ON RELIABLE AND SCALABLE MANAGEMENT OF WIRELESS SENSOR NETWORKS

DISSERTATION

Presented in Partial Fulfillment of the Requirements for

the Degree Doctor of Philosophy in the

Graduate School of The Ohio State University

By

Sandip Shriram Bapat, B.E.

* * * * *

The Ohio State University

2006

Dissertation Committee:

Dr. Anish Arora, Adviser

Dr. Paolo A.G. Sivilotti

Dr. Ten H. Lai

Approved by

Adviser

Graduate Program in
Computer Science and
Engineering

ABSTRACT

Wireless sensor networks have shown great potential as the technology that will change the way we interact with the physical world around us and have forced researchers and system designers to reconsider the way in which they think about distributed systems. However, existing deployments show that application designers and network managers for these networks have to deal with a great deal of uncertainty during sensor network execution. This uncertainty arises in part because of the unique differences in the sensor network model such as inherently unreliable broadcast communication, severely resource constrained devices and vulnerability to different types of faults. To meet these challenges, we must first understand the different reliability issues related to wireless sensor networks and then design appropriate mechanisms to deal with them. We believe network management to be a key enabler for such networks and applications to successfully deal with these challenges.

To address these problems, in this dissertation, we first present a comprehensive study of different types of node and network faults that occur in wireless sensor networks. Based on data collected from numerous indoor and outdoor experiments, we propose a fault model for wireless sensor networks. For anticipated faults, our model provides failure rate and distribution information that can be used by system designers and network managers to check application quality. Our model also identifies unanticipated faults that may occur in critical sensor network operations

such as deployment, reconfiguration and localization. By studying the cause, effect and response actions required to deal with these faults, we identify key elements of a network management architecture for wireless sensor networks.

We then present MASE, a unified **M**anagement **A**rchitecture for **SE**nsor networks, that addresses network management issues at all levels in a wireless sensor network: at individual nodes, in the network, and also at the base station or network manager. We realize, from our fault studies, the value of self-stabilization in dealing with both anticipated and unanticipated faults and emphasize self-stabilizing designs for the various elements in MASE. The MASE architecture is compositional and extensible in nature, allowing easy addition of new management services.

We describe in this dissertation, key network management services that we have already designed and implemented as part of MASE. We especially focus on services that were either not provided hereto or whose existing solutions faced serious reliability and scalability issues.

The *Stabilizing Reconfiguration* service for instance, solves the problem of version number cycling in existing reconfiguration protocols using a novel approach called *Human-In-The-Loop* stabilization. The *Chowkidar* health monitoring service reliably collects node and link status from the entire network at a base station and guarantees consistency of collected results despite the occurrence of ongoing faults, a property unique to our solution in the sensor network model. The *Reporter* service allows a network manager to detect termination of application protocols in a black-box manner and is highly message-efficient, requiring only about 5% of messages needed by existing solutions. Finally, the network-based experiment orchestration framework tries to close the loop in sensor network management by providing software libraries,

instrumentation tools and execution control logic for automating common patterns in sensor network execution and experimentation.

The different architectural components presented in this dissertation have been validated not only through extensive simulations and testbed experiments, but also in field deployments for managing large scale sensor network systems such as *A Line In The Sand* and *ExScal*. Implementations for existing services and tools developed as part of the MASE architecture, for mote, Stargate and server platforms, are also publicly available in the form of a MASE toolkit.

To my grandparents

Vimal & (Late) Keshav Bapat

Sudha & Dinakar Vaidya

ACKNOWLEDGMENTS

The completion of this thesis would not have been possible without the guidance, encouragement and support I received from several people.

I am deeply grateful to my advisor Professor Anish Arora who, back in 2001, motivated me to pursue a PhD. He has been a great mentor as well as a role model whose tireless dedication to research and careful attention to detail I shall always strive to emulate. One of his best attributes as an advisor was the perfect balance he achieved between granting me the freedom to pursue research problems that interested me and always being involved to guide the overall direction of my research, that kept me on the right track.

I am thankful to my dissertation committee members Professor Paul Sivilotti and Professor Steve Lai for their valuable comments and suggestions. Their ability to ask the questions that were truly fundamental to the problem at hand forced me to think deeper about my ideas and in several instances, come up with more general or more elegant solutions.

During the DARPA-NEST project, I had the pleasure of working with distinguished researchers such as Professor Mohamed Gouda, Professor Ted Herman, Professor Sandeep Kulkarni, Professor Mikhail Nesterenko, Professor Prasun Sinha, Professor Rajiv Ramnath and Dr. Emre Ertin. Their unique perspectives on different research problems have definitely influenced and enriched my way of thinking.

My research would not have been possible without financial support from The Ohio State University and various research grants from DARPA, NSF and Microsoft Research. I am indebted to these institutions for their support. I am also thankful to the Department staff including Tamera, Tom, Ewana, Marty and Catrena for their help in dealing with various administrative matters.

During my PhD study, I have greatly enjoyed interacting with fellow graduate students - Vinodkrishnan Kulathumani, Vinayak Naik, Hongwei Zhang, Santosh Kumar, Mukundan Sridharan, Prabal Dutta, Murat Demirbas, Bill Leal, Taewoo Kwon, Pihui Wei, Vineet Mittal and Sukhdeep Sidhu. I will forever remember some of the memorable experiences shared with this group such as performing sensor network experiments in the sub-zero temperatures of Ohio and the rain and storms of Florida.

Columbus has been my home away from home for the past six years and has given me the opportunity to forge some great friendships with some truly wonderful people. I shall always cherish the help and support received and the fun and laughter shared with my friends Vinayak, Mukta, Prashant, Niket, Omkar, Ameya, Swapna, Janhavi, Sheetal, Neha and Shrikant, among others. I also greatly enjoyed volunteering for Sankalpa and Columbus Maharashtra Mandal during my stay in Columbus.

There are no words of acknowledgment that can do full justice to the unconditional love, support, encouragement and sacrifice of my family, especially my parents and my sister Meghana. I simply could not have reached this stage without them. I am indebted to my grandparents, who instilled in me, a sense of discipline and a love for learning at a young age and have been a constant source of encouragement and inspiration for me.

VITA

March 27, 1979	Born - Mumbai, India.
2000	B.E. (Computer Technology), V.J.T.I. – University of Mumbai, India.
2000-2001	University Fellow, The Ohio State University, USA.
June-August 2001	Summer Intern, Microsoft Corporation, USA.
2001-present	Graduate Research Associate, The Ohio State University, USA.

PUBLICATIONS

Research Publications

S. Bapat, and A. Arora "Message Efficient Termination Detection in Wireless Sensor Networks". *Technical Report, The Ohio State University, OSU-CISRC-10/06-TR75*, 2006.

S. Bapat, and A. Arora "Stabilizing Reconfiguration in Wireless Sensor Networks". *International Conference on Sensor Networks, Ubiquitous and Trustworthy Computing (SUTC)*, pages 52-59, 2006.

S. Bapat, W. Leal, T. Kwon, P. Wei, and A. Arora "Chowkidar: A Health Monitor for Wireless Sensor Network Testbeds". *Technical Report, The Ohio State University, OSU-CISRC-10/06-TR76*, 2006.

W. Leal, S. Bapat, T. Kwon, P. Wei, and A. Arora "Stabilizing Health Monitoring for Wireless Sensor Networks". *The 8th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, pages 395-410, 2006.

S. Bapat, V. Kulathumani and A. Arora "Analyzing the Yield of ExScal, a Large-scale Wireless Sensor Network Experiment". *The 13th International Conference on Network Protocols, (ICNP)*, pages 53-62, 2005.

S. Bapat, V. Kulathumani and A. Arora "Reliable Estimation of Influence Fields for Classification and Tracking in unreliable Sensor Networks". *The 24th Symposium on Reliable Distributed Systems (SRDS)*, pages 60-72, 2005.

E. Ertin, A. Arora, R. Ramnath, M. Nesterenko, V. Naik, S. Bapat, V. Kulathumani, M. Sridharan, H. Zhang, and H. Cao "Kansei: A Testbed for Sensing at Scale". *The 5th International Conference on Information Processing in Sensor Networks (IPSN) for Sensor Platform, Tools and Design Methods for Networked Embedded Systems (SPOTS) track*, pages 399-406, 2006.

A. Arora, R. Ramnath, P. Sinha, E. Ertin, S. Bapat, V. Naik , V. Kulathumani, H. Zhang, H. Cao, M. Sridhara, S. Kumar, N. Seddon, C. Anderson, T. Herman, N. Trivedi, C. Zhang, M. Gouda, Y. Choi, M. Nesterenko, R. Shah, S. Kulkarni, M. Aramugam, L. Wang, D. Culler, P. Dutta, C. Sharp, G. Tolle, M. Grimmer, B. Ferriera, and K. Parker "ExScal: Elements of an Extreme Scale Wireless Sensor Network". *The 11th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 102-108, 2005.

A. Arora, R. Ramnath, P. Sinha, E. Ertin, S. Bapat, V. Naik , V. Kulathumani, H. Zhang, H. Cao, M. Sridhara, S. Kumar, N. Seddon, C. Anderson, T. Herman, N. Trivedi, C. Zhang, M. Gouda, Y. Choi, M. Nesterenko, R. Shah, S. Kulkarni, M. Aramugam, L. Wang, D. Culler, P. Dutta, C. Sharp, G. Tolle, M. Grimmer, B. Ferriera, and K. Parker "Project ExScal". *International Conference on Distributed Computing in Sensor Systems (DCOSS)*, pages 393-394, 2005.

A. Arora, P. Sinha, E. Ertin, V. Naik , H. Zhang, M. Sridharan, and S. Bapat "ExScal Backbone Network Architecture". *The 3rd International Conference on Mobile Systems, Applications, and Services (Mobisys)*, 2005.

A. Arora, P. Dutta, S. Bapat, V. Kulathumani, H. Zhang, V. Naik, H. Cao, M. Demirbas, M. Gouda, Y. Choi, T. Herman, S. Kulkarni, U. 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 Journal*, pages 605-634, 2004.

V. Naik, A. Arora, S. Bapat, and M. Gouda "Whisper: Local Secret Maintenance in Sensor Networks". *IEEE Distributed Systems Online*, 2003.

V. Naik, A. Arora, S. Bapat, and M. Gouda "Whisper: Local Secret Maintenance in Sensor Networks". *Workshop on Principles of Dependable Systems (PoDSy) in conjunction with The International Conference on Dependable Systems and Networks (DSN)*, 2003.

FIELDS OF STUDY

Major Field: Computer Science and Engineering

Studies in:

Computer Networks	Prof. A. Arora
Software Systems	Prof. P. Sivilotti
	Prof. P. Sadayappan
Theory and Algorithms	Prof. Michael Rathjen
	Prof. Ten H. Lai

TABLE OF CONTENTS

Page

LIST OF TABLES

Table **Page**

LIST OF FIGURES

Figure **Page**

CHAPTER 1

INTRODUCTION

1.1 Background

A wireless sensor network, as the name suggests, is formed by a group of nodes that are capable of sensing one or more physical attributes of their environment such as temperature, light, sound, etc., processing and storing these sensed values locally and coordinating with other sensor nodes using their wireless radios. Additionally, some network nodes may also have actuation capabilities by which they can control or manipulate their physical environment. A typical characteristic of wireless sensor/actuator networks in their present day form, that distinguishes them from traditional networks such as the Internet, is that they are built out of low-cost, resource constrained components, following the general principle that although individual nodes may have limited capabilities and be subject to faults, their low cost makes deployments at large scales feasible.

Indeed, the proliferation of new hardware platforms, both research and commercial, as well as the development of software architectures and development tools over the past few years, has helped sustain the vision of wireless sensor networks in the

Year	Nodes	Area	Program size
2002	10	10 sq.m.	5KB
2003	100	500 sq.m.	30-100KB
2004	1000	250,000 sq.m.	200KB-2MB

Table 1.1: Evolution of sensor networks.

scale of 10^5-10^6 nodes being deployed in the near future in industrial, military, agricultural, medical and several other applications. Some commonly found applications of such sensor networks include environmental or habitat monitoring [16, 58, 71], structural monitoring [76], shooter localization [68] and intrusion detection [1, 2]. Based on our experience in building wireless sensor networks, we have seen sensor networks scale in several dimensions as shown in Table 1.1, the largest of these being *ExScal* [1], in which we deployed more than 1000 sensor nodes called XSMs [29] and 200 802.11b enabled devices called XSSs [43] over a 1.3km x 300m outdoor area.

1.2 The Case for Management in Wireless Sensor Networks

The design, implementation, deployment and maintenance of such large scale wireless sensor networks differs from and is made especially more challenging than traditional distributed systems like the Internet due to factors such as constraints imposed by the hardware and software platforms and by the environment in which wireless sensor networks are deployed as described below.

Wireless sensor networks are typically built out of low-cost devices having limited computational power, memory, energy and communication range. The most popular sensor hardware platforms such as the mica2 [42], XSM and the TMote [22], used

for both research and commercial deployments, have processor speeds of less than 10 MHz, up to 10 KBs of RAM and a few hundred KBs of external flash. The wireless communication rates for these nodes range from 30-250 kbps, however, the actual bandwidth obtained in multi-hop networks of such devices is much lower. These low-cost hardware devices are also prone to several types of faults, some of which may actually produce quite complex behaviors. For instance, a simple fault occurring in a sensor node is a failstop where a node stops working once it runs out of battery. However, before a node failstops, it may operate at a critical battery level where its processor can operate correctly but other components such as sensors of flash memory cannot, thereby producing arbitrary behaviors during sensing or reprogramming.

Hardware resource constraints in sensor devices also impose significant limitations on the type of software that runs on wireless sensor nodes. Limitations on processing and memory affect the amount of sophistication that can be built into the operating systems and other software for sensor networks. Properties such as atomicity, mutual exclusion, deadlock freedom, fairness, etc., which are often taken for granted in traditional distributed systems have to be sacrificed for minimality and efficiency in the presence of resource constraints. Consequently, faults such as the corruption of state variables, non-execution of certain tasks to completion or component deadlocks due to task failures or event losses may occur during the execution of a program.

Further, these nodes are often deployed in environments where they may be subjected to harsh terrains, severe weather and changing environmental noise conditions. Such harsh and dynamic environments also lead to different types of faults in sensor networks such as false positives during sensing, loss of connectivity and network partitioning.

Hardware, software and environmental factors thus contribute to faults being the norm rather than the exception in wireless sensor networks. Moreover, faults occurring at one sensor node may have a non-local effect, e.g. a node producing false detections due to a faulty sensor imposes significant communication overhead on nodes in its routing path.

One way of dealing with faults is to design a system that is fault-tolerant to begin with. However, this requires network designers to be fully aware, at design time, of the different types of faults and the extent to which they may occur once the network is deployed. As seen from Table 1.1, wireless sensor networks are becoming increasingly complex not only in terms of deployment area and scale, but also in terms of program complexity. Further, these networks are being deployed for increasingly long-lived missions. Over their extended lifetime, these networks are subject to different forms of changes. First, the physical deployment environments of these networks could change over time, e.g. the temperature or wind conditions in a region depend on the climate of that region and the current seasonal conditions. Further, wireless sensor devices themselves undergo changes, e.g. the battery levels of sensor nodes may decrease, thereby leading to changes in sensor sensitivity, communication range, etc. Finally, the programs that need to run on these sensor nodes may evolve as a result of changing application requirements, protocol enhancements, bug fixes, etc.

The problem of fully understanding and anticipating all possible faults that may occur in such complex, evolving deployments is therefore quite hard. Even if such a design could somehow be achieved, implementing it in resource constrained wireless

sensor networks might be too expensive. Moreover, wireless sensor network applications are often built by composing reusable modules that are designed and implemented independently. Even if the individual components were designed to be fault-tolerant, it may not be easy to argue that their composition would also in fact be fault-tolerant.

We therefore contend that network management is critical for dealing with the unreliability, change and resource constraints that are inherent in complex, large scale wireless sensor networks and accordingly address different aspects of wireless sensor network management in this dissertation.

1.3 Wireless Sensor Network Management

Network management can be simply defined as the process in which different network entities, which represent the managed devices, provide information about their state to a manager entity, which then reacts to this information by executing one or more actions such as logging, notification, reset, shutdown or repair. Managed devices may send information to the manager on their own, either periodically or when certain triggers such as exceptions or fault detections are fired, or upon being polled by the manager.

The International Organization for Standardization (ISO) has defined a conceptual model that defines the five main areas of network management as fault management, configuration management, performance management, accounting management and security management. In this section, we re-ask the same question about how management should be defined in the context of wireless sensor networks.

1.3.1 Definition

We derive our definition for management of wireless sensor networks from the standard ISO definition with some changes related to the specific issues and requirements that need to be emphasized in the wireless sensor network model. There exists a significant overlap between fault management and performance management, even in the standard ISO definition. For sensor networks, this line is even more blurred since faults are the norm as opposed to the exception and network performance is closely related to the ability of the system to cope with faults. Wireless sensor networks are complex systems that may include different types of devices with different hardware and software capabilities. Tracking network devices and ensuring the correctness of their hardware and software configurations is therefore an important requirement. Analogous to accounting management, managing resources is a critical task since these networks are severely resource constrained. The efficient utilization of resources not only results in better system performance but also increased system lifetime. Security, as in all networks, is certainly an important requirement in sensor networks. However, traditional security algorithms and techniques cannot be applied directly to provide security for wireless sensor networks because of differences in the attack models and the resource constrained platforms. We thus need to address protocol design issues for wireless sensor network security to make security management more meaningful. Based on these requirements, we now define the key areas of wireless sensor network management as follows:

1. **Fault management:** As discussed earlier, faults are the norm rather than the exception in wireless sensor networks. The main goal of fault management in

wireless sensor networks is to detect, log and respond to different types of faults that may occur in a sensor network.

2. **Configuration management:** Wireless sensor networks need to adapt in response to the different types of changes they undergo as described earlier. Also, different groups of nodes in the network may have different roles and therefore require different configurations, e.g. in heterogeneous networks with different types of sensors, each node needs to load the correct drivers for its sensor type(s). Configuration management involves making sure that the correct hardware and software configurations are always maintained in the network.

3. **Resource management:** As discussed earlier, wireless sensor networks are resource constrained to begin with. Resource management is therefore critical for deciding what set of resources needs to be allocated in order to best satisfy a particular system specification or user request. Moreover, as these resources get depleted due to battery exhaustion or failure, the resource manager the manager needs to figure out how to reassign their tasks to other nodes in the network so that the overall application quality metrics such as communication and sensing coverage are not affected.

1.4 Contributions

In this dissertation, we present the following important solutions related to problems in reliable and scalable management of wireless sensor networks:

1. Based on data collected from numerous indoor and outdoor experiments, including the large scale *ExScal* deployment [1, 11], we propose a fault model for

wireless sensor networks. The proposed fault model provides an empirical characterization such as frequency and distribution for well-known faults and also reveals several new types of faults that are unique to the wireless sensor network domain.

2. We present MASE, a compositional architecture for wireless sensor network management. The different components in our architecture are designed to be self-stabilizing so that they are themselves tolerant to different types of faults.

3. Reconfiguration is an important management task in wireless sensor networks as it allows a network manager to program a network once it has been deployed, change some or all application modules on some or all nodes or update critical system parameters to improve performance. However, existing reconfiguration protocols are not self-stabilizing and may enter fault states from which they never converge. We propose a self-stabilizing reconfiguration protocol [10] that solves this problem. This protocol uses a novel approach for stabilization which we call *Human-In-The-Loop Stabilization* as it involves a two-part convergence process in which the network first self-stabilizes to a semi-correct state from which a human manager can restore it to the ideal state. We show that this reconfiguration protocol guarantees convergence and is therefore reliable and also is local, has low communication and computation overhead and is therefore scalable.

4. Being able to accurately monitor the state of a wireless sensor network is important for a manager to determine the appropriate management response. For instance, in the case of network health monitoring, a manager needs to receive

information about the status of each node and link in the network. We propose two protocols, one centralized and the other distributed, which form the core of a wireless sensor network monitoring service called Chowkidar [13] which we have developed. The distributed Chowkidar protocol [51] solves the well-known problem of message-passing rooted spanning tree construction and its use in PIF (propagation of information with feedback) for the case of a wireless sensor network. This protocol is guaranteed to terminate with accurate results, including detection of ongoing failure and restart during the monitoring process and is thus reliable. The protocol is initiated upon demand; that is, it does not involve ongoing maintenance and only requires few messages per node, and is therefore scalable.

5. The Chowkidar protocols described above address the problem of reliable collection of information such as node and link health from every node in the network. However, for many management tasks, it is not necessary to collect information from every node to infer the global state of the network. Termination detection is an example of such a task. We present a message-efficient protocol, called Reporter [9], that detects termination of application protocols. Reporter is reliable because it satisfies the standard safety and liveness requirements of termination detection and is highly scalable because it only requires a small fraction (around 5%) of nodes to send termination reports and reuses application traffic for selecting reporters and creating a structure for collecting termination reports.

