in-place* are proposed. Such protocols are orthogonal to the design objective of ProSe but are useful in disseminating the code generated by ProSe across the network in a reliable manner.

5.5 Chapter Summary

In this chapter, we presented a tool, *ProSe*, for programming sensor networks. ProSe allows one to (1) hide low-level details (e.g., message collisions, message losses, sensor failures, synchronization, etc) of sensor networks from the designer, thereby, enabling the designer to focus only on the functionality of the protocol, (2) reuse existing fault-tolerant/self-stabilizing algorithms considered in distributed systems and traditional networking in the context of sensor networks, and (3) automate code generation and deployment. Since the programs are specified in abstract models considered in distributed systems literature, ProSe allows the designer to verify the behavior of the programs. Thus, ProSe allows the designer to gain assurance about the programs deployed in a sensor network. Moreover, the abstract models let the designer to manipulate the programs to meet new properties. In particular, approaches in [17, 49, 75, 103] can be used to modify the programs to meet new properties (including fault-tolerance properties).

Chapter 6

Case Studies on Rapid Prototyping with ProSe

In this chapter, we illustrate ProSe to generate network and application services for sensor networks. Specifically, in Section 6.1, we specify a routing tree maintenance program in shared-memory model and evaluate the performance of the program generated by ProSe. And, in Section 6.2, we specify an intruder-interceptor program, an application service for tracking intruders in the network, in shared-memory model and evaluate the performance of the program generated by ProSe. Finally, in Section 6.3, we specify a power management program and evaluate the performance of program generated by ProSe. In addition, in Section 6.4, we discuss the impact of the proposed tool.

6.1 Network-Level Service: Routing Tree Maintenance

In this section, we consider the routing program proposed in [25]. This program is similar to the routing program we used to illustrate our transformation algorithms

in Chapter 3 (cf. Section 3.3). This program is a variation of the balanced routing program [27].

6.1.1 Description of Routing Tree Maintenance Program (RTMP)

We specify RTMP in shared-memory model in ProSe as shown in Figure 6.1. In this program, sensors are arranged in a logical grid. The program constructs a spanning tree with the base station as the root. The base station is located at $\langle 0, 0 \rangle$ of the logical grid. Each sensor classifies its neighbors as *high* or *low* neighbors depending on their (logical) distance to the base station. Also, each sensor maintains a variable, called *inversion count*. The inversion count of the base station is 0. If a sensor chooses one of its low neighbors as its parent, then it sets its inversion count to that of its parent. Otherwise, it sets its inversion count to inversion count of its parent + 1. Furthermore, to deal with the problem of cycles, if the inversion count exceeds a certain threshold ($CMAX$), the sensor removes itself from the tree.

In this program, each sensor (say, j) maintains three *public* variables: (i) $inv.j$, the inversion count of j, (ii) $dist.j$, the (logical) distance of j to the base station, and (iii) $up.j$, the status variable for j (indicates whether j has failed or not). ProSe provides implementation of $up.j$ using heartbeat protocol, as discussed in Section 5.1.2.

Whenever j finds a low/high neighbor that provides a better path (in terms of inversion count) to the base station, it updates its *private* variable, $p.j$, the parent of j, and inversion count $inv.j$. Whenever a sensor fails or inversion count is not consistent with its parent, the sensor sets its parent to $NULL$ and its inversion count to $CMAX$ (i.e., it removes itself from the routing tree). Subsequently, when it finds a neighbor with a better inversion count value, it rejoins the tree.

```
1  program RoutingTreeMaintenance
2  sensor j;
3  const int CMAX;
4  var
5    public int inv.j, dist.j;
6    public boolean up.j;
7    private int p.j;
8  begin
9 (dist.k < dist.j) && (up.k == TRUE) &&
10         (inv.k < CMAX) && (inv.k < inv.j)
11    -> p.j = k; inv.j = inv.k;
12 | (dist.k < dist.j) && (up.k == TRUE) &&
13         (inv.k+1 < CMAX) && (inv.k+1 < inv.j)
14    -> p.j =k; inv.j = inv.k+1;
15 | (p.j != NULL) &&
16   ((up.(p.j) == FALSE) ||
17   (inv.(p.j) >= CMAX) ||
18   ((dist.(p.j) < dist.j) && (inv.j != inv.(p.j))) ||
19   ((dist.(p.j) > dist.j) && (inv.j != inv.(p.j)+1)))
20    -> p.j = NULL; inv.j = CMAX;
21 | (p.j == NULL) && (inv.j < CMAX)
22    -> inv.j = CMAX;
23 end
```

Figure 6.1: Routing tree maintenance program in shared-memory model

6.1.2 Transformation of RTMP

We use ProSe to generate the corresponding nesC/TinyOS implementation and subsequently use TinyOS platform to build the binary image. We use the TDMA based transformation from Chapter 3 to transform the program into WAC model and generate the nesC/TinyOS code. We integrate SS-TDMA [80] with the generated program to implement the write-all action. The memory footprint of the generated code (*routingM* component) is shown in Table 6.1.2. As observed from the table, the memory requirements of the generated code is significantly small.

Table 6.1: Memory footprint of the generated routing tree maintenance program

	Program ROM (in bytes)	RAM (in bytes)
routingM + NeighborStateM	42	106
SS-TDMA	108	586
other components (Timer, FramerM, LedsC, etc)	15934	404

6.1.3 Experimental Results of Generated RTMP Program

We deploy the code generated by ProSe on 3x3 and 5x5 XSM [41] networks in a classroom setting, where the inter-sensor separation is 8 ft and the base station is located at top-left corner of the grid. Each sensor executes the write-all action of the routing tree maintenance program once in every 1.95 seconds. We keep this value intentionally high in order to reduce the frequency of execution of actions. After the initial routing tree is constructed, we fail some sensors (simultaneously) to determine how the sensors converge to a new routing tree. We measure the latency for constructing the initial routing tree, convergence time (i.e., the time required to converge to new tree after failure of some sensors), energy spent (in terms of number of messages sent, number of messages received and active radio time), and latency in transmitting a message from middle of the network (and the farthest sensor) to the base station. The results of these experiments are presented in Table 6.2.

Figure 6.2 shows the initial tree and the converged tree (after sensors fail) on a 5x5 XSM network, where one of the sensors fail at the start of the experiment. After the initial tree is constructed (cf. Figure 6.2(a)), we fail sensors 3, 6, 8, 10, 17, 18, and 20 simultaneously. The active sensors then converge to the new routing tree (cf. Figure 6.2(b)) within 21 seconds.

We note that the performance of the program generated by ProSe is within the acceptable performance guidelines of a typical sensor network application (e.g., [9, 10]).

Table 6.2: Experimental results of the code generated by ProSe in presence of 2 (respectively, 7) failed sensors in case of 3x3 (respectively, 5x5) network

(a) Initial latency and convergence time

Network size	Latency for initial tree construction (no failures)	Convergence time (after failures)
3x3	8 s	4 s
5x5	20 s	21 s

(b) Energy consumption in maintaining the routing tree

Network size	Energy (every 10 s per sensor)		
	Messages sent	Messages received	Active radio time
3x3	5	13.33	550 ms
5x5	5	15.83	625 ms

(c) Routing cost

Network size	Routing cost	
	Sender: middle of network	Sender: farthest sensor
3x3	1.974 s	1.998 s
5x5	1.998 s	2.046 s

To illustrate this, we compare the convergence time and the number of messages transmitted in the code generated by ProSe with that of MintRoute [135], a reliable multi-hop routing program designed *manually* for sensor networks. We note that the purpose of this comparison is to show that program generated by ProSe is competitive to manually designed protocols. Also, it shows that the generated program can be used in practice. (We do not compare other parameters due to lack of *implementation data* about MintRoute.) The convergence time of the routing tree maintenance program generated by ProSe is better than that of MintRoute (4 s with the code generated by ProSe vs. 6 s with MintRoute for a 3x3 network). However, the average number of message transmissions with MintRoute is less than that of the routing tree maintenance program generated by ProSe (1.25 messages with MintRoute vs. 5 messages with the code generated by ProSe every 10 s per sensor).

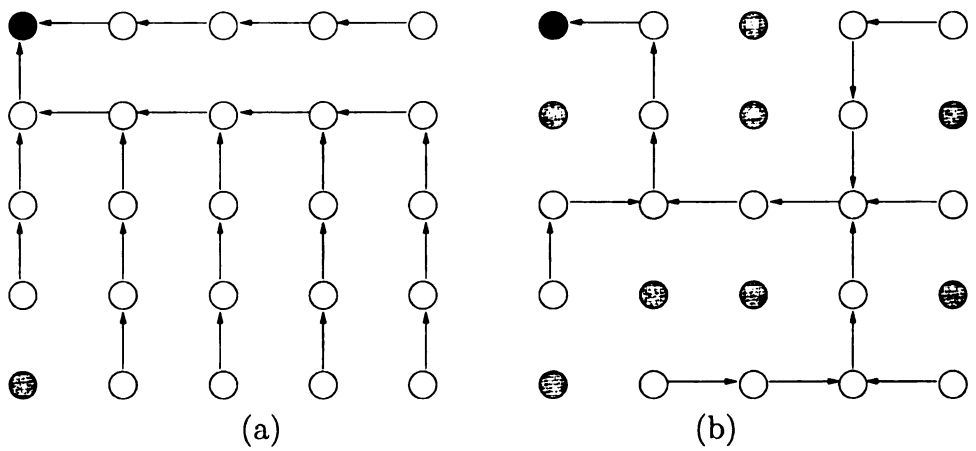


Figure 6.2: Routing tree construction and maintenance on a 5x5 XSM network with base station (filled circle) at the top-left corner. (a) initial tree and (b) converged tree after failure of some sensors (shown in gray circles)

6.1.4 Simulation Results of Generated RTMP Program

We also simulated the generated routing tree maintenance program (ProSe-RTMP) and MintRoute [135] using TOSSIM [89], a discrete event simulator for TinyOS sensor networks. In our simulations, the base station is located at $\langle 0, 0 \rangle$ (i.e., sensor 0) and the inter-sensor separation is 10 ft. In the absence of any interference, we have observed that probability of successful communication is more than 98% among the neighbors. However, random channel errors can cause the reliability to go down. Hence, we choose conservative estimates of 95% and 90% link reliability (that correspond to the analysis in [46, 141]) in our simulations.

Similar to the experimental setup, in ProSe-RTMP, each sensor executes the write-all action of the program once in every 2 seconds. And, in MintRoute, the sensors exchange routing information every 2 seconds. Once the initial routing tree is constructed, we simultaneously fail some sensors (up to 100 sensors in case of 20x20 network) and measure the convergence time. The simulation results of the generated routing program are shown in Figure 6.3. (Please note that the images in this

dissertation are presented in color.)

Figures 6.3(a)-(b) show the results for the case where the link reliability is 95%. The initial latency to construct the routing tree for ProSe-RTMP and MintRoute are similar. MintRoute maintains link estimates of the active links of a sensor and updates the estimate periodically. As a result, the radio is active all the time. By contrast, with ProSe-RTMP, the active radio time of the sensors during this period is significantly less (i.e., around 20% of the initial latency). Thus, ProSe-RTMP provides an energy-efficient tree maintenance service.

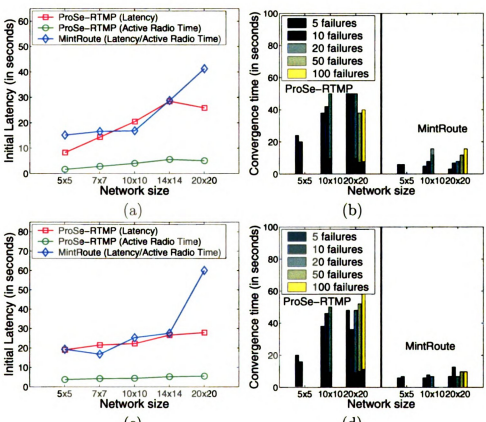


Figure 6.3: Simulations results of the generated program. With 95% link reliability: (a) initial latency to construct the routing tree and (b) convergence time after failure of some sensors. And, with 90% link reliability: (c) initial latency and (d) convergence time. Note that the black bars in the convergence time graph shows the active radio time during the convergence period.

Figure 6.3(b) presents the convergence time of the protocols in the presence of

failed sensors. As observed from Figure 6.3(b), MintRoute converges to a new routing tree quickly. By contrast, ProSe-RTMP converges within 30-50 seconds. We note that this behavior is *not* because of prototyping with ProSe. Rather, it is because of the nature of the original protocol specified with ProSe. More specifically, MintRoute is *pessimistic* in nature, i.e., it maintains a moving average of link estimates of all active links of a sensor all the time. Hence, when sensors fail, it converges to a new tree quickly. By contrast, the routing tree maintenance program specified in Figure 6.1 is *optimistic* in nature. In other words, whenever a sensor chooses one of its neighbors as its parent, it does not change its parent unless the parent has failed or the tree is corrupted. As a result, when sensors fail, it takes sometime for the protocol to discover such sensors and update the tree. On the other hand, the active radio time during recovery is small with ProSe-RTMP (i.e., in the order of the convergence time with MintRoute).

Figures 6.3(c)-(d) show similar results for the case where the link reliability is 90%. In this case, the message loss is extensive and, hence, the initial latency increases as the network size increases. Also, the convergence time increases as the number of failed sensors increases. Thus, the results shown in Figure 6.3 demonstrate the potential of ProSe to generate competitive network-level services for sensor networks.

6.2 Application-Level Service: Pursuer-Evader Tracking

Sensor networks are often used in intruder-interception games, where the sensors guide the pursuer (e.g., a robot, a soldier, etc) to track and intercept the evader (e.g., intruder, hostile vehicle, etc). In this section, we demonstrate the potential of ProSe to program such application level services. We consider the pursuer-evader tracking program proposed in [37].

6.2.1 Description of Tracking Program

We specify the evader-centric program from [37] in shared-memory model as shown in Figure 6.4. In this program, sensors maintain a tracking structure rooted at the evader. The pursuer follows this tracking structure to intercept the evader. Whenever the pursuer arrives at a sensor (say, k), it consults k to determine its next move. Specifically, the pursuer moves to the parent of k. And, since the pursuer is faster than the evader, it eventually intercepts the evader.

```
1  program PursuerEvaderTracking
2  sensor j;
3  var
4    public int dist2Evader.j, detectTimeStamp.j, p.j;
5    private boolean isEvaderHere.j;
6  begin
7  (isEvaderHere.j == TRUE)
8    -> p.j = j; dist2Evader.j = 0;
9       detectTimeStamp.j = TIME;
10 | (detectTimeStamp.k > detectTimeStamp.j) ||
11   ((detectTimeStamp.k == detectTimeStamp.j) &&
12         (dist2Evader.k+1 < dist2Evader.j))
13   -> p.j = k;
14      detectTimeStamp.j = detectTimeStamp.k;
15      dist2Evader.j = dist2Evader.k+1;
16 end
```

Figure 6.4: Pursuer-evader tracking program in shared-memory model

In this program, each sensor (say, j) maintains three public variables: (i) *dist2Evader.j*, distance to the root of the tracking structure, (ii) *detectTimeStamp.j*, the timestamp that j knows when the evader was detected at the root, and (iii) $p.j$, the parent of j. Whenever j detects the evader, it sets *detectTimeStamp.j* to its current clock value (using the $TIME$ keyword), *dist2Evader.j* to 0 and $p.j$ to itself. If it finds one of its neighbors (say, k) has the latest detection timestamp, then it updates its public variables accordingly and sets its $p.j$ to k. In Figure 6.4, for simplicity, we do not show the actions of the pursuer. Since the pursuer can listen to the messages

transmitted by the sensors, whenever the pursuer is near j, it reads the public variable $p.j$ and moves to $p.j$.

6.2.2 Transformation of Tracking Program

We use the TDMA based transformation from [81] to transform the program into WAC model. We integrate SS-TDMA [80] with the generated program to implement the write-all action. In this program, we need to *wire* components that detect whether the evader is present near a sensor. For example, if the goal of the application is to intercept vehicles then we need to integrate components that can signal whether a vehicle is present or moving near a sensor (e.g., magnetometer components, accelerometer components). Based on this signal, the variable *isEvaderHere.j* at sensor j is either set to *TRUE* or *FALSE*. We assume that the intruder detection service is independent of the tracking service.

The memory footprint of the generated code (*trackingM* component) is shown in Table 6.3. The memory requirements of the generated code is significantly small.

Table 6.3: Memory footprint of the generated tracking program

	Program ROM (in bytes)	RAM (in bytes)
trackingM + NeighborStateM	N/A*	91
SS-TDMA	108	586
other components (Timer, FramerM, LedsC, etc)	14920	404

* The perl script `tinyos-1.x/contrib/SystemC/module_memory_usage` used to obtain the breakdown of program ROM and RAM used by various components only provides the RAM usage data for trackingM; it does not report the ROM usage for trackingM. We expect this value to be small.

6.2.3 Simulation Results of Generated Tracking Program

We simulated the generated program using TOSSIM [89]. The inter-sensor separation is 10 ft and the TDMA period in SS-TDMA is 0.78 seconds. Similar to Section 6.1,

we choose the link reliability to be 95% and 90%.

In our simulations, we use a *virtual* pursuer and a *virtual* evader. Specifically, we model the actions of pursuer and evader using global variables. In our simulations, evader randomly moves in the network. We set the variable $isEvaderHere.j$ at j to $TRUE$ or $FALSE$ depending on where the evader is currently located. Whenever the pursuer is at sensor j, it reads the public variable $p.j$ of j and moves to $p.j$. The ratio of the speed of pursuer movement to evader movement is 1:2. We did two sets of simulations; one, the pursuer is initially located at $\langle 0, 0 \rangle$ (i.e., sensor 0) and two, the pursuer is initially near the center of the network. In both these scenarios, the evader is initially located at the corner (i.e., at $\langle N-1, N-1 \rangle$ on a NxN grid). The simulation results of the generated tracking program are shown in Figure 6.5. (We note that the purpose of these simulations is to demonstrate that the program generated by ProSe is competitive to manually designed protocols and can be used in practice.)

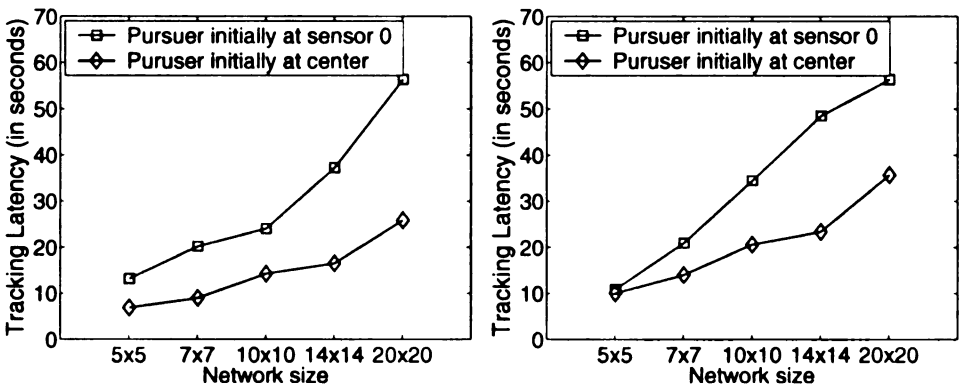


Figure 6.5: Tracking latency of the generated program: (a) with 95% link reliability and (b) with 90% link reliability

Figure 6.5(a) shows the tracking latency for the pursuer to intercept the evader, where the link reliability is 95%. The latency increases as the network size increases. Also, as expected, the latency when the pursuer is initially near the center is signif-

icantly less than the case where the pursuer is initially at $\langle 0, 0 \rangle$. In addition, during tracking, the active radio time is at most 20% of the time required by the pursuer to intercept the evader. (We observe similar results with link reliability = 90%). Thus, the results shown in Figure 6.5 demonstrate the potential of ProSe to generate competitive application-level services for sensor networks.

6.3 Prototyping Power Management Protocols

Sensor network applications require the network to operate for a long time (usually, several weeks to several months). However, the sensors are typically battery-powered (e.g., Mica [61], XSM [41], Telos [113]) and, hence, they can operate continuously only for a few days. In addition, since the sensors are deployed in large numbers and mostly in inaccessible fields, it is difficult to change the batteries after deployment. Therefore, power management is crucial for extending the lifetime of the network.

To rapidly prototype and quickly evaluate protocols, the designers of existing power management protocols (e.g., [54, 116, 126, 129, 130, 136, 137]) implement their own simulators or model their protocols in specialized simulators (e.g., GloMoSim [139]). However, it is desirable that the designers prototype their protocols in nesC/TinyOS platform as it provides a framework for generating both simulation as well as production code from the same source.

In this section, we model power management protocols with ProSe. Specifically, we consider *pCover* [129], a simple power management protocol that provides partial (but high) sensor coverage of the target field. pCover maintains a certain degree of coverage through sleep-awake scheduling of sensors. By trading little sensor coverage of the field, in [129], the authors show (using C++ discrete event simulator) that pCover substantially improves the network lifetime.

6.3.1 Description of pCover Program

The pCover program written in shared-memory model is shown in Figure 6.6. The basic idea of pCover is that a sensor should turn itself off if and only if its *local coverage* is higher than a certain threshold, called *OnThreshold*. Local coverage of a sensor is the percentage of the sensor's sensing area that is covered by other awake sensors.

In this program, each sensor is in one of 4 states: *probe*, *awake*, *readyoff*, and *sleep*. Each sensor j maintains one public variable $st.j$ that identifies the state of the sensor. In addition, j maintains a copy of the public variables of its neighbors (in NeighborStateM as described in Chapter 5). We discuss the actions of the pCover program shown in Figure 6.6 in detail, next.

Probe state. A sensor in probe state probes the environment, determines whether it should stay awake or go to sleep. After a timeout Y, the sensor computes its local coverage. Note that the designer has to provide the *LocalCoverage* component that returns the local coverage of a sensor. This component acts only on the state information of the neighbors maintained at the sensor. The sensor starts working if its local coverage is lower than the OnThreshold. Otherwise, the sensor switches to sleep state. The timeout Y is used to ensure that when the sensor decides whether it should stay awake or go to sleep, it has the *fresh* state information of its neighbors.

Awake state. A sensor in awake state actively monitors the area within its sensing range. It remains active until the timer reaches the timeout value Z. Since we do not want all awake sensors to timeout at the same time, the timer is initialized to a random value. Once the awake timer expires, the sensor changes its state to *readyoff*.

Readyoff state. In readyoff state, the sensor still provides sensing coverage. However, the neighbors of a readyoff sensor (say j) consider j as a sleeping sensor. In other words, the neighbors of j do not count it when they compute local coverage.

```
1  program pCover
2  sensor j
3  const int X, Y, Z, S, W, OnThreshold, OffThreshold;
4  var
5     public int st.j;
6     private int timer.j;
7  component LocalCoverage;
8  begin
9  (st.j == SLEEP) && (timer.j >= X)
10         -> st.j = PROBE; timer.j = 0;
11 | (st.j == PROBE) && (timer.j >= Y) &&
12                 (LocalCoverage.compute() > OnThreshold)
13         -> st.j = SLEEP; timer.j = 0;
14 | (st.j == PROBE) && (timer.j >= Y) &&
15                 (LocalCoverage.compute() <= OnThreshold)
16         -> st.j = AWAKE; timer.j = Random(0, S);
17 | (st.j == AWAKE) && (timer.j >= Z)
18         -> st.j = READYOFF; timer.j = 0;
19 | (st.j == READYOFF) && (timer.j >= W)
20         -> st.j = AWAKE; timer.j = Random(0, S);
21 | (st.j == READYOFF) && (LocalCoverage.compute() > OffThreshold)
22         -> st.j = SLEEP; timer.j = 0;
23 | ((st.j == SLEEP) && (timer.j <= X)) ||
24   ((st.j == PROBE) && (timer.j <= Y)) ||
25   ((st.j == AWAKE) && (timer.j <= Z)) ||
26   ((st.j == READYOFF) && (timer.j <= W))
27         -> timer.j = timer.j + 1;
28 end
```

Figure 6.6: pCover program in ProSe

If a readyoff sensor finds that its local coverage is greater than *OffThreshold*, it will change its state to sleep. Also, if a sensor is in readyoff state for a long duration, it can switch to awake state. This action allows one to deal with the case where a lot of sensors are in readyoff state although none of them can go to sleep state (due to local coverage being less than OffThreshold).