6. Execution and experimentation with wireless sensor networks currently involves significant human involvement. Towards making this process automated, we identify common execution and experimentation patterns in sensor networks and design management services and tools that allow users to orchestrate network operation in an automated manner.

1.5 Organization of the Dissertation

The rest of this dissertation is organized as follows.

- Chapter 2 describes the proposed fault model for wireless sensor networks.
- Chapter 3 describes the different components in our management architecture.
- Chapter 4 describes the reconfiguration protocol for wireless sensor networks which uses the novel *Human-In-The-Loop* stabilization approach.
- Chapter 5 describes the two approaches for reliable collection of management information, viz. the Chowkidar protocols for reliable collection of node and link health from all nodes at a base station and the Reporter protocol for efficient termination detection in wireless sensor networks.
- Chapter 6 describes the network orchestration and control service in our management architecture.
- Chapter 7 summarizes the findings of this dissertation and discusses related future work.

CHAPTER 2

A FAULT MODEL FOR WIRELESS SENSOR NETWORKS

2.1 Introduction

As described in Chapter 1, the deployment size and program complexity for wireless sensor network applications is expected to increase steadily in the near future. However, past experiences in sensor network research such as [2] have shown that scaling even an individual protocol for a network that is 10 times larger often involves redesigning the protocol itself.

One of the main challenges in scaling network deployments and application protocols is the occurrence of a variety of faults. In addition to the usual distributed systems faults such as node fail-stops, wireless sensor networks are subject to network faults such as channel contention, interference and fading over the wireless medium. The extent to which these faults affect a network is determined by several factors such as internode separation, antenna polarization, presence of obstacles and the traffic pattern in the network. Network faults can thus have significant spatial and temporal variability making the design of scalable sensor network protocols a challenge.

Sensor network applications are often built out of multiple protocols for low-level services such as medium access, reliable communication, sensing, time synchronization, etc., and integrating these protocols raises several challenges. First, system correctness is hard to reason about in the presence of complex interactions between the various protocols. This is especially true in resource-constrained operating systems such as TinyOS [37] that do not guarantee mutual exclusion, deadlock freedom and other properties taken for granted in traditional distributed system design. Secondly, optimizing the performance of such complex systems often involves simultaneous tuning of parameters across multiple protocols which may have conflicting requirements. For example, increasing the memory allocation for routing buffers may improve communication performance, but perhaps at the cost of memory available for filtering windows in sensory processing, leading to increased noise of false positives.

Designing large scale wireless sensor networks is further complicated by the fact that there is little data on faults, their variability or their impact on applications at small and large scales. Due to the lack of sufficient data about faults occurring in a large scale network, there is also a dearth of simulation and analytical tools that realistically model scaling effects. This makes it hard to predict the behavior of sensor network protocols and often forces network designers to be conservative in allocating resources to the network.

Node, network and other faults also critically affect the management of wireless sensor networks. Node faults may result in the management data structures becoming disconnected, network faults may result in management information being lost while other hardware and software faults may produce corruption of management data.

Given such faulty information, a manager might end up selecting an incorrect response that could further degrade system performance and perhaps even affect correctness.

In this chapter, we therefore present a model for different types of faults that occur in wireless sensor networks. This fault model is derived using data from several experiments, including those performed in the indoor Kansei [31] testbed at Ohio State and experiments performed over a 14 day period in an outdoor setting using the *ExScal* deployment in Florida.

The rest of this chapter is organized as follows. In Section 2.2, we describe the *ExScal* system since it is a typical example of a wireless sensor network application and since many of our fault data measurements were taken in the context of the *ExScal* application. We then present models for different types of faults, deployment in Section 2.3, reprogramming in Section 2.4, localization in Section 2.5 and routing in Section 2.6. Finally, we discuss some other commonly observed faults in wireless sensor networks in Section 2.7. For each fault type, we discuss causes and effects of the fault and also the experimental methodology used to obtain the data from which the model is derived, wherever applicable.

2.2 *ExScal* System Overview

The *ExScal* system is designed as a large scale wireless sensor network for detecting, classifying and tracking intruders over an extended geographical area. In this section, we present an overview of *ExScal* with respect to its architecture and operation.

2.2.1 Multi-tier Design

ExScal usesd a three-tier network design to bound unreliability and end-to-end latency in the multi-hop network. The lowest tier, Tier-1, consists of XSMs (for

eXtreme Scale Motes) [29] which are derivatives of the Mica2 mote [42]. XSMs perform the tasks of sensing and detection using onboard magnetometer, acoustic and PIR (for passive infrared) motion sensors and communicate detected events to a local base mote. Each local base mote aggregates detections from an average of 50 XSMs and is connected to a Tier-2 node called the XSS (for eXtreme Scaling Stargate) [43] through a 51-pin connector interface. XSS nodes have a 400 MHz processor, 64MB RAM and 32MB flash memory and are thus more resource-rich than XSMs. Each XSS has a GPS device and can communicate reliably over several hundred meters using a 2.4GHz radio, connected to a 5 ft tall, 9dBi omnidirectional antenna. XSS nodes form their own peer-to-peer ad hoc communication network using the IEEE 802.11b MAC protocol. This network is rooted at a special XSS node that is connected via wired ethernet to a Tier-3 node. The Tier-3 node is a laptop or PC running the classification, tracking and visualization applications and also serves as the command and control station for network management.

Network topology Figure 2.1 shows the topology of the *ExScal* network which consists of 983 XSMs, represented by dots, arranged in two regions. The dense region at the top consists of 5 rows with 140 XSMs each at a separation of 9m arranged in a regular hexagonal grid. The sparse region consists of two XSM lines starting at 90m from the dense region and 90m from each other. These XSM lines enable us to track intruder motion after it has left the dense region. Figure 2.1 also shows 45 XSS nodes, represented by triangles, arranged in a 15 x 3 grid with 90m separation. The dense region thus has a total of 686 XSMs and 15 XSSs resulting in about 50 XSM nodes per XSS.

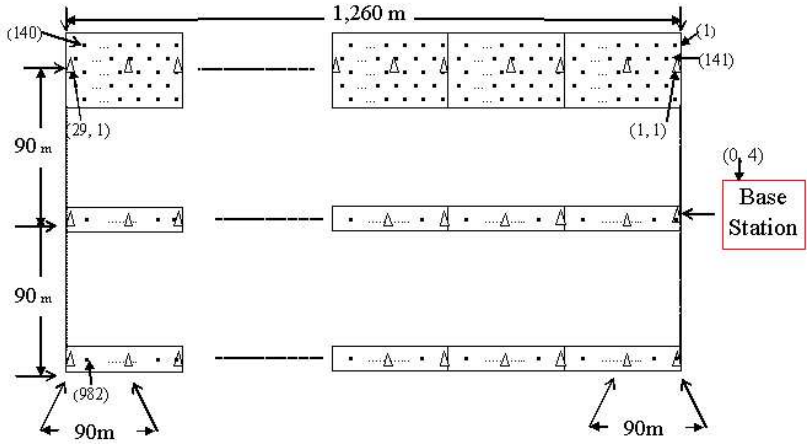


Figure 2.1: *ExScal* network topology.

To demonstrate scalability of multi-tier design, *ExScal* also uses 203 XSS nodes in a 29 x 7 grid over the same area for experiments involving the Tier-2 ad hoc communication protocols [5].

2.2.2 Multi-phase Operation

To manage application complexity, the operation of *ExScal* is broken down into the following phases.

- **Pre-deployment** The first phase of *ExScal* consists of a default application for all XSMs. This application, which we call *Trusted Base* has the ability to download new programs using the Deluge [38] reprogramming protocol and the ability to perform certain management functions like sleep/wakeup and network health querying using the Sensor Network Management System (SNMS) [72] protocol. The *Trusted Base* also includes a *deployer response* application. This application is enabled when the XSM is turned on during deployment, and sends

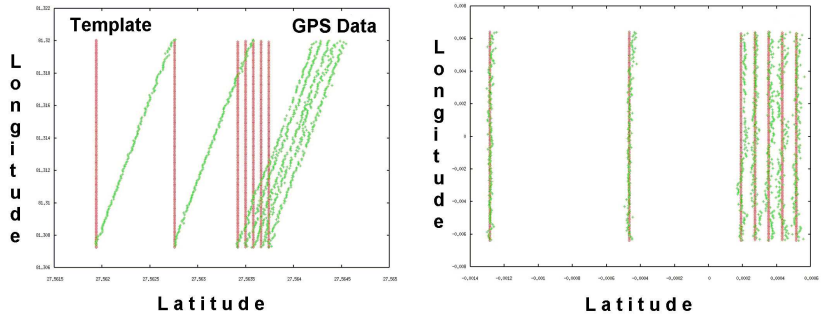
out *Hello* messages containing the unique identifier of the XSM and emits an audible beep as confirmation of its liveness.

- **Deployment** The deployment process consists of two steps. In the first step, the grid topology is marked on the ground using techniques from civil engineering to an accuracy of a few centimeters. Marked grid points represent ideal node positions. In the second step, human deployers place the XSMs and the XSSs at the marked grid points and power them on. The *Hello* messages sent by the *deployer response* application on an XSM are received by a mote attached to a hand-held XSS node carried by the deployer. The deployer's XSS records the node-id in this message along with the GPS location of that point where the node has been deployed in a file on the XSS. Thus, at the end of the deployment process, the network deployers have a list of node-ids and their corresponding GPS co-ordinates.

- **Reprogramming** The reprogramming phase is used to download new application programs from the Tier-3 node to the entire network and is a recurring phase in *ExScal* operation. Tier-2 applications and protocols run on XSSs as Linux processes hence reprogramming Tier-2 nodes consists of replacing an existing executable with a new one. To download a new program on Tier-1 XSMs, we use the Deluge protocol in the *Trusted Base*. The new Tier-1 program is first downloaded to the XSS nodes which in turn execute the Deluge protocol to download it to the entire XSM network.

- **Localization** The next phase of *ExScal*, which we call *OASLOC* for Operator Assisted Localization, involves localization of deployed nodes. To perform

localization, (node-id, GPS) pairs collected by each deployer's hand-held XSS are first downloaded on the Tier-3 node and merged. This list is then fed to a geometric program which we call *Snap-to-Grid*, running on the Tier-3 node. *Snap-to-Grid* uses a template of ideal grid positions, as shown in Figure 2.2(a), and performs a series of rotation, translation and heuristic-based matching operations to map each node-id to a grid position in the template, as shown in Figure 2.2(b).



(a) Template vs. GPS data. (b) GPS data snapped to template.

Figure 2.2: OASLOC Snap-to-Grid process.

A similar strategy is used to localize Tier-2 XSSs, the key distinction being that XSSs directly communicate their node-id and GPS locations to the Tier-3 node using an efficient flooding-based algorithm. Tier-2 grid positions are communicated back to the XSSs using the same flooding service, while Tier-1 grid locations are first communicated to the nearest XSS node which then initiates a controlled flooding algorithm called *Epicast* to forward these to the respective XSMs.

- ***ExScal Op-Ap*** Upon localization, the nodes are ready to execute the main sensing and intrusion detection application, which we call *Op-Ap* for *Operator App*. *Op-Ap* uses a routing protocol called GridRouting [21] to communicate its detections to the local base node. GridRouting uses the output of *OASLOC* to conservatively select a set of potential parents with stable, reliable links for each XSM. The *ExScal Op-Ap* also uses an implicit acknowledgement based retransmission protocol called ReliableComm [78] to improve per-hop reliability. The routing reliability of *Op-Ap* at Tier-1 is thus the reliability provided by GridRouting using ReliableComm.

  Detections received at a Tier-2 node are communicated to the central Tier-3 node where they are used to classify and track the detected intruders. This Tier-2 convergecast uses a beacon-free routing protocol called LOF [79], which uses data traffic to perform link-estimation for selecting next-hop parents.

Figure 2.3 shows the component diagram of the *ExScal Op-Ap* at Tier-1.

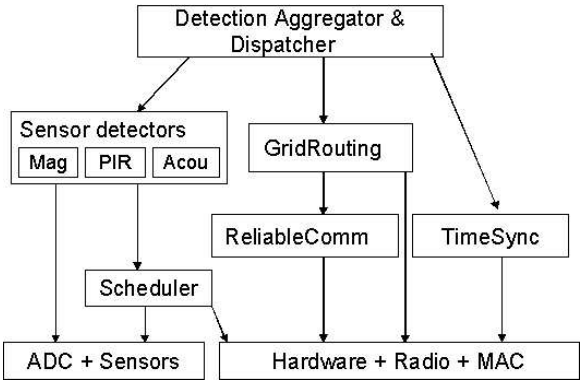


Figure 2.3: *ExScal Op-Ap* Tier-1 components.

2.3 Deployment Faults

2.3.1 Definition

Many wireless sensor networks, like *ExScal*, are designed for outdoor settings and use a large number of (previously untested) devices, hence deployed nodes are subject to environmental elements. During the 15 day deployment period in *ExScal* for instance, deployed nodes were left in an open area where they were exposed to passing vehicles or wildlife that crushed some nodes, heavy rain that caused leakages resulting in failures in others and heavy wind that toppled yet others and reduced their communication range significantly, thereby disconnecting them from the rest of the network. We define a deployment fault as follows:

> *Deployment fault:* A node is defined to be affected by a deployment fault if it fails physically or it cannot be reached by any other deployed node.

In all of the examples described above, once a deployment fault occurred at a node, the node was rendered useless during the rest of the deployment period. Deployment faults are thus permanent.

2.3.2 Experimental Measurements

To measure deployment faults accurately, we need reliable ground-truth information about the entire network, which is especially challenging and expensive for large scale networks. Hence, in *ExScal*, we only measured ground truth for a section of 100 XSM nodes. This 100 node section is small enough that we could reliably collect fault data from it on an ongoing basis, yet large enough to capture most interesting faults that could occur in other similar sections. We then extrapolated the data from this section to estimate the fault rate for the whole network. We then verified

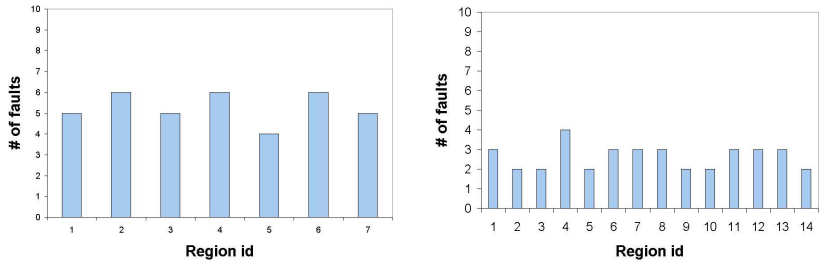
this estimate by logging all messages received from the network during the entire 15 day deployment period. We then counted all unique nodes from which at least one message is received, i.e. the number of *up* nodes as defined above. This number is then compared to the estimated value to derive a lower bound on the yield of the deployment phase. Since these messages were generated during different application phases and communicated using different routing protocols, we argue that with very high probability, an *up* node would be able to communicate at least one message to the base station.

Number of deployed nodes	686
Number of deployment faults in one section	5
Estimated number of total deployment faults	35
Estimated number of *up* nodes	651 (686 - 35)
Measured number of *up* nodes	647

Table 2.1: Deployment fault data.

Results. Table 2.1 compares the estimated and measured values for deployment faults. Based on the fault data collected for one section, the number of XSM failures for the whole network is then estimated to be 35 implying that 651 XSMs should be *up*. As seen from Table 2.1 this estimated closely matches the measured number of 647 *up* nodes.

Perhaps more important than the actual number of deployment faults, is the nature of their distribution. We therefore calculated histograms of the number of deployment faults measured in different regions to obtain their spatial distribution. Figure 2.4 plots two such histograms for regions of sizes 100 nodes and 50 nodes



(a) 100 node region. (b) 50 node region.

Figure 2.4: Spatial uniformity of deployment faults.

respectively. The flat shape of both histograms in Figure 2.4 demonstrates that the spatial distribution of deployment faults in *ExScal* is uniform. We also observe deployment failures to have a uniform temporal distribution. At the time of deployment, each node was verified to be *up* using the audible beep in the *deployer response* application. In the 100 node section that we closely monitored, we discovered 1 *failed* node after 3 days, 3 *failed* nodes after one week and 5 *failed* nodes at the end of the 15 day period, indicating a temporal attrition rate for deployed nodes.

> *Deployment fault model:* Deployment faults occur with non-trivial frequency (5.7% in *ExScal*) in wireless sensor networks and are spatially and temporally uniform.

2.4 Reprogramming Faults

2.4.1 Definition

Once sensor nodes are deployed in their environment, they cannot be recollected every time their software needs to be changed nor can we connect a cable to each node to program it. Thus, reprogramming a wireless sensor network typically involves downloading a new program to each nodes using bulk transfer protocols such as

Deluge [38], Sprinkler [61], MNP [49], etc. Since a node may need to store multiple programs at the same time, this cannot be done in the main memory of nodes, hence downloaded programs are first stored in the onboard flash memory of nodes from where they can be copied into the instruction memory as per user control.

In *ExScal*, we use the Deluge protocol in the *Trusted Base* to reprogram XSMs with a new application image. Deluge divides an application image into smaller *pages* which are downloaded one at a time and stored in the external flash on an XSM. An XSM can be rebooted to this image only if it has downloaded all pages correctly. Deluge is a flooding based epidemic protocol, so a node with a partial application image continually tries to download missing pages from neighbors that may have them, using version numbers to distinguish new images from old ones.

Based on the possible sources of failure, we define three types of reprogramming faults in a wireless sensor network.

> *Initialization faults*: We say a node has an initialization fault if it cannot execute the reprogramming protocol due to flash initialization errors during startup.

Restarting nodes with initialization faults often results in successful flash reinitialization, however, it is not feasible to detect and restart individual failed nodes in large scale deployments.

> *Lagger nodes*: We say a node is a lagger if it can participate in the reprogramming protocol, but progresses at a much slower rate than its neighbors. In the worst case, a lagger is always stuck trying to download the same program data and makes no progress.

Lagger nodes have a significant adverse impact on other network nodes, especially in epidemic protocols such as Deluge as they repeatedly request program data from

neighboring nodes causing them to waste substantial energy in message transmissions and flash operations, thereby reducing their lifetime. Our offline measurements show that the current drawn by an XSM is nearly doubled due to extra message transmission and flash read operations. The number of neighbors for an XSM node in the *ExScal* topology is between 10 and 20. Thus, even a small fraction of lagger nodes can significantly reduce the lifetime of a large number of nodes. Another problem caused by agger nodes is persistent reprogramming traffic in the network that causes higher contention for application messages. This leads to reduced reliability, increased latency and degraded application performance.

> *Non-stabilization faults*: We say a non-stabilization fault has occurred when the network cannot converge to the correct program and oscillates between multiple versions of the same program

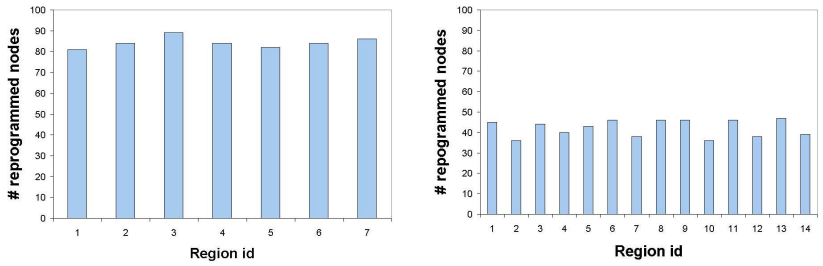
Non-stabilization faults may in turn be caused due to other faults such as transient corruption of version data or network faults such as partitions. A non-stabilization fault may persist forever, in which case reprogramming would never complete and nodes would continue to waste valuable resources performing redundant operations. We discuss non-stabilization faults in further detail in Chapter 4 and also present a solution that guarantees convergence when such faults occur in bounded time.

2.4.2 Experimental Measurements

To measure reprogramming faults in *ExScal*, the network manager initiated the download of the *ExScal* application using the Deluge protocol in the *Trusted Base*. The manager then queried the network repeatedly to monitor the progress of this download over the network using the SNMS management protocol. Nodes with initialization faults responded to this query with invalid results due to flash errors while

lagger nodes were detected based on the relative progress made by nodes with respect to their neighbors. When a safe time interval had passed for reprogramming the entire network and when the query results indicated that the observed faulty behaviors were stable, i.e. the observed lagger nodes were not making any progress, the manager issued a reboot command to switch to the newly downloaded *ExScal* application. Again, as before, we correlated the number of faults measured in the querying phase with the number of nodes from which application messages were not received and found these to be nearly identical.

Results. The data collected by repeated querying of XSMs during reprogramming shows that 5% of the XSMs are affected by initialization faults and thereby cannot download new applications and that the number of lagger nodes is 0.5%. These numbers represent the lower bound on reprogramming faults since there could be other faulty nodes whose query responses were lost. However, in the data collected from the subsequent application phase we observed messages received from 93% of nodes that ran the reprogramming protocol, implying that the upper bound on reprogramming faults was 7% .



(a) 100 node region. (b) 50 node region.

Figure 2.5: Spatial uniformity of reprogramming faults.