Sleep state. A sensor in sleep state wakes up every X minutes. When it wakes up, it changes its state to probe and proceeds to execute actions in that state.

6.3.2 Transformation of pCover Program

We use ProSe to generate the nesC/TinyOS implementation of the pCover program and subsequently build the binary image. Towards this end, we use the TDMA based transformation from [81] to transform the program into WAC model and generate the nesC/TinyOS code. We integrate SS-TDMA [80] with the generated program to implement the write-all action. As mentioned in Chapter 5, since the pCover program includes component invocation (LocalCoverage) in the actions, we require the designer of the protocol to implement this component in nesC/TinyOS. We discuss how the designer implements this component and how ProSe integrates it with the generated code, next.

LocalCoverage component. Based on the state information of the neighbors of a sensor (say, j), LocalCoverage component computes the percentage of j's sensing area that is covered by its *awake* neighbors. This component provides a method (*compute()*) that could be invoked in the guarded commands program. This method returns the local coverage of the sensor.

In order to compute the local coverage of the sensor, LocalCoverage requires the state information of the neighbors of the sensor. This information is maintained by NeighborStateM component. Since, ProSe wires NeighborStateM with LocalCoverage

when generating the nesC/TinyOS code for pCover, LocalCoverage component can obtain the state information of the neighbors of the sensor by invoking NeighborState. Note that all accesses to NeighborStateM are local and ProSe is responsible for updating NeighborStateM with *fresh* values. Thus, the designer does not have to deal with programming level challenges of nesC/TinyOS platform and low-level challenges of sensor networks (e.g., communication, collisions, corruption, etc).

Memory footprint. The memory footprint of the generated code (*pCoverM* component) is shown in Table 6.4. The memory requirement of the generated code is small.

Table 6.4: Memory footprint of the generated pCover program

	Program ROM (in bytes)	RAM (in bytes)
pCoverM + NeighborStateM	N/A*	132
LocalCoverageM	322	432
SS-TDMA	108	586
other components (Timer, FramerM, LedsC, etc)	14588	1057

* Again, the perl script `tinyos-1.x/contrib/SystemC/module_memory_usage` used to obtain the breakdown of program ROM and RAM used by various components only provides the RAM usage data for pCoverM; it does not report the ROM usage for pCoverM. We expect this value to be small.

6.3.3 Simulation Results of Generated pCover Program

We evaluate the performance of the generated nesC/TinyOS code for pCover with TOSSIM [89].

Simulation settings. We use the simulation setting similar to [129]. We deploy the sensors in a grid topology over a 100m X 100m area. We set the sensing range r of the sensors to 10m and the radio interference range to 50m. We did two simulations: one with sensor density of 1 node/r^2 and the other with 2 nodes/r^2. Inter-sensor separation and the number of sensors deployed varies depending on the density. With

1 node/r^2 (respectively, 2 nodes/r^2), the inter-sensor separation is 10m (respectively, 7m) and the network size is 10x10 (respectively, 14x14). SS-TDMA [80] sets the TDMA period depending on the number of sensors falling in the interference range of a sensor. With 1 node/r^2 (respectively, 2 nodes/r^2), SS-TDMA sets the period to 50 (respectively, 100) slots, where one time slot = 30ms.

We assume that the lifetime of a sensor is 20 minutes. We choose this value in order to ensure that the simulation completes within a reasonable time. (With density of 2 nodes/r^2, the simulation takes 3 days to complete. Typically, sensors are expected to work continuously for 1000 minutes. Simulating a sensor lifetime of 1000 minutes in TOSSIM, however, would approximately take 150 days to complete.) We simulate the lifetime of each sensor by maintaining a variable and decrementing it appropriately in each time slot.

In all our simulations, we set the timeout values for pCover as follows: X = 1 minute, Y = 2 TDMA slots, Z = 3 minutes, and S = W = 2 minutes. We randomly initialize the state of each sensor. We set OnThreshold and OffThreshold to 0.7 and 0.6. We consider that a network is "dead" when the *global coverage* of the network is less than a certain threshold even if all the alive nodes are working. Global coverage (or degree of coverage) is the percentage of the field that is covered by the working nodes. We define network lifetime as the duration from the beginning of deployment until the network is dead. We use 50% as the threshold in our simulations.

In our simulations, each link in the network has a bit error probability, representing the probability that a bit can be corrupted if it is sent along the link. Bit errors for each link is decided independently (using LossyBuilder, a Java program in TinyOS release) based on empirical loss data gathered from real world [46]. Next, we discuss our simulation results.

Coverage and network lifetime. In Figure 6.7, we show the degree of coverage and number of active sensors over time. In our simulations, we compute the global

coverage for the entire 100m X 100m field and for the inner 80m X 80m field. The border sensors contribute only a part of their sensing range in the field and, hence, we consider the inner 80m X 80m field, where there is no such edge effect. As we can see from Figures 6.7(a) and 6.7(b), the sensors maintain the coverage at approximately the same level. With density = 2 nodes/r^2, initially (i.e., around 3 minutes), we observe a drop in the coverage. This is due to the fact that large number of sensors are initially set to active state (as a result of random initialization) and the number of active sensors fluctuate before converging to an appropriate number that maintains the coverage at a certain level (around 88.4%). Figure 6.7(c) shows the number of active sensors over time. As we can observe from the figure, this number remains at the same level until the point where the coverage starts dropping.

From Figures 6.7(a) and 6.7(b), we observe that the coverage is well maintained until one point, after which, the coverage drops suddenly, and the network dies in a short period. This shows that pCover maintains a balanced energy consumption as all sensors run out of power at around the same time. Also, we confirm the result in [129]; by sacrificing little coverage, the network lifetime is extended. Specifically, the lifetime with densities of 1 nodes/r^2 (respectively, 2 nodes/r^2) is around 39.55 minutes (respectively, 57.9 minutes).

Quality of coverage. As mentioned in [129], in partially covered sensor networks, quality of coverage is an important metric. For example, in surveillance networks, it is measured in terms of how fast the sensors detect a target object. Since the sleep interval (i.e., X) is 1 minute, time to detect stationary objects in the sensor field is bounded by 1 minute. Additionally, since the sensors rotate their roles (working vs. sleeping), the set of active sensors changes continuously. Hence, an undetected "hole" is likely to be detected as the set of active sensors changes. In Figure 6.8, we show the snapshots of the field at different times.

From Figure 6.8, we observe that the location of "holes" change continuously. In

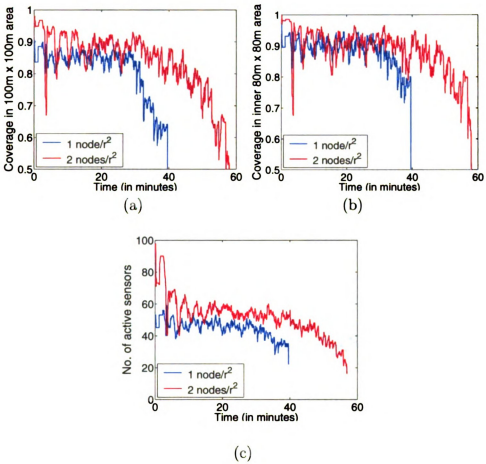
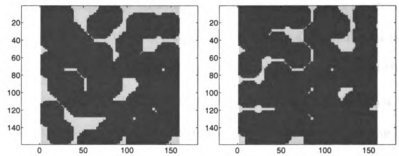


Figure 6.7: Coverage and number of active sensors over time; (a) coverage of entire 100m X 100m area, (b) coverage of inner 80m X 80m area, and (c) number of active sensors.

surveillance networks, the intruder does not know the location of such holes. Hence, it is unlikely that the intruder can choose to move along the uncovered path. Therefore, the time to detect the intruder is small on average.

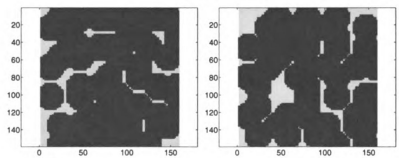
6.4 Chapter Summary

First, we note that protocols proposed in the literature can be reused with ProSe as is (e.g., routing program in Figure 6.1 is similar to [25], tracking program in Figure 6.4 is the same as the evader-centric program in [37], pCover program in



(a) 10 minutes (b) 20 minutes

(coverage: 83.5%/87.6%) (coverage: 89.9%/90.9%)



(c) 30 minutes (d) 40 minutes

(coverage: 89.2%/91.6%) (coverage: 85.2%/89.8%)

Figure 6.8: Snapshot of the field with density = 2 nodes/r^2 (dark regions are covered). Coverage data below each subfigure shows the coverage of entire area and the coverage of inner 80m X 80m area respectively at that time.

Figure 6.6 is the same as [129]). Based on the case studies discussed in Sections 6.1 and 6.2, we find that the programs generated by ProSe are competitive to manually designed protocols. In this context, we note that properties of the protocol obtained from ProSe depend upon the initial protocol in guarded commands. However, since ProSe allows the designer to automatically transform an abstract program to generate the corresponding nesC/TinyOS code, it can be used to rapidly prototype a given protocol. Thus, a designer can utilize ProSe for designing different protocols needed for a given sensor network application. Of these, some protocols will meet the desired constraints of the application. (Since the process of generating code for these protocols

is automated, the development time for this is very small.) The designer will then have to design only those protocols for which an acceptable protocol is not available.

Additionally, we observe that the abstract model used in ProSe is the same as that is used in tools that automatically add new properties (e.g., FTSyn [84] that adds fault-tolerance properties). Thus, ProSe can be integrated with such tools to add properties of interest and generate nesC/TinyOS code that preserves such properties. Furthermore, designers can gain assurance about their programs by verifying them with model checkers. Since ProSe preserves several properties of interest, we expect such verification to be highly valuable.

Chapter 7

Infuse: Data Dissemination Protocol for Sensor Networks

Sensor networks are usually deployed in hostile/inaccessible fields (e.g., battlefield, national borders, hazardous sites, etc). And, the sensors are deployed in large numbers over a large field. Hence, it is difficult –if not impossible– to manually collect them after deployment. As a result, to quickly deploy and evaluate sensor network protocols and applications, sensors must be reprogrammed with a new program *in place* over the wireless radio medium. Such reprogramming is important to deal with bug-fixes, reconfigurations, and tuning application parameters. Towards this end, the new program must be disseminated over the wireless radio medium.

Challenges in network reprogramming. Since the payload of a message in the sensors (in the order of few bytes) is significantly less than the program (in the order of KBs), the new program must be split into several small packets. Therefore, reliable dissemination of such bulk data (e.g., new program) is important. One of the important challenges in reliable dissemination of bulk data is that the network is multi-hop in nature. The communication range of the sensors is limited and, hence, they need to collaborate with each other to forward the data across the network. As

a result, approaches such as XNP [33], where the sensors are assumed to be within the communication range of the base station, are not suitable.

Another challenge in sensor networks is the nature of message communication. As mentioned in Chapter 1, the basic mode of communication in sensor networks is *local broadcast with collision.* Thus, if multiple messages are sent to a sensor simultaneously then, due to collision, it receives none. This can also occur due to the *hidden terminal effect*, where a given message may collide at one sensor and be correctly received at other sensors.

To provide reliable message communication, different medium access control (MAC) protocols are proposed for sensor networks. Collision-avoidance protocol like carrier-sense multiple access (CSMA) offers only probabilistic guarantees about message communication. Most of the existing solutions for multi-hop data dissemination (e.g., [63, 64, 74, 86, 107, 123]) use CSMA and, hence, rely on other mechanisms such as acknowledgments/negative-acknowledgments, advertise/request schemes, and/or error correcting codes for reliability.

Collision-free MAC protocol like time division multiple access (TDMA) offers deterministic guarantees about message communication and, hence, it is desirable for reliable data dissemination. Other collision-free protocols include frequency division multiple access (FDMA) and code division multiple access (CDMA). FDMA is often used with TDMA where each sensor knows when to listen to a particular frequency. CDMA requires special hardware for encoding/decoding messages and, hence, it is not desirable for resource poor sensors.

In this chapter, we propose *Infuse*,[1] a TDMA based data dissemination protocol for sensor networks. Specifically, in Sections 7.1 and 7.2, we present the protocol and in Section 7.3, we present an optimization to reduce the energy consumption during

[1] infuse v.; to cause to be permeated with something (as a principle or quality) that alters usually for the better ⟨*infuse* the team with confidence⟩. Source: Merriam-Webster Online, `http://www.m-w.com/`.

dissemination. Then, in Section 7.4, we discuss the properties of Infuse. Subsequently, in Section 7.5, we present the simulation results.

7.1 Infuse: Protocol Architecture

In this section, we present the protocol architecture of *Infuse*. We assume that there is a base station that is responsible for communicating with the outside world. The base station initiates the bulk data transfer whenever it receives a *startDissemination* message from the outside world (e.g., the monitoring/visualization station in the Line in the Sand experiments [9, 10]). The data is split into fixed size packets called *capsules*. The startDissemination message includes the ID of the new data and the number of capsules. Additionally, it may include the location (in EEPROM) where the sensors should store this data. Note that Infuse is not concerned with the contents of the data. For example, in difference-based reprogramming [115], the data is the difference between the old and the new programs. Moreover, the data could be encrypted (e.g., using link-layer encryption mechanism like TinySec [68]) to prevent malicious reprogramming or dissemination. Upon receiving the startDissemination message, the base station sends the *startDownload* message (that includes the ID of the new data, number of capsules, and optionally, the location where the sensors should store the data) to all its neighbors; these neighbors, in turn, forward it to their neighbors. Whenever a sensor receives the message, it initializes appropriate data structures in order to store the new data. Furthermore, it signals the application that a download is in progress. And, when the sensor receives the complete data, it signals the application that the download is complete. Figure 7.1 shows the Infuse protocol architecture.

TDMA Service. The dissemination layer relies on a TDMA service (e.g., [57, 60, 80]) that may in turn use a localization service (e.g., [56]). The TDMA service

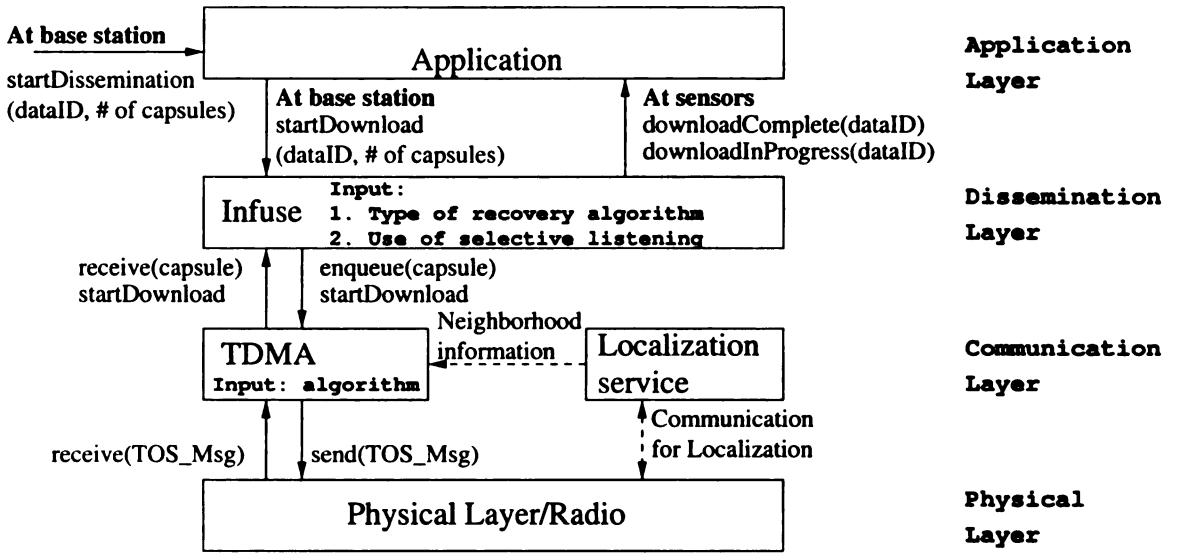


Figure 7.1: Protocol diagram of Infuse

identifies the slots in which a sensor can transmit and the slots in which it should listen to its neighbors. We assume that the TDMA service provides a fair share of bandwidth to each sensor. One way to achieve this is to ensure that between every two slots assigned to a sensor, at least one slot is assigned to its neighbors. The TDMA service from [80] ensures fairness. Also, we assume that the slots assigned to the sensors are periodically revalidated to deal with transient errors and/or clock drift.

Ideal scenario. Once the base station sends the startDownload message, in its subsequent TDMA slots, it sends data messages. Each message contains a capsule, say c_l, its sequence number, say n_l, and information for providing recovery (cf. Section 7.2). Whenever a sensor receives a capsule (say, c), it stores c at the appropriate location and enqueues it in the TDMA queue; thus, c will be forwarded to additional sensors farther from the base station.

7.1.1 Protocol Parameters

Infuse takes two (compile time) parameters: (1) the choice of recovery algorithm, and (2) whether a sensor should listen to only a subset of its neighbors to reduce active radio time (at the cost of increasing the latency). We describe the role of these parameters, next.

Parameter 1: Recovery algorithm. Although TDMA guarantees collision-freedom, background noise can cause random message losses. While dealing with these problems, the padding added to a message should be minimized since the payload size of a message is often limited (e.g., 29 bytes in Mica motes [61]). Also, the preamble added to a message in the lower layers of communication stack is high (e.g., 20 bytes in Mica). Hence, unnecessary communication (in terms of explicit acknowledgments) needs to be avoided.

During dissemination, whenever the successors of a sensor (say, j) forward the capsule (say, c), j gets an *implicit acknowledgment* for c. We use this information to recover from lost capsules. We compare two recovery algorithms based on the sliding window protocols [28, 125]. The recovery algorithms use implicit acknowledgments unlike the explicit acknowledgments used in the traditional sliding window protocols. The first algorithm, *Go-back-N* (cf. Section 7.2.1), does not add any padding to a message. The second algorithm, *selective retransmission* (cf. Section 7.2.2), adds $2b$ bits to a message, where $2b$ is the size of the window.

Parameter 2: Selective listening to neighbors. In the context of bulk data dissemination, a sensor may receive a message several times, once from each of its neighbors. To reduce duplicate messages, it is desirable that a sensor listens to only a subset of its neighbors. However, in this case, the latency may increase since the duplicate messages may assist in dealing with random message losses. When such selective listening is desired, each sensor classifies its neighbors as predecessors and

successors. Initially, all neighbors are (potential) predecessors as well as (potential) successors. Now, given two neighbors j and k, if k forwards a majority of new packets before j then j marks k as its predecessor (i.e., removes k from successor list) and k marks j as its successor (i.e., removes j from predecessor list). Once a sensor classifies its neighbors, it can choose to listen to 1 or more predecessors (for new capsules) and 0 or more successors (for implicit acknowledgments and recovery). (Note that when selective listening is not desired, all neighbors are treated as potential predecessors and successors.)

Based on these parameters, we obtain four possible versions of Infuse. Note that the version is selected at compile time, i.e., all sensors will be running the same version in any given experiment. In Section 7.5, we compare these versions to assist a designer in selecting the appropriate version based on the network characteristics.

7.2 Infuse: Recovery Algorithms

In our protocol, each sensor transmits a capsule in its TDMA slot. In order to deal with channel errors, in this section, we consider two recovery algorithms; these algorithms identify the capsule a sensor should forward in its TDMA slot. This is unlike CSMA based dissemination protocols (e.g., Deluge [63], MNP [86]) where extra steps (e.g., mechanism to reduce concurrent senders, transmitting meta-data about the availability of new data using advertisements, etc) need to be taken to prevent congestion and reduce collisions. Since there is no collision with TDMA, there is no need for elaborate control and each sensor can independently decide what capsule to send in its TDMA slot.

7.2.1 Go-Back-N Based Recovery Algorithm

In this algorithm, a sensor transmits a capsule with sequence number n_f iff it has received all capsules with sequence number smaller or equal to n_f. Thus, when a sensor transmits a capsule with sequence number n_f, it provides implicit acknowledgment for all capsules $0, \ldots, n_f$. To provide recovery in presence of channel errors, each sensor maintains (in RAM) a window of capsules with sequence number $n_{ia}+1, \ldots, n_{ia}+2b$, where n_{ia} is the highest sequence number for which the sensor has received an implicit acknowledgment from all its successors and $2b$ is the size of the window. Note that some of the window locations may be empty, if the corresponding capsules are not yet received. When a sensor receives a capsule in this window, it stores the capsule both in its RAM and EEPROM. Now, a sensor (say, j) will forward capsule c_f (with sequence number n_f) only when all its neighbors have forwarded capsules with sequence number $n_f - b$ or higher (i.e., $n_{ia} \geq n_f - b$). Otherwise, j will start retransmitting its current window, i.e., it will transmit the capsule with sequence number $n_{ia}+1$. This creates a *back pressure* in the network and, hence, the rate of dissemination of new capsules is reduced during recovery (cf. Figure 7.2 for the algorithm).

```
sensor j:
  highest_acknowledged_seqno = min(highest sequence number for which
        implicit acknowledgment is received from all successors of j);
  next_seqno++;
  if next_seqno > highest_acknowledged_seqno + b
    // start retransmitting from the start of the current window
    next_seqno = highest_acknowledged_seqno + 1;
  if the capsule with sequence number next_seqno has been received
    enqueue it in the TDMA queue;
```

Figure 7.2: Implicit acknowledgments and Go-back-N algorithm

Illustration. Consider the data dissemination process shown in Figure 7.3, where the window size is 2. In Figure 7.3(i), sensors transmit capsules $10, 9, 7, 6$ in time slots s_1, s_2, s_3, s_4, $s_1 < s_2 < s_3 < s_4$, respectively. In Figure 7.3(ii), the second sensor

forwards capsule 8 in slot s_2+P, where P is the period between successive TDMA slots. This is due to the fact that the second sensor did not receive implicit acknowledgment for capsule 8 from its successor. Hence, instead of forwarding capsule 10, it goes back and starts retransmitting from capsule 8. Similarly, in Figure 7.3(iii), the first sensor forwards capsule 10 instead of 12. Thus, lost capsules are recovered.

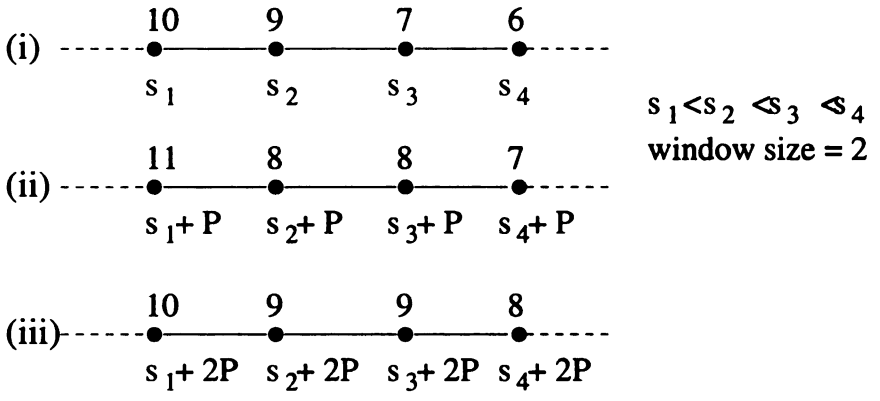


Figure 7.3: Illustration of Go-back-N algorithm on a linear topology

Dealing with failed sensors. In the presence of failed sensors, neighboring sensors will not get implicit acknowledgments. To deal with this problem, whenever a sensor fails to get an implicit acknowledgment from its successors after a fixed number of retransmissions, it declares that neighbor as failed. Now, a sensor will retransmit a capsule only when it does not receive an implicit acknowledgment from its active neighbors.

7.2.2 Selective Retransmission Based Recovery Algorithm

Similar to Go-back-N, in this approach, each sensor maintains a window of $2b$ capsules, where b is any integer. However, unlike Go-back-N, a sensor (say, j) will transmit the capsule with sequence number n_f even if it has not received some capsules with sequence number smaller than n_f. (This can happen due to channel errors.) Rather, j transmits capsule with sequence number n_f only if it has received all capsules with

sequence number $0, \ldots, n_f - b - 1$. Also, the sensor piggybacks acknowledgments for capsules with sequence number $n_f - b, \ldots, n_f - 1, n_f + 1, \ldots, n_f + b$. The piggybacked acknowledgments are used by its predecessors to determine the highest sequence number for which acknowledgment is not yet received ($n_{unacked}$) from some neighbor. To recover from lost capsules, j will forward capsule c_f (with sequence number n_f) only if $n_{unacked} > (n_f - b)$. Otherwise, j will retransmit the capsule containing the sequence number $n_{unacked}$. After retransmission, j will try to forward capsule with sequence number n_f in its next TDMA slot. The intuition behind selective retransmission is that even if a sensor misses a capsule transmitted by one of its predecessors, it may still receive the capsule from other neighbors. (This is due to the fact the sensors may have more than one path to the base station.) Furthermore, the piggybacked acknowledgments update the predecessors about the missing capsules at the successors. This also allows the predecessors to listen infrequently to the implicit acknowledgment of successors. Thus, it can be used to reduce message communication and active radio time (cf. Figure 7.4 for the algorithm).

```
sensor j:
  min_unacked_seqno = min(sequence number for which implicit acknowledgment
        is not received by j from some of its successors);
  next_seqno = smallest sequence number in the window
        {min_unacked_seqno, min_unacked_seqno+1, ..., min_unacked_seqno+2b-1}
        for which the corresponding capsule has been received but not yet forwarded;
  if min_unacked_seqno ≤ next_seqno - b
    // selectively retransmit the capsule with sequence number min_unacked_seqno
    next_seqno = min_unacked_seqno;
  enqueue [the capsule with sequence number next_seqno along with status flags for
        sequence numbers next_seqno ± x, 1 ≤ x ≤ b] in the TDMA queue;
```

Figure 7.4: Implicit acknowledgments and selective retransmission

Illustration. Consider the data dissemination process in Figure 7.5, where the window size is 2. Each sensor is shown transmitting $c[m|n]$, where c is the actual capsule forwarded by the sensor, m and n are the piggybacked acknowledgments

for capsules $c-1$ and $c+1$ respectively. If m (respectively, n) is $c-1$ (respectively, $c+1$), the predecessors get an implicit acknowledgment for the corresponding capsule. If m or n is represented as "X", it indicates that the sensor is yet to receive that capsule. In Figure 7.5(i), sensors forward capsules $10, 9, 7, 6$ in time slots s_1, s_2, s_3, s_4, $s_1 < s_2 < s_3 < s_4$ respectively. The second sensor missed capsule 10 due to channel errors and it forwards $9[8|X]$ in slot s_2. Since the window size is 2 (i.e., $b=1$), the first sensor cannot forward capsule 11 in slot s_1+P, where P is the period between successive TDMA slots. Hence, the first sensor retransmits capsule 10 (cf. Figure 7.5(ii)). Similarly, the second sensor forwards capsule 9 in slot s_2+P and s_2+2P (cf. Figure 7.5(ii-iii)). Thus, lost capsules are recovered through selective retransmissions.

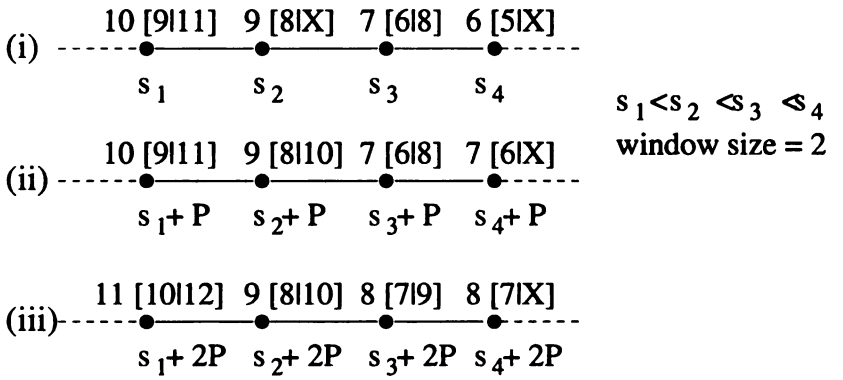


Figure 7.5: Illustration of selective retransmission based recovery algorithm on a linear topology

Remark. In the presence of failed sensors, the modifications proposed for Go-back-N algorithm can be applied for this approach as well. Also, we can make the recovery algorithms proposed in this section self-stabilizing using the framework for one-to-many sliding window protocol presented in [28].

7.3 Infuse: Optimization to Reduce Energy Usage

As discussed in Section 7.1.1, a sensor (say, j) classifies its neighbors as its predecessors and successors. Then, j selects one of the predecessors as the *preferred predecessor*. Note that the choice of preferred predecessor of one sensor is independent of that of others. This preferred predecessor is responsible for listening to implicit acknowledgments and recovering lost capsules at j. Now, whenever j forwards a capsule, it includes its preferred predecessor in the message. This can be achieved with $\log(q+1)$ bits, where q is the number of neighbors that a sensor has (and the $+1$ term is for the case where preferred predecessor is not yet chosen). Since the predecessors listen to the transmissions of their successors (to deal with channel errors), they learn about the sensors for whom they are the preferred predecessors. Once the preferred predecessor information is known, a sensor (say, k) will listen to the transmissions of j only if j's preferred predecessor is k. Otherwise, k will not listen in the time slots assigned to j. Thus, during data dissemination, the number of message receptions is reduced by allowing only the preferred predecessors to recover lost capsules at their successors.

However, if the preferred predecessor of j fails, j cannot recover from lost capsules. Towards this end, other predecessors will listen to the transmissions of their successors occasionally. In other words, a sensor (say, l) will listen in the time slots assigned to j with a small probability, if l is not the preferred predecessor of j. This will allow the successors to change their preferred predecessors and recover from lost capsules.

Remark. We note that the number of messages can be reduced even further as follows. If k is the preferred predecessor of j, k can choose to listen in the time slots assigned to j with a certain probability. This will allow k to listen to the transmissions of j occasionally. However, this is sufficient to recover lost capsules at j, since k learns about the lost capsules at j with the help of the implicit acknowledgments.

7.4 Infuse: Properties

In this section, first, we discuss how the data is propagated in a pipeline and estimate the latency in presence of no channel errors. Next, we argue that our approach is energy-efficient.

Pipelining. In Infuse, whenever a sensor receives a capsule, it stores the capsule in the flash at the appropriate address. Then, it forwards the capsule in the next TDMA slot. Hence, the capsules are forwarded in a pipeline fashion. If P is the period between successive TDMA slots, it takes at most $d * P$ time to forward one capsule across the network, where d is the diameter of the network ($= 2(n-1)$, in case of $n \times n$ grid network). If c_{tot} is the number of capsules in the data, as a result of pipelining, once the first capsule is forwarded, the remaining capsules can be forwarded within $(c_{tot}-1) * P$ time. Thus, in the presence of no channel errors, the time required to disseminate data with c_{tot} capsules is $((c_{tot}-1)+d) * P$. This provides an analytical estimate for the dissemination latency. For bulk data, $c_{tot} \gg d$. Therefore, dissemination latency is independent of the network size.

Energy-efficiency. In Infuse, the energy spent during dissemination is equal to the sum of energy spent in (1) idle-listening, (2) message receptions, (3) message transmissions, and (4) writing to EEPROM or external flash. Since all the sensors are required to write the data to their external flash, the energy spent in writing to external flash is a constant across the network. Additionally, in Infuse, each sensor forwards every capsule at least once. Hence, the number of message transmissions remains almost a constant across the network. Finally, in most sensor network platforms (e.g., Mica-2 [61]), the energy spent in idle-listening is equal to the energy-spent in receiving a message. Therefore, the energy spent during dissemination is determined by the amount of active radio time.

With TDMA, a sensor remains in active mode only in its TDMA slots (if it

needs to send any capsule) and in the TDMA slots of its neighbors. Hence, in the remaining slots, sensors can save energy by turning their radio off. Additionally, the use of preferred predecessor allows a sensor (say, j) to save energy by turning the radio off in the slots allotted to its successors for whom j is not the preferred predecessor. In case of Mica-2 sensors, the energy savings from turning the radio off in this manner is substantial, since the energy spent in the off state is only 3 μW whereas the energy spent in idle listening/message reception (respectively, message transmission) is 24 mW (respectively, 48 mW) [41]. Also, the Mica-2 sensors can switch to off (respectively, on or active) state instantaneously (respectively, in 2.5 ms) [41], whereas the timeslot interval in a typical TDMA algorithm is an order of magnitude more (e.g., 30 ms in SS-TDMA).

7.5 Infuse: Results

We simulated Infuse in Prowler [119], a probabilistic wireless network simulator for Mica motes [61]. The goal of these simulations is to validate the properties from Section 7.4 and to evaluate the performance of different versions of Infuse to enable a designer to choose the appropriate version of Infuse based on the network characteristics. We use one of the TDMA algorithms from [80]. We disseminate data consisting of 1000 capsules (unless specified otherwise) over a 3x3, 5x5, and 10x10 grid networks, where the base station is located at the top-left/north-west corner (i.e., location $\langle 0, 0 \rangle$).

Simulation model and parameters. In our simulations, we assume that the inter-sensor separation is 10 m, communication range is 10 m, interference range is 32 m, interference ratio (i.e., y) used by the TDMA algorithm is 4, and the time slot interval is 30 ms. These values correspond to our experience in the Line in the Sand experiment [9, 10].

In the absence of any interference, we have observed that the probability of successful communication is more than 98% among the neighbors. Since we use TDMA for message communication, interference from other sensors does not occur. However, random channel errors can cause the reliability to go down. Hence, we choose a conservative estimate of 95% link reliability in our simulations. This value also corresponds to the analysis in [46, 141].

Infuse parameters. In our simulations, the capsule size is 16 bytes. To deal with failed sensors, whenever a sensor fails to receive implicit acknowledgment from its successor, it retransmits 5 times before declaring failure. In case of preferred predecessors, if l is not a preferred predecessor of j, l will listen to the slots assigned to j with a probability of 20%. The parameters used in our simulations are listed in Table 7.1.

Table 7.1: Infuse simulation parameters

Parameter	Value
Network parameters:	
Inter-sensor separation	10 m
Link reliability	95%
Communication range	10 m
Interference range	35 m
TDMA parameters:	
Interference ratio	4
Time slot (time to transmit one message)	30 ms
Infuse parameters:	
Capsule size	16 bytes
Maximum number of retransmissions	5
Probability of listening to successors by their non-preferred predecessors	20%

Analytical estimate. Now, we compute the analytical estimates for dissemination using a specific TDMA algorithm [80] on $n \times n$ grid network. The estimate for (i) latency is $((c_{tot} - 1) + d) * P$, where c_{tot} is the number of capsules, $d = 2(n-1)$ is the diameter of the network, and P is the TDMA period, (ii) active radio time is 1

slot for forwarding the capsule and at most 4 slots for listening to the grid neighbors per capsule, (iii) message transmissions are equal to the number of capsules, and (iv) message receptions are 1 reception from the predecessor and 2 receptions for implicit acknowledgments from 2 sensors farther from the base station per capsule. We use the analytical estimate to compare the simulation results.

To disseminate 1000 capsules across a 10x10 network, where the timeslot interval is 30 ms and interference range = 4, the analytical estimate for (i) dissemination latency is 13.22 minutes, (ii) active radio time is 2.5 minutes, (iii) the number of message transmissions for each sensor is 1000, and (iv) the number of message receptions for each sensor is 3000.

7.5.1 Pipelining of Data Capsules

In this section, we verify the pipelining property of Infuse and show that this result is different from the dynamic behavior discussed in Deluge [63]. Figure 7.6(a-b) shows the progress of data dissemination for a data sequence consisting of 1000 capsules with Go-back-N algorithm. The window size used in these simulations is 6. At 5% (respectively, 50%) of time taken to disseminate 1000 capsules, all sensors have received 49-50 capsules (respectively, 502-505 capsules). Thus, the program capsules are transmitted in a pipeline.

The dissemination progress shown in Figure 7.6(a-b) contradicts the dynamic behavior presented in Deluge [63]. Specifically, in [63], it has been shown that the data capsules reach the edge sensors in the network first before reaching the middle of the network. This dynamic behavior causes congestion (due to CSMA based MAC) in the middle and, hence, message communication and latency are increased. However, with Infuse (cf. Figure 7.6(a-b)), we observe that all the sensors receive the data capsules at approximately the same time. And, Figure 7.6(c-d) shows the dissemination progress with selective retransmission algorithm. Again, this result shows that the

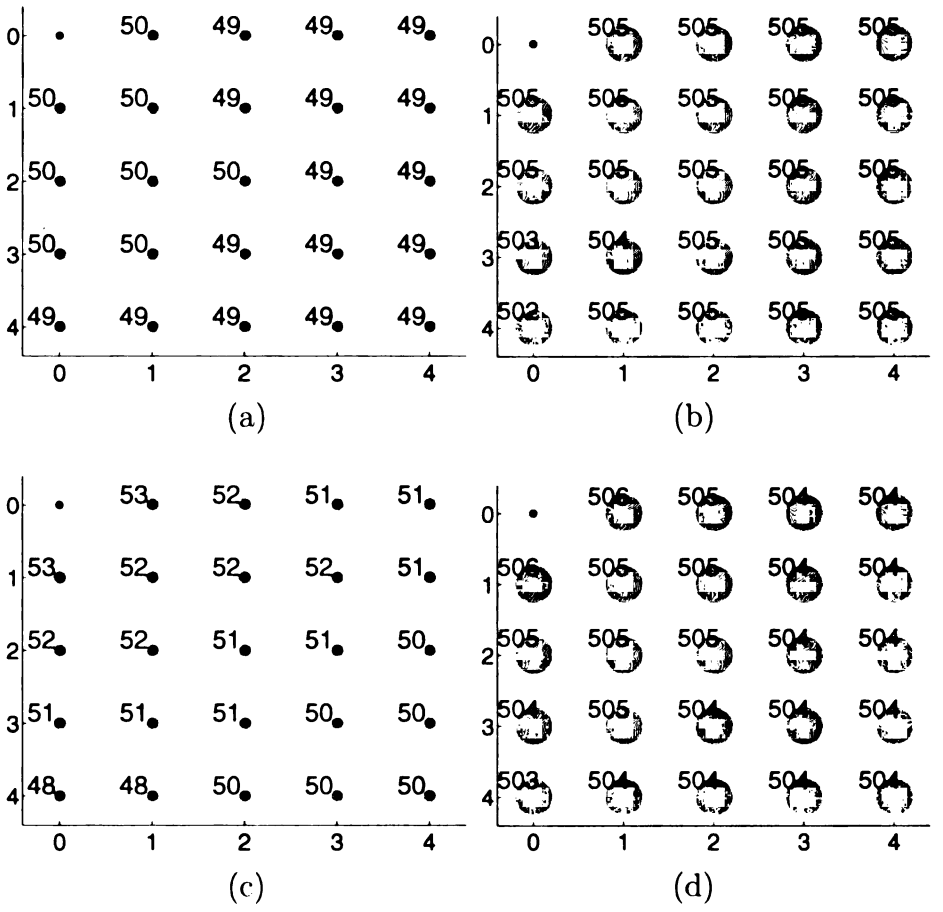


Figure 7.6: Dissemination progress for data of 1000 capsules with Go-back-N when (a) 5%, and (b) 50% of the time elapsed, and with selective retransmission when (c) 5%, and (d) 50% of the time elapsed. The radius of the circle at each sensor in the figure is proportional to the number of capsules received by the corresponding sensor.

dissemination latency along the edges is similar to the latency along the diagonal.

7.5.2 Performance of the Recovery Algorithms

In this section, we show that (1) due to pipelining, dissemination latency remains almost the same for different network sizes, (2) active radio time is significantly less than dissemination latency (and, hence, Infuse is energy-efficient), and (3) latency and active radio time grow linearly with respect to the data size.

Go-back-N algorithm. Figure 7.7 shows the results for dissemination with 1000

capsules for Go-back-N algorithm. With window size = 6, the latency is close to the analytical estimate (cf. Figure 7.7(a)). If a sensor (say, j) missed a capsule, its predecessor (say, k) will retransmit the capsule. Since j could still get the same capsule from its other predecessors or its successors, unnecessary retransmissions are reduced with window size = 6. Furthermore, the latency and the active radio time remains almost the same for different network sizes. This result is also expected based on the pipelining property of the proposed protocol (cf. Section 7.5.1) and the analytical estimate, as $c_{tot} \gg d$. Additionally, when the window size increases, the sensors have to transmit more messages during recovery, although most of the retransmissions may not be necessary. As a result, the recovery is too slow and, hence the latency increases. The same result can also be observed for message transmissions and receptions.

Selective retransmission algorithm. Figure 7.8 shows the results for selective retransmission algorithm. Once again, the latency is close to the analytical estimate and remains almost the same for different network sizes (due to pipelining). If a sensor misses a capsule, its predecessors selectively retransmit the capsule, thereby reducing the number of retransmissions. Thus, the latency and the active radio time are reduced. Likewise, message transmissions/receptions are reduced.

Latency and active radio time growth functions. Figure 7.9 shows how latency and active radio time grow with respect to the data size for both Go-Back-N and selective retransmission algorithms. As we can observe from the figure, both latency and active radio time grow linearly with respect to the data size.

7.5.3 Use of Preferred Predecessors

Figure 7.10 shows the results for dissemination with 1000 capsules. The window size used in these simulations is 6. As expected, selective retransmission (SR) and selective

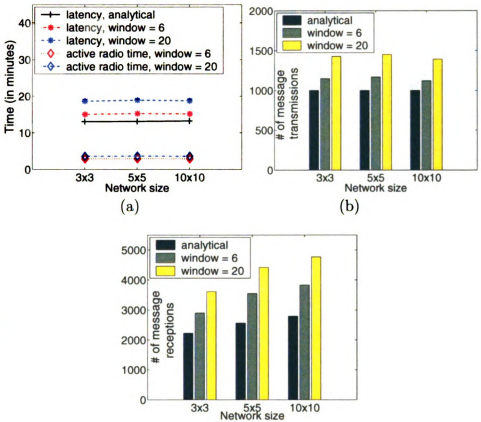


Figure 7.7: Simulation results for disseminating data with 1000 capsules using Go-back-N algorithm. (a) dissemination latency and active radio time, (b) number of message transmissions, and (c) number of message receptions

retransmission algorithm with preferred predecessors (SR-PP) perform better than Go-back-N algorithm (GBN) and Go-back-N algorithm with preferred predecessors (GBN-PP) respectively (cf. Figure 7.10(a)). This is due to the fact SR and SR-PP selectively retransmits lost capsules unlike GBN and GBN-PP. Moreover, the latency for SR-PP (respectively, GBN-PP) is more than SR (respectively, GBN).

For most situations, SR-PP (respectively, GBN-PP) has lower active radio time than SR (respectively, GBN). Thus, as expected, the use of preferred predecessor enables us to reduce the active radio time at the cost of increased latency. However, for large networks, the advantage of GBN-PP is no longer available; this occurs

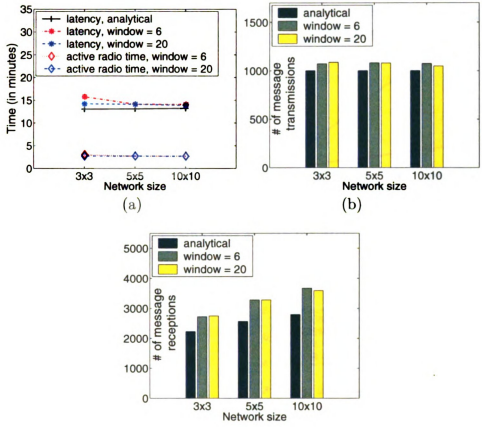


Figure 7.8: Simulation results for disseminating data with 1000 capsules using selective retransmission algorithm. (a) dissemination latency and active radio time, (b) number of message transmissions, and (c) number of message receptions

due to excessive retransmissions, as a sensor receives capsules from only one of its neighbors. By contrast, for GBN, the need for retransmissions is less, as a sensor receives redundant copies of a capsule. This effect is not seen while comparing SR and SR-PP, as unlike GBN and GBN-PP, a predecessor only retransmits the missing capsules and not the whole window. Hence, the effect of lost capsules in GBN-PP is more severe than that of SR-PP. With SR-PP, as the network size grows, active radio time reaches closer to that of SR, as the number of retransmissions in SR-PP is close to that of SR. (This is due to the fact the number of retransmissions by the preferred predecessors alone becomes closer to the number of retransmissions by all

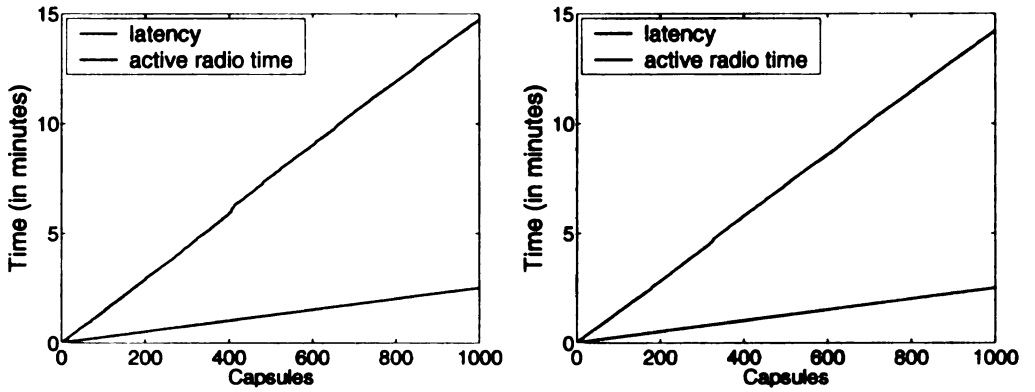


Figure 7.9: Latency and active radio time growth functions for (a) Go-back-N based recovery algorithm and (b) selective retransmission based recovery algorithm.

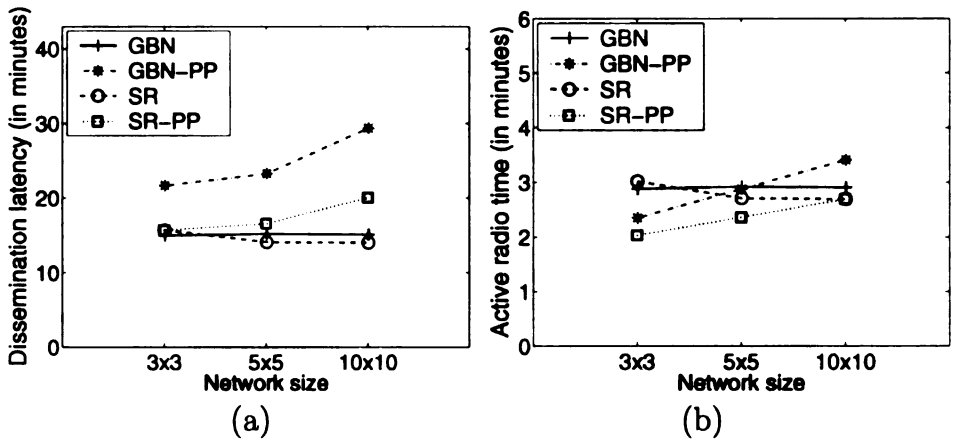


Figure 7.10: Simulation results with preferred predecessors with 1000 capsules, (a) dissemination latency and (b) active radio time. Note that the scale is different for the two graphs.

the predecessors of a sensor in SR.) Based on this result, we prefer SR and SR-PP compared to GBN and GBN-PP respectively. However, GBN and GBN-PP are easy to implement and GBN does not add any overhead to a message.