We also measured the spatial distribution of reprogramming faults by calculating the histograms of the number of correctly reprogrammed nodes in regions of different sizes. This spatial distribution is uniform as expected, since initialization faults occur due to timing errors at startup and lagger nodes occur due to flash errors, low battery or transient corruption, and are thus both independent events. Figure 2.5 shows the histograms for regions of 100 and 50 nodes respectively and once again demonstrates the spatial uniformity of reprogramming faults.

> *Reprogramming fault model*: Reprogramming faults have non-trivial frequency of occurrence (5.5%-7% in *ExScal*) in wireless sensor networks and have uniform spatial distribution.

2.5 Localization Faults

2.5.1 Definition

Localization is an important wireless sensor network task that aims to estimate the absolute or relative positions of deployed nodes. Localization can be performed by using GPS devices that provide absolute locations or using some distance estimation technique based on radio and/or acoustic signals. Localization is used by network services such as routing for selecting nodes in its reliable transmission distance as parents, or by services such as tracking to estimate the actual position of the target. As described in Sec. 2.2.2, *ExScal* uses *OASLOC* for localizing deployed nodes. We define three types of localization faults which were observed during *OASLOC* and which apply to other localization techniques as well.

> *Hole fault*: We say a node has a hole fault if it is not assigned a grid position by the localization protocol.

Hole faults may occur due to deployment faults, procedural faults such as not waiting long enough for GPS devices to obtain an accurate reading or environmental factors

such as poor GPS reception in an area leading to a burst of hole faults. In estimation based techniques, hole faults may occur as a result of malfunctioning devices such as buzzers, acoustic sensors, radios, etc. on some nodes.

> *Mapping fault*: We say a node has a mapping fault if it is assigned a grid position other than the one it is deployed at.

Mapping faults may occur as a result of drifts in GPS values collected for nodes that are deployed close to each other; in *ExScal* nodes were deployed only 9m apart. Exact algorithms that map localization data to grid positions often require exponential time, hence localization systems typically use heuristics or approximation algorithms, which may also introduce mapping faults.

> *Network localization fault*: We say a node has a network localization fault if it does not receive its assigned grid position.

A network localization fault may occur due to message losses while communicating the calculated grid positions to the nodes.

2.5.2 Experimental Measurements

During the localization phase in *ExScal*, network deployers manually recorded the order in which XSM node-ids were deployed on the field in addition to the (id,GPS) data collected automatically on the hand-held XSS device. The output of the *OASLOC Snap-to-Grid* program was then compared with this *ground truth* to detect holes and mapping faults. The spatial distribution of these faults is used to calculate the burstiness of each type. As before, network faults cannot be directly measured and are inferred from messages received in the *Op-Ap* phase as follows. *Op-Ap* uses a node's grid position as the source address in its messages. If a node has not received its grid position, it uses its software id whose domain is different from the

domain of grid positions. Thus, by observing the source field for messages received during the *Op-Ap* phase, we calculate the number of nodes affected by network faults.

Fault type	**Number of faulty nodes**	**Size of fault bursts**		
		Min	Avg	Max
Hole	6.1%	1	1.4	6
Mapping fault	3.2%	1	1.7	8
Network fault	2.1%	1	1	1
Total	11.4%			

Table 2.2: Localization fault data.

Table 2.2 shows the number of nodes affected by each type of localization fault in *ExScal*. Unlike deployment or reprogramming faults however, we find that hole and mapping localization faults may not be spatially uniform and may occur in bursts. We also find that the maximum distance by which a node affected by a mapping fault is displaced is one grid position.

> *Localization fault model*: Localization faults have a significant probability of occurrence (11.4% in *ExScal*) in wireless sensor networks. Localization faults may not be spatially uniform and may occur in bursts.

2.6 Routing Faults

2.6.1 Definition

Although the routing problem in a general network means any-to-any routing, wireless sensor networks rarely use this form of routing. Instead, data traffic in a wireless sensor network has two dominant patterns - one from the base station to all nodes, i.e. broadcast, as is needed for reprogramming, reconfiguration, management, etc. and the other from all nodes to the base station, i.e. convergecast, as is needed

for periodic or event based data collection from a sensor network. In this section, we consider the convergecast pattern for routing faults as it is the dominant one for application data.

> *Routing fault*: We say a message originating at any node in the network is affected by a routing fault if it does not reach the central base station node.

Recall that wireless sensor networks can consist of multiple tiers as is the case in *ExScal*, thus a routing fault could occur at any point along the path from the node to the base station. Routing faults are typically caused by network contention due to which messages transmitted at the same time collide on the shared wireless channel and are lost or due to congestion where messages are dropped in the forwarding buffers of intermediate hops due to too much traffic in the network.

2.6.2 Experimental Measurements

To measure routing faults in *ExScal*, we instrumented each node to send messages according to some traffic pattern. Each message contained the id of the node, a sequence number, a time-stamp and a traffic pattern identifier. These messages were first received and logged at the local Tier-2 node and then sent to the Tier-3 node where they were also logged. Routing faults at each tier and for end-to-end communication were then computed by taking the ratio of the number of received messages to the number of sent messages.

Traffic patterns The basic traffic pattern in *ExScal Op-Ap* is one in which a set of nodes detecting an intruder report their detections to the nearest Tier-2 node, which then forwards them to the Tier-3 classifier and tracker. While this traffic pattern is suitable for event detection systems, it may not necessarily apply to other kinds

of applications. Also, in practice, periodic traffic like timesync and routing beacons, or bursty traffic like false positives, interferes with *Op-Ap* detection messages. We therefore selected three traffic patterns as defined below, which we feel are general enough to be used in other types of wireless sensor network applications and are better indicators of routing faults for application traffic.

• *Low-freq*: In the low-freq traffic model, each XSM sends a message to the Tier-3 node every 120 seconds.

• *High-freq*: In the high-freq traffic model, each XSM sends a message to the Tier-3 node every 20 seconds.

• *Bursty*: In the bursty traffic model, each XSM sends a message at the same time, creating a message burst. The bursty traffic pattern may be applicable for event detections that are geographically far-reaching, e.g., acoustic shooter localization [68], or caused by a network wide false alarm, e.g., thunder or an airplane triggering acoustic detections across a large region.

Note that in all 3 experiments, no aggregation at Tier-2 was used. Aggregation reduces Tier-2 traffic to very few, if not a single message, in which case LOF [79] has been shown to be almost 100% reliable.

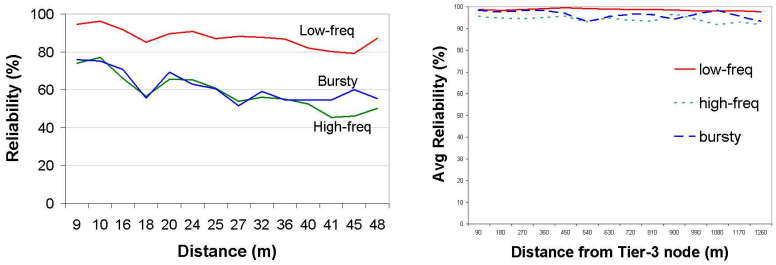
Remark on message generation. For the low-freq and high-freq traffic patterns, each XSM used a local timer to generate periodic messages. In the bursty case, a command was flooded from the Tier-3 node to all Tier-2 nodes which then flooded the Tier-1 network with this message. Nodes waited a fixed amount of time for the flood to subside before sending their message. Our experimental measurements show that the effects of the reliability and latency of the command message propagation on traffic burstiness are relatively insignificant.

Results. Table 2.3 lists the average routing reliability for all nodes and the average routing reliability for nodes with and without localization faults, as measured at the Tier-1 base station. The data in Table 2.3 shows that the low-freq traffic has

	Low-freq	**High-freq**	**Bursty**
Net average reliability	86.72%	58.32%	60.17%
Average reliability			
with no localization fault	89.36%	70.95%	79.7%
with localization faults	74.74%	50.97%	60.35%

Table 2.3: Reliability of *ExScal Op-Ap* Tier-1 routing.

the highest average reliability and the least variance as expected. The traffic load generated by intrusion detection systems like *ExScal* is in fact lower than in the low-freq case. The common case Tier-1 routing reliability of *Op-Ap* is thus even higher and more uniform than in Table 2.3. The data in Table 2.3 also indicates that localization faults do have a non-trivial impact on the performance of routing protocols that use localization information to select parent links.



(a) Tier-1 routing. (b) Tier-2 routing.

Figure 2.6: Spatial distribution of routing reliability.

Figure 2.6(a) plots the average routing reliability at Tier-1 as a function of the distance from the local base node. The maximum distance in this graph represents the farthest distance between an XSM and its nearest base-XSS node. Although reliability gradually decreases with increasing distance, we again observe that the spatial reliability distribution is uniform for low-freq traffic.

End-to-end reliability Table 2.4 lists the end-to-end routing performance of *ExScal* and its breakdown at both tiers. It can be seen that Tier-2 has high routing reliability

	Low-freq	**High-freq**	**Bursty**
Tier-1 reliability	86.72%	58.32%	60.17%
Tier-2 reliability	98.73%	94.55%	96.86%
End-to-end reliability	85.61%	55.14%	58.28%

Table 2.4: End-to-end routing reliability.

for all three traffic loads. Also, as seen in Figure 2.6(b), the routing reliability at Tier-2 is uniform across the entire 1.26 km distance. The end-to-end routing reliability thus has a distribution similar to the one at Tier-1 as shown in Figure 2.6(a).

> *Routing fault model*: The probability of routing faults is a function of the amount of network traffic (15% for low-freq versus 45% for high-freq traffic in *ExScal*). Localization faults have non-trivial impact on routing faults. Routing faults have uniform spatial distribution (with careful design of the routing protocol).

2.7 Other Faults

2.7.1 Sensor Faults

Detecting events of interest based on changes in the sensed environment is a common function of wireless sensor networks. For instance, a magnetometer senses

the value of the magnetic field in its neighborhood which changes significantly when a metallic object enters it. Similarly, acoustic sensors can detect objects based on sound produced by them. Sensors in a wireless sensor network are subject to different types of faults which may not only lead to incorrect sensing and detection, but also affect other network functions as seen below.

> *Stuck-at faults*: We say a node is affected by a stuck-at fault if its sensed value is always constant. A special instance of stuck-at faults is railing, in which the sensed value is stuck at the low or high limit of sensor values.

Stuck-at faults may occur in a node because of hardware or software faults. For instance, we observe magnetometers to become desensitized over prolonged exposure to high magnetic field or in severely hot or cold environments. Also, misconfiguring or reading the wrong ADC port could result in sensors producing incorrect values.

> *Debonding faults*: We say a node is affected by a debonding fault if its sensor (or actuator) becomes physically detached or debonded from the rest of the node.

Sensor nodes are often designed in a modular fashion to allow the same processing and radio module to be coupled with a number of different sensors that are mounted on different sensor boards. Debonding faults may occur in such nodes because of prolonged physical wear and tear such as in a control application for monitoring industrial plants where the node is subject to extensive vibrations. A debonded sensor or actuator could produce partial results, e.g. a partially debonded actuator could only exert part of the desired output force; or become completely ineffective producing random output.

> *Byzantine sensor faults*: We say a sensor node is byzantine if its rate of false detections exceeds a certain threshold.

A sensor may become byzantine if it is deployed in adverse environments, e.g. wind-induced motion may trigger false PIR detections for nodes deployed in tall grass. Extreme heat, rain and wind may also lead to persistent false positives. Finally, nodes may observe arbitrary values while sensing if their battery voltage falls below a critical value. Byzantine sensors not only produce incorrect outputs locally, but also impose high load on other nodes that have to route their false detections and create network traffic that interferes with real detection messages. In *ExScal*, we experimentally measured about 1% of the sensors to be byzantine, with these nodes producing as many as 15 false detections per minute.

Kulathumani et al. provide further experimental analysis of the impact of sensor faults such as stuck-at, debonding and byzantine on a vibration control system for fairing structures [45] and present a control scheme that tolerates these faults [46].

2.7.2 Software Faults

The limited computational, memory and bandwidth resources at a node impose restrictions on the amount of processing that can be successfully performed at the node. If this limit is exceeded, processing tasks may not run to completion causing non-deterministic behavior and various kinds of failures. Pointers and memory locations may get corrupted, message buffers may be overwritten, and certain sensing and processing events might get lost. The node might even be forced into deadlock or livelock states from which it cannot recover on its own. Our experiments with the TinyOS [37] operating system show that even some system components could enter deadlock states or become corrupted due to software failures. Although, some of these faults can be avoided by safe programming approaches such as scheduling

application tasks, especially those that access shared resources such as radio, ADC, etc. the decision of abandoning such guarantees at the operating system level in favor of higher efficiency in popular wireless sensor network operating systems like TinyOS implies that software faults may still occur.

2.8 Conclusions

In this chapter, we proposed models for different types of faults that occur in wireless sensor networks based on experimental data. Given the lack of good analytical fault models and adequate simulation tools, we believe that such empirical fault models are extremely valuable to developers and managers of wireless sensor networks. For example, from our deployment fault model, we know that about 5% of deployed nodes will fail and become useless. Similarly, the reprogramming model tells us that of the remaining 95% nodes, only 93% will be successfully reprogrammed and will run the user application. Thus, using the fault models presented above, we can calculate the net effective output or *yield* of *ExScal* to be about 73%. However, since at the time of deploying *ExScal*, these models were not available, our system design was conservative and used a redundancy factor of 2 for safety. Knowing that the net *yield* is in fact much more than 50% will allow us in the future to deploy *ExScal* with much less redundancy, reducing the total system cost.

CHAPTER 3

MANAGEMENT ARCHITECTURE

In Chapter 1, we discussed the important role of management in wireless sensor networks. However, although research on hardware and software platforms and network services and protocols has grown substantially in the last few years, wireless sensor network management has received little attention in the literature. One of the main reasons for this is that the research community has yet to agree on standard protocols for basic sensor network functions such as medium access, routing and reliable transport. Consequently, wireless sensor network systems, both in research and industrial domains, are designed in a vertical manner with each application using its own solution stack for the various networking functions. Naturally, there is tight coupling between most application and management protocols in existing wireless sensor networks.

In this chapter, we first discuss some of the key issues that differentiate wireless sensor network management from that for traditional networks such as the Internet or cellular networks. We then present a brief overview of some existing management solutions for wireless sensor networks. Finally, we propose our management architecture called MASE for wireless sensor networks.

3.1 Differences in Wireless Sensor Network Management

Although the goals of management in wireless sensor networks are similar to those in traditional networks such as the Internet or cellular networks as described in Chapter 1, there are several important distinctions between the two arising out of differences in the network and fault models, deployment and usage scenarios and other such factors. The following are the main distinguishing factors for wireless sensor network management.

1. Standardized architectures and protocols. A key distinguishing feature of traditional networks is their use of standardized protocols. Protocols such as TCP/IP [41], OSPF [40], etc. have been long accepted as the standard in Internet communications. Even for cellular networks, the various communication protocols for data and control communications and for dealing with mobility, handoffs, etc. are agreed upon. An advantage of this is that efficient management protocols, tailored to the specific protocol issues and faults can be designed for these standardized network protocols and can be uniformly applied. For instance, in [35], Hao et al. describe an efficient fault management protocol designed for link state routing protocols such as OSPF that can be easily deployed over existing Internet infrastructures.

In fact, the TCP/IP protocol suite also includes the Simple Network Management Protocol (SNMP) [39] which has standardized the representation of management information in the form of an Abstract Syntax Notation (ASN) and the message formats for exchanging management information; thereby facilitating the easy exchange of management information across the network. Even though SNMP does have its limitations in terms of being reactive as opposed to proactive and some

scalability concerns, it still remains the most widely used network management protocol because it enables network managers to easily manage network performance, find and solve network problems, and plan for network growth. A lot of subsequent research in Internet management has focused on intelligent analysis, correlation and information extraction techniques for data gathered using SNMP. Similarly, GSM and other telecommunications networks use the Telecommunication Management Network (TMN) [73] standard proposed by ITU.

By contrast, the use of vertical solutions in existing wireless sensor network applications raises an interesting paradox for management; on the one hand, the principle of generality dictates that management should not be tied to application-specific protocols, thereby forcing designers to include a separate communication stack for management along with that of the application. On the other hand, it is wasteful to duplicate these functions, especially in networks where resources are extremely scarce. Moreover, applications are likely to choose network protocols that perform best for their deployment and traffic scenarios, thus a general management protocol would be sub-optimal. This was in fact the case in our *ExScal* [1] network, where we experimentally observed [11] that the *ExScal* application routing protocols [21,79] performed significantly better than the management routing protocol [72].

2. Available resources. A lot of the existing network management designs have been in the context of wired networks, not only for the Internet, but also for cellular networks, where the connections from base stations to the Mobile Switching Centers (MSCs) and among MSCs are wired. These wired networks are provisioned with enough bandwidth to support the network information gathering required for management. In SNMP for instance, network information is polled periodically from the

network or sent to the managing entity whenever certain interesting events are detected. Even in the case of wireless cellular networks, dedicated forward and reverse control channels are provisioned in the wireless link between base stations and mobile units for management functions such as registration, channel assignment, handoff, etc. Traditional network management schemes also employ centralized or distributed management agents to monitor incoming network information, search for patterns that may resemble faults in the network and even automatically take corrective actions to repair the detected faults. These management entities have significant processing and memory requirements. In fact, the complexity of management agents is one of the reasons why network management in the Internet is not built into the routers themselves, e.g. the online fault management approach for link state protocols described in [35] introduces a separate agent to monitor the link state database at a router for faults. Finally, these networks are perpetually powered so they do not have to worry about energy or system lifetime issues.

Wireless sensor networks on the other hand use devices with severely constrained resources such as processing, memory and communication bandwidth. Even in hierarchical networks such as *ExScal*, which use some devices with better capabilities, e.g., XSS [43], the amount of resources available for management are much lower than the distributed management agents in traditional networks. Moreover, most network deployments are not wall-powered and hence need to conserve their limited energy. Radio communication is one of the most energy expensive operations in wireless sensor networks, hence the amount of data that can be pushed by a node to central

manager is limited. A delicate balance thus needs to be achieved in wireless sensor networks between the management tasks performed at a central node using data collected from the network and those distributed in the network.

3. Unanticipated fault scenarios. The Internet and cellular networks have been deployed and extensively used for a long time by billions of people worldwide. Coupled with the fact that these networks have adhered to standardized network protocols, a lot of data has been collected about the behavior and performance of these networks and the specific protocols used under different conditions. This had led to the creation of well-developed fault models for these traditional networks. Certainly, simply knowing the fault model is not good enough, especially for management in complex, large-scale systems such as the Internet where searching for and identifying even known fault patterns from the enormous amount of data generated by these systems is a challenging task. Nonetheless, knowing the types of faults that can occur in the system allows the network manager to instrument the various network elements to detect, report and correct these faults. The SNMP management protocol has two main modes of information gathering, one is a periodic collection of critical system variables by a managing entity which then pieces together data from different network elements to detect anomalies, while the other is an asynchronous mode in which certain fault events or errors are trapped and reported asynchronously to the managing entity.

In addition to typical networking faults, wireless sensor networks have to deal with faults arising out of unreliable hardware, limited energy and other resource constraints and physical deployment conditions that traditional networks do not. Some of these faults generate unanticipated system behaviors whose impact is not predictable, e.g.,

if the energy of a sensor node falls to a level sufficient to drive the processor but not to read/write other node components such as flash memory, sensors, etc., the node may produce arbitrary behaviors during reprogramming or sensing. Also, the use of vertical solutions results in a number of different application-specific protocols to be managed, each of which introduce faults of their own.

4. Reliability of information gathering. Internet management protocols such as SNMP are generally implemented using the UDP transport protocol. Although UDP does not guarantee reliable delivery like TCP, the reliability of UDP in the common case is quite high (>98%). Further, since energy and bandwidth are not critical constraints, management protocols can themselves implement mechanisms to improve the reliability of data collection. SNMP for instance, uses timeout based retransmission in case it does not hear back from an agent upon sending a request. Cellular networks use a wired backbone to manage network resources in the base stations and MSCs. The last hop to the mobile unit, which is wireless, uses an assigned channel for data traffic connections. These connections are managed by continuously monitoring the link quality at the base station. If the link quality falls below a threshold, a handoff to a base station with a better link to the mobile unit is initiated. The mobile unit is informed of this change over a dedicated control channel. Connection and mobility management in cellular networks thus also use reliable links.

By contrast, management in wireless sensor networks is implemented on top of unreliable wireless links. Further, management traffic uses the same radio channel and thus has to contend with application traffic in this bandwidth constrained environment. Radio communication is energy expensive, hence the number of redundant transmissions cannot be too high. The reliability of collecting management data can

be somewhat improved by piggy-backing it on application traffic or opportunistically using the application routing protocol for management, however this reliability is still nowhere close to the near-100% reliability of traditional networks. For example, in *ExScal*, the average end-to-end reliability of the SNMS routing protocol was measured to be only about 50% while the reliability of the *ExScal* application routing was 85% which means that a significant fraction of management data could be lost in the network.