7.5.4 Effect of Window Size

In these simulations, data consisting of 100 capsules are propagated across a 5x5 network. Figure 7.11(a) shows the dissemination latency and active radio time for

Go-back-N algorithm. With window size = 2, whenever a sensor (say, k) observes that its successor (say, j) misses a capsule, it initiates recovery by retransmitting the corresponding capsule. However, j can still receive the capsule from its other neighbors, as j has multiple paths to the base station. Hence, in this case, the recovery is initiated too early. Therefore, the latency is higher for window size=2. For other values, predecessors allow the successors to recover from lost capsules through other neighbors. However, from Figure 7.11(a), we observe that as the window size increases, the latency also increases. This is due to the fact if j misses a capsule, its predecessor k starts retransmitting the *whole* window from the lost capsule, although most of the retransmissions are not necessary. Thus, with Go-back-N, we observe that the window size should be chosen such that recovery is neither initiated too early nor too late. In Figure 7.11(a), we note that with window size = $4, 6, \ldots, 12$, the latency remains almost the same. Hence, we choose window size = 6 in the rest of the simulations.

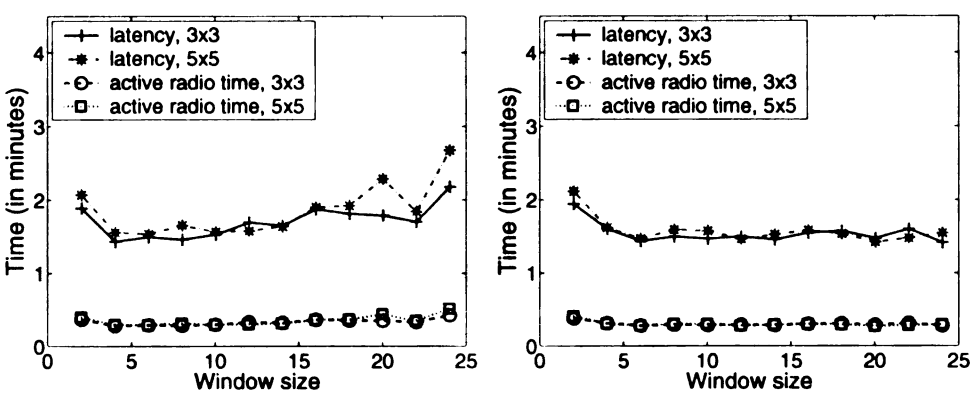


Figure 7.11: Effect of window size. (a) Go-back-N and (b) selective retransmission algorithms.

Figure 7.11(b) shows the effect of window size on selective retransmission algorithm. In this figure, we observe that the dissemination latency (and active radio time) remains constant for window sizes ≥ 6. This is due to the fact that the pre-

decessors selectively retransmit lost capsules, unlike Go-back-N algorithm. We note that the dissemination latency for window size $= 2, 4$ is slightly higher than that of other values. As discussed earlier in Go-back-N, in this case, if a sensor (say, j) misses a capsule, its predecessor (say, k) retransmits the corresponding capsule immediately, although j may receive the same capsule through other neighbors.

7.5.5 Effect of Failed Sensors

In these simulations, data consisting of 1000 capsules are propagated across the network. The window size used in these simulations is 6. The number of failed sensors in these simulations is 3 and 5 on a 5x5 network, and $3, 5, 10, 20$, and 27 on a 10x10 network. Figure 7.12 shows the effect of failed sensors on Go-back-N (GBN) and selective retransmission (SR) algorithms. From Figure 7.12, the additional time required for dissemination in presence of failed sensors is small. When the number of failed sensors increases, this additional time also increases. This is due to the fact that the pipeline is disrupted when the sensors fail.

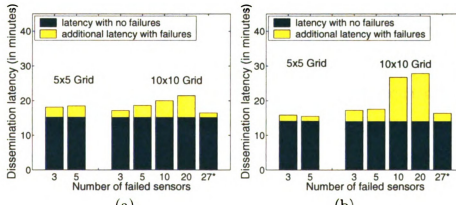


Figure 7.12: Effect of failed sensors. (a) Go-back-N and (b) selective retransmission. * indicates bulk failure of 3 sub-grids of size 3x3.

With GBN, whenever a sensor observes that its successors miss a capsule, it retransmits the entire window. In other words, an inherent redundancy is available in

GBN, where the sensor recovers all the successors (and possibly some predecessors) that have missed a capsule in the current window. However, with SR, in order to reduce the number of retransmissions, whenever a sensor observers that one of its successors has missed a capsule, it retransmits only the corresponding capsule. In other words, the level of redundancy is less in SR. Now, in presence of failed sensors, the number of paths to base station is reduced. Due to the built in redundancy in GBN, the effect of the reduction in paths in GBN is less severe than that in SR (cf. Figure 7.12).

Additionally, we did a simulation, where 3 sub-grids of size 3x3 are randomly selected as the failed sensors on a 10x10 network. From Figure 7.12, only 1.35 (respectively, 2.38) additional minutes are required to disseminate the data with GBN (respectively, SR). This value is close to the latency for dissemination in presence of 3 failed sensors. In other words, the cumulative effect of failure of nearby sensors shows up as a single disturbance in the pipeline. Therefore, the additional time required is less than the case where the failures are random.

Remark. In these simulations, we assumed that the sensors fail before the dissemination starts. Even if sensors fail during dissemination, the latency increases only by a very small percentage. Specifically, the latency is less than or equal to the latency in the case where the sensors have failed up front + the time required to detect the failure of sensors independently. Based on our simulations, the time required to detect failures is approximately 0.3 minutes. Thus, the latency in presence of dynamic failures increases only by a small percentage.

7.5.6 Effect of Other TDMA Algorithms and Topologies

The goal of this section is to illustrate that Infuse could be used with different TDMA algorithms and with different topologies. We compare the performance of Infuse on a uniform deployment of 100 sensors in a 10x10 grid with a random deployment. For

the uniform deployment, we use SS-TDMA, whereas we use the algorithm in [60] for random topology. (We note that the number of colors used to obtain TDMA schedule in both are approximately the same: 26 for grid topology and 31 for random topology.) The window size used in both simulations is 6. Table 7.2 summarizes the results of our simulations.

Table 7.2: Infuse on a random topology

Data size (in capsules)	Dissemination latency (in minutes)		Active radio time (in minutes)	
	Random topology	Grid topology	Random topology	Grid topology
Go-back-N (GBN)				
50	0.85	0.78	0.14	0.15
100	2.15	1.66	0.31	0.31
Selective retransmission (SR)				
50	1.34	0.72	0.36	0.14
100	2.86	1.54	0.57	0.29

With Go-back-N algorithm (GBN), the latency required to disseminate 50 capsules on a random topology is 0.85 minutes and the active radio time is 0.14 minutes. In case of grid topology, the latency required is 0.78 minutes and the active radio time is 0.15 minutes. For disseminating data with 100 capsules, the latency required in grid topology is lesser than that of random topology. However, the active radio time is less than that of SR in random topology.

With selective retransmission algorithm (SR), for random topology, the latency required is higher than that of GBN. In case of GBN, during recovery, a sensor retransmits all the capsules in the window. Hence, it helps the successors that have missed a particular capsule in the current window to recover. By contrast, in case of SR, when a sensor detects that one of its successors lags behind, it retransmits the particular capsule. This helps only that successor to recover, unlike GBN, and, hence, the sensor has to learn the status of other successors in future slots. Thus, with random topologies, GBN performs better. We did not experience this behavior in case of grid topology as a sensor has at most two successors and, hence, retransmitting

the whole window is expensive. Therefore, SR performs better on a grid topology and the active radio time is significantly less compared to random topology.

7.5.7 Comparison: Go-Back-N and Selective Retransmission

In this section, we compare the performance of our recovery algorithms. First, in Section 7.5.1, we show that both Go-back-N and selective retransmission algorithms provide a uniform, fine-grained pipelining service. This ensures that Infuse does not have the dynamic behavior expressed in [63]. Also, in Section 7.5.2, we observe that the latency and the active radio time grow linearly with respect to data size for both the algorithms. Second, in Section 7.5.3, to our surprise, we observe that the use of preferred predecessors does not significantly improve the performance of Go-back-N algorithm. This is due to the fact that with preferred predecessors, duplicate sources are reduced. As a result, the probability of successfully retransmitting the entire window during recovery is reduced. By contrast, we do not observe this behavior with selective retransmission algorithm, as a sensor selectively retransmits only lost capsules during recovery. Third, in Section 7.5.4, we observe that the window size should be chosen carefully in case of Go-back-N. On the contrary, window size (≥ 6) does not affect the performance of selective retransmission algorithm. While selective retransmission performs better on a grid topology with no failures, from Sections 7.5.5 and 7.5.6, we observe that Go-back-N performs better in presence of failed sensors and on random topologies. Table 7.3 summarizes the results.

In general, in traditional networking, it is expected that selective retransmission be better than Go-back-N. From Table 7.3, we observe that for a grid topology with no failures, this is valid. However, if the network has a random topology or can be affected by failures, Go-back-N is better than selective retransmission. This is due to the fact that unlike SR, inherent redundancy is available in GBN, where a sensor recovers all the successors that have missed a capsule in the current window during

Table 7.3: Comparison of recovery algorithms

	Go-back-N	Selective retransmission
Message overhead	none	2b bits, 2b = window size
Preferred predecessors	does not reduce active radio time	reduces active radio time
Pipelining	uniform, fine-grained	uniform, fine-grained
Latency/active radio time	linear	linear
Window size	affects latency	does not affect latency
Failed sensors	tolerates random failures	increases latency considerably
Random topology	does not increase active radio time	increases active radio time

recovery. Thus, this shows a somewhat counter-intuitive result that if the deployment may not be uniform or where sensors may fail, Go-back-N is preferable to selective retransmission.

7.5.8 Effect of Interference Ratio

In [46, 141], it has been shown that a signal from a (Mica based) sensor (say, j) reaches a sensor within distance 10 m (called, *connected region* or communication range) with probability $\geq 98\%$. And, the sensors at distance $10-32$ m (called, *transitional region* or interference range) receive the signal from j with a reduced probability. Finally, the sensors at distance > 32 m (called, *disconnected region*), do not receive the signal from the sender. Based on this discussion, the ratio of the interference range to communication range is around 3.2. If the network density increases (or a different sensor/radio hardware is used) then this ratio may increase. As a result, more sensors may fall in the transitional region. Therefore, the interference ratio would have to be increased. Figure 7.13 shows the analytical estimate on dissemination latency of Infuse with different interference ratios. As the ratio increases, the latency also increases. For a given interference ratio, the latency grows linearly with respect to the data size. Moreover, in case of grid topology, the number of slots for which a sensor keeps its radio on is approximately $5\times$ the number of capsules (one slot to forward,

4 slots to listen to its grid neighbors). Hence, the active radio time is independent of the interference ratio.

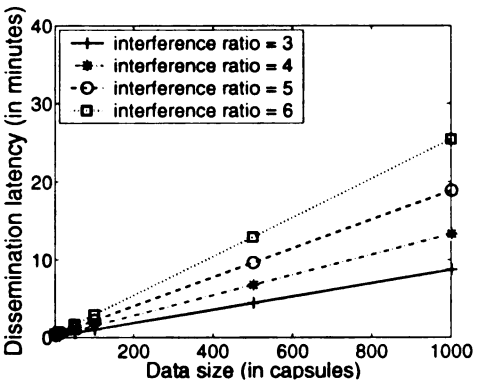


Figure 7.13: Effect of interference ratio on dissemination latency

In case of even larger values of interference ratio, the probability that a given message reaches a longer distance increases. Towards this end, we can use the SS-TDMA algorithm customized for larger communication ranges [80]. In this approach, not all sensors are required to forward the data. Specifically, the sensors that are farther away from the sender can forward the data. In case of random topologies, we can organize the network into clusters and elect cluster heads/leaders [57]. Alternatively, we can compute the minimum connected dominating set or MCDS (similar to Sprinkler [102]). Once the leaders or the sensors in MCDS are identified, we can establish a TDMA schedule (e.g., using [57, 60]) for them. The remaining sensors can then listen to the slots assigned to their closest leaders or sensors in MCDS. Thus, Infuse can be easily modified to disseminate data in high density networks.

7.5.9 Comparison with Related Work

Related work on dissemination has been addressed for wired networks in [69] where reliable transmission of multicast messages using multiple multicast channels is proposed. One of the important concerns in dissemination for wireless networks is the

broadcast storm problem [106]. Specifically, in dissemination using naive flooding based algorithms, a broadcast storm is created where redundant broadcasts, contention, and collisions occur. Infuse is not affected by the broadcast storm problem since contention/collisions are managed by TDMA.

Network programming. Related work on dissemination protocols, especially for network programming in sensor networks, include Deluge [63] and multi-hop network reprogramming (MNP) [86]. Deluge is an epidemic protocol for disseminating large data objects that uses Trickle [90] to suppress redundant advertisements and requests, and to minimize the set of concurrent senders. MNP is a network reprogramming service that uses a sender selection algorithm to reduce the number of concurrent senders. Additionally, in MNP, sensors are allowed to turn their radio off whenever they are not transmitting or receiving new packets.

Comparison of Deluge and MNP with Infuse. In Table 7.4, we compare the simulation results of Deluge and MNP protocols with that of Infuse. Specifically, we compare the latency and the active radio time during dissemination of data of size 5.4 KB on a 10x10 network, where the interference ratio = 4. (We have chosen the data size as 5.4 KB based on the availability of results from [63, 86].) The latency with Go-back-N (GBN) and selective retransmission (SR) algorithms is less than that of Deluge and MNP. Furthermore, the active radio time with Infuse (= 1.0 minutes for GBN) is significantly less than that of Deluge (= 11.67 minutes) and MNP (= 5.87 minutes). This is due to the fact that Infuse allows each sensor to turn its radio off in the slots not assigned to itself and its neighbors. In particular, for the grid topology, a sensor keeps its radio on only in the slots assigned to itself and to its 4 grid neighbors. Therefore, Infuse offers an energy-efficient dissemination service.

If the network density increases or the radio hardware is different then the interference ratio increases. Since Deluge and MNP use a CSMA based communication service, due to hidden terminal effect and network congestion, we expect that the

Table 7.4: Go-back-N (GBN) and selective retransmission (SR) Vs. Deluge and MNP for dissemination on a 10x10 grid, where interference ratio = 4

Protocol		Data size	No. of packets or capsules	Latency (in minutes)	Active radio time (in minutes)
Deluge [63]		5.4 KB	240	11.67	11.67
MNP [86]		5.6 KB	256	9.61	5.87
Infuse	GBN	5.4 KB	345	5.21	1.0
	SR	5.4 KB	345	4.83	0.93

latency and the active radio time would increase in such cases. In Infuse, the TDMA period increases with interference ratio. As a result, the latency increases. However, the active radio time remains the same (cf. Table 7.5), as the sensor keeps its radio on only for 5 slots in each TDMA period.

Table 7.5: Results for Go-back-N (GBN) and selective retransmission (SR) for dissemination of data of size 5.4 KB (= 345 capsules) on a 10x10 network with different interference ratios (extrapolated from the results for interference ratio = 4)

Recovery Algorithm	Latency (in minutes)	Active radio time (in minutes)
Interference ratio = 6		
GBN	10.02	1.0
SR	9.29	0.93
Interference ratio = 8		
GBN	16.43	1.0
SR	15.23	0.93

Based on this comparison, we expect that Infuse will be highly beneficial in scenarios where the network is sparse and already deployed. However, in laboratory environments and in dense networks, the TDMA period may be very high. Hence, Infuse is not intended for such scenarios.

Other dissemination protocols. Other dissemination protocols include sensor protocols for information via negotiation (SPIN) [74], multi-hop over the air programming (MOAP) [123], and transport protocols [122, 128]. In SPIN, a 3-way handshake protocol (ADV/REQ/Data) is used to disseminate the data. Furthermore, meta-data

(i.e., high-level descriptors of data) is used to declare the availability of new data. Infuse differs from SPIN in that no negotiations using meta-data are necessary and reliability is achieved through implicit acknowledgments. In MOAP, a publish-subscribe interface similar to [64] is used to provide dissemination (especially, reprogramming) service. In this scheme, a sensor has to receive the entire code before it can send meta-data about the availability of new code. MOAP uses sliding window mechanisms and negative acknowledgments for loss recovery. By contrast, with Infuse, each capsule is forwarded as soon as possible. And, Infuse uses sliding window mechanisms with implicit acknowledgments for loss recovery.

Work related to transport protocols in sensor networks is also used for data dissemination. Examples of transport protocols for sensor networks include pump slowly, fetch quickly (PSFQ) [128] and reliable multi-segment transport (RMST) [122]. These protocols rely on negative acknowledgments for loss recovery. By contrast, Infuse uses implicit acknowledgments in order to recover lost capsules. Additionally, Infuse takes advantage of the underlying TDMA based MAC in providing a pipelined service.

7.5.10 Application of Infuse in Reprogramming

We implemented Infuse on TinyOS for Mica-2 and XSM motes. In the DARPA NEST meeting on extreme scaling in sensor networks (ExScal demonstration, Avon Park, FL, December 2004) [10], we demonstrated Infuse on Mica-2 motes to reprogram the network with a new program, using Go-back-N based recovery algorithm.

We demonstrated Infuse on a 5x5 grid in an outdoor setting, where the inter-sensor separation of 8 ft and the base station is located at the top-left corner of the grid. We integrated Infuse with SS-TDMA [80] to assign time slots to each sensor. We used a conservative estimate of interference ratio, $y=6$. Initially, the base station contained the new data (i.e., new program for the sensors). The size of the new program was 2 KB (=128 capsules). First, the base station established the TDMA schedule

according to the SS-TDMA algorithm. Then, it disseminated the program capsules, one in each of its TDMA slots. In this experiment, the dissemination latency was 3.5 minutes. The active radio time during the dissemination process was approximately 25 seconds. These results are close to the analytical estimate for latency (= 3.37 minutes) and active radio time (= 19.2 seconds).

We found similar results in other experiments at Michigan State University. We have experimented with Infuse for reprogramming the network with programs of size from 2 KB-15 KB, interference ratio of 4-6 and window size of 6-12. In all these experiments, the results were consistent with the analytical estimate/simulation results.

7.6 Chapter Summary

In this chapter, we presented *Infuse*, a TDMA based data dissemination protocol for sensor networks. To deal with random message losses caused by varying link properties and message corruption, we considered two recovery algorithms based on the sliding window protocols that use implicit acknowledgments. The first algorithm, Go-back-N, adds no extra information to the payload of a message. With Go-back-N, we showed that the window size should be chosen carefully. And, we observed that Go-back-N tolerates failed sensors without significant degradation in performance. The second algorithm, selective retransmission, adds $2b$ extra bits to the message, where $2b$ is the size of the window. With selective retransmission, we showed that window size (≥ 6) does not affect the protocol. However, in presence of failed sensors, we showed that it increases latency considerably. Thus, we find a somewhat counter-intuitive result that Go-back-N is preferable to selective retransmission if topology is not uniform or if failures may occur.

In presence of no channel errors, we estimated the dissemination latency. We

showed that the data is propagated in a pipeline and, hence, the latency is reduced. We argued that Infuse is energy-efficient. Specifically, we showed that message transmissions/receptions are reduced. Since Infuse uses a TDMA based MAC protocol, sensors need to listen to the radio only in the slots assigned to their neighbors. In the remaining slots, sensors can turn off their radio. Moreover, we proposed an algorithm to reduce messages receptions and the active radio time further by using the notion of preferred predecessors.

Reliable data dissemination is a bandwidth intensive and time consuming operation. Hence, it has the potential to disrupt the communication of the underlying application. In a CSMA based network, this disruption is expected to be severe, as the network is highly congested. By contrast, a TDMA based protocol can provide some guarantees about the communication of the underlying application. Since the application messages (e.g., event messages) are rare and are time critical, the TDMA algorithm can be extended to provide high priority for such messages. The TDMA algorithm can also be customized (e.g., using SS-TDMA) for the communication pattern of the application. Thus, the TDMA algorithm can communicate such rare messages reliably. In this context, in [80], we have compared the performance of CSMA with SS-TDMA. Specifically, if the only communication consists of event messages, SS-TDMA improves the reliability from 50% to 100% with a only small increase in the delay. It follows that if event messages have to compete with bandwidth intensive dissemination then a TDMA based service such as Infuse will be especially useful.

Chapter 8

Tradeoffs in Sensor Communication

In the last three chapters, we developed tools and protocols that enable rapid prototyping and quick deployment of sensor network protocols and applications. In order to facilitate the designers to quickly evaluate the performance of their protocols, it is desirable to fine-tune the parameters of the protocols depending on the tradeoffs in the execution environment. Specifically, the tradeoffs will permit the designers to choose the appropriate operating point for their protocols.

In this chapter, we identify the tradeoffs in sensor communication. First, in Section 8.1, we identify the tradeoff between energy and latency in communicating event messages (e.g., reporting the presence of intruders) to the external world, i.e., the base station. Next, in Section 8.2, we identify the tradeoff between causal delivery and timely delivery of messages to the base station. Finally, in Section 8.3, we discuss the related work.

8.1 Energy Vs. Latency

The ability to reliably communicate an event of interest to the base station or the outside-world is an essential function in sensor networks. For example, in applications such as ExScal and A Line in the Sand [9, 10], where sensors detect, classify, track,

and visualize intruders along an area, and habitat monitoring [124], where sensors monitor the activities of a habitat, the sensors are required to communicate their observed values to the base station. Such many-to-one (or source-to-sink) communication is often referred as *convergecast*. One of the important requirements in such communication is that the latency involved during convergecast should be minimized.

As discussed in earlier chapters, an important constraint with sensor networks is limited power. To deal with this constraint, as part of an energy management scheme, sensors are allowed to turn their radio off or switch to low-power mode, where the amount of idle listening and overhearing are reduced. For example, time division multiple access (TDMA) algorithms proposed in [57, 60, 80] can be effectively used to reduce the amount of idle listening by allowing a sensor to turn its radio off in the time slots not assigned to itself and its neighbors.

It is easy to observe that there is a potential conflict between energy-management and low-latency convergecast. Specifically, if the radio is always on then the latency may be reduced. Similarly, if the sensors execute as part of an energy-management scheme then the latency may increase.

With this motivation, in this section, we focus on the problem of energy-efficient convergecast while ensuring that the latency is within the application requirements. Specifically, we propose a TDMA based convergecast algorithm. With the help of simulations, we analyze the performance of the proposed convergecast algorithm and show that it provides a better energy-latency tradeoff.

8.1.1 Convergecast Algorithm

Suppose each sensor listens to the medium always. Whenever a sensor (say, j) observes an event, it sends a convergecast message (say, m_c) to the base station. Now, consider a sensor (say, k) that is on the path between j and the base station. We observe that until k receives m_c from j, k spends time on *idle listening* and, hence,

most of its energy is wasted. To provide a better energy-latency tradeoff, we propose a TDMA based algorithm for convergecast in sensor networks.

In this algorithm, each sensor listens to the medium in the slots assigned to its neighbors at certain distance. Whenever a sensor observes an event, it sends a convergecast message in its TDMA slot. The sensors may use a routing algorithm (e.g., logical grid routing protocol or LGRP [25]) to forward the message to the base station for visualization and monitoring. Since multiple sensors observe an event, a TDMA based algorithm for reliably communicating all the messages to the base station increases the latency considerably.

To improve the latency in convergecast, in this algorithm, each sensor operates in one of the following two modes; *TDMA mode* or *active mode*. Initially, sensors execute in the TDMA mode (to conserve energy and reduce the amount of idle listening). Whenever a sensor receives a convergecast message, it forwards the message in its TDMA slot, and switches to active mode after a timeout. Furthermore, the sensor sets another timer, called *TDMA timer*; when this timer expires, the sensor returns to TDMA mode. In the active mode, the sensor listens to the medium always and forwards messages according to a CSMA protocol. The sensors may use a reliable communication layer (e.g., ReliableComm [140]) on top of the TDMA layer. Thus, the convergecast algorithm is shown in Figure 8.1.

8.1.2 Simulation Model

We simulated our algorithm in the framework based on Prowler [119], a probabilistic wireless network simulator for MICA motes [61]. We use SS-TDMA [80] to provide collision-free TDMA service. And, we use the logical grid routing protocol (LGRP) from [25] for routing. Finally, we use ReliableComm [140] to provide the reliable communication service to deal with message corruption and message collision (in CSMA networks). Next, we give a brief overview of SS-TDMA and ReliableComm.

```
j executes in TDMA mode
  listen to the TDMA slots of neighbors;
  if j receives a convergecast message m_c
    forward m_c in the next TDMA slot;
    setup wakeup-timer
      when timer fires, switch to active mode;
    setup TDMA-timer
      when timer fires, switch to TDMA mode;
j executes in active mode
  listen to the medium always;
  if j receives a message m
    forward m using a CSMA based algorithm;
```

Figure 8.1: TDMA based convergecast algorithm for sensor networks

(We refer the reader to Chapters 3 for more details on LGRP).

Self-stabilizing TDMA (SS-TDMA). We note that our algorithm does not depend on a specific TDMA algorithm. We have chosen SS-TDMA [80] due to its applicability in [9]. Moreover, SS-TDMA can be customized for convergecast. In [80], the sensors are arranged in a 2 dimensional grid and the base station is located at $\langle 0, 0 \rangle$. If the interference ratio of the sensors is y then the initial slot of a sensor located at $\langle i, j \rangle$ is $(P-1)i+(P-(y+1))j$, where $P=(y+1)^2+1$ is the TDMA period. If x_a is the initial slot of sensor a then a can transmit in slots: $x_a + c * P$, where $c \geq 0$.

ReliableComm (RComm). ReliableComm [140] is a CSMA based reliable communication protocol, designed to improve per-hop and end-to-end reliability in presence of fading, collisions and congestion. It maintains a queue, where a given message is removed from the queue when the sensor receives an *implicit acknowledgment* from its parent. If it fails to receive the acknowledgment within a timeout, it retransmits the message. The number of retransmissions is bounded by a threshold. Since the base station does not require to forward the messages, it explicitly acknowledges the receipt of a message.

Simulation model and parameters. In our simulations, we assume that the

base station is located at one corner of the grid and the sensors that observe an event are located diagonally opposite to the base station. Furthermore, we require at least 50% of the messages to be delivered to the base station. This value is based on the reliability requirements of the LITeS experiment [9].

We assume that the sensors can communicate with high reliability among their neighbors. Also, we assume that the signal from a sensor may reach sensors within distance 6 although the probability of successful communication is very low. However, the actual radio propagation is based on the distance-fading model, where the strength of a signal from a sensor is inversely proportional to the square root of the distance. The parameters used in our simulations are listed in Table 8.1.

Table 8.1: Convergecast simulation parameters

Parameter	Value
Network/event.	
Network size	7x7 grid
Sub-grid sending event messages	1x1 - 5x5
Convergecast algorithm.	
Wakeup timeout	1 s
TDMA timeout	20 s
SS-TDMA/LGRP/RComm.	
Time slot interval	50 ms
Interference range	6
H	2
Maximum number of retransmissions	3
Retransmission timeout	55 ms

8.1.3 Simulation Results

We compare the performance of our algorithm using LGRP and ReliableComm (i.e., SS-TDMA+RComm+LGRP) with (i) SS-TDMA+LGRP, where sensors execute in TDMA mode always, and (ii) RComm+LGRP, where sensors execute in active mode always.

Latency. In Figure 8.2, we show the time required to convergecast event messages

to the base station. With SS-TDMA+RComm+LGRP and RComm+LGRP, only 50% of the event messages are delivered. The latency of the proposed convergecast algorithm follows closely that of the RComm+LGRP scheme. This meets the requirement of the LITeS experiment, where the desired latency for reporting an event is less than 13 seconds (with 50% reliability) [9].

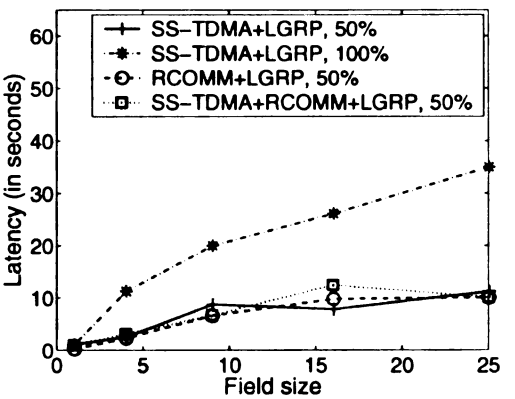


Figure 8.2: Latency during convergecast

With SS-TDMA+LGRP, all the messages are delivered to the base station. This is due to the fact that SS-TDMA ensures collision-free communication. Moreover, from Figure 8.2, we observe that the latency for delivering 50% of the messages with SS-TDMA+LGRP is almost equal to the other two schemes. This is due to the fact that SS-TDMA is customized for convergecast. Specifically, the slots assigned to sensors closer to the base station are after the slots assigned to sensors farther from the base station. Hence, whenever a sensor receives a message, it can forward it within 50 ms (= timeslot interval). Thus, the latency is close to RComm+LGRP scheme.

Suppose TDMA is customized for broadcast. Whenever a sensor receives a message, it has to wait at most P slots (=2.5 seconds, in our simulations) before it can forward, where P is the TDMA period. This delay is added at each hop and for each message. Hence, the latency may increase. With SS-TDMA+RComm+LGRP, a sensor switches to RComm once it forwards the first message. Hence, it offers better

latency.

Active radio time (ART). In Figure 8.3, we show the average ART of the sensors. As expected, SS-TDMA+LGRP is energy-efficient, since the sensors listen to the medium only in the slots assigned to their neighbors. In the remaining slots, the sensors conserve energy by turning their radio off. Thus, ART is approximately 40% of the time required for convergecast.

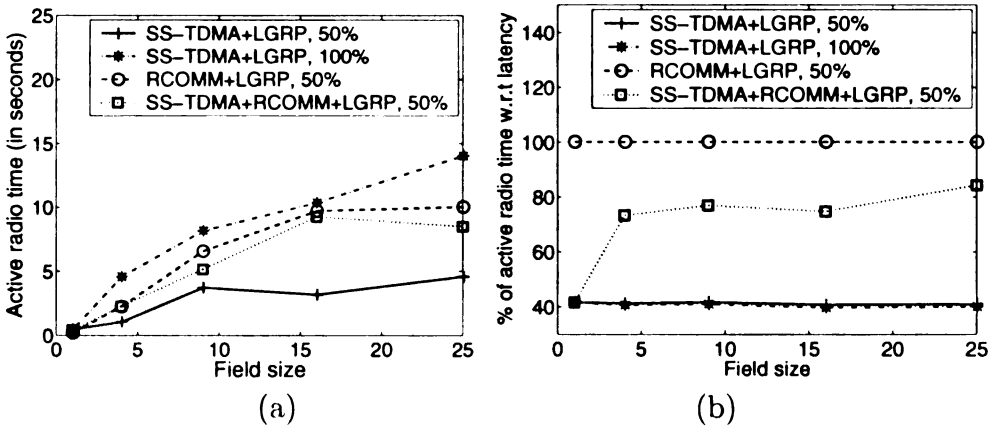


Figure 8.3: (a) Average ART and (b) % of average ART with respect to latency. Note that the scale is different in each figure.

With SS-TDMA+RComm+LGRP, the sensors conserve energy until the actual communication starts. From Figure 8.3(b), the sensors remain in active mode for 76.89% of the time (i.e., 5.13 seconds in 6.67 seconds), when a sub-grid of 3x3 sensors send messages. With RComm+LGRP, the sensors spend 100% of the time in active mode. Hence, it is not energy-efficient.

Message communication. In Figure 8.4, we show the number of transmissions/receptions during convergecast. The number of transmissions is in order of 500 when a sub-grid of 3x3 sensors send messages with 50% reliability for different schemes. Similarly, the number of receptions for different schemes is approximately the same. In case of SS-TDMA+LGRP, 100% reliability is achieved with increased

communication.

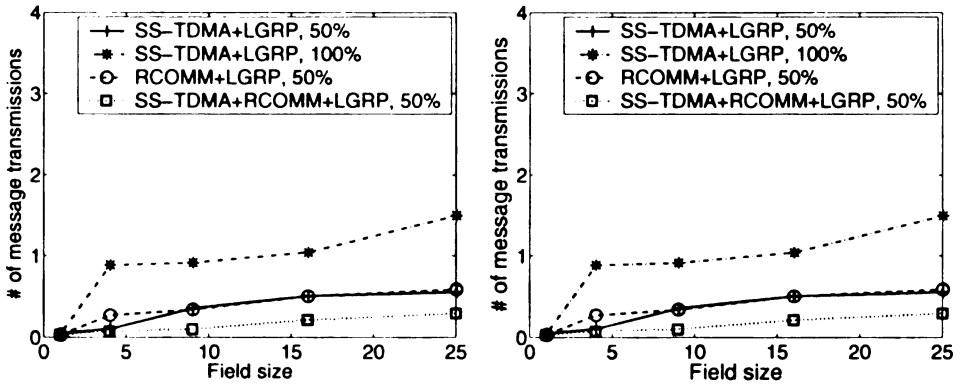


Figure 8.4: (a) No. of transmissions and (b) No. of receptions. Note that the scale is different in each figure.

Network lifetime. In Figure 8.5, we show the analytical estimate of network lifetime with respect to the probability of occurrence of an event at any instant for SS-TDMA+RComm+LGRP. If the probability is 0, the sensors execute in the TDMA mode always. The sensors can turn their radio off in the slots not assigned to itself and its neighbors. Thus, the network lifetime improves by 3.8 times (when the interference range = 6 and $H = 2$). When the probability of occurrence of an event increases, ART also increases. As a result, the network lifetime decreases. When this probability is close to 1, the sensors start to operate in the active mode more frequently and, hence, energy conservation is negligible.

In a typical application (e.g., LITeS [9, 10]), the probability of occurrence of an event at any instant is less than $10 - 15\%$. From Figure 8.5, SS-TDMA+RComm+LGRP improves the lifetime of the sensor network by approximately 3 fold.

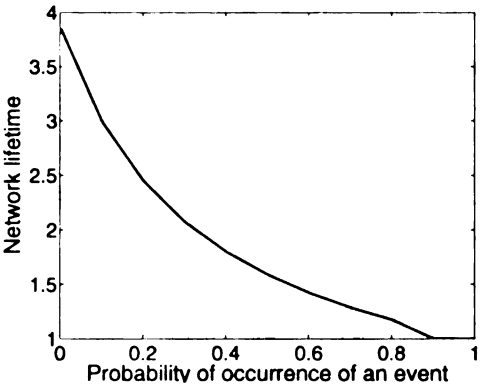


Figure 8.5: Analytical estimate of network lifetime with SS-TDMA+RComm+LGRP

8.2 Causal Delivery Vs. Timely Delivery

The ability to *observe* distributed computations is one of the important problems in many systems. Consider an application in sensor networks (e.g., MICA motes [61]) where a group of sensors need to track a moving object (e.g., [37]). In such a system, the sensors communicate their observations about the object they are tracking with each other. However, due to limited memory/computing power and small size, a sensor cannot provide human readable output. Hence, these applications typically include a more powerful *visualization unit* (e.g., a PC) that is responsible for providing the required human readable output. Thus, the visualization unit (or observer) *observes* the communication among sensors and uses its high computing power/memory to display/interpret the communication among sensors.

Since the order in which the observer receives messages may be different from the order in which the communication occurred in the underlying system, the observer needs to reorder messages consistently. While it may be impossible to recreate the exact scenario that occurred in the underlying system, it is desirable to obtain at least a consistent view at the observer. One way to obtain such consistent view is to ensure that the observer *delivers* the messages in a causal order.

One additional requirement for *online* observers is that they should be *timely*, i.e.,

the observer needs to reconstruct the underlying computation quickly. Specifically, the visualization of the underlying computation should be in real-time and, hence, the visualization unit must act quickly on the messages received so that the visualization does not significantly lag behind the original computation.

It is easy to observe that there is a potential conflict between achieving causal delivery and delivering messages quickly. Causal delivery requires that a message be buffered until all messages that causally depend on it are delivered. However, timely delivery requires that a message be delivered as soon as possible. Since these goals are contradicting, it is necessary to develop protocols where the observer can choose the level of causality violations it can accept to ensure timely delivery of messages. We call this the problem of *approximate causal delivery.* And, an observer that provides approximate causal delivery is called *approximate causal observer.* With this motivation, in this section, we adapt the algorithm in [85] to obtain approximate causal delivery.

8.2.1 Logical Timestamps and Causal Delivery

In this section, we discuss the algorithms for logical timestamp and causal delivery from [85]. We use the causal delivery algorithm for achieving approximate causal delivery.

System Model

A distributed system (e.g., sensor networks) consists of finite set of processes (e.g., sensors) which communicate via passing messages. Each process j has a physical clock $rt.j$. In the absence of faults, a distributed system must provide some guarantees that will enable the observer to obtain a tradeoff between causal delivery and timely delivery. In the presence of faults, these guarantees may be violated temporarily. We focus on the following two guarantees about the bound on maximum clock drift (for

example, using [58]) among different processes and the bound on message delay.

Guarantees of the distributed system.

*G*1. The value of $rt.j$ is non-decreasing, and at any time, the difference between the clock values of any two processes is bounded by ϵ. In other words,

$$\forall j, k : |rt.j - rt.k| \leq \epsilon$$

*G*2. Let m_j be a message sent by process j to k. Also, let st_m denote the clock value of j when j sent m_j, and let rd_m denote the clock value of j when k received m_j. We require that k should receive m_j within time δ unless m_j is lost. In other words,

$$((rd_m \leq (st_m + \delta)) \ \vee \ rd_m = \infty)$$

Execution of a process consists of a sequence of events; an event can be a local event, a send event, or a receive event. In a local event, a process neither receives nor sends a message. In a send event, a process sends one or more messages, and in a receive event, a process receives one or more messages. For simplicity, we assume that one clock tick of j corresponds to at most one event at process j. Note that, we can weaken this assumption so that one clock tick corresponds to at most K events, where K is any constant.

We assume that there is a special *observer* process in the system. A copy of relevant messages sent by any process is also sent to the observer. The observer buffers the messages and delivers them in such a way that the number of causality violations is acceptable. Note that we do not assume that the observer can precisely determine causal relations between two messages.

Notation. We use i, j, k and l to denote processes. We use e, f and g to denote events. Where needed, events are subscripted with the process at which they occur, thus, e_j is an event at j. We use m to denote messages. Messages are subscripted with the process that sends the message. Thus, m_j is a message sent by j.

Logical Timestamp Program

Before presenting the program, we define the notion of *happened-before*, $\longrightarrow$ among events.

Happened-before. The happened-before relation [87] is the smallest transitive relation that satisfies, for any events e, f, $e \longrightarrow f$ if (1) e and f are events on the same process and e occurred before f, or (2) e is a send event in one process and f is the corresponding receive event in another process. □

Solution to logical timestamp. In the solution to the logical timestamps proposed in [85], the timestamp of an event e_j at process j is of the form $\langle rt.e_j, c.e_j, kn.e_j \rangle$, where $rt.e_j$ denotes the physical clock value of j when e_j was created. The variable $c.e_j$ denotes the difference between the knowledge j had about the maximum clock value in the system and the physical clock value of j. The variable $kn.e_j$ is an array of size 2ϵ. The variable $kn.e_j[t]$, $-\epsilon \leq t < \epsilon$, captures the knowledge about the number of events f such that $r.f = r.e_j + t$ and $f \longrightarrow e$.

Each process j in the system maintains $rt.j$, $r.j$, $c.j$ and $kn.j$. (The algorithm works correctly even if j maintains $(rt.j \ mod \ B)$ instead of $rt.j$, where $B \geq \epsilon + \delta + 1$. Thus, the space required for maintaining $rt.j$ is bounded.) The variable $rt.j$ represents the physical clock value at j and $\langle r.j, c.j, kn.j \rangle$ represents the timestamp of the last event at j. The initial and update rules for different events are presented in Figure 8.6. Note that, for simplicity of presentation, we assume $kn.j[t] = 0$ if $t < -\epsilon$ or $t \geq \epsilon$.

Comparing timestamps. Let $\langle r.e_j, c.e_j, kn.e_j \rangle$ and $\langle r.f_k, c.f_k, kn.f_k \rangle$ be two timestamps. The *less* function for comparing timestamps based on the logical timestamp program in Figure 8.6 is as follows:

$less(\langle r.e_{,j}\, c.e_j, kn.e_j \rangle, \langle r.f_k, c.f_k, kn.f_k \rangle)$

iff

Initially:

$rt.j, r.j, c.j = 0,$

$\forall t : t \neq 0 : kn.j[t] = 0, kn.j[0] = 1$

Local event e_j or Send event e_j (message being sent is m_j)

$c.j := max(0, r.j + c.j - rt.j)$

$\forall t : -\epsilon \leq t < \epsilon : kn.j[t] := kn.j[t + rt.j - r.j]$

$kn.j[0] := kn.j[0] + 1$

$r.j := rt.j$

$r.e_j, c.e_j, kn.e_j := r.j, c.j, kn.j$

if e_j is a send event then $r.m_j, c.m_j, kn.m_j := r.j, c.j, kn.j$

Receive event e_j (message m received with timestamp $\langle r.m, c.m, kn.m \rangle$)

$c.j := max(0, r.j + c.j - rt.j, r.m + c.m - rt.j)$

$\forall\, t : -\epsilon \leq t < \epsilon : kn.j[t] := max(0, kn.j[t + rt.j - r.j], kn.m[t + rt.j - r.m])$

$kn.j[0] := kn.j[0] + 1$

$r.j := rt.j$

$r.e_j, c.e_j, kn.e_j := r.j, c.j, kn.j$

Figure 8.6: Logical timestamp program

$$(r.e_j + c.e_j, kn.e_j[c.e_j], kn.e_j[c.e_j - 1], \ldots, kn.e_j[c.e_j - \epsilon + 1], j)$$

$<$ // lexicographic comparison

$$(r.f_k + c.f_k, kn.f_k[c.f_k], kn.f_k[c.f_k - 1], \ldots, kn.f_k[c.f_k - \epsilon + 1], k)$$

In the above comparison, kn values are compared only when $r.e + c.e$ equals $r.f + c.f$. Thus, $kn.f[c.f]$ is compared with $kn.e[r.f + c.f - r.e] (= kn.e[c.e])$. Since $kn.f[t]$ denotes the knowledge about events at $r.f + t$, the comparison of kn values allows us to determine if f_k was aware of more events than e_j.

Properties of the logical timestamp program. The logical timestamp program presented above has the following properties. (We refer the reader to [85] for proof.)

- $\forall e, f :: e \longrightarrow f$

  $\Rightarrow less(\langle r.e, c.e, kn.e \rangle, \langle r.f, c.f, kn.f \rangle).$

- The value of $c.e$ is less than ϵ and the value of each element in $kn.e$ is less than n, where n is the number of processes in the system. Hence, the space needed by the timestamp is $O(\epsilon\ log\ n\ +\ log\ \delta)$. Further, it does not grow as the computation proceeds.

Causal Delivery Program

The causal delivery program proposed in [85] is as follows: Whenever a process j receives a message m, j buffers the message until $delcond(m, j) = (rt.j = r.m + c.m + \delta + \epsilon)$ is satisfied. As soon as the $delcond(m, j)$ is satisfied, the message is delivered. If two or more messages satisfy the delivery condition simultaneously then process j determines the causal relation among the messages and delivers them accordingly. If m_j and m_k satisfy the delivery condition simultaneously and $less(\langle r.m_j, c.m_j, kn.m_j\rangle, \langle r.m_k, c.m_k, kn.m_k\rangle)$ is $true$, then m_j is delivered before m_k.

Properties of the causal delivery program. The causal delivery program presented above has the following properties (cf. [85] for proof).

- If process j sends a message m when its physical clock value was $r.m$ then the message would be delivered before the physical clock value of j reaches $r.m + \delta + 3\epsilon$.

- If two messages m_1 and m_2 such that $send(m_1) \longrightarrow send(m_2)$ arrive at any process j then m_1 is delivered before m_2.

- The causal delivery program is self-stabilizing.

8.2.2 Approaches for Approximate Causal Delivery

The algorithm presented in Section 8.2.1 uses the delivery condition $delcond(m, j)$ to deliver a message m to process j. This condition is necessary for correctness, i.e.,

to ensure *all* messages are delivered in causal order. In other words, there exists messages for which this condition is optimal.

In the algorithm in Section 8.2.1, message m is delivered at process j when $rt.j = r.m+c.m+\delta+\epsilon$. Thus, $c.m+\delta+\epsilon$ is the approximate delay in obtaining causal delivery. We consider the case where messages are delivered before this delivery condition is satisfied. However, instead of choosing fixed values for the reduced delay, we let the delay be proportional to the underlying system guarantees and any other information that m carries.

In the algorithm in Section 8.2.1, $r.m + c.m$ captures the knowledge that the sender of m had about the maximum clock in the system. Also, ϵ and δ depend on the underlying system guarantees. Hence, we let the reduced delay to be a certain percentage of the delay incurred while ensuring causal delivery. Thus, the actual delay incurred by messages depends on the underlying system guarantees (ϵ, δ) and the knowledge $(c.m)$ that m had 'about the future'.

Based on this approach for reducing the delay, we present two algorithms for approximate causal delivery: (1) deliver after partial wait and (2) check before delivery.

Deliver After Partial Wait (DAPW)

In this algorithm, we use the following delivery condition: $delcond(m, j) = (rt.j = r.m + ¢(c.m + \delta + \epsilon))$, where $0\% < ¢ < 100\%$. Thus, $¢ = 0\%$ means that the messages are delivered to the observer when the clock of observer is at least $r.m$, or as soon as the message arrives at the observer, whichever is later. And, $¢ = 100\%$ means that the messages are delivered in perfect causal order. Thus, by using different values for $¢$, the application can choose the delay in delivery.

Check Before Delivery (CBD)

In DAPW, whenever a message, say m_1 is about to be delivered to process j, if there is a casually related message m_2 such that $send(m_2) \longrightarrow send(m_1)$ is true and m_2 is scheduled for delivery at a later time than m_1 then a causality violation is inevitable. Hence, we propose our second algorithm that checks the queue to determine causally related messages. Specifically, whenever message m_1 is about to delivered at process j, j checks the message queue to determine if there is any message m_2 such that $less(\langle r.m_2, c.m_2, kn.m_2\rangle, \langle r.m_1, c.m_1, kn.m_1\rangle)$ is *true*. If there exists such a message m_2 then j sets the delivery time of m_1 as $delcond(m_1, j) = delcond(m_2, j)$. If there are no such messages then m_1 is delivered based on the DAPW algorithm.

8.2.3 Simulation Model

Our simulation model consists of n ordinary processes and one special process (observer). The ordinary processes communicate with each other. Every message sent by an ordinary process is also sent to the observer. Now, we show how our simulation model ensures system properties stated in Section 8.2.1.

Ensuring $G1$. At each step of the simulation, one process is selected at random based on a uniform distribution of $n + 1$ processes (i.e., n ordinary process and a special observer process). The selected process (say, j) can increment its physical clock ($rt.j$) and send messages to other processes. The simulation program ensures $G1$ by selecting another process from the uniform distribution if incrementing $rt.j$ leads to violation of $G1$.

Ensuring $G2$. Whenever a process sends a message, the destination receives the message within $x, 0 \leq x \leq \delta$, unit(s) of time, thereby ensuring $G2$. Message delay is determined using a normal distribution $N(\mu, \sigma)$, where μ is the mean delay and σ is the standard deviation of the delay. In our simulations, we use $N(\frac{\delta}{2}, \frac{\delta}{4})$ (approximately

95% messages are received in $[0 \ldots \delta]$) and $N(\frac{\delta}{4}, \frac{\delta}{8})$ (approximately 95% messages are received in $[0 \ldots \frac{\delta}{2}]$) for message delay. If the random delay from the distribution is greater than δ, we treat it as a lost message. Since the message delay cannot be less than 0 in a real system, if the random delay from the distribution is less than 0, we choose another random delay from the same distribution.

After the system properties are met, the selected process can increment its physical clock and send messages to other processes.

Implementing message rate. Whenever a process (say, j) increments its physical clock, it sends a message to other processes with certain probability. We implement this using message rate. Process j chooses a random number between 1 and 1/message rate. If the random number is 1, j sends a message to another process. Also, whenever a process sends a message to another process, it sends a copy of the message to the observer.