5. Planned, well-configured network topology. Network elements such as routers, DNS servers, etc. in the Internet and base stations, MSCs, etc. in cellular networks are deployed and configured in a careful, planned manner. By contrast, wireless sensor networks are often deployed in an ad hoc manner. For instance, in regions that are physically inaccessible due to terrestrial or security constraints, sensor nodes may be aerially deployed. Thus, instead of a planned, well-configured network deployment, managers of wireless sensor networks have to deal with one that may have a high degree of variability.

3.2 Related Work

In this section, we look at existing work that addresses one or more of the management issues in wireless sensor networks as outlined earlier.

An important class of existing research has focused on developing SNMP-like services for wireless sensor network management. One of the first efforts in this area was the development of query processing systems such as TinyDB [57] and Cougar [77], among others. These general purpose query systems could be used to easily and efficiently define various declarative queries to get application or management data

from the network and provided a foundation on which other management tools could be built. The Tiny Application Sensor Kit (TASK) [14] is an example of one such system. The TASK system is typically installed on a handheld PDA-like device and includes tools for easy deployment, system reconfiguration, health monitoring and system lifetime estimation of a wireless sensor network to facilitate sensor network deployment, operation and management even for non-expert users.

Another sensor network management system of note is the Sensor Network Management System (SNMS) [72]. In addition to providing a querying mechanism for wireless sensor networks, SNMS also provides support for easy logging and log retrieval of management information and enables the manager to command and control the network through network based operations such as power management (sleep & wakeup) and node reset. SNMS is, to a large extent, application independent and includes its own networking stack for management. Additionally, SNMS provides middleware and tool support to easily allow applications to expose certain attributes of interest to the management system. However, SNMS only focuses on mechanisms that enable different management tasks and not the actual policies and actions for these tasks; for instance the job of interpreting the collected results is external to SNMS as is deciding what to do when the collected information is incomplete or perhaps incorrect.

The Sympathy [65] system is designed for fault detection at a central base station in a data collection application in which nodes periodically send data to a base station. Sympathy is thus not entirely application independent and exploits knowledge of a specific application traffic pattern to define certain fault metrics such as link reliability, path reliability, etc. Sympathy agents at each node monitor the flow of application

traffic and evaluate these defined metrics which are then communicated to the base station using additional messages. This information is then used by a failure detector program at the base station which tries to localize the type and the source of the faults in the network and notifies the user. A similar approach is used in [69] where the fault management system exploits not only the continuous data traffic flow in the network to piggy-back management information, but also uses the route update messages in the routing protocol to effect changes in routing paths for suspected nodes in order to trace failed nodes. By contrast, the MANNA [67] network management architecture is defined in the context of event-driven systems. Fault management under MANNA is therefore similar to the traditional SNMP, where a central manager periodically polls the nodes for their status and tries to locate faults in the network. However, as expected, MANNA imposes additional communication overheads and suffers from false positives in fault detection due to unreliability of multi-hop wireless communication. Although the above architectures represent a gradual move towards autonomous fault detection in wireless sensor networks, they do not address the issue of reacting to these faults in an automated manner — this is ultimately left to the human manager.

As sensor networks undergo several changes in hardware and software after their deployment due to changing environment conditions, faults and application scenarios, reconfiguration is an important management task that has received considerable attention in existing literature. Several protocols and network services have been proposed for performing configuration updates in the network. These include protocols for updating parameters or small code capsules such as Trickle [56] and for bulk

dissemination of entire programs such as Deluge [38], MNP [49], etc. These reconfiguration services expose current configuration state to the network manager and allow new configurations to be downloaded and installed in the network. However, as we shall see in Chapter 4 ([10]), existing reconfiguration services are non-stabilizing under certain types of faults. Further, in these faulty states, these services do not allow the network manager to install a new, correct configuration either, thereby causing the network to be unmanageable.

Since energy is a critical resource for battery powered sensor nodes, power management has received considerable attention in sensor network research. A common approach for power management in sensor networks is determining the optimal number of nodes that need to be active in order to maintain certain application QoS guarantees such as coverage and connectivity. This has led to research on investigating policies for scheduling sleep and wake-up of sensors [15, 50]. Another approach to power management which is especially popular in systems for detecting rare events such as intrusions, is the use of sentries [33, 36]. Sentries are a group of nodes that are kept active in the network to guarantee just enough coverage for detecting an intruder and waking up neighboring nodes upon detection to perform more fine-grained detection, classification and tracking. The management issues involved in this process are the selection, rotation and load-balancing of sentry responsibilities in the network.

The systems described above thus present solutions for many of the management tasks outlined earlier for different network deployment and application scenarios. However, in general, no single architecture that unifies the different management functions exists today.

3.3 Management Schema

In this section, we describe the schema for wireless sensor network experimentation and execution which illustrates the different tasks that a network manager has to perform and the management services that are required to do so. In the rest of this dissertation, we use the term "experiment" to denote not only experimentation in the traditional testing context but also to denote the normal execution of a (well-tested) wireless sensor network such as *ExScal*.

The schema for managing a wireless sensor network experiment typically involves the following key steps:

- *Specification.* First, a network manager needs to precisely specify the different elements that form the experiment to be performed. This specification consists of the programs that will be run as part of the experiment, the different platforms that each program runs on, configuration parameters, dependency information such as order of invocation and the expected output. In scenarios such as testbed experimentation, users may not know internal configuration details about the network. The manager might also want to include certain management components such as health monitoring or data logging in addition to the user specified programs. In such cases, the manager needs to append the user provided specification to include this information.

- *Configuration.* Once an experiment is specified, it then needs to be correctly configured on the network. Configuration involves loading the specified programs on the respective devices, setting device parameters such as frequency, power level, etc. and starting the program processes in the correct order as

specified. Experiments may be configured using a reliable control channel such as Ethernet for testbed networks or over a multi-hop wireless network as was the case in *ExScal*. In either case, the manager needs to ensure that the correct configuration is loaded in the entire network.

- *Online monitoring and interaction.* Once an experiment is started, the manager needs to monitor its execution. This monitoring could be local, e.g. locally checking if a user process has crashed, or it could be global, e.g. periodically collecting node and link status information at the base station. Faults may occur at any time during an experiment and compromise its output, hence monitoring needs to be reliable and fast.

  Conversely, a manager may also need to provide certain information to an ongoing experiment. This information could be trace data injected into the network at appropriate times, parameter or other configuration updates or response to a detected fault.

- *Analysis and evaluation.* Upon completion of an experiment, the manager needs to assess its output. This involves analyzing management data logged at nodes or received at the base station during the experiment. If the analysis reveals that the data is consistent, the user could be notified of the experiment results. Alternatively, the analysis could reveal faults or inconsistencies during experiment execution such as half of the nodes not receiving the correct configuration parameters, in which case the manager might need to re-run the experiment.

3.4 The MASE Architecture

In the previous subsection, we described a schema that outlined the key steps in wireless sensor network management We now present our management architecture, called MASE for **M**anagement **A**rchitecture for wireless **SE**nsor networks, that takes a unified view of these different management tasks. We describe in this section, the various components in the MASE architecture and the specific tasks they address. Detailed descriptions, correctness and performance issues for the algorithms used for each component are discussed in subsequent chapters.

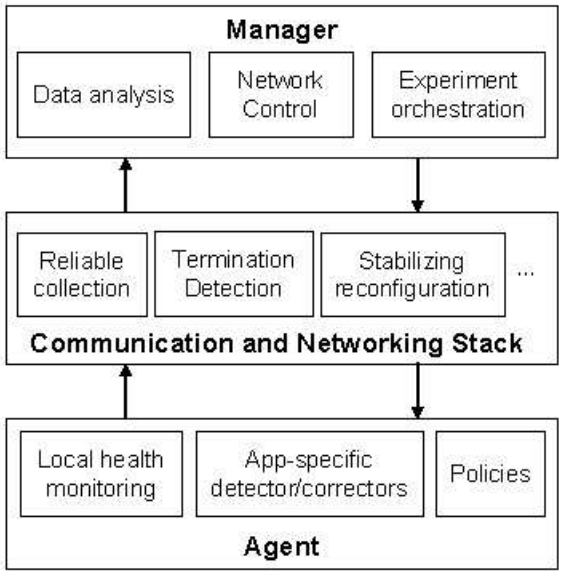


Figure 3.1: MASE architecture for wireless sensor network management.

Figure 3.1 shows the overall component structure of MASE. As seen in the figure, MASE is a composition of management elements at three different levels, viz., distributed management agents deployed at individual nodes in the network, middleware

components for communication and networking deployed across network nodes and a central network manager. We discuss the roles of each of these elements in further detail.

3.4.1 Communication and Networking Stack

The Communication and Networking stack is the key component that bridges the local management agents with the network manager in MASE. Primarily, this stack provides middleware support for two important management functions identified in the management schema.

1. Stabilizing reconfiguration. As described in Section 3.3, the deployed sensor network needs to be configured properly for it to work correctly. This is an especially challenging problem for field deployments where sensor nodes are configured over unreliable wireless channels. Due to changing requirements or detected faults, application parameters, modules or programs may need to be changed during an experiment, thus maintaining consistency across the network in the presence of these changes is equally important.

As described earlier, existing research has focused on services and tools that reliably and efficiently allow the network to be reconfigured in the field. However, we find that due to certain faults such as network partitioning or transient message and state corruption, existing reconfiguration services could enter bad states from which a manager might never be able to guarantee convergence to the correct configuration.

MASE therefore provides a stabilizing reconfiguration layer using which a network manager can guarantee that the sensor network always converges to the desired correct

configuration. Our solution uses a novel approach to stabilization called *Human-In-The-Loop* Stabilization in which the network first converges to a safe configuration state from which the manager can then restore the correct configuration.

2. Reliable monitoring. In Section 3.3, we described the importance of monitoring in the overall schema of wireless sensor network management. However, unlike traditional wired networks such as the Internet or private Local Area Networks, the manager of a wireless sensor network does not have the ability to remotely log on to deployed nodes over a reliable transport layer such as TCP, nor are network devices accessible over a single wireless hop from some wired node as is the case in cellular networks.

Reliably collecting monitoring information over a multi-hop network of unreliable wireless links is thus a challenging management task. Moreover, perfect reliability is critical for some wireless sensor networks such as experimental testbeds or control applications. For instance, without reliable knowledge about the status of devices in an experimental testbed, users would not be able distinguish testbed faults from those in their programs, thereby rendering experimentation ineffective. Similarly, incorrect information about the state of sensors and actuators in a control application could lead to incorrect actuation outputs being applied.

MASE provides two services using two different approaches for reliable network monitoring. The first approach is well-suited for scenarios such as testbed management where some information needs to be reliably collected from every network node. For this, we present Chowkidar, a reliable monitoring service that guarantees the strong fault-masking property despite faults that occur during the collection process. The second approach is suited for field deployment scenarios where energy efficiency

is critical. In this approach, we identify a small fraction of nodes in the network whose monitoring information is sufficient to deduce the state of the entire network. We present the Reporter service, which demonstrates this approach in the context of the well-known management problem of termination detection.

3.4.2 Distributed Agents

The distributed agents in the MASE architecture are components that run at the node level in the deployed wireless sensor network. These agent components thus co-exist with application software components and may in turn share or monitor the state of the application. Similar to SNMP, MASE agents may generate data periodically or when certain interesting states or events are detected; which could then be sent to the base station using the networking stack. Additionally, distributed agents in MASE can also perform operations such as correcting local state, enforcing management policies, etc. Based on the roles the play, different types of agents are defined in MASE as described below.

1. Health monitoring agents. These are responsible for monitoring the health of the node and its local neighborhood. Some health attributes such as upness, remaining battery power, quality of sensor data (and hence sensor health) can be monitored by an agent locally, while others such as status of radio transmitter and receivers require exchanging messages with neighbor nodes. By aggregating information from all nodes in the network, a manager can infer the total network health.

2. Application specific detector/corrector agents. These agents are instrumented along with the application and can access application state to detect when certain types of predefined faults might occur and in some instances to correct these

faults. Even though these agents are tailored to specific application fault scenarios, there exist common fault patterns across multiple application components, hence, in MASE, the same detectors and correctors can be instrumented to monitor multiple application components.

For example, a common pattern in sensor network applications using TinyOS is the use of a busy flag to prevent multiple concurrent accesses to shared components that provide a split phase response. Thus, a component wishing to send a message over the radio invokes the MAC layer in TinyOS and if the call is accepted, sets a busy flag waiting for the current message send to finish. When the MAC layer finishes transmitting the message over the radio, it notifies the calling component through a sendDone event at which time, the calling component resets its busy flag. However, as described in Section 2.7.2, under certain fault conditions, this notification event may be lost, in which case the calling component will stay busy forever and not be able to send any messages. This scenario can be detected by defining an application-specific upper bound on the time taken to complete the split-phase operation and by monitoring the state of the busy flag. If the busy flag is continuously enabled for a time exceeding this upper bound, a fault can be detected. Correspondingly, this fault can be corrected locally by resetting the busy flag in the calling component.

This same detection/correction pattern also applies to other split-phase operations such as reading sensor values from an ADC port, reading and writing to external flash memory, etc. MASE detector/corrector agents are therefore instrumented to accept, in this case, the address of the busy variable and the component-specific timeout value, as input parameters so that the same agent can monitor multiple application components.

3. Policy agents. These agents are responsible for maintaining and enforcing the policies specified by the network manager. As described earlier, it is sometimes impossible or too expensive to design sensor network systems with perfect fault-tolerance. A common strategy in such scenarios is to design tolerance for common case fault scenarios and manage network operation through predefined policies under the occurrence of rare faults.

For example, our stabilizing reconfiguration design is guaranteed to converge to the correct configuration. However, during convergence these nodes may in fact exchange application messages even though they are running different versions of that application. Worse, these messages may have different formats and could lead to incorrect behavior if interpreted and acted upon. e.g., if the values for wake-up and sleep in an application are reversed between two versions of an application, a node may receive a wake-up message from a node running a different version and go to sleep. Moreover, as part of entering sleep state, it could turn off its radio and shut down other components, thereby preventing reconfiguration from stabilizing.

A manager may specify one of several policies to deal with such a fault condition. One policy could be for nodes to disregard messages if they belong to different versions. Alternatively, applications could be designed to be backward compatible in which cases, nodes with newer versions would be allowed to accept messages from older versions but not vice versa.

Although enforcing network policies involves detecting and controlling faulty behavior, policies are not necessarily application specific. In fact, one of the policy agents in MASE governs whether or not an application should be allowed to execute. This policy agent, which was part of the *ExScal* design, uses a watchdog timer to

monitor how many times an application misbehaves, in the form of deadlocks, live-locks, or crash/restarts, and in accordance with the specified policy may decide not to invoke it and enters the *Trusted Base* mode in which the faulty application can be replaced.

3.4.3 Network Manager

The network manager is the top-level component in MASE and is responsible for deploying management services and agents in the network, collecting data from them, analyzing it and determining the appropriate response to maintain the network in the optimal state. The network manager in MASE thus has two main components as described below.

1. Data analysis and interpretation. This component receives management data from the various node agents in the network and is responsible for analyzing and interpreting the data to derive higher-level results.

One example of such data analysis is the predicate evaluation framework which we are developing as part of the Chowkidar testbed health monitoring system [74] for our Kansei testbed [31]. Chowkidar provides node and link health data from local monitoring agents at the manager. The job of predicate evaluation is to detect whether additional, higher-level faults have occurred from this data. For instance, if an Ethernet hub in Kansei has failed, Chowkidar reports failures individually for all nodes attached to this hub. However, the actual fault in this case is the failure of the hub which can only be inferred by correlating the failures of these attached devices. Another example is detecting sensor faults in Kansei where the data analysis component at the manager spatially correlates sensor data from nodes in the same

geographical region to detect sensing outliers. Another example of predicate evaluation is determining the state of the network, for example, have all (or more than x%) nodes in the network initialized or terminated successfully?

2. Network control. This component is responsible for determining the appropriate management actions that need to be taken during or after an experiment in response to the management information collected during its execution. It is also responsible for orchestrating the experiment execution to implement this response.

For example, network health data analysis might reveal that only 50% of the expected number of nodes are functional whereas the application requires at least 60% nodes to maintain the desired quality. The network control component could then respond by activating sleeping nodes in the network so that the effective application quality is maintained at 60%. Similarly, if the sensitivity of deployed sensors is detected to change over time, the manager could invoke a reconfiguration operation to update the sensing threshold parameter.

MASE provides support, wherever appropriate, for both *Human-In-The-Loop* and automated network control. The different MASE services provide well-defined interfaces using which a human manager can easily obtain management data from the network and implement the necessary response actions.

3. Experiment orchestration. Network control tasks often require some human assistance. In the example described above, if there are no sleeping nodes to be activated, the appropriate response could be to deploy new nodes or repair failed ones, which can only be achieved with human intervention. Similarly, if an unanticipated fault is detected, the human manager needs to determine the detector, corrector or policy agents that need to be included to deal with them in the future.

However, there also exist several tasks which can be automated. For example, as described in the management schema, the manager, given a user program as input, may need to instrument this program with certain management components such as data logging or health monitoring or determine other network components that this program may depend on for correct execution in the particular network environment. These tasks can be automated by defining appropriate language constructs and configuration interfaces that can be easily parsed by a management tool which can also instrument the experiment with the appropriate management components.

We also identify patterns of sensor network experiments whose execution can be automated. One such pattern is multi-phase execution in systems such as *ExScal*, described in Chapter 2. In multi-phase systems, different applications are invoked one after the other according to a pre-defined sequence. Before invoking a new phase, a manager needs to verify that the previous phase has completed successfully. Thus, given a multi-phase user experiment, we can instrument each application therein with a termination detection service, such as Reporter, so that phase termination can be detected by the manager. Automated phase control of these experiments can then be achieved by letting users specify, through a standard configuration interface, the order in which phases need to be executed and a transition rule for switching between phases. One such rule could be to switch from one phase to the next if a certain fraction of nodes in the network (say 90%) have finished executing that phase.

MASE therefore provides an experiment orchestration framework which includes components for automatic code synthesis, experiment configuration and execution that automate management tasks such as the ones described above during a wireless

sensor network experiment. We present the MASE experiment automation framework in more detail in Chapter 6.

3.5 Conclusions

In this chapter, we presented the schema for wireless sensor network management which identifies the key tasks that need to be performed by a network manager. We then described our MASE architecture for wireless sensor network management which addresses these tasks. The MASE architecture is compositional hence different components that support the same interface can be used interchangeably. For example, using the Chowkidar service for reliable information collection is well-suited for testbed scenarios, but for outdoor deployments, the more efficient Reporter service might be more appropriate.

In the following chapters, we will discuss the key architectural elements of MASE in further detail.

CHAPTER 4

STABILIZING RECONFIGURATION

4.1 Introduction

The ability to reconfigure wireless sensor networks after they have been deployed is critical for dealing with the effects of changing environment conditions, node and network faults, software bug fixes and changing application requirements. Reconfiguration of a given application can have several forms. In some cases, reconfiguration could mean changing the values of certain application parameters. Changing the signal-to-noise threshold for sensor detection or changing the transmission power level are examples of parameter reconfiguration. Alternatively, reconfiguration could imply replacing certain modules within the application; e.g., a sensor-based detection application could replace a low-pass filter module with a band-pass one depending on the environmental noise conditions. Existing systems such as Mate [54], SOS [34], Contiki [27], etc. allow specific modules within an application to be replaced dynamically. Finally, in cases where the previous application program is faulty or application requirements change, reconfiguration could also mean replacing the entire application itself.

Regardless of the type of change involved, existing reconfiguration protocols allow a network operator to specify new configurations to be run on sensor nodes in the network. They also ensure dissemination of updates to the desired nodes in the network in a reliable and timely manner. Trickle [56], MOAP [70], Deluge [38] are some examples of reliable data dissemination protocols used for reconfiguration. Most reconfiguration protocols, including the above, use version numbers to distinguish new configurations from old ones. Due to physical constraints, version numbers are bounded in size, hence reconfiguration protocols use wraparound arithmetic to handle rollovers. For example, in a reconfiguration protocol with 3 version numbers {0,1,2}, a configuration would be updated from version 0 to 1, then to version 2 and then again to version 0. In the common case where version numbers in the network are consistent, these reconfiguration protocols can successfully update the network configuration with a new one. However, under certain fault conditions, such as when nodes start executing the reconfiguration protocol with arbitrary initial version numbers, this may not hold - in fact the reconfiguration protocol may not stabilize.

Given that most implementations use a large version space – Deluge uses 16-bit version numbers, allowing for 65536 distinct versions – one might be misled into believing that rollovers would never occur in practice. However, data or message corruption, operator errors and other such faults could lead to the version numbers in the network being changed to arbitrarily high values, thereby necessitating rollover.