8.2.4 Simulation Results

For our simulation, we developed an event simulation program in Java. The program takes number of ordinary processes, ϵ, δ, message rate, the mean of message delay, the standard deviation of message delay and the type of algorithm as input. We conducted experiments for $\delta = 10$ with the following values of ϵ: 5, 10, 20, and 30. Similarly, we conducted experiments for $\epsilon = 10$ with the following values of δ: 5, 10, 20, and 30. Note that, we have not associated a unit for ϵ and δ. If $\epsilon = 5$ and $\delta = 10$, it can be used to represent a system where the maximum clock drift is $5ms$ ($10ms$) and message delay is $10ms$ ($20ms$), etc. Further, we find that the ratio $\frac{\epsilon}{\delta}$ is important than the individual parameters.

For these values of ϵ and δ, we use the following values for message rate: 0.5, 0.1, and 0.01. Likewise, we use the following values for ¢: 100%, 80%, 60%, 40%, 20%, and 0%. For each input, we perform at least 3 experiments to compute the

causality violations. The results presented here are average of these experiments. For a given value of the input parameters, the percentage of causality violations in different experiments are similar.

In these experiments, we compute the number of causality violations at the observer as follows: For each m_1, we compute the number of messages delivered before m_1 (say, m_2) such that $send(m_1) \longrightarrow send(m_2)$ is true. We say that these messages violate backward causality. Likewise, for each m_1, we computer the number of messages delivered after m_1 (say, m_2) such that $send(m_2) \longrightarrow send(m_1)$ is true. We say that these messages violate forward causality. The number of causality violations is obtained by taking the average of messages that violate backward/forward causality.

To compute these causality violations, for each event/message, we also maintain vector timestamps [42, 99] in addition to the logical timestamps from Section 8.2.1. These vector timestamps identify the actual causal relation among events in the system. They are not used in any way to determine when messages are delivered.

Effect of Maximum Clock Drift

The effect of ϵ on causal delivery of messages using DAPW and CBD is shown in Figure 8.7. The graphs show the number of causality violations as a function of the percentage of delay, ¢, used in *delcond*. In these experiments, we use $\delta = 10$ and message rate = 0.1. The simulation consists of 10 ordinary processes and the special observer process.

DAPW. When the ratio $\frac{\epsilon}{\delta}$ is larger, the number of causally dependent messages for a given message m is large. Thus, for larger values of $\frac{\epsilon}{\delta}$, there is a higher probability that one or more of these messages are delivered before m. Hence, as $\frac{\epsilon}{\delta}$ increases, the number of causality violations increase (cf. Figures 8.7 (a) and 8.7 (c)).

When message delay is determined from the distribution $N(\frac{\delta}{4}, \frac{\delta}{8})$, 95% of the messages are received within $\frac{\delta}{2}$. By contrast, when message delay is determined from

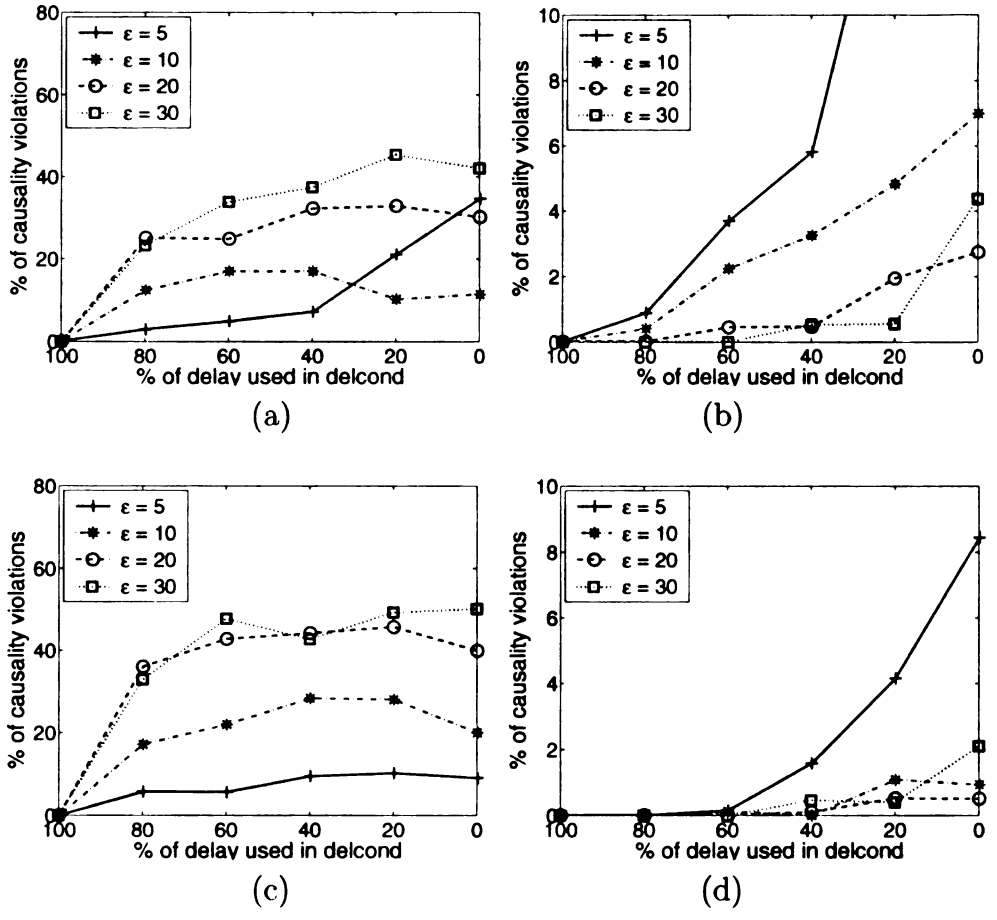


Figure 8.7: Effect of ϵ on causal delivery using (a) DAPW with message delay $N(\frac{\delta}{2}, \frac{\delta}{4})$, (b) CBD with message delay $N(\frac{\delta}{2}, \frac{\delta}{4})$, (c) DAPW with message delay $N(\frac{\delta}{4}, \frac{\delta}{8})$, and (d) CBD with message delay $N(\frac{\delta}{4}, \frac{\delta}{8})$. (Note that, the scale of DAPW and CBD graphs are different.)

the distribution $N(\frac{\delta}{2}, \frac{\delta}{4})$, 95% of the messages are received in δ. Thus, for $N(\frac{\delta}{4}, \frac{\delta}{8})$, the number of messages that causally depend on a given message is more than that for $N(\frac{\delta}{2}, \frac{\delta}{4})$. Hence, the probability of causality violations is more when the distribution $N(\frac{\delta}{2}, \frac{\delta}{4})$ is used for message delay (cf. Figures 8.7 (a) and 8.7 (c)).

For small values of $\frac{\epsilon}{\delta}$, there exists a threshold T such that, the causality violations increase suddenly when $¢ < T$. For example, in Figure 8.7 (a), for $\epsilon = 5$, there is a sudden rise in causality violations for $¢ < 40\%$. The number of causally related messages is less when the ratio $\frac{\epsilon}{\delta}$ is small. When $T \leq ¢ < 100\%$, the delay in delivery captures most of the causal relation among messages. When this delay is reduced

(i.e., $¢ > T$), the messages are delivered faster and, hence, the causal relation among messages is not captured. For larger values of $\frac{\epsilon}{\delta}$, more causally related messages are present for a message m. Hence, the observer captures most causally related messages even when the delay in delivery is less.

CBD. From Figures 8.7 (b) and 8.7 (d), we observe that as $\frac{\epsilon}{\delta}$ ratio increases, the number of causality violations decrease. This result is exactly opposite to DAPW. In CBD, before delivering a message m_1, the message queue is checked to determine if there is any message, say m_2 such that m_1 causally depends on m_2. If there is such a message, CBD postpones the delivery of m_1. Thus, as $\frac{\epsilon}{\delta}$ ratio increases, CBD can detect/prevent most of the causality violations since there is higher probability that the message queue contains one or more causally related messages.

Contrary to the observation for DAPW, we note that the number of causality violations is less when CBD is used with message delay of $N(\frac{\delta}{4}, \frac{\delta}{8})$. This is due to the fact that there are more causally related messages for a given message m, and there is a high probability that at least one of them will be present in the message queue of the observer when m is about to be delivered.

Comparison. From Figure 8.7, we conclude that the number of causality violations in CBD are an order of magnitude less than that in DAPW. For small values of $\frac{\epsilon}{\delta}$, CBD performs almost similar to DAPW. When $\frac{\epsilon}{\delta}$ is small, the number of causally related messages for any message is less. Therefore, CBD has limited or no information in the message queue to detect/prevent causality violations, as opposed to the case where the ratio $\frac{\epsilon}{\delta}$ is large. Thus, for small values of $\frac{\epsilon}{\delta}$, it may be better to use DAPW and save the overhead of checking the queue as one in CBD. Also, we note that the processing overhead with DAPW is significantly lower than that of CBD. Hence, for small values of $\frac{\epsilon}{\delta}$, we recommend DAPW. As the ratio $\frac{\epsilon}{\delta}$ increases, we prefer CBD, since the small addition in processing overhead reduces the number of causality violations considerably. (We observe similar results in Figure 8.8, where

the effect of varying message delay is studied and in Figure 8.9, where the effect of varying the message rate is studied.)

Effect of Maximum Message Delay

The effect of δ on causal delivery of messages using DAPW and CBD is shown in Figure 8.8. The graphs show the number of causality violations as a function of percentage of delay, ϵ, used in *delcond*. In these experiments, we use $\epsilon = 10$ and message rate $= 0.1$. The simulation consists of 10 ordinary processes and the special observer process.

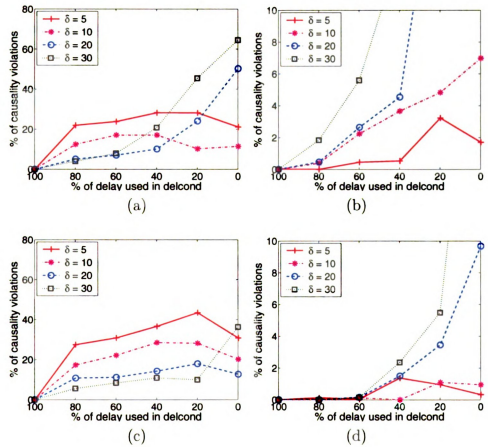


Figure 8.8: Effect of δ on causal delivery using (a) DAPW with message delay $N(\frac{\delta}{2}, \frac{\delta}{4})$, (b) CBD with message delay $N(\frac{\delta}{2}, \frac{\delta}{4})$, (c) DAPW with message delay $N(\frac{\delta}{4}, \frac{\delta}{8})$, and (d) CBD with message delay $N(\frac{\delta}{4}, \frac{\delta}{8})$. (Note that, the scale of DAPW and CBD graphs are different.)

DAPW. From Figures 8.8 (a) and 8.8 (c), we observe that as $\frac{\epsilon}{\delta}$ increases, the number of causality violations increase. These results are similar to that in Figure 8.7.

Further, in Figure 8.8 (c), we observe that the number of causality violations is more when DAPW is used with message delay of $N(\frac{\delta}{4}, \frac{\delta}{8})$. Once again, these results are similar to that in Figure 8.7.

Similar to Figure 8.7, for small values of $\frac{\epsilon}{\delta}$, there exists a threshold T such that causality violations increase suddenly when $¢ < T$. For example, in Figure 8.8 (a), for $\delta = 20$ (respectively, $\delta = 30$), there is a sudden rise in causality violations when $¢ < 60\%$ (respectively, $¢ < 40\%$).

CBD. From Figures 8.8 (b) and 8.8 (d), we observe that as $\frac{\epsilon}{\delta}$ increases, the number of causality violations decrease. Once again, this is exactly opposite to DAPW.

Effect of Message Rate

The effect of message rate on causal delivery of messages using DAPW and CBD is shown in Figure 8.9. The graphs show the number of causality violations as a function of percentage of delay used in *delcond*. In these experiments, we use $\epsilon = \delta = 10$. The simulation consists of 10 ordinary processes and the special observer process.

As the message rate increases, more causally dependent messages for a message m are present in the system. Thus, the probability of causality violations is higher. Further, the number of causality violations in CBD is significantly less than that in DAPW.

When δ is an overestimate of message delay (i.e., message delay of $N(\frac{\delta}{4}, \frac{\delta}{8})$), the number of causality violations in CBD is in the order of $0\% - 2\%$ (cf. Figure 8.9 (d)). This is due to the fact most messages arrive within $\frac{\delta}{2}$, and, hence, CBD has more information present in the message queue to detect/prevent causality violations

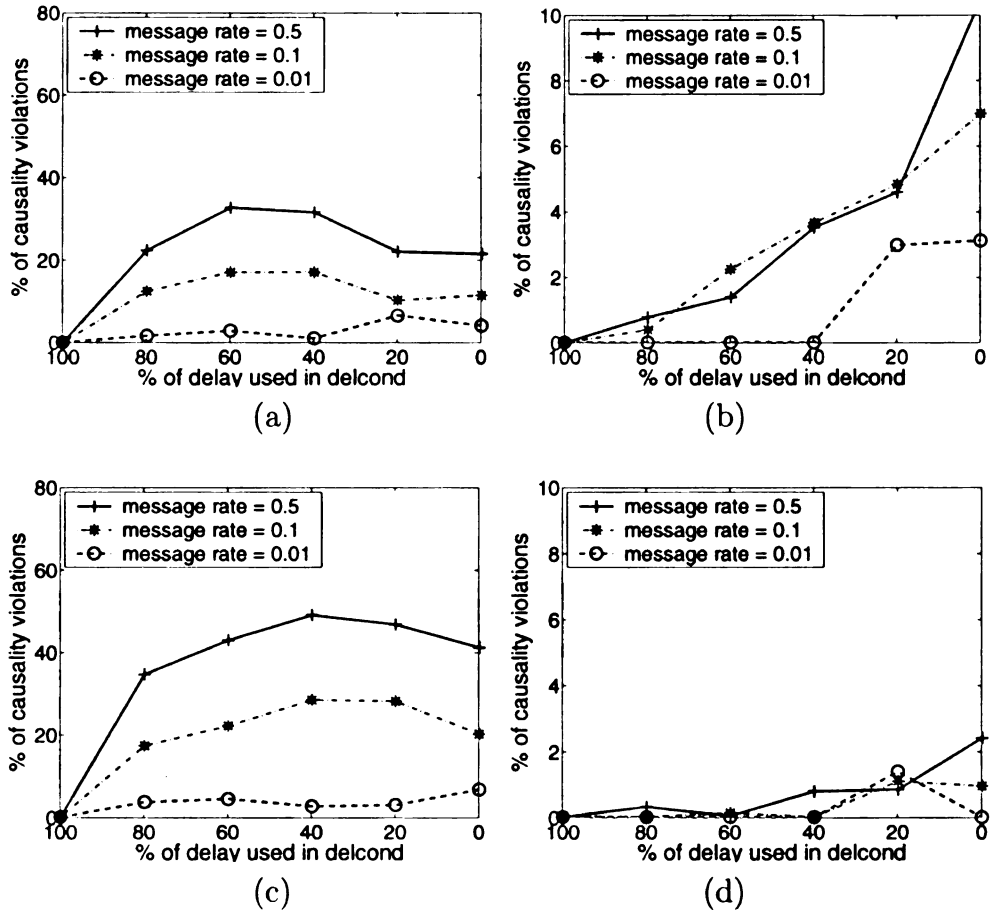


Figure 8.9: Effect of message rate on causal delivery using (a) DAPW with message delay $N(\frac{\delta}{2}, \frac{\delta}{4})$, (b) CBD with message delay $N(\frac{\delta}{2}, \frac{\delta}{4})$, (c) DAPW with message delay $N(\frac{\delta}{4}, \frac{\delta}{8})$, and (d) CBD with message delay $N(\frac{\delta}{4}, \frac{\delta}{8})$. (Note that, the scale of DAPW and CBD graphs are different.)

before delivering a message.

Further, for small values of message rate, causality violations in DAPW and CBD are nearly equal. This is due to the fact that at low message rates, CBD has very limited information to exploit the messages in the queue. Further, as CBD incurs an additional overhead of processing the message queue, we expect that DAPW will be preferred for small values of message rate.

Effect of Number of Processes

The effect of number of processes on causal delivery of messages using DAPW and CBD is shown in Figure 8.10. The results are for $\epsilon = \delta = 10$ and message rate $= 0.1$. We use the following values for the number of ordinary processes: 5, 10 and 50.

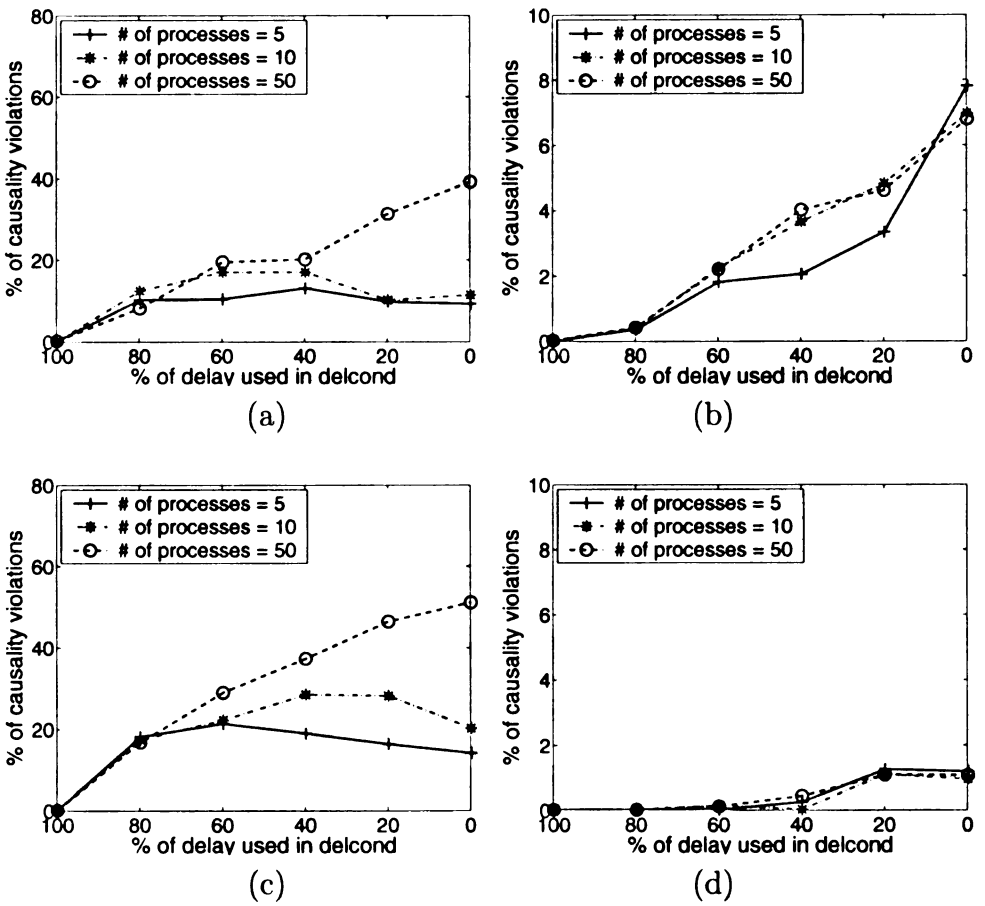


Figure 8.10: Effect of number of ordinary processes on causal delivery using (a) DAPW with message delay $N(\frac{\delta}{2}, \frac{\delta}{4})$, (b) CBD with message delay $N(\frac{\delta}{2}, \frac{\delta}{4})$, (c) DAPW with message delay $N(\frac{\delta}{4}, \frac{\delta}{8})$, and (d) CBD with message delay $N(\frac{\delta}{4}, \frac{\delta}{8})$. (Note that, the scale of DAPW and CBD graphs are different.)

As the number of processes increases, more causally dependent messages for a message m are present in the system. Thus, the probability of causality violations is higher.

Furthermore, when δ is an overestimate of message delay (i.e., message delay of

$N(\frac{\delta}{4}, \frac{\delta}{8})$), the number of causality violations in CBD is in the range of $0\% - 3\%$ (cf. Figure 8.10 (d)). Since most messages arrive within $\frac{\delta}{2}$, CBD detects/prevents most of the causality violations.

Physical Clocks Vs. Partial Timestamps

In this section, we argue that the information maintained in CBD, although small, is important in reducing the number of causality violations. Towards this end, we compute the causality violations for the case where only physical clock is used to determine when a message should be delivered. To obtain an implementation that uses physical clock alone, we set the c value and all elements in kn to 0. We call this algorithm DPC1. We also consider the algorithm DPC2 where the c value is used but kn values are reset to 0. Other points on this continuum can be obtained by maintaining a subset of the kn values in the timestamp.

Notation. By *"2 kn.e elements"* we mean that the simulation uses $kn.e_j[c.e_j]$ and $kn.e_j[c.e_j - 1]$ elements instead of the $kn.e_j$ array for an event e_j. Similarly, by *"k kn.e elements"* we mean that the simulation uses the first k $kn.e$ elements.

Figure 8.11 shows the simulation results for $\epsilon = \delta = 10$, message rate $= 0.1$ and 10 processes. (Figure 8.12 shows the results for 50 processes.)

From Figure 8.11, we observe that using physical clocks alone for causal delivery of messages is not enough. Specifically, even when ¢ $= 100\%$, DPC1 and DPC2 have around 30%–50% of causality violations. And, maintaining just 2 $kn.e$ elements provides a significant reduction in number of causality violations ($10 - 15\%$). Moreover, if we increase the number of $kn.e$ elements in the timestamp, the causality violations can be further reduced. Maintaining just 6 $kn.e$ elements gives the same result as CBD. Thus, the timestamp provides a continuum in which the application developer can choose the size of the timestamps based on the requirements. This result is especially important in sensor networks. Specifically, in MICA motes [61], the payload

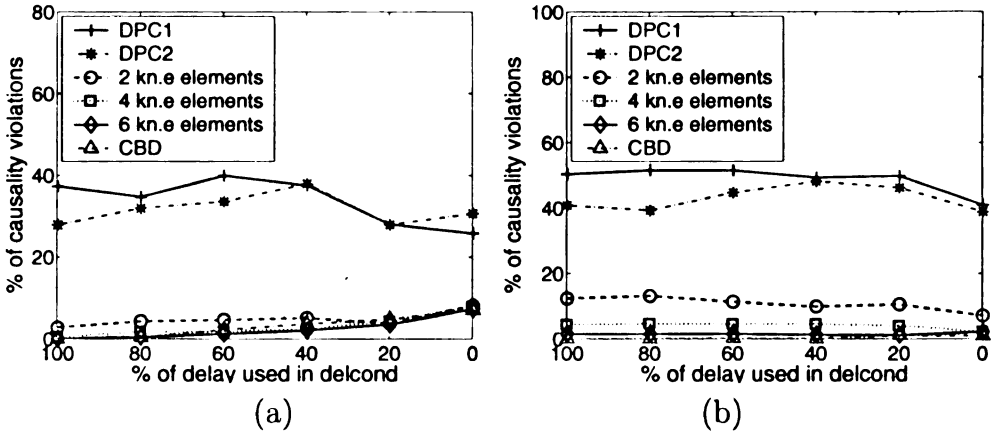


Figure 8.11: Effect of using partial timestamps with 10 processes on (a) CBD with delay $N(\frac{\delta}{2}, \frac{\delta}{4})$, (b) CBD with delay $N(\frac{\delta}{4}, \frac{\delta}{8})$.

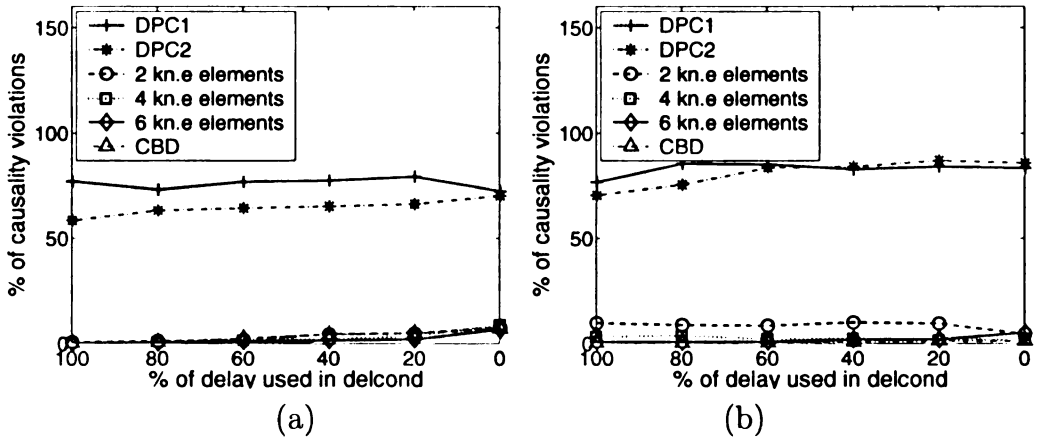


Figure 8.12: Effect of using partial timestamps with 50 processes on (a) CBD with delay $N(\frac{\delta}{2}, \frac{\delta}{4})$, (b) CBD with delay $N(\frac{\delta}{4}, \frac{\delta}{8})$.

size is just 29 bytes. Hence, the overhead in achieving approximate causal delivery should be small. Depending on the percentage of causality violations processes can handle and the overhead involved, the developer can choose an appropriate size for the timestamp. For example, choosing 2 $kn.e$ elements (i.e., 4 bytes including $rt.j$ and $c.j$) will result in $10-15\%$ causality violations (as opposed to $30-50\%$ causality violations when using the physical clocks alone). Thus, small additional information maintained in the timestamp plays a significant role in reducing the number of causality violations.

8.3 Related Work

In this section, we discuss the related work on identifying tradeoffs in sensor networks.

Energy vs. latency. Related work on analyzing the tradeoff between energy and latency in convergecast communication include [5, 62, 71, 138]. In [5], a TDMA based convergecast is investigated. Specifically, the paper proposes a tree construction algorithm for convergecast. Once the tree is constructed, it assigns schedules to the sensors for collision-free communication. By contrast, in our convergecast algorithm, we use an existing TDMA algorithm. Moreover, our solution improves the network lifetime by reducing the amount of idle listening.

In [62], a randomized convergecast algorithm is proposed. This paper identifies the lower bound on the running time for an arbitrary network. Moreover, this paper studies the energy-latency tradeoff. In [138], for an offline problem, dynamic programming based approximation solution is proposed, where the energy dissipation of sensors in the data aggregation tree is minimized. For a real-time scenario, this paper proposes an online protocol for data aggregation. Unlike [62, 138], our solution uses a TDMA based algorithm to conserve energy and to reliably switch to active mode when the network observes events of interest. Furthermore, we show that the network lifetime improves by 3 fold for a typical application.

In [71], a randomized algorithm for convergecast is proposed for ad hoc networks. One of the assumptions in [71] is that the nodes have collision detection capability. By contrast, we do not assume that collisions are detectable. Collision detection may not be possible since the sensors have limited communication capabilities and limited power.

Causality Vs. timeliness. Related work on causal delivery includes [3, 8, 18, 29, 30, 43, 58, 118, 120]. Based on [43], one cannot design solutions for approximate causal delivery in pure asynchronous systems where process speeds, process clocks and message delays are arbitrary. In other words, the underlying system must provide

some simple guarantees that enable the observer to obtain the tradeoff between causal delivery and timely delivery. Therefore, as identified in Section 8.2.1, we considered two simple guarantees: (1) clock drift among different processes in the system is bounded by ϵ and (2) messages reach their destinations within some bound (δ). The first guarantee is met by using GPS clocks, network time protocol, atomic clocks or clock synchronization programs (e.g., [8, 29, 58]). The second guarantee can by met by using protocols that characterize messages as being timely or late (e.g., [30]).

One can use solutions such as matrix clocks [120] to solve the problem of approximate causal delivery. However, this approach suffers from four problems; for one, matrix clocks do not use the underlying physical clock and, hence, cannot easily handle timely delivery. For two, the protocol in [120] cannot handle lost messages; all subsequent messages that causally depend on the lost message become undeliverable. Thirdly, the size of the timestamp used in [120] is $O(n^2)$ where n is the number of processes in the system. Such a large size could be especially problematic in systems where the number of processes is large. Finally, in [120], as the computation proceeds, the size of the timestamps grows without a bound. While solutions in [3, 18] deal with the first two problems, these solutions still suffer from the overhead of timestamps whose size is quadratic in the number of processes and whose size grows unbounded as the computation proceeds.

Another related work on causal delivery in sensor networks is [118] where a temporal message ordering service is proposed. In this approach, the senor network is modified to achieve temporal message ordering at base station. Towards this end, they send each message by multiple routes, one or more short and one or more long. By contrast, in our work, we do not assume such multiple messages. In fact, we do not modify the underlying communication in the sensor network. The reordering is done only at the base station. It follows that our approach allows one to make use of any optimizations that can be performed in the routing layer.

8.4 Chapter Summary

In this chapter, we considered the tradeoffs in sensor communication. The analysis presented in this chapter would assist the designers of sensor network protocols to fine tune their protocols in order to quickly evaluate them. We presented two tradeoff studies in convergecast communication among the sensors: (1) tradeoff between energy and latency and (2) tradeoff between causal delivery and timely delivery.

Energy Vs. latency. We presented a TDMA based convergecast algorithm for sensor networks. In this algorithm, each sensor is allowed to save energy whenever the network remains idle. And, whenever a sensor receives convergecast messages, the algorithm switches to a CSMA based protocol after forwarding the first message. Thus, it reduces the amount of idle-listening and improves the latency. We studied the performance of our algorithm and showed that it meets the requirements of a typical application (e.g., ExScal [9, 10]). We showed that active radio time is within 75% of the time required for convergecast (cf. Section 8.1.3). Moreover, for a typical application where the probability of occurrence of an event is less than $10-15\%$, we argued that our solution improves the network lifetime by approximately 3 fold.

Causality Vs. timeliness. We presented a solution for approximate causal delivery that allows the designer to choose an acceptable level of causality violations while ensuring timely delivery of messages. We discussed the effect of the parameters such as maximum clock drift, maximum message delay, and message rate on causal delivery of messages. We showed that by using physical clocks alone, the number of causality violations increase significantly. By adding new variables to the timestamp, the number of causality violations can be reduced. In other words, we showed that our solution provides a continuum such that the application developer can choose the size of timestamps used in the system based on the number of causality violations the application can tolerate. This result is especially useful in sensor networks, since

the sensors are resource constrained and the size of the payload in a message is very limited (e.g., 29 bytes in MICA). We note that maintaining just 2 $kn.e$ elements (i.e., 4 bytes) provides a significant reduction in causality violations ($10 - 15\%$) compared to using physical clocks alone ($30 - 50\%$). Hence, causal delivery of messages at the base station can be achieved easily in sensor networks with a small message overhead. To our knowledge, this result is the first of its kind for providing approximate causal delivery in sensor networks.

Chapter 9

Conclusion and Future Research

In this dissertation, we proposed theoretical models and algorithms and practical tools to enable rapid prototyping and quick deployment of sensor network protocols. In this chapter, first, in Section 9.1, we summarize the contributions made towards defending the thesis. Then, in Section 9.2, we state the impact of the proposed solutions. Finally, in Section 9.3, we discuss the open problems and future research directions.

9.1 Contributions

The contributions of this dissertation are two fold: *foundational* and *experimental*. Regarding foundational contributions, we developed algorithms for reusing abstract models and distributed programs developed using them from distributed systems literature in the context of sensor networks. Regarding experimental contributions, we developed a programming tool and middleware services that enable the designer to rapidly prototype and quickly deploy sensor network protocols.

9.1.1 Foundational Contributions

In this work, we first identified that (variations of) several problems considered in sensor networks are already addressed in the context of distributed computing and traditional networking. These include, but are not limited to, routing, spanning tree maintenance, leader election, reliable broadcast, synchronization, distributed tracking, and consensus. Also, we observed that these programs are specified in abstract models (cf. Chapter 2) that enable the designer to verify the correctness of these programs as well as to manipulate them to meet new properties using the approaches in [17, 49, 75, 103].

Since the abstract models do not meet the difficulties and challenges in sensor networks, in Chapter 3, we developed transformation algorithms that allow the designer to transform programs written in abstract models into a model consistent with sensor networks, i.e., *write all with collision* (WAC) model. We also showed that the transformation algorithms preserve the self-stabilization [38, 40] property of the original programs. In other words, we showed that if the original program is self-stabilizing then the transformed program in WAC model is also self-stabilizing.

Additionally, in our transformation algorithms, we showed that any time division multiple access (TDMA) algorithm can be effectively used to transform a program into WAC model. We argued that if the TDMA algorithm is self-stabilizing and deterministic then the transformation provides deterministic guarantees about the transformed program and preserves the self-stabilization property of the original program. To enable such deterministic transformation, in Chapter 4, we developed a self-stabilizing deterministic TDMA algorithm in WAC model. To the best of our knowledge, this is the first TDMA algorithm in WAC model to achieve such properties. Also, this is the first TDMA algorithm to enable stabilization-preserving deterministic transformations for WAC model.

9.1.2 Experimental Contributions

In order to enable the designer to rapidly prototype and quickly deploy protocols, we developed the following: (1) a tool for simplifying the construction and deployment of sensor network protocols and (2) a reliable data dissemination service for quick deployment of protocols. In addition, we identified the tradeoffs in sensor communication that allows the designers to fine tune their protocols.

Programming tool for sensor networks. In Chapter 5, we presented *ProSe*, a tool for programming sensor networks based on the theoretical foundations established in this dissertation. We showed that ProSe enables the following: (1) specify programs in abstract models (e.g., read/write model, shared-memory model) considered in distributed systems literature while hiding low-level concerns of sensor networks (e.g., message collision, race conditions, synchronization) and programming level challenges (e.g., manual stack management, manual buffer management, flow control), (2) reuse existing fault-tolerance/self-stabilizing algorithms in the context of sensor networks, (3) preserve properties such as fault-tolerance and self-stabilization of the input program, (4) abstract failure of sensors and provide support to deal with message corruption, arbitrary state corruption, and faulty/malicious sensors, and (5) automate code generation and deployment.

We used ProSe to generate sensor network binaries for (i) *network services* such as routing, leader election, and spanning tree maintenance, (2) *distributed reset service* to reset the state of the network to a consistent global state, (3) *application services* such as distributed tracking, and (4) *power management protocols* [11, 16]. Since the programs are specified in abstract models, the development time of a typical sensor network protocol is small. Furthermore, we showed that ProSe enables quick deployment since the transformation is automated.

Data dissemination protocol for network reprogramming. Sensor networks

are usually deployed in inaccessible fields and in large numbers. Hence, it is essential that the network is reprogrammed *in place*. Towards this end, in Chapter 7, we presented *Infuse*, a TDMA based reliable data dissemination protocol for sensor networks. We showed that Infuse provides quick deployment of sensor network protocols and ensures 100% reliability (both in terms of the number of sensors receiving the program and the program image being delivered). In order to deal with arbitrary message loss (due to random channel errors), we adapted sliding window based flow control mechanisms using implicit acknowledgments. Additionally, we proposed optimizations in order to reduce energy usage during dissemination.

Tradeoffs in sensor communication. To assist the designers in quickly evaluating their protocols, we identified tradeoffs in sensor communication, especially, in convergecast [13, 77, 78]. The designers can fine tune the parameters of their application by identifying appropriate operating points and protocols to achieve the desired performance levels. In Chapter 8, we identified the tradeoff between: (1) energy and latency and (2) causality and timeliness. In the first tradeoff analysis, we showed that the lifetime of the sensor network can be improved by using a simple TDMA based convergecast algorithm. For a typical sensor network, where the probability of occurrence of an event is $< 10 - 15\%$, this algorithm improves the lifetime by approximately 3 fold. In the second tradeoff analysis, we showed that the designer can choose the time a message has to be buffered and the size of the timestamp used to achieve causal delivery depending on the level of causality violations the application can handle.

9.2 Impact

In this section, we discuss the impact of this dissertation. First, we note that most of the current work in sensor networks is focused on developing: (1) applications for

a particular task (e.g., long-range surveillance [9, 10], habitat monitoring [98, 124], environment monitoring [93]) and (2) middleware services (e.g., routing [25, 135], power management [50, 110], TDMA [57, 60, 80]). These solutions focus on dealing with constraints imposed by sensor networks.

To advance the state of the art in sensor networks, it is necessary to develop abstractions that hide these constraints from the designer, thereby, enabling them to simplify the protocol design. The importance of the need for such abstractions has been identified in recent work [81, 97, 105, 131]. In this dissertation, we proposed mechanisms to reuse abstract models considered in distributed systems literature. Such models will make the protocol design highly intuitive and concise. Furthermore, such models allow reuse of existing tools for verification and manipulation to add new properties such as fault-tolerance. For these reasons, this dissertation work will make it significantly simpler to design and deploy sensor network protocols as well as to gain confidence in them.

Since the abstract models hide the low-level details from the designer, it would enable the transition where protocols are designed by *domain experts* rather than *experts in sensor networks.* In addition, with the improvements in underlying wireless technology, it would be possible to improve the efficiency of transformation algorithms. As a result, this would improve the applications using them. Thus, this work not only enables rapid prototyping and quick deployment of sensor network protocols but also has the potential to allow the designer to automatically benefit from future advances in underlying sensor network technology.

9.3 Future Research

This dissertation has created the possibility of several new research directions. Some of these are outlined below.

Foundational aspects. Regarding foundational aspects, we will focus on developing efficient transformation algorithms and designing fault-local stabilizing protocols for sensor networks.

- *Efficient transformation algorithms for WAC model.* In Chapter 3, we showed the feasibility of stabilization preserving transformations for WAC model. Towards this end, we showed that a TDMA algorithm in WAC model can be effectively used to obtain such transformation. The TDMA algorithm proposed in Chapter 4 ensures stabilization preserving deterministic transformation for WAC model. However, this algorithm is sequential in nature. In other words, this algorithm assigns time slots to each sensor in a sequential fashion. Existing TDMA algorithms [7, 22, 26, 57, 60, 80, 121] for concurrent slot assignment are randomized in nature and/or assume that the network has a grid topology. Hence, an interesting future research direction is to investigate whether deterministic concurrent coloring/time slot assignment is feasible in WAC model. We expect that such algorithms allow the sensors to recover from arbitrary state corruption quickly.

- *Transformation algorithms for WAC model with collision detection capability.* Current transformation algorithms for WAC model assume that the collisions are not detectable. Collision detection in wireless radio networks have been addressed in [19, 20, 24]. Specifically, in [19, 20], the authors propose a probabilistic algorithm that allows one to emulate a single-hop radio communication with collision detection capability on a multi-hop radio network without collision detection capability. The authors show how this emulation can be applied

in the design of efficient algorithms for multi-hop networks (e.g., leader election). And, in [24], the authors augment the sensors with collision detection. Unlike Ethernet networks [100] that allow transmitters to detect collision, the collision detectors in [24] allow only receivers to detect them. Such collision detectors are shown to be useful in solving problems such as consensus. Based on the results from [19, 20, 24], an interesting future work is to develop efficient transformations for WAC model using such collision detection approaches.

- *Fault-local stabilization.* In order to improve the stability, availability, and scalability of the network, it is desirable that the services designed for sensor networks are fault-local stabilizing. Intuitively, a service is fault-local stabilizing if the time taken to stabilize is proportional to the size of the network perturbed by faults. Fault-local stabilization ensures that the effect of faults is contained locally. Hence, another future direction to this dissertation is to design middleware services that achieves fault-local stabilization in the presence of faults. These services will be useful in designing various network protocols and also in the transformation for WAC model. Moreover, these services will assist in achieving fault-local stabilization in the protocols and also in the transformed programs.

Experimental aspects. Regarding experimental aspects, we plan to augment ProSe with several new features to simplify the design of adaptive sensor network protocols.

- *High-level primitives to simplify protocol design.* In this dissertation, we developed a tool that allows one to hide several low-level concerns of sensor networks during prototyping. To simplify the design of sensor network protocols, high-level primitives are highly useful. For example, primitives such as distance-k broadcast, distance-k consensus, and distance-k leader election are found to be

the basic building blocks of sensor networks in [24, 34, 94, 111, 132]. In [34, 111], the authors argue that a single-hop reliable broadcast is an essential primitive in order to reduce the complexity of the application software. In [24], the authors show the importance of multi-hop consensus in sensor networks. With these primitives, the design of several protocols (e.g., in-network aggregation, power management, target detection and classification) can be simplified. Therefore, an important future work is to identify such high-level primitives and develop reliable and efficient protocols for them. Moreover, it would be worthwhile to implement such primitives with ProSe to enable rapid prototyping and quick deployment of sensor networks.

- *Adding an adaptation framework with ProSe.* Since it is desirable that the sensors adapt to given environment conditions, we plan to provide support for component adaptation in sensor networks that ensures transparency and provides assurance guarantees. To achieve this goal, we will investigate how to ensure that the adaptation succeeds at all sensors and provides the required level of functionality during adaptation. This work will leverage on existing tools and protocols (e.g., [11, 15, 63, 72, 79, 82, 83, 86]) in generating sensor network components and deploying them. Also, this work will extend ProSe to build an adaptation framework for sensor networks.

Bibliography

Bibliography

[1] M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, 1991.

[2] T. Abdelzaher, B. M. Blum, Q. Cao, Y. Chen, D. Evans, J. George, S. George, L. Gu, T. He, S. Krishnamurthy, L. Luo, S. H. Son, J. Stankovic, R. Stoleru, and A. D. Wood. EnviroTrack: Towards an environmental computing paradigm for distributed sensor networks. *In Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS)*, pages 582–589, March 2004.

[3] F. Adelstein and M. Singhal. Real-time causal message ordering in multimedia systems. *In Proceedings of the International Conference on Distributed Computing Systems (ICDCS)*, pages 36–43, May-June 1995.

[4] A. Adya, J. Howell, M. Theimer, W. J. Bolosky, and J. R. Douceur. Cooperative task management without manual stack management or, event driven programming is not the opposite of threaded programming. *In Proceedings of 2002 USENIX Annual Technical Conference*, June 2002.

[5] V. Annamalai, S. K. S. Gupta, and L. Schwiebert. On tree-based convergecasting in wireless sensor networks. *In Proceedings of the Wireless Communications and Networking Conference (WCNC)*, 3:1942–1947, March 2003.

[6] G. Antonoiu and P. K. Srimani. Mutual exclusion between neighboring nodes in an arbitrary system graph tree that stabilizies using read/write atomicity. *In Proceedings of 5th International Euro-Par Conference, Euro-par'99 Parallel Processing*, LNCS:1685:823–830, August-September 1999.

[7] K. Arisha, M. Youssef, and M. Younis. Energy-aware TDMA-based MAC for sensor networks. *In Proceedings of the IEEE Workshop on Integrated Management of Power Aware Communications, Computing and Networking (IMPACCT)*, May 2002.

[8] A. Arora, S. Dolev, and M. G. Gouda. Maintaining digital clocks in step. *Parallel Processing Letters*, 1(1):11–18, September 1991.

[9] A. Arora, P. Dutta, S. Bapat, V. Kulathumani, H. Zhang, V. Naik, V. Mittal, H. Cao, M. Demirbas, M. Gouda, Y-R. Choi, T. Herman, S. S. Kulkarni,

M. Arumugam, M. Nesterenko, A. Vora, and M. Miyashita. A line in the sand: A wireless sensor network for target detection, classification, and tracking. *Computer Networks*, 46(5):605–634, 2004.

[10] A. Arora, R. Ramnath, E. Ertin, P. Sinha, S. Bapat, V. Naik, V. Kulathumani, H. Zhang, H. Cao, M. Sridharan, S. Kumar, N. Seddon, C. Anderson, T. Herman, N. Trivedi, C. Zhang, M. Nesterenko, R. Shah, S. Kulkarni, M. Arumugam, L. Wang, M. Gouda, Y. Choi, D. Culler, P. Dutta, C. Sharp, G. Tolle, M. Grimmer, B. Ferriera, and K. Parker. ExScal: Elements of extreme scale wireless sensor network. *In Proceedings of the International Conference on Real-Time and Embedded Computing Systems and Applications (RTCSA)*, August 2005.

[11] M. Arumugam and S. S. Kulkarni. Programming sensor networks made easy. Technical Report MSU-CSE-05-25, Department of Computer Science, Michigan State University, September 2005.

[12] M. Arumugam and S. S. Kulkarni. Self-stabilizing deterministic tdma for sensor networks. *In Proceedings of the International Conference on Distributed Computing and Internet Technology (ICDCIT)*, pages 69–81, December 2005.

[13] M. Arumugam and S. S. Kulkarni. Tradeoffs between energy and latency for convergecast. *In Proceedings of the Second International Workshop on Networked Sensing Systems (INSS)*, June 2005.

[14] M. Arumugam and S. S. Kulkarni. Self-stabilizing deterministic time division multiple access for sensor networks. *AIAA Journal of Aerospace Computing, Information, and Communication (JACIC)*, 3:403–419, 2006.

[15] M. Arumugam, S. S. Kulkarni, and K. N. Biyani. Adaptation in sensor-actuator networks: A case study. *In Proceedings of the Third International Conference on Networked Sensing Systems (INSS)*, June 2006.

[16] M. Arumugam, L. Wang, and S. S. Kulkarni. A case study on prototyping power management protocols for sensor networks. *In Proceedings of the Eighth International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, November 2006, To Appear.

[17] P. C. Attie, A. Arora, and E. A. Emerson. Synthesis of fault-tolerant concurrent programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 26(1):125–185, January 2004.

[18] R. Baldoni, M. Mostefaoui, and M. Raynal. Causal deliveries in unreliable networks with real-time delivery constraints. *Journal of Real-Time Systems*, 10 (3):1–18, 1996.

[19] R. Bar-Yehuda, O. Goldreich, and A. Itai. On the time-complexity of broadcast in radio networks: An exponential gap between determinism and randomization.

In Proceedings of the 6th Annual SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC), pages 98–108, August 1987.

[20] R. Bar-Yehuda, O. Goldreich, and A. Itai. Efficient emulation of single-hop radio network with collision detection on multi-hop radio network with no collision detection. *Distributed Computing*, 5(2):67–71, September 1991.

[21] P. Buonadonna, D. Gay, J. Hellerstein, W. Hong, and S. Madden. TASK: Sensor network in a box. *In Proceedings of the European Workshop on Wireless Sensor Networks (EWSN)*, pages 133–144, January-February 2005.

[22] C. Busch, M. M-Ismail, F. Sivrikaya, and B. Yener. Contention-free MAC protocols for wireless sensor networks. *In Proceedings of the 18th Conference on Distributed Computing (DISC)*, pages 245–259, October 2004.

[23] K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, 1988.

[24] G. Chockler, M. Demirbas, S. Gilbert, C. Newport, and T. Nolte. Consensus and collision detectors in wireless ad hoc networks. *In Proceedings of the 24th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*, pages 197–206, July 2005.

[25] Y-R. Choi, M. G. Gouda, H. Zhang, and A. Arora. Stabilization of grid routing in sensor networks. *AIAA Journal of Aerospace Computing, Information, and Communication (JACIC)*, To Appear.

[26] V. Claesson, H. Lönn, and N. Suri. Efficient TDMA synchronization for distributed embedded systems. *In Proceedings of the 20th IEEE Symposium on Reliable Distributed Systems (SRDS)*, pages 198–201, October 2001.

[27] J. A. Cobb and M. G. Gouda. Balanced routing. *In Proceedings of the International Conference on Network Protocols (ICNP)*, pages 277–284, October 1997.

[28] A. M. Costello and G. Varghese. Self-stabilization by window washing. *In Proceedings of the Symposium on Principles of Distributed Computing (PODC)*, pages 34–44, May 1996.

[29] J. M. Couvreur, N. Francez, and M. G. Gouda. Asynchronous unison. *In Proceedings of the 12th International Conference on Distributed Computing Systems (ICDCS)*, pages 486–493, June 1992.

[30] F. Cristian and C. Fetzer. The timed asynchronous distributed system model. *IEEE Transactions on Parallel and Distributed Systems*, 10(6):642–657, 1999.