Experiences from a real deployment. In December 2004, as a part of the *ExScal* [1] experiments, we deployed over 1000 sensor nodes in a 1.3km x 300m area, which is one of the largest wireless sensor networks deployed till date. During the manufacturing process for these nodes, the factory installation of the baseline program

was performed in batches, with the downloading software automatically incrementing the version number between batches. Consequently, sensor nodes delivered from the factory were incorrectly initialized with different version numbers. Worse, this set of version numbers was such that no single version dominated all the others (this is analogous to having all 3 versions – 0, 1 and 2 – for the 3-version reconfiguration protocol). Unfortunately, since Deluge, the reprogramming protocol used in *ExScal*, is non-stabilizing under this fault, network operators in *ExScal* could not deploy these nodes and assume the risk of letting them download the same program onto each other forever. As a result, a manual, cumbersome procedure had to be executed which involved wirelessly reprogramming all nodes with the same version number in a one-hop setting. During this procedure, operators had to carefully monitor all nodes to ensure that they were re-initialized to the same version. This procedure imposed an additional overhead of almost 10% on the entire human effort in *ExScal*, thereby motivating the need for stabilizing reconfiguration. Our calculations show that without stabilization, the deployed network would have consumed 10-15% more power without doing any useful work due to persistent reprogramming operations involving flash read/writes and message transmissions.

4.2 System Model

Our system consists of two parts – a network of wireless sensor nodes and a framework for reconfiguring these nodes. In this section, we describe the network and the reconfiguration framework setup that is assumed in the rest of the chapter.

4.2.1 Network Model

We assume a multi-hop wireless sensor network of n nodes. Nodes can join or leave the network at any time, as would be the case with mobile nodes or nodes that are duty-cycled for power management, with the exception of one (or more) base station(s) that initiate network reconfiguration and are always assumed to be up. We also assume that at any time the nodes that are up and running the reconfiguration protocol form a connected network.

We assume that wireless links between neighboring nodes are reliable. Although this is a strong assumption, experimental studies such as the one by Zhao and Govindan [80] have shown that there exists an *inner-band* radius in which the packet reception probability is uniformly high. The per-hop reliability of wireless links can also be substantially improved by mechanisms such as explicit or implicit acknowledgements with retransmission, TDMA scheduling, etc. as shown in [48, 64, 78]. We define the diameter D of the network to be the maximum number of such reliable hops in the network.

4.2.2 Reconfiguration Framework

Configuration. Associated with each node in the network are one or more configurations. As discussed in Section 4.1, a configuration can be of several forms such as a parameter, a module or a program. For simplicity, we assume that all nodes in the network share the same configuration, although our results also apply to networks with multiple configurations per node or where groups of nodes have different configurations.

We thus define a configuration C as a block of data that is stored on each node in the network. Each configuration C is associated with a version number V and meta-data M, such as a CRC or hash value which is different for different configurations. The configuration of a network can be updated by downloading a new configuration on *all* nodes in the network – this is the reconfiguration problem.

Reconfiguration. Reconfiguration consists of replacing an existing configuration C in the network with a new configuration C'. The reconfiguration service is specified by the following interface – *reconfigure(C', M', V')* – where C' is the new configuration, M' is its associated meta-data and V' is the version number for this configuration. Intuitively, a reconfiguration invocation will succeed in replacing the existing configuration C having version V with the new C' if $V' > V$.

We assume a reconfiguration system with 3 versions – 0, 1 and 2 – in which newer versions are distinguished from older ones according to the following update rules – (i) version 1 > version 0, (ii) version 2 > version 1 and (iii) version 0 > version 2. It can be easily seen that 3 versions are *necessary* to identify new configurations correctly because a 2-version protocol would have update rules – (i) version 1 > version 0 and (ii) version 0 > version 1 – which are contradictory.

A common generalization of the update rules assumed in this chapter to an N-version system, which is used in Trickle, Deluge and other reconfiguration protocols, is the use of a sliding window of size N/2. Under this scheme, version v_2 is newer than v_1 if (v_2 - v_1) mod N < N/2. Implementations of such schemes often use signed integers with normal arithmetic operations. We use positive integers with wraparound for simplicity. It should thus be noted that in the rest of the chapter, the {>,<} operators for comparing two versions follow the version update rules described above.

Update protocol. Any practical reconfiguration protocol must be reliable and execute in bounded time. Reliability implies that starting from a correct initial state, the reconfiguration protocol updates all nodes in the network and does not leave the network in an inconsistent state. Bounded time execution implies that nodes propagate configuration updates in bounded time. Since the up nodes in the network are connected, the total time in which an update is propagated in the entire network is also bounded, although this bound is a function of the diameter of the network.

Protocol *PeriodicBroadcast*
Const T : integer
Var $vnum$: integer
m : message
Actions

$\langle A_1 \rangle :: \; Timer.fired \overset{[kT..(k+1)T]}{\longrightarrow} \; bcast \; m(vnum)$
[]
$\langle A_2 \rangle :: \; rcv \; m(v) \longrightarrow \underline{\textbf{if}} \; (v > vnum) \; vnum := v \; \underline{\textbf{fi}}$
[]
$\langle A_2' \rangle :: \; rcv \; m(v) \longrightarrow \underline{\textbf{if}} \; (v > vnum) \; vnum := v \; \underline{\textbf{else}}$
$\underline{\textbf{if}} \; (v < vnum) \; bcast \; m(vnum) \; \underline{\textbf{fi}}$

Figure 4.1: A canonical periodic broadcast based reconfiguration protocol.

We consider two variants of a canonical periodic broadcast based reconfiguration protocol as shown in Figure 4.1. In both variants, each node advertises its version at a random instant in each interval of time T (Action A_1). Neighboring nodes receive this advertisement and update their state if they have an older version (Action A_2). The only difference in variant 2 is that if a node hears that a neighbor has an older version, it broadcasts its own data to enable its neighbor to catch up faster (Action

A_2'). Thus, from a steady state, if a new version is introduced in the network, variant 2 propagates it more aggressively. Also, once every node has acquired the update, the protocol falls back to the slower broadcast rate. Since the second variant subsumes the actions of the first, we will restrict subsequent protocol presentations to this variant, except when we evaluate and compare the performance and stabilization properties of the two variants.

It can be easily seen that this protocol satisfies our assumptions of reliability and bounded latency. As each node continually broadcasts its version number and the network is connected, a new version will eventually propagate in the entire network. Further, since a node broadcasts its version at least once every interval, this propagation will complete in bounded time. This update protocol is similar to the one used in Mate, Deluge and SOS, among others, with minor differences in the exact timing parameters of the various actions.

It should also be noted that the protocol actions presented above only deal with updating the version information in the network during reconfiguration. Upon updating its state after learning of a newer version in the network, a node initiates a download phase to get the actual configuration associated with this version.

4.2.3 Rate of Updates

We assume a bounded rate of invocation for the reconfiguration protocol. Specifically, we require that the time between successive updates must be at least 2D$\times$T where T is the broadcast interval in the update protocol and D is the network diameter. As we shall see later, this rate of updates corresponds to the worst-case convergence time of the reconfiguration protocol.

4.3 The Problem of Non-stabilizing Reconfiguration

In this section, we describe the problem of cycling or non-convergence of version numbers that may occur in the reconfiguration protocol presented above, under certain fault conditions. The protocol presented in Figure 4.1 guarantees that starting from a *good* state – one in which all nodes have the same version number v, executing the protocol with a new version update v' such that $v' > v$ results in all nodes eventually acquiring the new version v'. However, an important question now arises: how does this protocol behave when started from, or somehow driven into a *bad* state – one in which not all nodes have the same version number. In particular, we focus on the problem of stabilizing from a global state in which all 3 version numbers exist simultaneously in the network.

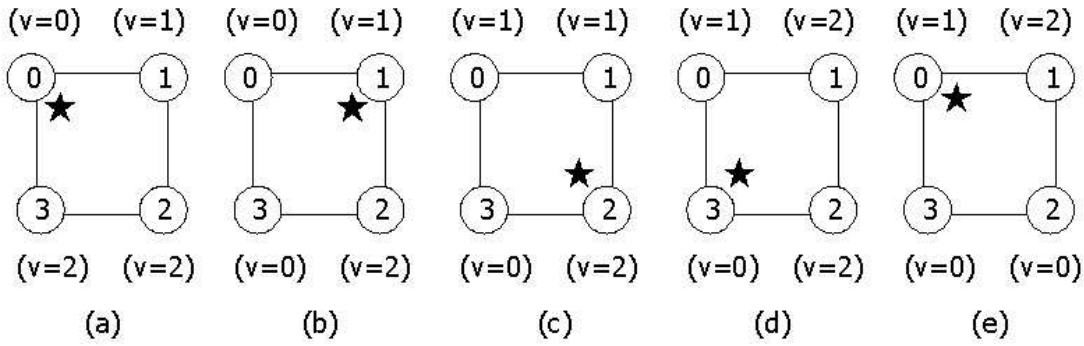


Figure 4.2: Cycling of version numbers.

Figure 4.2 illustrates this problem in a 4-node network with initial state as shown in (a). Edges between nodes indicate wireless connectivity. The $\star$ symbol indicates which node will transmit next in the given execution trace. As shown in the figure, after nodes 0, 1, 2 and 3 transmit in that order (Figure 4.2(b)-(e)), the network reaches

a state in which all 3 version numbers still exist in the network. If the same sequence of broadcasts occurs in each interval, the version numbers will never converge. It can be argued that due to randomization of transmission times in each interval (protocol action A_1), it is unlikely that such a worst-case sequence can be obtained forever, and that version numbers will eventually stabilize. However, the same argument may not hold for a much larger network in which nodes have arbitrary versions to begin with, and in fact the version numbers may indeed keep cycling in the network forever. In a general, N-version system, such a scenario may occur if there exist at least 3 distinct version numbers in the network such that no one version is newer than all of the others. We call such sets of version numbers as inconsistent.

Definition. A set S of distinct version numbers is defined to be *inconsistent* if no element of S is newer than all other elements of S. Formally,

$$\text{S is inconsistent} \equiv (\exists x, y, z : x, y, z \in S : x > y \wedge y > z \wedge z > x)$$

The above definition works for our reconfiguration protocol model where the sliding window is of size N/2 and hence 3 versions form a cycle. However, in general, a reconfiguration protocol may have arbitrary length windows which would imply an arbitrary number of nodes required to form a cycle. We therefore present the following generalized definition for an inconsistent set:

$$\text{S is inconsistent} \equiv (\forall x : x \in S : (\exists y, z : y, z \in S : x > y \wedge x < z))$$

4.3.1 Causes of Non-stabilization

In this subsection, we present some of the reasons due to which a reconfiguration protocol may enter a globally inconsistent state from which version numbers cycle through the network and may never stabilize.

1. Operator errors. Reconfiguration protocols are often designed under the assumption that updates are initiated by a single operator or node. However, this is not always the case in practice. Large-scale wireless sensor networks may be administered by several operators who manage different regions of the network at the same time. A possible scenario in such networks is one where conflicting updates are initiated by multiple operators. Alternatively, the same operator may also mistakenly initiate multiple updates in an unsafe manner. As illustrated by the *ExScal* experience in Section 4.1, an operator could also initialize the network in an arbitrary manner, leading to non-stabilization.

2. Transient corruption. One of the main challenges in sensor networks is the severely constrained nature of node and network resources. Sensor nodes have limited processing capability, memory and battery power - all of which serve as potential sources of data corruption. Configuration updates are communicated as messages that are subject to corruption either due to undetected packet errors due to collisions or accidental buffer overwrites within the send/receive messaging stack in a node. Often, nodes store critical data like version numbers and configurations in non-volatile flash memory to survive crashes and reboots. However, flash read/write operations may fail or return arbitrary values when executed under low battery power. These and other such factors could result in transient corruption of data in one or more nodes, thereby producing globally inconsistent sets.

3. Network dynamics. Sensor networks are often deployed in harsh physical environments in which they may be subject to loss of radio connectivity and network partitioning effects. Also, nodes are often put into a low-power sleep state for power

management. Similarly, in solar-powered networks, some nodes deployed in sunlight-denied areas may only intermittently join the network. Thus, a partitioned or sleeping node may miss subsequent configuration updates. Node mobility may result in nodes from a different region or even from a different network joining the network. Under all of the above conditions, a network could be left in a state where different nodes or regions of nodes in the network have different configurations that form an inconsistent set.

4.3.2 Self-stabilization and Related Work

In his seminal paper [25], Dijkstra defined a self-stabilizing protocol as one which starting from an arbitrary state, converges to a legitimate state in finite time. In [44], Katz and Perry describe a reset scheme based on a global snapshot by a central leader to achieve self-stabilization. In [4], Arora and Gouda present a distributed reset protocol where nodes locally initiate reset requests that are processed along a self-stabilizing spanning tree. In [6], Awerbuch et al. present a generalized reset protocol for arbitrary graphs which uses local checking and correction to achieve self-stabilization. In [8], Awerbuch et al. extend this idea to deal with problems that can be locally checked but require global correction to achieve self-stabilization.

Our work, though closely related, differs from the above in two key aspects. First, we assume a wireless network model in which we do not require unique identifiers for nodes and neighbor links in the connectivity graph. The second difference is that in our problem, we cannot achieve self-stabilization without the involvement of an external agent - the Human-In-The-Loop.

Proposition 1. There does not exist a reconfiguration protocol with bounded version numbers that is self-stabilizing.

The ideal invariant (or legitimate state) for a network reconfiguration protocol can be stated as:

Every node in the network has the most recent configuration.

To be self-stabilizing, a reconfiguration protocol should converge to this invariant on its own. However, in the presence of an inconsistent set, it is not possible for any node or group of nodes in a wireless sensor network to decide which version number represents the most recent configuration. Further, there may exist multiple configurations in the network that share the same version number due to rollover. We thus contend that a reconfiguration system with bounded version numbers cannot self-stabilize to the ideal invariant stated above.

We therefore break down the process of stabilization into two phases. In the first phase, we require the reconfiguration protocol to guarantee convergence to the following weaker invariant:

Every node in the network has the same configuration.

In the second phase, we let the decision of which configuration is the most recent one be made by the Human-In-The-Loop and present an external reset mechanism for converging to the ideal invariant.

Despite not being self-stabilizing, a reconfiguration protocol guaranteeing the weaker invariant stated above is quite useful in practice. Recall from Section 4.2.2 that reconfiguration involves not only identifying the latest configuration but also actually obtaining it from other nodes. An epidemic protocol such as Deluge, for

instance, keeps trying to download an entire program from neighbors that advertise a newer version. Thus, as long as version numbers keep cycling in a wireless sensor network and do not stabilize, nodes continue to expend precious resources such as processing, radio and battery, resulting in degraded performance and reduced lifetime of the network. Even if this cycling problem is detected by an operator, there is no version number that the operator can download to break the cycle. However, once the protocol stabilizes to the weaker invariant with a single version, an operator can download a higher version to update all the nodes.

4.4 Stabilizing Reconfiguration Protocol

In this section, we present a stabilizing solution to the reconfiguration problem discussed above. Our solution uses local detectors and correctors to guarantee convergence to the weaker invariant without imposing any additional communication or coordination overhead. We also present an external reset mechanism initiated by the Human-In-The-Loop to guarantee convergence to the ideal invariant.

4.4.1 Local Detection and Correction

We first prove that the existence of a stable inconsistent set in the network can be locally detected by a node in bounded time. By stable, we mean an inconsistent set that does not converge before this bounded time. For inconsistent sets that are not stable, stabilization is trivially achieved. To prove consistency of our detectors, we show that local detection at a node implies the existence of an inconsistent set within a bounded computational history. Our detector thus does not generate any false assertions.

Theorem 1 *(Completeness of local detection).* For every computation starting from a global state having an inconsistent version set, and in which the inconsistency persists during the entire computation, there exists at least one node with an inconsistent version set in its local computation history within time D×T.

Proof. We first show that at least one node detects the inconsistency locally. Assume for contradiction that this does not hold. This implies that each node in our 3-version system can assume at most 2 distinct versions in the given computation implying that its version number can change at most once. To achieve the contradiction we show that the total number of version changes in the network is a monotonically increasing function. Since an inconsistent set exists in the network, there must exist at least one neighbor pair with different version numbers. It therefore follows from the protocol actions that in the next interval, at least one of these nodes must change their version number. Since the inconsistent set persists, the number of version changes exceeds n at some point implying that at least one node undergoes two version changes, thereby contradicting our assumption.

We now prove that the version history of at least one node contains an inconsistent set within an interval D×T. Since inconsistency persists in the network, there must be at least two nodes with different version transitions enabled in the initial state. This is because if all nodes have no transitions or the same transitions enabled, it implies that the network has converged or will converge in the next interval. Without loss of generality, consider two nodes n_1 and n_2 that undergo transitions $0 \rightarrow 1$ and $1 \rightarrow 2$ respectively. Consider the shortest path between n_1 and n_2 and consider the immediate neighbors of n_1 and n_2 along this path (n_1' and n_2' respectively). In the next interval, n_1 and n_2 will either be dominated by their neighbors, dominate them

or discover them to be the same. If either of n_1 or n_2 is dominated in the next interval, we immediately have the witness that locally detects the inconsistency. If n_1 or n_2 dominate their neighbors then we now have the same transitions occurring at nodes n_1' and/or n_2' and we can carry out the same proof for these new node pairs which are at least one hop closer to each other than the original pair. Finally, we have the case where n_1 and n_2 are the same as their neighbors. However, since n_1 and n_2 had different versions to begin with, this symmetry will be broken at some point along the path. Also, since the path between n_1 and n_2 can have length at most D, this symmetry-breaking will take place in D×T time which means that at least one node along this path will undergo more than one transition in a window of time D×T, leading to an inconsistent set in its local history. □

Theorem 2 *(Consistency of local detection).* In any computation with correct protocol invocation, if the local history at any node in the last D×T time contains an inconsistent set, there must exist a global state with an inconsistent set in the last 2D×T time in the history of this computation.

Proof. Consider that a node whose local history at time t contains an inconsistent version set in the last D×T time. This implies that this node assumed all 3 versions at some point in the interval [t-D×T .. t]. Without loss of generality, assume that the version observed by the node at time t is version 2. This scenario is depicted in Figure 4.3.

From the rate assumption in Section 4.2.3, we know that there can be at most one reconfiguration invocation in the interval [t-D×T .. t]. If there was no invocation in this interval, no new versions could be added to the network after time t-D×T. Thus,

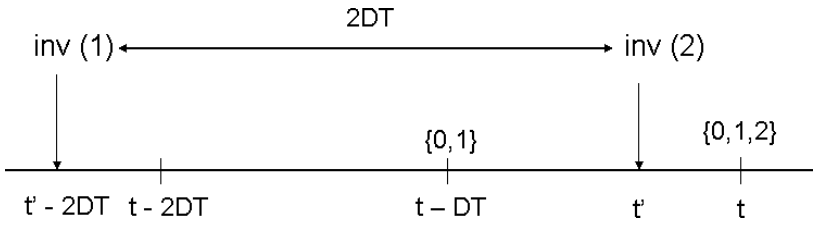


Figure 4.3: Consistency of local detection.

if this node assumed versions 0,1 and 2 in this interval, they must all have been present in the network at time T-D×T, thereby proving the theorem. Now consider that there was one invocation in this interval. Again, without loss of generality, assume that this invocation was for version 2 as shown in Figure 4.3. This implies that at time t-D×T, versions 0 and 1 must have been present in the network to be assumed by the node later. Now consider the time interval [t-2D×T .. t-D×T]. According to our rate assumption, there could not have been any invocations after time t'-2D×T as shown in Figure 4.3, hence there was no invocation and now new versions added during this interval. This implies that versions 0 and 1 must have been present at time t-2D×T. However, if these were the only 2 versions present in the network at time t-2D×T, we know that they must have converged before time t-D×T. This is a contradiction. □

Having established the completeness and consistency of local detection, we now present an algorithm that performs these functions. We augment the original set of versions with a special version number, ϕ and the update rules with the condition $\forall i : i \in \{0, 1, 2\} : \phi > i$. ϕ thus indicates a special reset state. Also, we update A'_2 from the original protocol with a set of special actions for local detection and correction. Our proofs of correctness guarantee that the reset state will only be

entered if indeed there is a problem in the network. This modified action DCA'_2, shown in Figure 4.4 guarantees stabilization to the weaker invariant described in Section 4.3.

Const D : integer
Var $log[\,]$: integer $\times$ real
$hist[\,]$: integer
$curr\_time$: real
Actions

$\langle DCA'_2 \rangle$ :: $rcv\ m(v) \longrightarrow$ **if** $(v > vnum)$
$curr\_time := get\_Local\_Time();$
$hist[\,] := get\_Version\_History(log,$
$curr\_time - D \times T, curr\_time);$
if $is\_inconsistent(hist[\,])$
$vnum := \phi;$
$clear\_log();$
else
$vnum := v;$
$log := log \cup (v\ ,\ curr\_time);$ **fi**
elseif $(v < vnum)$ $bcast\ m(vnum)$ **fi**

Figure 4.4: Local detection and correction.

In this stabilizing action, each node logs the version numbers assumed by it and the local time at which each version was assumed. We require only a local clock value that is easily accessible in all hardware platforms and do not assume any global clock synchronization. During each version change, a node checks its logs for versions assumed in the last D×T time. If an inconsistent set is detected, the node executes a local correction action and resets its version to ϕ. The node also resets its logs and its configuration C. From the protocol action A_1, we know that this node will now periodically broadcast version ϕ and since ϕ dominates all other versions, every

node in the network will reset its configuration to version ϕ within time D×T. Since we know from Theorem 1 that an inconsistent set either converges or is detected by at least one node within D×T time, we can prove the following theorem for this stabilizing action.

Theorem 3 (*Local Stabilization*). Starting from an arbitrary state, every execution of the locally stabilizing reconfiguration protocol stabilizes to a state where all nodes have the same version within time 2D×T.

4.4.2 Correction using Human-In-The-Loop

The protocol actions described in the previous subsection guarantee local stabilization to the weak invariant that all nodes eventually have the same version number. An alternate, centralized approach to stabilization is one where versions present in the network are monitored by collecting a global snapshot using query protocols such as SNMS [72]. However, experimental studies [11] have shown that global state collection in large-scale networks may only be about 50% reliable. It is therefore quite likely that a single global snapshot collected by an operator may miss the inconsistency in the network. The operator would thus need to continuously query and monitor the network - an energy expensive solution. However, since our local detectors and correctors reset the versions of the entire network to ϕ in bounded time using the original protocol actions only, the operator can easily learn about the inconsistency by querying only its local neighborhood. Our approach thus does not require any additional messages to be sent and also eliminates the need for global querying, making it energy efficient.

Upon learning of the inconsistent state in the network, the Human-In-The-Loop or the operator can restore the network to the ideal invariant by re-downloading the latest configuration in the network. Before doing this, the operator may also choose to wake up nodes that may be in low power sleep states to avoid inconsistencies that may arise in the future.

Stabilizing *Reset*. In the state where all nodes have version ϕ, the operator cannot simply download a new configuration since all nodes will reject the update due to the dominance of ϕ over all versions. To enable the operator to restore the network to the ideal state, we modify the original reconfiguration interface to include a special *reset* operation. The purpose of the *reset* operation is to restore the network to a consistent state from which new invocations will succeed. To guarantee stabilization in the presence of multiple initiating operators, nodes use a policy of updating their version to the predetermined value 0 on receiving a *reset*. Also, a *reset* is a one-time operation as opposed to an epidemic one and it terminates when every node forwards the *reset* message once. Note that a one-time forwarding of the *reset* message is sufficient because of our assumption of a connected network with reliable links. A simple way of relaxing this assumption is to forward each *reset* message k times. When the *reset* operation terminates, all nodes have version 0 and the operator can now restore the network by downloading the correct (most recent) configuration with version 1.

During the propagation of a *reset* operation, a node which has received a *reset* message and updated its version to 0 may roll back to a different version if it hears an advertisement from a neighbor that has not yet received the *reset*. To guarantee stabilization, we require that after receiving the first *reset* message, a node ignores

all advertisements for a bounded time δ. This δ time bound is a function of the time taken for the *reset* to propagate in the network. We can thus prove the following theorem about the *reset* operation.

Theorem 4 *(Stabilizing Reset).* Starting from an arbitrary state, executing the *reset* operation results in a stable state where all nodes have version 0 in bounded time.

The complete stabilizing reconfiguration protocol is presented in Figure 4.5 where DCA_2' and RA_3 denote the local detection/correction and *reset* operations respectively.

4.5 Performance

In this section, we present simulation and experimental results showing the likelihood of non-stabilization of the canonical reconfiguration protocols, its impact on performance and also the effectiveness of our local detectors and correctors in stabilization.

4.5.1 Setup

We used TOSSIM [55] to simulate the two canonical protocols described in Section 4.2.2. By choosing suitable link reception probabilities, we created a regular grid topology for our simulations in which each node could communicate only with its four grid-neighbors. Our experimental setup consisted of 105 XSM [29] sensor nodes deployed in a 15x7 grid in the indoor Kansei [31] testbed with 3 ft grid spacing. We used a very low power level (2/255) to limit the transmission range and create a multi-hop network for our experiments.

During the initialization of each simulation or experiment run, each node chose its initial version number randomly from {0,1,2}. We chose the version advertisement

Protocol *StabilizingPeriodicBroadcast*
Const T : integer
D : integer
δ : real
Var $vnum$: integer
m : message
$log[\,]$: integer $\times$ real
$hist[\,]$: integer
$curr\_time$: real
$reset\_start\_time$: real
$sent\_reset$: boolean
Actions

$\langle A_1 \rangle :: \; Timer.fired \overset{[kT..(k+1)T]}{\longrightarrow} \; bcast\ m(vnum)$
[]
$\langle DCA'_2 \rangle :: rcv\ m(v) \longrightarrow$
$curr\_time := get\_Local\_Time();$
if $(curr\_time - reset\_start\_time < \delta) skip;$
elseif $(v > vnum)$
$hist[\,] := get\_Version\_History(log,$
$curr\_time - D \times T\ ,\ curr\_time);$
if $is\_inconsistent(hist[\,])$
$vnum := \phi;$
$clear\_log();$
$sent\_reset := FALSE;$
else
$vnum := v;$
$log := log \cup (v\ ,\ curr\_time);$ **fi**
elseif $(v < vnum)\ bcast\ m(vnum)$ **fi**
[]
$\langle RA_3 \rangle :: rcv\ m(reset) \longrightarrow$
$reset\_start\_time := get\_Local\_Time();$
$vnum := 0;$
if $(sent\_reset = FALSE)$
$sent\_reset := TRUE;$
$bcast\ m(reset);$ **fi**

Figure 4.5: Stabilizing reconfiguration protocol.

interval (T) in these protocols as 15 seconds. To factor out the choice of this specific value, the experimental measurements for convergence time in this section are presented in terms of number of *intervals*.

4.5.2 Results

We first show that the non-stabilizing version problem occurs in practice in the canonical protocols presented earlier. To say that a run is non-stabilizing, we would ideally have to let the simulation or experiment run forever. For the results presented in this section, we call a run as non-stabilizing if it did not stabilize within 50D intervals of execution time where D is the diameter of the network. We argue that this is a reasonable choice especially since we know from Theorem 1 that our local detectors and correctors will detect an inconsistency in no worse than D intervals. Hence, even if the run were to stabilize after 50D intervals, there would be a significant performance overhead.

Our results are tabulated in Table 4.1. The first row in the table measures the likelihood of a run being non-stabilizing for the canonical protocols. As the data shows, a significant number of runs (>46%) do not stabilize even for networks as small as 10×10 and 15×7 grids. Also, variant 2 of the protocol, where nodes aggressively update their neighbors has slightly better convergence properties than variant 1, although this difference is less pronounced for the larger networks.

The performance impact of non-stabilization can be seen in row 2 which shows the average number of messages transmitted by a node per broadcast interval. As expected, variant 1 transmits at the fixed rate of one message per interval. However,

Setup	5 x 5 sim		10 x 10 sim		15 x 7 expt	
	V1	V2	V1	V2	V1	V2
% of non-stabilizing runs	70	0	100	82	55	46
Avg. # msgs per node per interval	1	4.16	1	6.44	1	6.15
Avg. # intervals for convergence	2.66	3.66	N/A	20.33	4.15	4.75
Max. # intervals for local detection	2	1	4	2	2	2

Table 4.1: Performance evaluation of stabilizing reconfiguration.

the aggressive nature of variant 2 implies that for non-stabilizing runs, it consumes up to 6.5 times more resources, depleting the network at a much faster rate.

The third row in the table lists the average number of intervals required for convergence in runs that stabilize on their own (for non-stabilizing runs, this number is infinite), while the fourth row lists the worst case time for local detection and hence local stabilization. As seen from the data, local stabilization results in faster convergence than even those runs that stabilize on their own. Recall that reconfiguration also involves a data download phase operating concurrently with version updates. Hence, the faster the network converges, the lesser is the bandwidth and energy consumed while trying to download data.

Our simulation and experimental results thus prove that non-stabilization is indeed a problem that occurs in practice in existing reconfiguration protocols. Our results also validate the efficiency of the local detection and correction algorithm presented in this chapter.

4.6 Conclusions

In this chapter, we demonstrated, both theoretically and using simulations and experimental studies, that due to different types of faults such as operator errors, data corruption, etc., there is a non-trivial probability that existing protocols for reconfiguration of wireless sensor networks may not stabilize. We presented, both qualitatively and quantitatively, the adverse impact of this non-stabilization on the quality and lifetime of the network. Our findings were validated in real life during the large-scale *ExScal* experiments when we had to expend significant manual resources due to this problem.

We also presented a stabilizing reconfiguration protocol that solves this problem. Our solution uses a novel mix of autonomous and Human-In-The-Loop stabilization by first detecting and correcting version inconsistencies locally, allowing the human operator to then easily restore the network to the desired configuration. Our solution can be easily incorporated into existing reconfiguration protocols to make them stabilizing or can also be composed with them in the form of wrappers or interceptors to provide stabilizing reconfiguration services for our MASE architecture.

CHAPTER 5

RELIABLE MONITORING

5.1 Introduction

As noted in Chapter 1, the manager of a wireless sensor network must know the state of the network to determine the appropriate management actions that need to be taken to maintain the most optimal configuration and performance in the network. However, as described in Chapter 2, multi-hop routing of messages in wireless sensor networks can be unreliable. This may pose a problem for sensor network management wherein a manager, provided with unreliable network information, implements the wrong management response in the network.

Consider the scenario where the manager of a wireless sensor network testbed wishes to deploy a user application on a k×k grid of nodes in the testbed. The manager queries the testbed network to obtain the current health status of all nodes and selects a k×k sub-grid of nodes that responded to this query with a healthy status. However, if the collection of health information is unreliable, the responses from some healthy nodes may be lost. Thus, the testbed manager may be forced to deny the user's request even though it actually could have been satisfied.

We therefore present Chowkidar, a health monitoring service that provides accurate information about the health of a deployed network to the network manager. Chowkidar is designed to be masking fault-tolerant, which means that even though faults may occur during its operation, it shields these from the end-user and guarantees that the output is consistent. The Chowkidar service has both a centralized and a distributed implementation, although, as we shall see later, the distributed version is more efficient in terms of time and energy.

Although Chowkidar is reliable, it requires every network node to be involved in the collection process and to provide a health response. While this approach is suitable and may be essential for testbeds or sensor-actuator based control networks where reliability of collected information is of utmost importance, it may be too expensive to implement in large scale field deployment scenarios where energy efficiency might be equally important. Moreover, there exist management scenarios where not all nodes in the network may be required to respond.

We identify termination detection, which is an important element of wireless sensor network management, as once such problem, and present Reporter, a reliable, yet message efficient protocol. Although Reporter has been designed in the context of termination detection, the key ideas in Reporter can be adapted to provide efficient solutions for other management problems, including health monitoring.

We describe the Chowkidar and Reporter protocols in further detail in the rest of the chapter.

5.2 Chowkidar

In this section, we present the Chowkidar service for reliable health monitoring in wireless sensor networks. Although the main use case for our Chowkidar design and implementation was the monitoring of Kansei [3,31], a large scale, heterogenous, wireless sensor network testbed, which we have developed at Ohio State, the algorithms for health monitoring presented in this section are applicable to any sensor network that satisfies our network model. For simplicity of presentation, we restrict our attention here to health monitoring in the context of wireless sensor network testbeds and comment on applicability to general network deployments wherever appropriate.

(a) Physical layout of Kansei. (b) A multi-device Kansei node.

Figure 5.1: The Kansei testbed at Ohio State.

5.2.1 Motivation

Large scale wireless sensor networks, whether deployed outdoors or in a testbed environment, have hundreds or thousands of devices that use different types of hardware and software. In Chapter 2, we described the multi-tier *ExScal* network which

used a number of different types of hardware and software platforms. Another example of such a large, multi-platform network is our Kansei testbed shown in Figure 5.1. Kansei currently houses several hundred devices of different types, whose characteristics are listed in Table 5.1.

	XSM	**TelosB**	**Stargate**
Processor	4MHz	8MHz	400MHz
RAM	4KB	10KB	32MB
OS	TinyOS	TinyOS	Linux
Interfaces	CC1000, Serial	CC2420, USB	Ethernet, 802.11b, Serial, USB
Bandwidth	38.4kbps	250kbps	11Mbps

Table 5.1: Different platforms in Kansei.

Given the relatively unreliable nature of wireless sensor devices, faults may occur before or during an experiment that affect the quality of data produced in a run. Even if devices are reliable, software bugs or incorrect device configurations could lead to faults. In the fault model developed in Chapter 2, we classified and measured faults from a network performance or yield standpoint. From a health monitoring and diagnosis standpoint, we further categorize these faults, based on our experiences with designing, maintaining and experimenting using Kansei over the last two years, as follows.

- *Device fail-stop faults.* A fail-stop results in the complete failure of a device. Fail-stops may occur due to hardware failure or software crashes that render the device completely unresponsive. The impact of a fail-stop depends on the type of the affected device in heterogenous testbeds like Kansei. For example,

the fail-stop of an XSM mote simply results in that mote being unavailable for experimentation. However, the fail-stop of a Stargate is worse since a Stargate is used by Kansei to program and log data from XSM and TelosB motes attached to it via their serial and USB interfaces. Similarly, the fail-stop of an Ethernet hub results in loss of wired connectivity to all of its attached Stargates and in turn their attached motes. Since the wired network is used by Kansei for instrumentation and data retrieval during an experiment, these fail-stops render the affected testbed regions unavailable.

- *Network interface faults.* A network interface fault at a device results in the device being unable to communicate over that interface. Network interface faults may occur due to driver failure or loose hardware connections such as an unplugged Ethernet cable, an unseated 802.11b wireless card or a detached Stargate-mote connector. Since most testbeds have separate experimentation and instrumentation interfaces, the failure of a single network interface does not render a device unreachable. Failure of all its network interfaces, however, results in a device being partitioned from the testbed.

- *Software faults.* A software fault results in a correct device or network interface being driven into a corrupted or bad state. These faults may occur due to buggy code or misconfigured software. For example, a user program may change the radio power level to a very low value or change the transmission frequency, so that neighbors cannot receive radio messages sent by this device. Another type of configuration fault occurs if the Stargate or its attached mote changes the configuration of some pin in their serial connector to an incompatible state.

If these faults are not detected and accounted for while analyzing experimental results, users could end up deriving incorrect conclusions and making incorrect choices that adversely affect the performance of subsequent runs. It is therefore imperative for health status information to be available before, during and after an experiment so that the results produced by a testbed experiment can be accurately analyzed.

In addition to node health status, diagnostic services provided by monitoring can help administrators distinguish between a testbed fault and an error caused by the experiment itself. This is important since some faults may have the same visible effect as others. For example, a 802.11b wireless driver process crash has the same effect as the card becoming unseated. Diagnosis of such faults can help administrators determine the appropriate correction actions needed to restore the testbed to a fully functional state. Knowing, for example, via alternate network interfaces that the wireless driver has crashed, an administrator could simply reload it, whereas an unplugged card would require the administrator to go to the testbed and physically re-insert it.

We therefore developed Chowkidar, Hindi for "watchman", as a service that provides health data about testbed resources, periodically as well as on demand. Users (or a testbed scheduling service) can use the information provided by Chowkidar to ensure that experiments run only on working devices by checking health before and after an experiment. They can also better assess experimental results knowing whether or not some failures occurred during the experiment. Administrators can use Chowkidar to learn about testbed faults and any available diagnostic information to help determine the most effective response.

5.2.2 Monitoring Requirements

The general network health monitoring problem may be stated as follows: to identify which network objects are functioning correctly, and, for those that are not, to identify them and so far as possible, indicate why they are not functioning. Objects typically include network resources such as links, nodes, and so forth, but may also include things such as application components and processes.

We identify the key requirements for a network health monitoring, some of which are derived from the standard problem definition stated above and others arising out of the wireless sensor network model, for which Chowkidar is designed.

- *Reliability.* For monitoring results to be useful, they should be reliable. Reliability has two important dimensions; first, the reported results should be consistent with the actual state of the testbed and second, they should be complete so that if a node is working and is reachable, this status should be known to the monitor. Due to resource constraints such as limited bandwidth and energy, some wireless sensor networks use sampling-based approaches where the overall state of the network is estimated based on data received from a subset of nodes. The goal of a testbed health monitoring service however, is to provide ground truth information against which users can compare obtained experimental results. We therefore require that the status of each resource in the network be monitored and that these results be reliably exfiltrated.

  Reliability also implies that monitoring must be independent of an experiment's semantics or communication structure, otherwise design or implementation errors in an experiment could yield incorrect monitoring results.

- *Efficiency.* Regardless of whether a monitor collects health status from some or all nodes, monitoring should be both time and energy efficient. By this we mean that monitoring must complete quickly, using few messages.

- *Handling heterogeneity.* As exemplified in Table 5.1, wireless sensor network testbeds may have different hardware and software platforms with varying capabilities and limitations. A monitor must therefore handle heterogeneous devices and networks. This might include PCs, embedded Linux systems such as Stargates and motes such as Mica2s, XSMs or TelosBs, as well as a variety of networks, including mote radio, WiFi, Ethernet, and others. There can be several instances of each network (such as multiple Ethernets). A monitor must be able to check the status of each of these.

- *Diagnosing failure types.* Administrators need to be able to distinguish between complete device failure and interface failure. In the former case, devices usually have to be physically repaired or replaced; in the second, a remote correction might be possible. Hence the monitor must distinguish between these failures.

- *Adaptability.* Testbeds keep evolving due to factors such as device failures and replacements, new technology and changing user requirements. Testbed configurations may also change as a result of rewiring and physical relocation. A monitor must therefore not be dependent on any particular testbed architecture and must be able to easily accommodate different types of networks and devices and adapt to changes in network configuration with minimal effort.

- *Usability.* Monitoring results must be available centrally to users and administrators. Administrators need status information periodically to detect permanent failures or identify failure patterns while users want to know the network status before and after an experiment. Hence monitoring must be performed both automatically or on demand.

- *Co-existing with experiments.* Users may be interested in monitoring node health during an experiment. Hence monitoring should be easy to compose with user applications. At the same time, a monitor must have at most bounded interference with experiments during concurrent operation. This is of particular concern for radio networks, but can affect others such as Ethernet as well.

Existing monitoring approaches such as Motelab [75], SNMP [39], SNMS [72], fault tracing [69] and Sympathy [65] satisfy some of these requirements but none of them satisfy all of them. In particular, none distinguish between node and interface faults, and none are heterogeneous. This is discussed in more detail in Section 5.2.9. It was because of these limitations that we developed Chowkidar.

5.2.3 Chowkidar Features

As stated earlier, our Chowkidar service can use one of two different protocols for collecting monitoring information from devices: the first uses centralized control while the second one is distributed. Although these two Chowkidar protocols differ in a number of ways, they both satisfy the following properties.

- Monitoring information is collected and evaluated centrally at a base station. The base station knows the topology of the network and also knows which nodes

are in use by experiments. This information is used to perform monitoring, to assess the results, and to provide current and historical status.

- Monitors are correct in the presence of node and link fail-stops and restarts that occur during the run or between runs. A given collection either gives a consistent result (corresponding to testbed state that existed during the run) or reports failure. In the latter case, the monitor can be run again.

- All devices are explored for reachability along all available networks, with a typical path consisting of multiple networks.

- Paths to resources are least-cost, where the cost of a network link is assigned by the administrator, taking into account factors such as bandwidth, reliability, and interference.

- Monitoring runs are done periodically and, at the request of the user or the testbed scheduler, can be done on demand.

- Between monitoring runs, no resources are used except for components that listen for monitoring messages.

Properties 2 and 3 imply that information provided by Chowkidar is reliable. This is important since faults can happen at any time, including during a monitoring run. In such cases, we want monitoring to either report the failure so that it can be retried, or we want the collection to be consistent in the sense that the state reported actually occurred in the system during the run.

For example, suppose a reachable node's status has been confirmed and, during the balance of the monitoring run, it is not accessed again. If this node fails before the

end of the monitoring run, the report (that the node is reachable) will be inconsistent with the final state of the system (in which it is not); but it will be consistent with the state of the system before the node failed.

On the other hand, if a node was found to be reachable at one point during the run but found to be unreachable later, then the run would terminate with an error value. Chowkidar can be easily extended to provide masking fault-tolerance by retrying automatically whenever errors occur until a consistent result is obtained. If we assume that faults stop, or at least stop long enough, then eventually Chowkidar will succeed.

Property 4 implies that Chowkidar is efficient since it performs monitoring over least-cost paths, thus requiring fewest resources.

Our Kansei implementation uses a scheduling service called Director that maintains an SQL database of resource status. Chowkidar accesses this database to assess resource status for monitoring and can be configured to monitor all resources, only free resources or resources specifically used in user experiments provided users include Chowkidar components in their applications. Similarly, Director is configured to automatically load Chowkidar components on resources when they become free.

However, Chowkidar is configuration–driven, hence it is testbed or network–independent. Thus, for a given wireless sensor network, if the communication components for the different networks are provided along with a configuration of the network as a whole, Chowkidar can be readily adapted for use on another testbed or network.