[31] *Mica 2 Datasheet.* Crossbow Technology, Inc., . Available at: `http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/MICA2_Datasheet.pdf`.

[32] *Mica Z Datasheet.* Crossbow Technology, Inc., . Available at: `http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/MICAZ_Datasheet.pdf`.

[33] *Mote In-Network Programming User Reference Version 20030315.* Crossbow Technology, Inc., 2003. Available at: `http://www.xbow.com/Support/Support_pdf_files/Xnp.pdf`.

[34] D. Culler, P. Dutta, C. T. Eee, R. Fonseca, J. Hui, P. Levis, J. Polastre, S. Shenker, I. Stoica, G. Tolle, and J. Zhao. Towards a sensor network architecture: Lowering the waistline. *In Proceedings of the Tenth Workshop on Hot Topics in Operating Systems (HotOS X)*, June 2005.

[35] P. Danturi, M. Nesterenko, and S. Tixeuil. Self-stabilizing philosophers with generic conflicts. *In Proceedings of the Eighth International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, November 2006, To Appear.

[36] A. K. Datta, C. Johnen, F. Petit, and V. Villain. Self-stabilizing depth-first token circulation in arbitrary rooted networks. *Distributed Computing*, 13:207–218, 2000.

[37] M. Demirbas, A. Arora, and M. Gouda. Pursuer-evader tracking in sensor networks. In S. Phoha, T. F. La Porta, and C. Griffin, editors, *Sensor Network Operations.* Wiley-IEEE Press, May 2006.

[38] E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 1974.

[39] E. W. Dijkstra. *A Discipline of Programming.* Prentice Hall PTR, 1997.

[40] S. Dolev. *Self-Stabilization.* The MIT Press, 2000.

[41] P. Dutta, M. Grimmer, A. Arora, S. Bibyk, and D. Culler. Design of a wireless sensor network platform for detecting rare, random, and ephemeral events. *In Proceedings of the Fourth International Symposium on Information Processing in Sensor Networks (IPSN), Special Track on Platform Tools and Design Methods for Networked Embedded Sensors*, pages 497–502, April 2005.

[42] J. Fidge. Timestamps in message-passing systems that preserve the partial ordering. *In Proceedings of the 11th Australian Computer Science Conference*, 10(1):56–66, Feb 1988.

[43] M. J. Fischer, N. A. Lynch, and M. S. Peterson. Impossibility of distributed consensus with one faulty processor. *Journal of the ACM*, 32(2):374–382, 1985.

[44] C-L. Fok, G-C. Roman, and C. Lu. Rapid development and flexible deployment of adaptive wireless sensor network applications. *In Proceedings of the 25th International Conference on Distributed Computing Systems (ICDCS)*, pages 653–662, June 2005.

[45] S. Ganeriwal, R. Kumar, and M. Srivastava. Timing sync protocol for sensor networks. *In Proceedings of the First International Conference on Embedded Networked Sensor Systems (SenSys)*, pages 138–149, November 2003.

[46] D. Ganesan, B. Krishnamachari, A. Woo, D. Culler, D. Estrin, and S. Wicker. An empirical study of epidemic algorithms in large scale multihop wireless networks. Technical Report IRB-TR-02-003, Intel Research, March 2002.

[47] D. Gay, P. Levis, R. voh Behren, M. Welsh, E. Brewer, and D. Culler. The nesC language: A holistic approach to networked embedded systems. *In Proceedings of ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI)*, pages 1–11, June 2003.

[48] M. Gouda and F. Haddix. The linear alternator. *In Proceedings of the 3rd Workshop on Self-Stabilizing Systems (WSS)*, pages 31–47, August 1997.

[49] M. Gouda and F. Haddix. The alternator. *In Proceedings of the 1999 ICDCS Workshop on Self-Stabilizing Systems (WSS)*, pages 48–53, June 1999.

[50] M. G. Gouda, Y-R. Choi, and A. Arora. Sentries and sleepers in sensor networks. *In Proceedings of the 8th International Conference on Principles of Distributed Systems (OPODIS)*, pages 384–399, December 2004.

[51] M. G. Gouda and T. M. McGuire. Accelerated heartbeat protocols. *In Proceedings of the 18th International Confernece on Distributed Computing Systems (ICDCS)*, pages 202–209, May 1998.

[52] M. G. Gouda and N. J. Multari. Stabilizing communication protocols. *IEEE Transactions on Computers*, 40(4):448–458, 1991.

[53] B. Greenstein, E. Kohler, and D. Estrin. A sensor network application construction kit (snack). *In Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems (SenSys)*, pages 69–80, November 2004.

[54] C. Gui and P. Mohapatra. Power conservation and quality of surveillance in target tracking sensor networks. *In Proceedings of the Tenth Annual International Conference on Mobile Computing and Networking (MobiCom)*, pages 129–143, September-October 2004.

[55] R. Gummadi, O. Gnawali, and R. Govindan. Macro-programming wireless sensor networks using kairos. *In Proceedings of the First IEEE International Confernece on Distributed Computing in Sensor Systems (DCOSS)*, pages 126–140, June-July 2005.

[56] T. He, C. Huang, B. Blum, J. Stankovic, and T. Abdelzaher. Range-free localization schemes for large scale sensor networks. *In Proceedings of the 9th Annual International Conference on Mobile Computing and Networking (MobiCom)*, pages 81–95, September 2003.

[57] W. B. Heinzelman, A. P. Chandrakasan, and H. Balakrishnan. An application-specific protocol architecture for wireless microsensor networks. *IEEE Transactions on Wireless Communications*, 1(4):660–670, October 2002.

[58] T. Herman. Models of self-stabilization and sensor networks. *In Proceedings of the 5th International Workshop on Distributed Computing (IWDC)*, LNCS:2918:205–214, December 2003.

[59] T. Herman. NestArch: Prototype time synchronization service. `http://www.ai.mit.edu/people/sombrero/nestwiki/index/ComponentTimeSync`, 2003.

[60] T. Herman and S. Tixeuil. A distributed TDMA slot assignment algorithm for wireless sensor networks. *In Proceedings of the First International Workshop on Algorithmic Aspects of Wireless Sensor Networks (ALGOSENSORS)*, LNCS:3121:45–58, 2004.

[61] J. Hill and D. E. Culler. Mica: A wireless platform for deeply embedded networks. *IEEE Micro*, 22(6):12–24, 2002.

[62] Q. Huang and Y. Zhang. Radial coordination for convergecast in wireless sensor networks. *In Proceedings of the First International Workshop on Embedded Networked Sensors*, November 2004.

[63] J. W. Hui and D. Culler. The dynamic behavior of a data dissemination protocol for network programming at scale. *In Proceedings of the Second International Conference on Embedded Networked Sensor Systems (SenSys)*, pages 81–94, November 2004.

[64] C. Intanagonwiwat, R. Govindan, and D. Estrin. Directed Diffusion: A scalable and robust communication paradigm for sensor networks. *In Proceedings of the Sixth International Conference on Mobile Computing and Networks (MobiCOM)*, pages 56–67, August 2000.

[65] K. Ioannidou. Transformations of self-stabilizing algorithms. *In Proceedings of the 16th International Conference on Distributed Computing (DISC), Springer-Verlag*, LNCS:2508:103–117, October 2002.

[66] C. Johnen, G. Alari, J. Beauquier, and A. K. Datta. Self-stabilizing depth-first token passing on rooted networks. *In Proceedings of the 11th International Workshop on Distributed Algorithms (WDAG)*, LNCS:1320:260–274, September 1997.

[67] H. Kakugawa and M. Yamashita. Self-stabilizing local mutual exclusion on networks in which process identifiers are not distinct. *In Proceedings of the 21st Symposium on Reliable Distributed Systems (SRDS)*, pages 202–211, October 2002.

[68] C. Karlof, N. Sastry, and D. Wagner. TinySec: A link layer security architecture for wireless sensor networks. *In Proceedings of the Second ACM Conference on Embedded Networked Sensor Systems (SenSys)*, pages 162–175, November 2004.

[69] S. K. Kasera, G. Hjálmtýsson, D. F. Towsley, and J. F. Kurose. Scalable reliable multicast using multiple multicast channels. *IEEE/ACM Transactions on Networking*, 8(3):294–310, 2000.

[70] O. Kasten and K. Römer. Beyond event handlers: Programming sensor networks with attributed state machines. *In Proceedings of the Fourth International Symposium on Information Processing in Sensor Networks (IPSN)*, pages 45–52, April 2005.

[71] A. Kesselman and D. Kowalski. Fast distributed algorithm for convergecast in ad hoc geometric radio networks. *In Proceedings of the 2nd International Conference on Wireless On demand Network Systems and Services*, pages 119–124, January 2005.

[72] Y. M. Kim, A. Arora, V. Kulathumani, M. Arumugam, and S. S. Kulkarni. On the effect of faults in vibration control of fairing structures. *In Proceedings of the ASME Conference on Multibody Systems, Nonlinear Dynamics, and Control (MSNDC), Smart Sensor Technology, Control, Diagnosis, and Distributed Networking (SMART-SENS) track*, September 2005.

[73] S. O. Krumke, M. V. Marathe, and S. S. Ravi. Models and approximation algorithms for channel assignment in radio networks. *Wireless networks*, 7(6): 575–584, November 2001.

[74] J. Kulik, W. R. Heinzelman, and H. Balakrishnan. Negotiation-based protocols for disseminating information in wireless sensor networks. *Wireless Networks*, 8(2-3):169–185, 2002.

[75] S. S. Kulkarni and A. Arora. Automating the addition of fault-tolerance. *In Proceedings of the 6th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT)*, pages 82–93, September 2000.

[76] S. S. Kulkarni and M. Arumugam. Collision-free communication in sensor networks. *In Proceedings of the Sixth Symposium on Self-stabilizing Systems (SSS), Springer*, LNCS:2704:17–31, June 2003.

[77] S. S. Kulkarni and M. Arumugam. Approximate causal observer. *In Proceedings of the First International Workshop on Networked Sensing Systems (INSS)*, pages 123–128, June 2004.

[78] S. S. Kulkarni and M. Arumugam. Approximate causal observer. *Transactions of Society of Instrument and Control Engineers (SICE), Special Issue on Networked Sensing Systems*, E-S-1(1):33–42, January 2006.

[79] S. S. Kulkarni and M. Arumugam. Infuse: A TDMA based data dissemination protocol for sensor networks. *International Journal on Distributed Sensor Networks (IJDSN)*, 2(1):55–78, 2006.

[80] S. S. Kulkarni and M. Arumugam. SS-TDMA: A self-stabilizing MAC for sensor networks. In S. Phoha, T. F. La Porta, and C. Griffin, editors, *Sensor Network Operations*. Wiley-IEEE Press, May 2006.

[81] S. S. Kulkarni and M. Arumugam. Transformations for write-all-with-collision model. *Computer Communications (Elsevier)*, 29(2):183–199, January 2006.

[82] S. S. Kulkarni and K. N. Biyani. Correctness of dynamic adaptation. *In Proceedings of the 7th International Symposium on Component Based Software Engineering (CBSE)*, LNCS:3054:48–58, May 2004.

[83] S. S. Kulkarni, K. N. Biyani, and M. Arumugam. Composing distributed fault-tolerance components. *In Proceedings of the Workshop on Principles of Dependable Systems (PoDSy)*, pages W127–W136, June 2003.

[84] S. S. Kulkarni and A. Ebnenasir. A framework for automatic synthesis of fault-tolerance. Technical Report MSU-CSE-03-16, Michigan State University, July 2003.

[85] S. S. Kulkarni and Ravikant. Stabilizing causal deterministic merge. *In Proceedings of the Fifth International Workshop on Self-Stabilizing Systems (WSS), Springer*, LNCS:2194:183–199, October 2001.

[86] S. S. Kulkarni and L. Wang. MNP: Multihop network reprogramming service for sensor networks. *In Proceedings of the 25th International Conference on Distributed Computing Systems (ICDCS)*, pages 7–16, June 2005.

[87] L. Lamport. Time, clocks, and the ordering of events in a disributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[88] P. Levis and D. Culler. Maté: A tiny virtual machine for sensor networks. *ACM SIGOPS Operating Systems Review*, 36(5):85–95, December 2002.

[89] P. Levis, N. Lee, M. Welsh, and D. Culler. TOSSIM: Accurate and scalable simulation of entire tinyOS applications. *In Proceedings of the First International Conference on Embedded Networed Sensor Systems (SenSys)*, pages 126–137, November 2003.

[90] P. Levis, N. Patel, S. Shenker, and D. Culler. Trickle: A self-regulating algorithm for code propagation and maintenance in wireless sensor network. *In Proceedings of the First USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI)*, pages 1–14, March 2004.

[91] J. Liu, M. Chu, J. Liu, J. Reich, and F. Zhao. State-centric programming for sensor-actuator network systems. *Pervasive Computing*, 2(4):50–62, October-December 2003.

[92] E. L. Lloyd and S. Ramanathan. On the complexity of distance-2 coloring. *In Proceedings of the Fourth International Conference on Computing and Information (ICCI)*, pages 71–74, May 1992.

[93] J. Lundquist, D. Cayan, and M. Dettinger. Meterology and hydrology in Yosemite national park: A sensor network application. *In Proceedings of the 2nd International Workshop on Information Processing In Sensor Networks (IPSN)*, pages 518–528, April 2003.

[94] L. Luo, T. Abdelzaher, T. He, and J. Stankovic. Envirosuite: An environmentally immersive programming framework for sensor networks. *ACM Transactions on Embedded Computing Systems (TECS*, 2006.

[95] N. Lynch and F. Vaandrager. Forward and backward simulations – Part 1: Untimed systems. *Information and Computation*, 121(2):214–233, September 1995. Also, Technical Memo MIT/LCS/TM-486.b, Laboratory for Computer Science, Massachusetts Institute of Technology.

[96] S. Madden, M. Franklin, J. Hellerstein, and W. Hong. TinyDB: An acquisitional query processing system for sensor networks. *ACM Transactions on Database Systems (TODS)*, 30(1):122–173, March 2005.

[97] G. Mainland, L. Kang, S. Lahaie, D. C. Parkes, and M. Welsh. Using virtual markets to program global behavior in sensor networks. *In Proceedings of the ACM SIGOPS European Workshop*, September 2004.

[98] A. Mairwaring, J. Polastre, R. Szewczyk, D. Culler, and J. Anderson. Wireless sensor networks for habitat monitoring. *In Proceedings of the First ACM International Workshop On Wireless Sensor Networks and Applications (WSNA)*, pages 88–97, September 2002.

[99] F. Mattern. Virtual time and global states of distributed systems. *Parallel and Distributed Algorithms*, pages 215–226, 1989.

[100] R. M. Metcalfe and D. R. Boggs. Ethernet: distributed packet switching for local computer networks. *Communications of the ACM*, 19(7):395–404, 1976.

[101] M. Mizuno and M. Nesterenko. A transformation of self-stabilizing serial model programs for asynchronous parallel computing environments. *Information Processing Letters*, 66(6):285–290, 1998.

[102] V. Naik, A. Arora, P. Sinha, and H. Zhang. Sprinkler: A reliable and energy-efficient data dissemination service for wireless embedded devices. *In Proceedings of the 26th IEEE Real-Time Systems Symposium (RTSS), Real-Time Communication and Sensor Network Track*, pages 277–286, December 2005.

[103] M. Nesterenko and A. Arora. Self-stabilization preserving atomicity refinements. *Journal of Parallel and Distributed Computing*, 62(5):766–791, 2002.

[104] R. Newton, Arvind, and M. Welsh. Building up to macroprogramming: An intermediate language for sensor networks. *In Proceedings of the Fourth International Symposium on Information Processing in Sensor Networks (IPSN)*, pages 37–44, April 2005.

[105] R. Newton and M. Welsh. Region streams: Functional macroprogramming for sensor networks. *In Proceedings of the Workshop on Data Management for Sensor Networks (DMSN)*, August 2004.

[106] S.-Y. Ni, Y.-C. Tseng, Y.-S. Chen, and J.-P. Sheu. The broadcast storm problem in a mobile ad hoc network. *Wireless Networks*, 8(2-3):153–167, 2002.

[107] S-J. Park, R. Vedantham, R. Sivakumar, and I. F. Akyildiz. A scalable approach for reliable downstream data delivery in wireless sensor networks. *In Proceedings of the ACM International Symposium on Mobile Ad Hoc Networking and Computing (MobiHoc)*, pages 78–89, May 2004.

[108] F. Petit. Fast self-stabilizing depth-first token circulation. *In Proceedings of the 5th Workshop on Self-Stabilizing Systems (WSS), Springer*, LNCS:2194: 200–215, October 2001.

[109] F. Petit and V. Villain. Color optimal self-stabilizing depth-first token circulation. *In Proceedings of the International Symposium on Parallel Architectures, Algorithms, and Networks (ISPAN)*, pages 317–323, December 1997.

[110] J. Polastre, J. Hill, and D. Culler. Versatile low power media access for wireless sensor networks. *In Proceedings of the Second International Conference on Embedded Networked Sensor Systems (SenSys)*, pages 95–107, November 2004.

[111] J. Polastre, J. Hui, P. Levis, J. Zhao, D. Culler, S. Shenker, and I. Stoica. A unifying link abstraction for wireless sensor networks. *In Proceedings of the Third International Conference on Embedded Networked Sensing Systems (SenSys)*, pages 76–89, November 2005.

[112] J. Polastre, R. Szewcyzk, C. Sharp, and D. Culler. The mote revolution: Low power wireless sensor network devices. *In Proceedings of the 16th Symposium on High Performance Chips (HotChips)*, August 2004.

[113] J. Polastre, R. Szwecyzk, and D. Culler. Telos: Enabling ultra-low power wireless research. *In Proceedings of the Fourth International Symposium on Information Processing in Sensor Networks (IPSN), Special Track on Platform Tools and Design Methods for Network Embedded Sensors*, pages 364–369, April 2005.

[114] S. Ramanathan and E. L. Lloyd. Scheduling algorithms for multihop radio networks. *IEEE/ACM Transactions on Networking*, 1(2):166–177, April 1993.

[115] N. Reijers and K. Langendoen. Efficient code distribution in wireless sensor networks. *In Proceedings of the Second ACM International Workshop on Wireless Sensor Networks and Applications (WSNA)*, September 2003.

[116] S. Ren, Q. Li, H. Wang, X. Chen, and X. Zhang. Analyzing object detection quality under probabilistic coverage in sensor networks. *In Proceedings of the 13th International Workshop on Quality of Service (IWQoS)*, pages 107–122, June 2005.

[117] M. Ringwald and K. Römer. BitMAC: A deterministic, collision-free, and robust MAC protcol for sensor networks. *In Proceedings of the European Workshop on Sensor Networks (EWSN)*, pages 57–69, January-February 2005.

[118] K. Römer. Temporal message ordering in wireless sensor networks. *In Proceedings of the Second Mediterranean Workshop on Ad-Hoc Networks (MED-HOC NET)*, June 2003.

[119] G. Simon, P. Volgyesi, M. Maroti, and A. Ledeczi. Simulation-based optimization of communication protocols for large-scale wireless sensors networks. *In Proceedings of the IEEE Aerospace Conference*, March 2003. Simulator (*Prowler*) available at: `http://www.isis.vanderbilt.edu/projects/nest/prowler`.

[120] M. Singhal and N. Shivaratri. *Advanced Concepts in Operating Systems: Distributed, Database, and Multiprocessor Operating Systems.* McGraw-Hill Publishing Company, New York, 1994.

[121] K. Sohrabi and G. J. Pottie. Performance of a novel self-organization protocol for wireless ad-hoc sensor networks. *In Proceedings of IEEE Vehicular Technology Conference*, pages 1222–1226, 1999.

[122] F. Stann and J. Heidemann. RMST: Reliable data transport in sensor networks. *In Proceedings of the First International Workshop on Sensor Net Protocols and Applications*, pages 102–112, April 2003.

[123] T. Stathopoulos, J. Heidemann, and D. Estrin. A remote code update mechanism for wireless sensor networks. Technical Report CENS-TR-30, University of California, Los Angeles, Center for Embedded Networked Computing, November 2003.

[124] R. Szewcyzk, A. M. Mainwairing, J. Polastre, J. Anderson, and D. Culler. An analysis of a large scale habitat monitoring application. *In Proceedings of the Second International Conference on Embedded Networked Sensor Systems (SenSys)*, pages 214–226, November 2004.

[125] A. S. Tanenbaum. *Computer Networks.* Prentice Hall, 4th edition, March 2003.

[126] D. Tian and N. D. Georganas. A node scheduling scheme for energy conservation in large wireless sensor networks. *Wireless Communications and Mobile Computing Journal*, 3(2):271–290, May 2003.

[127] J. van Greunen and J. Rabaey. Lightweight time synchronization for sensor networks. *In Proceedings of the Workshop on Wireless Sensor Networks and Applications (WSNA)*, September 2003.

[128] C.-Y. Wan, A. T. Campbell, and L. Krishnamurthy. Reliable transport for sensor networks: PSFQ - pump slowly fetch quickly paradigm. *Wireless Sensor Networks*, pages 153–182, 2004.

[129] L. Wang and S. S. Kulkarni. Sacrificing a little coverage can substantially increase network lifetime. *In Proceedings of Third Annual IEEE Communications Society Conference on Sensor, Mesh, and Ad Hoc Communications and Networks (SECON)*, September 2006.

[130] X. Wang, G. Xing, Y. Zhang, C. Lu, R. Pless, and C. Gill. Integrated coverage and connectivity configuration in wireless sensor networks. *In Proceedings of the First International Conference on Embedded Networked Sensing Systems*, pages 28–39, November 2003.

[131] M. Welsh and G. Mainland. Programming sensor networks using abstract regions. *In Proceedings of the 1st Symposium on Networked Systems Design and Implementation (NSDI)*, pages 29–42, March 2004.

[132] K. Whitehouse, C. Sharp, E. Brewer, and D. Culler. Hood: A neighborhood abstraction for sensor networks. *In Proceedings of the 2nd International Conference on Mobile Systems, Applications, and Services (MobiSys)*, June 2004.

[133] K. Whitehouse, F. Zhao, and J. Liu. Semantic streams: A framework for declarative queries and automatic data interpretation. Technical Report MSR-TR-2005-45, Microsoft Research, April 2005.

[134] A. Woo and D. Culler. A transmission control scheme for media access in sensor networks. *In Proceedings of the Seventh Annual International Conference on Mobile Computing and Networking (MobiCom)*, pages 221–235, July 2001.

[135] A. Woo and D. Culler. Taming the challenges of reliable multihop routing in sensor networks. *In Proceedings of the 1st International Conference on Embedded Networked Sensor Systems (SenSys)*, pages 14–27, November 2003.

[136] T. Yan, T. He, and J. A. Stankovic. Differentiated surveillance for sensor networks. *In Proceedings of the First International Conference on Embedded Networked Sensing Systems (SenSys)*, pages 51–62, November 2003.

[137] F. Ye, G. Zhong, J. Cheng, S. W. Lu, and L. X. Zhang. PEAS: A robust energy conserving protocol for long-lived sensor networks. *In Proceedings of the 23rd*

International Conference on Distributed Computing Systems (ICDCS), pages 28–37, May 2003.

[138] Y. Yu, B. Krishnamachari, and V. K. Prasanna. Energy-latency tradeoffs for data gathering in wireless sensor networks. *In Proceedings of the 23rd Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, March 2004.

[139] X. Zeng, R. Bagrodia, and M. Gerla. GloMoSim: A library for parallel simulation of large scale wireless networks. *In Proceedings of the Workshop on Parallel and Distributed Simulations*, May 2002.

[140] H. Zhang, A. Arora, Y-R. Choi, and M. G. Gouda. Reliable bursty convergecast in wireless sensor networks. *In Proceedings of the 6th ACM International Symposium on Mobile Ad Hoc Networking and Computing (MobiHoc)*, pages 266–276, May 2005.

[141] M. Zuniga and B. Krishnamachari. Analyzing the transitional region in low power wireless links. *In Proceedings of the IEEE Conference on Sensor and Ad hoc Communications and Networks (SECON)*, October 2004.