5.2.4 System Model

We assume a multi-hop, heterogenous sensor network with different types of devices, platforms and communication links. We do however assume that nodes connected to multiple networks share the same communication interfaces and can transparently forward packets from one network to another. Network links can be unidirectional or bidirectional. While we do not require all links to be bidirectional, both Chowkidar protocols use the same forward and reverse paths for collecting health information, so we do assume that some bidirectional path exists from the root, or the Chowkidar server, to each node. Thus, our protocols may try to use some unidirectional links and fail, but eventually they will discover a bidirectional path if it exists. Wireless links are subject to interference in the presence of multiple concurrent senders. Interference affects link reliability and may result in message losses due to collisions. Message losses during exfiltration of health data from the testbed may affect monitoring reliability.

We assume that if a node fail-stops, it does not communicate on any of its interfaces. When an interface fails, the node cannot communicate on it. When a node restarts, it does so cleanly, reinitializing variables as required by the monitoring component. We assume an interface does not have state, so a restart is always clean.

5.2.5 Centralized Chowkidar Monitoring Protocol

The centralized Chowkidar protocol for collecting health information from all nodes in the network is a simple, sequential protocol in which the base station or the Chowkidar server, using configuration information and knowledge of network topology, attempts to construct a path to each node whose status is unknown, avoiding

links that are known to be down. The process, described below, terminates either when all nodes are confirmed as up or when there are no more paths to check. Initially, the status of each node and link is unknown.

1. Using testbed configuration information, construct a least cost path (LCP) tree that covers the free nodes of unknown status, using links that are not down, with the constraint that all leaves are nodes whose status is unknown. The tree can be empty, which means that all reachable nodes have been found, and we are done.

2. For each path, construct a probe message that contains the path. Send the message to the first node in this path along the designated interface. As each node receives the message, it forwards it to the next node if any, and waits for a reply. If a node is the leaf in the path, it sends a reply to its parent in the specified path. If a node that is waiting for a reply receives it, it forwards it on; if it does not arrive after a timeout (which can be calculated from the path), it replies on its own.

3. When a reply arrives at the base station, the knowledge of the path and the node that replied lets the base station identify the nodes and links that were reachable along that path; these are marked as "up". If some node other than the leaf replied then the downward link is marked as down and is effectively removed from the topology for the duration of the run.

   A network link involves a sender and a receiver; hence lack of responsiveness on the link implies that either the sender's interface is down or the receiver's interface is down or that the receiver node itself is down (or any combination of

the above). In any case, we consider that link as failed and do not use it again in subsequent paths: it is removed from the configuration for the duration of the run.

4. Resume from #1.

If, during a monitoring run, a particular node or link was found to be up as the result of some probe but later when that node or link was reused, it was found to be down, then a node or link fault has occurred that affects consistency. In that case, we abort the run and restart. If the run did not abort with an error, then the collection is consistent. Hence, in the presence of restart or of fail-stop that does not affect consistency, centralized Chowkidar will terminate with a consistent collection. If a fail-stop happens that might affect consistency, it will terminate with an error.

Network configuration information can be provided in two ways. In the "high atomicity" view, the configuration consists of nodes with network links between them. In this case, a collection will give all reachable nodes but may not check all interfaces. Alternatively, the configuration can be given with "low atomicity" in which interfaces are explicitly included. This forces Chowkidar to explore paths that include each interface, thus checking each one.

As a protocol that checks reachability by exploring all paths sequentially, centralized Chowkidar does not scale well. Calculating an LCP tree takes $O(|N| \cdot |E|)$ time where N is the number of nodes and E is the number of edges per node. Since a network can be fully connected, the time complexity is $O(N^2)$. It results in $O(N)$ probes and, since the set of probes cover the nodes, there are $O(N)$ messages total. The number of times the LCP tree calculation process and probing has to be repeated depends on the pattern of failures. It can be as high as $O(N^2)$ in a network where

half the nodes are down and each is directly linked to an up node; in this case, paths will be created from each up node to each down one. Hence time complexity can be as high as $O(N^4)$ and message complexity as high as $O(N^3)$.

5.2.6 Distributed Chowkidar Monitoring Protocol

As seen in the previous subsection, in the worst case pattern of failures, the time and message complexity of the centralized protocol can be quite high. From our experience with Kansei, a collection for centralized Chowkidar can take up to 10 minutes for the low atomicity case for 210 nodes. Kansei is in the process of growing, with up to 630 TelosB motes to be added in the near future, thus clearly the centralized approach will not scale.

We have therefore developed a stabilizing distributed protocol [52] for reliable and scalable information collection in Chowkidar. This protocol solves an instance of the well-known problem of message-passing rooted spanning tree construction and its use in PIF (propagation of information with feedback) for the wireless sensor network model. Our protocol differs from previous work in message-passing PIFs in two ways that are critical for this model. First, it is message efficient in that it uses only a few messages per node, which is important given the resource constraints of wireless sensor networks. Second, it tolerates ongoing node as well as link faults, and their restart, which do indeed occur in sensor networks. Existing PIF-style approaches either assume that faults have stopped to assure termination, or else they run continuously, often using using substantial resources when a fault occurs.

Our distributed Chowkidar protocol has two components - the first is a simple, efficient protocol that produces in one shot, a spanning tree over the set of reachable

nodes in the network. If node or interface failures do not happen during execution, the protocol yields a tree with bidirectional edges: each child knows its parent and each parent knows its children. The second component is a PIF protocol that runs subsequently using the constructed tree. During a PIF wave, each parent waits on its children to report before it reports to its parent; if the parent fails to hear from a child in a timely fashion, it initiates a failure message to the root, in which case a new tree is constructed.

We now describe the stabilizing, distributed Chowkidar tree construction and PIF wave protocols in detail.

Notation and semantics. We use guarded commands with interleaving semantics for specifying the protocol. Parameters `j` and `k` range over nodes; `i` ranges over links. Among commands with true guards, one is chosen and is atomically executed. Communication actions are synchronous: after a node sends a message to `X`, `X` executes the corresponding receive before any other action of `X` and before any other node sends to `X`. Environment actions can interleave send-receive pairs, might or might not be executed when enabled, and execute at most finitely often.

We use Gouda's AP notation for timeouts, which are represented by global state predicates. In the case of the child-parent acknowledgement sequence during tree construction (T6 of Figure 5.2), for instance, the timeout can be implemented by setting a time to wait for an acknowledgement after notification has been sent. Termination of each protocol can be given via a timeout predicate that negates the conjunction of the guards.

Timing assumptions. We use delayed delivery of wave messages to construct a least-cost tree, implemented by an underlying broadcast service that delays actual

broadcast until a specified time. In our implementation we have the following delays: Ethernet, 0s; serial link between Stargates and XSMs, 0s; XSM radio, 5s, where higher delay corresponds to higher link cost. From the base station to a given XSM X_0 there can be many potential paths, including B-E-SG_0-S-X_0 and B-E-SG_1-S-X_1-R-X_0, where B is base, SG is a Stargate, X is an XSM, E is Ethernet, S is serial link and R is radio. The first path costs 0s and the second 5s, so the first is preferred. Delayed delivery means that the wave via the serial link arrives at X_0 before the wave via the radio, so SG_0 will be selected as X_0's parent.

We assume that internal processing in nodes takes zero time and that links have associated timing values, including the delay enforced by the broadcast primitive. This lets us associate an upper bound on the time to wait before concluding that a timeout is satisfied.

Constants, variables and messages. We describe the constants, node variables, environment variables and message types. `lktype` is a constant set of link types for a given node corresponding to interface types. `id` is the identifier for a node; we assume that nodes have unique identities.

`sn` is an unbounded integer session number, initially 0. `parent` is the (node id, interface) pair for a parent, initially (-1,-1) for unassigned. `child` is a set of of (node id, interface) pairs of children, initially $\emptyset$. `busy` ensures only one handshake at a time, initially false. `tk, ti, tsn`, initially -1, -1, 0, resp., store parent candidate information during handshake. `newtree` is true if a new tree should be formed, initially true. `pphase` is a Boolean that is true during the propagation phase, initially false. `pchldcnt` is a bounded integer that counts the number of children heard from during feedback. `up` is an array of Booleans indexed by node ids that indicates whether

the node is up or not. `link` is an array of Booleans indexed by node id pairs and link type that indicates whether the link is up or not; this is symmetric wrt nodes.

Each message begins with a message type. `'wave'` is for tree wave messages. `'notify'` is for a child to notify a potential parent in tree construction. `'ack'` is for the parent to acknowledge the child in tree construction. `'token'` is the propagation message for PIF. `'fb'` is the feedback message for PIF. `'err'` is the error message for the PIF when the structure is found to not be a tree. `'restart'` is issued by a node that restarts.

Distributed Chowkidar Tree Construction Protocol

The distributed Chowkidar tree construction protocol is given in Figure 5.2. After action T1 initiates a new wave in response to the need for a new tree, the normal sequence is as follows: a node gets a wave message from a neighbor in T2 over some interface and enqueues it; the queue stores other alternatives in case faults prevent the sender of the first wave message from being adopted as a parent. T3 removes a wave message and, if it has a higher session number, tries to adopt the sender as parent by sending it a notification. The parent receives the notification in T4, adds the node as a new child, and returns an acknowledgement to the child. The child receives the acknowledgement in T5 and propagates the wave. Note that the broadcast service ensures that wave messages are delivered in order of link cost.

Node and link faults may occur concurrently with the protocol. If a node or link fails before the wave reaches it then this is the same as failing before the protocol begins. If a node or interface used in the tree fails after the wave has passed, this is the same as failing after the protocol terminates and will be detected by a subsequent PIF since that node will fail to respond to its parent.

```
T1   (id=Base∧newtree) → // begin new tree
     sn:=sn+1; tsn:=sn; parent:=(-1,-1); newtree:=false;
     ∀pi : (pi ∈ lktype) : broadcast ('wave',sn) on pi
       atTime tdeliv[pi];
     child:=∅;
T2   rcv ('wave',msn) from k on i → // enqueue waves
     if (msn>maxqueue()) q:=(k,i,msn);
     elseif (msn=maxqueue()) enqueue(k,i,msn);
     fi
T3   hdqueue=(xk,xi,msn) ∧ msn>tsn → // begin handshake
     (tk,ti,tsn):=dequeue();
     send ('notify',tsn) to tk on ti;
T4   rcv ('notify',msn) from k on i → // acknowledge
     if(msn=sn)
       send ('ack',sn) to k on i;
       child:=child ∪ (k,i);
     fi
T5   rcv ('ack',msn) from k on i → // join parent & echo wave
     if (msn=tsn)
       sn:=tsn; parent:=(tk,ti);
       ∀pi : (pi ∈ lktype) : broadcast ('wave',sn) on pi
         atTime tdeliv[pi];
       child:=∅; fi
T6   timeout (tsn≠sn ∧ (¬upn.tk ∨ ¬upl.id.tk.ti)) →
     tsn:=sn;
```

Figure 5.2: Chowkidar tree construction protocol.

Now consider failures concurrent with the wave boundary. Suppose node Y broadcasts via action T1 or T5, but before delivery, neighbor X or the link fails. This is the same as failure before the protocol begins and X will not be included in the tree. Suppose X sends notification to Y via T3 but Y or the link fails before receipt. This is the same as failing before the protocol starts. Suppose Y receives a notification from X in T4 but the link fails before X receives the acknowledgement. Then Y adopts X as a child but X does not learn this; T6 will cause a timeout and X will solicit a new parent. Since Y has falsely recorded X as a child, in a subsequent PIF will detect that X does not respond to Y. Similarly, if Y receives a notification from X but X fails before receiving the acknowledgement, X will not be responsive in the subsequent PIF.

Node restarts are clean and would normally be benign. However, if X starts a handshake with Y1, then fails and restarts, it can start a new handshake with Y2, perhaps in a later session, and either acknowledgement could arrive first. To handle this, T5 will accept an acknowledgement only from the node/link it notified last.

The protocol tolerates simultaneous sessions. In the absence of faults, the protocol will form a spanning tree over the highest session number among non-tree nodes that were up when the wave was propagated by a neighbor. In the presence of faults that affect the tree structure, either the fault happens before the wave or else the structure contains nodes with non-responsive children. As we have seen, a node listed as a child can be non-responsive if it or its tree interface is down, or if it has identified a different parent. In this case, a subsequent PIF will detect the fact and initiate a new tree construction.

In the protocol, wave messages are delivered in order according to cost of the links. Hence the spanning tree formed is least-cost unless while handshaking with node Y,

the link fails before X's notification is sent, causing X to attempt another parent which, if available, might result in a path that is not least-cost. The only active nodes are those that are at the wave boundary. Timeouts guarantee that wave progresses until it is extinguished, so the protocol terminates. Based on our assumptions about delivery and link times, we can calculate an upper bound on the time required to terminate.

Distributed Chowkidar PIF Protocol

The Chowkidar PIF protocol is shown in Figure 5.3. P1 blocks if a tree fault has been detected and should be initiated only when the tree protocol has terminated. A token is initiated on the tree in the form of a message with a new sequence number. In P2, interior nodes forward the token to their children while leaf nodes begin the feedback response. In P3, a node that receives feedback from a child increments a counter; when all children have responded, it sends its feedback response to its parent. In P4, a node waiting on a child's response times out if it is unresponsive. This creates a message that is propagated up the tree by P5.

Implementation of timeout P4 can be based on the longest possible path in the network. An initial PIF can refine the timeouts based on a node's distance from its leaves to get tighter values.

A PIF execution terminates within some bounded time and, if the structure from the root is a spanning tree, then it completes with a report of success and otherwise with a report of failure. If a restart message is received by a node before it has completed feedback then the tree is not spanning; by our synchronous communication assumptions, the restart message will reach the root before the feedback message, triggering the creation of a new tree.

```
P1   (id=Base) ∧ (start new PIF) ∧ ¬newtree → //new PIF
     sn:=sn+1;
     ∀pk,pi :  (pk,pi) ∈ child :  send ('token',sn)
       to pk on pi;
     pphase:=true; pchldcnt:=0;
P2   rcv ('token',psn) from k on i → //propagate or respond
     if (psn > sn)
       sn:=psn;
       if (child≠ ∅)
         ∀pk,pi :  (pk,pi) ∈ child :  send ('token',sn)
           to pk on pi;
         pphase:=true; pchldcnt:=0;
       else
         send ('fb',sn) to parent.node on parent.link;
         pphase:=false; fi fi
P3   rcv (fb,psn) from k on i → //forward responses
     if (psn=sn)
       pchldcnt:=pchldcnt+1;
       if (pchldcnt=|child|)
         if (id≠Base) send ('fb',sn) to parent.node
             on parent.link; fi
         pphase:=false; fi fi
P4   timeout pphase ∧ (k,i)∈child ∧ (¬upn.k ∨ ¬link.id.k.i ∨
         parent.k≠(j,i)) → //timeout unresponsive child
     if (id=Base) newtree:=true;
     else send ('err') to parent.node on parent.link; fi
     pphase:=false;
P5   rcv ('err') from k on i → //send error to Base
     if (id=Base) newtree:=true;
     else send ('err') to parent.node on parent.link; fi
```

Figure 5.3: Chowkidar PIF protocol.

```
E1   id≠Base ∧ up → up:=false; //fail a node
E2   link.id.k.i → link.id.k.i,link.k.id.i:=false,false;
         //fail a link
E3   ¬up → //restart a failed node
     up:=true;
     (reset all variables)
     ∀pi :  (pi∈lktype) :  broadcast (‘restart’) on pi
         delivery now;
R1   rcv (‘restart’) from k on i → //send restart msg up tree
     if (id=Base) newtree:=true;
     else send (‘restart’) to parent.node on parent.link; fi
```

Figure 5.4: Chowkidar environment and restart actions.

Environment actions. The environment-related actions are shown in Figure 5.4. E1 causes a node to fail. E2 causes a link to fail. E3 causes a node to restart; a restart message is broadcast on all available interfaces.

If R1 receives a restart message, it resets the tree construction busy flag in case the node had been involved in a handshake. To ensure correctness, the restart message is forwarded towards the root so that a new tree can be formed. Consider the following cases. Suppose the restart message is received by a node with a tree path to the root that stays up sufficiently long. Then the message will arrive at the root. Suppose the receiving node's tree path has a failed node or interface closer to the root. Then a subsequent PIF will detect the problem and when the ensuing tree is formed, the restarted node will be included. Suppose the node is partitioned from tree nodes but there is a newly-restarted neighbor that is not. Then its reset message will trigger a tree rebuild that includes both. Otherwise the node is not reachable by any path of up nodes/links and would not be included the spanning tree even if it were rebuilt.

In joint related work with Leal et al. [53], we present a formal, invariant based proof of the stabilization of the Chowkidar tree construction and PIF protocols in the presence of ongoing fault actions described above.

5.2.7 Chowkidar Performance Evaluation

We implemented both the centralized and distributed Chowkidar protocols for the Kansei testbed, though both implementations can easily be adapted to other testbeds and network deployments with some modifications. Our implementations span the different hardware and software platforms in Kansei listed in Table 5.1. In this subsection, we first evaluate the performance of the distributed Chowkidar protocol and then compare it with that of the centralized one based on data collected from several experiments in Kansei.

Distributed Chowkidar performance. Our analysis and proofs for the distributed protocol assume that links are reliable and bidirectional. However, in reality, broadcast links such as wireless radio and even Ethernet suffer from transient message losses. Indeed, a naive implementation of Chowkidar where child-parent handshakes were initiated immediately upon receiving a wave message led to message implosion on these shared channels and loss of messages in the network due to contention. This network unreliability due to concurrent message transmission by all nodes was in fact so high that it affected not only the performance but also the correctness of our protocol since it resulted in an incomplete tree being constructed in several runs.

To avoid message implosion on shared channels, our implementation introduced a simple application-level backoff mechanism. Coarse-grained tuning of backoffs enabled us to obtain correct performance for our protocol implementation in that a tree spanning all correct nodes was constructed, albeit with increased execution time.

Percentage of failed Stargates	0%	8%	20%	40%
Average time for tree construction	1.2s	8.7s	9.9s	10.5s
Percentage of failed runs	0%	0%	10%	30%

Table 5.2: Effect of faults on performance for a 25 node network using a 2.5s backoff.

Table 5.2 shows the results of the first series of experiments performed on a 25 node network using a 2.5 second backoff on wireless links for congestion avoidance. We first ran our algorithm without introducing any failures. As expected, our algorithm always constructed a least-cost spanning tree using only Ethernet and serial links in very short time. We then injected failures by randomly stopping multiple Stargate nodes, which forced more and more wireless links to be used in tree construction. As the data shows, as the number of injected faults increases, not only does the average tree construction time increase, but in the worst case shown here, where 40% of the Stargates were failed, even correctness was affected in some runs due to excessive message loss. Fortunately, the degradation in performance is gradual and sublinear, so we can select a suitable backoff period based on the expected worst case failures in the network.

We validate this assertion in Table 5.3 in which we measure the minimum backoff period at which 100% of the runs are successful even in the worst failure case

Number of network nodes	25	25	50	50
Maximum backoff for radio	2.5s	5s	5s	10s
Percentage of failed runs	30%	0%	25%	0%

Table 5.3: Linear scaling of backoff with network size for 40% Stargate failures.

considered in Table 5.2, i.e. 40% failed Stargates for networks with different sizes. As seen from the data, using a 5 second backoff guarantees correct execution when up to 10 out of 25 (40%) nodes fail; however the same backoff does not guarantee correct execution if the network size is doubled to 50 nodes with the same failure rate. Nevertheless, as expected, with a linear scaling of the backoff period to 10 seconds, we observe correct protocol execution.

We thus conclude from our experiments that in wireless sensor networks, unreliability of message transmission affects both protocol correctness and performance and hence should be given careful consideration. However, as demonstrated by our experiments, using a simple backoff mechanism is sufficient to achieve correctness at the expense of increased completion time. Our assumption of reliable, bidirectional links can therefore be reasonably realized even in real network deployments.

Performance comparison. To evaluate the performance benefits of using the distributed protocol over the centralized one, we ran a number of experiments using both implementations in our Kansei testbed. We ran both protocols on the same sets of nodes. In the initial experiments, we tested correctness in the absence of failures by simply executing the protocols on nodes that were known to be working. We then injected failures by killing Chowkidar processes on randomly selected nodes (the same nodes for both cases). Table 5.4 shows the experimentally measured performance for

a set of 25 nodes. This data does not take into account the time taken to compute the paths in the centralized case as this is quite small on a powerful server. Recall that the distributed protocol first constructs a spanning tree over which subsequent PIFs can be collected, hence the total time for the distributed protocol is the sum of the times taken by each of these phases.

% of failed nodes	$\mathbf{T_{cent}}$	$\mathbf{T_{dist}}$	$\mathbf{T_{tree}}$	$\mathbf{T_{PIF}}$
0%	9s	7s	2s	5s
8%	54s	14s	8.5s	5.5s
20%	86s	16s	10.5s	5.5s
40%	153s	17s	11.5s	5.5s

Table 5.4: Performance comparison on a 25 node network.

As seen from the data, in the absence of faults, the performance of the centralized and the distributed protocols is quite comparable. However, the performance of the centralized protocol degrades substantially as the number of failures increases, even for a 25 node network. This is because the centralized protocol not only operates sequentially but also tries to explore all possible working paths in case of a failed node before giving up. By contrast, the distributed protocol finds existing paths concurrently instead of pruning failed ones sequentially, so its performance is only marginally affected. It should be understood though that the centralized protocol was inherently reliable due to its sequentially design that avoids interference losses whereas the distributed implementation had to be carefully tuned to select appropriate randomized backoff parameters to minimize network interference created by

concurrent execution. The centralized protocol could also be parallelized and similarly tuned for reliability, but it is clear that it will not outperform the distributed one.

We also experimentally measured the scalability of both protocols by varying the network size, the results of which are shown in Table 5.5.

Total # of nodes	**% of node failures**	$\mathbf{T_{cent}}$	$\mathbf{T_{dist}}$	$\mathbf{T_{tree}}$	$\mathbf{T_{PIF}}$
25	0%	9s	7s	2s	5s
50	0%	23s	9s	3s	5s
25	40%	153s	17s	11.5s	5.5s
50	40%	305s	23s	17s	6s

Table 5.5: Scalability of centralized vs. distributed protocols.

The first two rows in the table indicate the completion times for both protocols in the absence of any injected faults while the last two rows indicate the completion time when (the same) 40% nodes, selected randomly in the network, are failed. The data clearly demonstrates that as the network size increases, the performance of the centralized protocol degrades much faster than the distributed one.

Another important point to note in the distributed case is that the PIF completion time increases only slightly as the failure rate and network size are increased. This is because the PIF completion time is a function of the depth of the constructed spanning tree. Also, since the same tree is used when there are no new failures, the PIF cost is amortized over multiple successive runs, hence if failures occur rarely, the average completion time for the distributed protocol is even smaller.

Our experiments thus show that when carefully tuned for reliability, the distributed protocol outperforms the centralized one and scales much better as both failure rate and network size are increased.

5.2.8 Use Cases for Chowkidar

In addition to implementing Chowkidar for Kansei, we have also integrated it with the testbed to provide near real-time health information about testbed nodes to users and administrators. In this subsection, we briefly discuss applications and health monitoring utilities that benefit from the reliable information collection features provided by Chowkidar.

Node selection for experiments. In the past, Kansei users would schedule experiments on a set of nodes only to realize later that some of them were not working. This usually led to users having to retry several times before they finally ended up with a set of working nodes. However, users can now check the latest Chowkidar output or invoke Chowkidar on-demand prior to scheduling experiments so their experiments always run on working nodes. By running Chowkidar on-demand after an experiment, users are also able to verify that no new failures occurred during their experiment, increasing confidence in the obtained data.

Diagnosing dependencies. Basic monitoring gives reachability information about nodes and interfaces. If a node is reachable then it is up; but if not, we cannot automatically conclude its status, since it might be still be functional, in the sense that it would be reachable if placed in a suitable environment. Although unreachable, a node might be functional if the power is off, if all of its interfaces are not functional, or if it can only be reached through other nodes that are unreachable.

We have to consider the case of devices where checking status depends on some other device. These include "dumb" nodes such as Ethernet hubs that cannot be addressed directly as well as device interfaces and power supplies. To know whether a dumb node such as an Ethernet hub is working or not we have to try to reach an attached node. Hence if some attached node is reachable via the dumb node then the dumb node is reachable and therefore up. However, if no attached node is reachable via the dumb node then either the dumb node is not functioning, all of the attached devices are not functioning, all or the interfaces of the attached devices are not functioning, or some combination. Of course, there can be additional dependencies, since the interface of an attached node won't function if the node itself is not functioning, which in turn might be due to a power supply failure.

For power supplies, the supply is up if some associated node is up. If all associated nodes are not reachable then either the power supply is not functioning or all the nodes are unreachable for some other reason.

For interfaces, an interface on a node is reachable if it can be used to reach some other node; in this case, the interfaces are up as are both nodes and the link between the nodes. Although unreachable, an interface might be functional if its node is unreachable or if no other nodes are reachable via the interface.

The health information collected by Chowkidar is thus useful to network managers for diagnosing failure dependencies and determining the appropriate actions required to restore correct network state.

Monitoring user-defined predicates. After Chowkidar [74] went online, we received feedback from several Kansei users and administrators about additional information they would like to be monitored. For example, testbed administrators were

interested in monitoring whether the various testbed processes were running correctly on Stargates while users wanted to know whether the SerialForwarder program used to send/receive TinyOS packets to/from motes was running throughout an experiment. As a result, we identified several new predicates besides node and network health that can now be monitored using Chowkidar.

5.2.9 Related Work

Efficient PIF with ongoing faults. The notion of a self-stabilizing PIF has been well-studied in distributed computing as it is an enabler for many other tasks such as distributed reset, global snapshot, termination detection, and others; see [23] for an overview of non-stabilizing, self-stabilizing and snap-stabilizing PIF protocols; in fact, our protocol is snap-stabilizing, completing in zero rounds. However, existing protocols that terminate do not tolerate ongoing faults, and those that do are not terminating.

[47] presents a PIF in the form of distributed reset. It is tolerant to ongoing faults, returning either a correct completion or an error indicating a tree failure. However, the protocol is not terminating since the tree structure is checked periodically and repair occurs when a problem is detected. [7] and [24] give terminating tree construction protocols but they make the standard stabilization assumption that faults have stopped.

Network health monitoring. A variety of monitoring facilities have been developed for testbeds and for deployed wireless sensor networks, but all those we are aware of fall short of our needs. Experiments are assumed to run on homogeneous devices; existing support tools do not handle the network heterogeneity of testbeds.

Tools do not distinguish between the health of a node and the health of its links. For some, the reliability is too low to be useful and for others, there is a dependency on the communication structure of the application.

Traditional networks like the Internet use standard protocols such as SNMP [39] for monitoring network devices and identifying faults. However, SNMP assumes the IP routing layer in its operation and is therefore dependent on the fault-tolerance of IP to be able to reach the monitored devices; this is not sufficient for a wireless sensor network with non-IP networks such as mote radio. Other monitors like Sympathy [65] only handle radio networks and do not admit heterogeneity.

Similarly, Motelab [75], Tutornet [30] and Orbit [66] provide users with a ping-based status for each device, indicating whether it is reachable or not. However, simply detecting that a device is unresponsive on a given network is not sufficient since it does not support heterogeneous networks and does not distinguish network and link faults.

The Sensor Network Management System (SNMS) [72] supports monitoring of wireless sensor networks by providing its own network stack that includes routing. SNMS allows administrators to remotely query network devices and learn their status. However, SNMS does not deal with heterogeneous networks, and studies such as [11] show that reliability of SNMS does not suffice to provide accurate fault status.

Sympathy [65] is designed for fault detection at a central base station in a data collection application in which nodes periodically send data to the base. Sympathy thus exploits knowledge of a specific application's traffic pattern to define certain fault metrics. A similar approach is used in [69] where the fault management system

exploits the continuous data traffic flow in the network to piggy-back health information and uses route update messages to trace failed nodes. The dependency on an application makes these inappropriate for our purposes since monitoring is conducted only when an application is running.

5.2.10 Conclusions

Health monitoring is a critical network management service for dealing with the complexity inherent in large wireless sensor testbeds and network deployments due to network heterogeneity and the occurrence of different types of faults. In this section, we presented Chowkidar, a reliable service that provides a way to monitor the status of heterogeneous testbeds automatically. By reliable, we imply that Chowkidar only reports information that is consistent with the actual testbed state, so this information is useful to users interested in running experiments on healthy nodes and to administrators who additionally use diagnosis data provided by Chowkidar to keep the testbed fully functional. We presented two Chowkidar protocols, experimentally compared their performance and analyzed scalability issues. Our distributed Chowkidar protocol not only tolerates ongoing faults but also has low computation, message and time complexity, making it scalable for large scale networks.

5.3 Reporter

The Chowkidar protocols described above present a general solution to the standard problem of collecting management information from a wireless sensor network and as such can be used to solve a variety of management problems. However, there exist several management scenarios where Chowkidar may be too inefficient as it not only requires constructing an external tree structure, but also requires every node to

be involved. In this section, we identify termination detection, which is an important network management problem, as an example of such a scenario and present Reporter, a message-efficient termination detection protocol for wireless sensor networks.

5.3.1 Motivation

As demonstrated by several field deployments in environmental, military, industrial and other applications, wireless sensor networks have evolved from a network of dumb sensing devices that pump data to a central node to complex software systems. By way of illustration, the Multi-Target Tracking [28] and *ExScal* [1] systems consist of components for several tasks such as sensing, event detection, routing, tracking, network reprogramming, localization, scheduling, power management, health monitoring, parameter update, etc. The total processing and memory requirements of these software systems exceed the resources available on most existing wireless sensor devices. Consequently, multi-phase execution is a common paradigm in such complex sensor network applications.

In multi-phase execution, the application is broken down into multiple logical tasks that are executed at different times. For instance, the task of localizing the network to determine absolute or relative positions of nodes can be one application phase. Based on this information, the reprogramming of location aware application programs onto all nodes can be a subsequent application phase. Power management, health monitoring, and parameter updates can be regarded as separate phases. To switch between phases, a network manager relies on network management tools provided by services such as Deluge [38] and SNMS [72].

Requirements for application phases. The execution of phases in applications is typically governed by the following requirements.

1. *In-order execution*: Application phases are often dependent on the output of some earlier phases. For instance, an application phase cannot of course start until the network reprogramming phase, which downloads to the network nodes the programs of that application phase, is completed. Similarly, if an event detection and tracking phase involves execution of a location-aware routing service, it must be executed after the localization phase.

2. *Atomic execution*: It is often desirable that all up nodes executing a phase complete it before the next phase starts execution on any node. For one, this may be necessary for correctness of phases which require coordination between nodes for completion; i.e., if a node that completes downloading a new program or its localization operation starts executing its next phase unilaterally, its neighbors that are waiting to receive program download or localization information from it may never complete that phase. Also, even if phases are designed to execute correctly even when previous phases have not completed, the "synchronous transition" implied by atomicity may be desirable for performance reasons; i.e., it may avoid message interference between executions of the two phases and, in several cases, may help the new phase stabilize more efficiently in time or energy consumption. For example, if some nodes were to join a phase late they may cause many nodes to expend more effort, e.g., in routing to enter a beaconing phase to update their current link quality estimates [79] or in distributed time synchronization to choose the preferred time.

Given these requirements, we identify termination detection as a critical network management service. Generally speaking, robustness and minimality are two key design considerations for network management. Robustness implies that network management must function correctly even when an application has failed, thus management services should not critically depend on application services for their operation. For instance, even if the application routing structure is broken, network health and other management data should still be received from the network. Minimality implies that since management services need to co-exist with applications, they should be lightweight and not adversely affect application performance. Hence, while it is possible to add termination detection capabilities separately to individual protocols and application phases, we argue it would be inefficient to do so, given the node and network constraints in wireless sensor networks.

A baseline solution for termination detection is to use the standard pattern of Propagation of Information with Feedback (PIF), as is done by multi-hop network querying services such as Chowkidar, SNMS [72] and TinyDB [57]. In this approach, applications export a termination attribute to the network querying service. The querying service creates and maintains its own routing data structure (e.g., a spanning tree) for collecting the query results. The network manager then periodically sends a query wave to all nodes to determine their termination status and the results are collected over this structure. This solution is obviously message-inefficient because it requires the manager to collect periodic query results. A simple optimization to improve the efficiency of this baseline is to define a one-time query which is triggered at each node upon termination. Regardless, this approach requires every node in the

network to respond upon its completion. Moreover, this approach also incurs the additional overhead of having to maintain a data structure for query collection.

In this section, we reformulate the problem of termination detection for the reactive model that is basis of low power wireless sensor network protocols, and propose an efficient termination detection algorithm, Reporter. Our Reporter algorithm uses fewer messages than the baseline described above in two ways: first, it only requires a small subset of network nodes to send termination reports without compromising correctness; and second, it (re)uses existing network traffic to create a structure for report collection and thus reduces control messaging overhead. Reporter makes limited assumptions about the protocols whose termination it detects and, thus, is highly composable with most existing implementations. Since network based reprogramming is a critical global operation for most sensor networks, we use reprogramming as a running case study to concretely illustrate the working of our algorithm. We demonstrate the improvement in efficiency and reduction in overhead through real experiments based on the TinyOS [37] implementation of Reporter, that detect termination in Deluge [38] and Sprinkler [61], two popular wireless sensor network reprogramming protocols. Our experiments show that although these two protocols are quite different in their design and operation, Reporter yields comparable performance for both, providing further evidence of its applicability to other protocols.

5.3.2 System Model and Problem Definition

In this subsection, we state the system model assumed in the rest of the section and reformulate the standard problem of detecting termination of a distributed protocol for this model.

System model. We first present our model for the wireless sensor network, and then present the reactive computation model for the underlying application protocol whose termination is to be detected.

Network model. We assume a connected, multi-hop network of wireless sensor nodes, each having a unique identifier. We assume that wireless links are bidirectional; however, the reliability in both directions need not be the same: if a link exists in one direction with some non-trivial reliability, then a link exists in the opposite direction as well with some non-trivial, albeit different reliability. Given that communications are sent over a broadcast channel, we also assume that a node can *snoop* on message traffic in its radio neighborhood, i.e., receive messages sent by nodes in its one-hop neighborhood that are not intended for it.

We assume there is a single distinguished node called the base station from which the protocol is initiated and where termination reports are collected. The network manager uses the information collected at the base station to detect termination, hence the terms base station and manager are used interchangeably in this section.

Application protocol model. Network protocols may be classified as proactive or reactive. We assume that most sensor network protocols are inherently reactive; i.e., they become busy only in response to some input from the environment, such as a message received over the radio or data/events received from a sensor. Unlike traditional distributed systems, where a node can be in two possible states, idle or active, we identify a third state *done*, and a set of transition rules that must be followed by nodes in our reactive model:

1. A node is initially idle and continues to remain idle unless it receives a protocol message, in which case it becomes active.

2. Upon becoming active, a node can perform local computation, send, and receive protocol messages; however, if the node does not receive any protocol messages for an interval of time T, it goes to the done state. Note that a node can remain active indefinitely by continuing to receive protocol messages.
3. Once a node enters the done state, it does not receive any more protocol messages and does not become active again.

At first glance, this model may seem restrictive, but a closer inspection of wireless sensor network protocols reveals that most if not all terminating protocols do in fact satisfy it. For instance, during network reprogramming, sensor nodes remain active as long as someone in their neighborhood has not received some part of the program being downloaded. Further, nodes with incomplete programs initiate requests for missing data and receive it within bounded time specified by the protocol. However, once all nodes in the neighborhood of a node have acquired the program, it does not receive any more requests and enters the done state as described above. A similar argument holds for cooperative localization protocols that do RSSI or acoustic ranging between neighboring nodes to figure out relative distances from one another. Once a node has finished estimating its distance with respect to all its neighbors, it is done. Other examples of protocols that satisfy our reactive model include power management (sleep/wakeup) and parameter update, and for these the time bound T is a small constant that is independent of the size of the network. The PIF protocol itself also satisfies our reactive mode, but in this cases T depends on the size of the network, as a node that sends an outgoing/propagation wave along its subtree and has to wait for its subtree to finish before it can send its incoming/completion reply to its parent.

For ease of exposition, we consider the case where the underlying protocol is executed on all nodes in the connected network, however the ideas in this section apply to protocols that only execute on certain subsets or groups of nodes in the network.

We do not assume any knowledge of the internal operation of the protocol or its message formats for the purpose of termination detection. We do however assume knowledge of two protocol parameters. The first is the upper bound T on the reactive computation. This time bound T is usually specified as a protocol parameter by the designer or can be readily inferred from the performance of the protocol. The second parameter is the message type(s) used by nodes for exchanging protocol messages. This message type is similar to a port number in Internet applications and is usually well-advertised by protocol designers to avoid conflicts. We thus argue that these assumptions are reasonable and our approach is essentially a black-box one.

Problem. The problem of termination detection requires a network manager to detect that the underlying distributed protocol has terminated at all nodes. Termination detection requires the following standard properties of safety and liveness:

- *Safety*: If the manager detects termination, then the underlying protocol must indeed have terminated.
- *Liveness*: If the underlying protocol has terminated, then the manager must eventually detect termination.

Given the differences between traditional distributed systems and the wireless sensor network model, we impose the following additional requirements:

1. *Energy efficiency*: The detection must introduce minimal additional overheads, since wireless sensor nodes have limited energy resources and message transmission and reception is a significant energy drain.

2. *Composability*: For the detection to be generally used as a management service, it should easily compose with different types of network protocols. (This motivates that detection should not exploit any particular protocol data structures or implementation details.)

5.3.3 The Reporter Algorithm

We now describe our Reporter algorithm for efficient termination detection in wireless sensor networks. The baseline algorithm described in Section 5.3.1 requires continual querying of the entire network and hence is not only energy inefficient but yields more unreliability as responses from the entire network create a traffic burst leading to congestion and contention losses in the network. Studies such as [11, 78] have shown that in large scale networks, the reliability of such global data collection typically lies between 50-90%. Further, this algorithm depends on an external routing structure for actually collecting termination reports. Constructing and maintaining this structure imposes additional messaging and performance overheads.

The basic idea underlying Reporter is (i) to identify a small set of *local reporter* nodes that can detect termination locally by exploiting the reactive protocol model in wireless sensor networks, and (ii) to exploit existing protocol traffic to construct a routing tree over which termination reports can be collected. We can prove that receiving local termination reports from this small set of reporter nodes is sufficient for detecting global termination of the underlying protocol. Thus, the design of Reporter involves solving three subproblems, viz., **efficient selection of local reporter nodes, autonomous structure creation** and **detecting global termination from local reports**.

Algorithm Reporter

Var $is\_reporter$: boolean
$seen\_reporter$: boolean
$parent$: integer

Initially $(parent = -1) \ \wedge \ (!is\_reporter) \ \wedge \ (!seen\_reporter)$

Actions

$\langle A_1 \rangle :: send\_pdata = TRUE \ \longrightarrow$ **if** $(!seen\_reporter)$
$is\_reporter \ := \ TRUE;$
$seen\_reporter \ := \ TRUE;$
fi
$send(myid, \ is\_reporter, pdata);$

[]

$\langle A_2 \rangle :: rcv(id, flag, data) \ \longrightarrow$ **if** $(parent \ < \ 0) \ parent \ := \ id;$ **fi**
if $(flag) \ seen\_reporter \ := \ TRUE;$ **fi**
$process\_rcvd\_msg(data);$

[]

$\langle A_3 \rangle :: nbr\_activity\_timeout \ \longrightarrow$ **if** $(is\_reporter) \ send\_report(parent, myid);$ **fi**

[]

$\langle A_4 \rangle :: rcv\_report(i, \ j) \ \longrightarrow$ **if** $(i = myid) \ send\_report(parent, \ j);$ **fi**

Figure 5.5: The Reporter algorithm for termination detection.

5.3.4 Efficient Selection of Local Reporter Nodes

Given our model of reactive, broadcast-based wireless sensor network protocols, it is possible to infer termination from only a small fraction of nodes, where each report not only implies termination of that node, but also of other nodes around it. Reporter selects these local reporter nodes (which we henceforth abbreviate as reporter nodes) via the following rule:

> *Rule 1: Reporter selection*: If, at the time of sending an underlying protocol message, a node has not received a message from any local reporter, it elects itself as a local reporter.

The reporter selection rule is implemented by actions A_1 and A_2 in Figure 5.5. In action A_1, a node decides whether or not it should become reporter according to the above rule. Additionally, it also piggybacks its decision on the outgoing message. Conversely, in action A_2, for every received protocol message, a node checks whether or not the sender is a reporter and if so, records that it has now seen a reporter.

It follows from Rule 1 that the set of reporters is a subset of nodes that send protocol messages during its execution. In any protocol execution, some nodes both send and receive protocol messages while others only overhear messages sent by others. For example, in a reprogramming protocol, some nodes send protocol messages in the form of the new program while some send requests for missing parts of the program from neighbors, while others simply acquire the new program by overhearing these messages. Based on Rule 1, we can prove the following property about the reporter set selected by our algorithm.

Property 1: The set of reporter nodes in a protocol execution forms a Dominating Set over the set of nodes that send protocol messages.

Proof. A set of nodes D is defined to be a Dominating Set of a set of nodes N if every node in N is within one-hop of some node in D. Now consider the set S of nodes in a protocol execution that send protocol messages and assume that the set of reporters R is not a Dominating Set of S. Then there must exist a node s that sends a protocol message, yet there is no reporter in the one-hop neighborhood of s. However, according to Rule 1, s would itself have become a reporter, which violates the assumption. □

We now define the rule using which reporters can detect termination of the underlying application protocol in their local neighborhood and thereby report this to the manager.

> *Rule 2: Local termination detection*: If a reporter node does not receive any messages in an interval of time T, it detects termination locally.

Given the time bound T in the reactive protocol model and the fact that our algorithm snoops on protocol traffic in the broadcast wireless model, the above rule readily detects local termination. This rule is implemented by action A_3 in Figure 5.5. Upon detecting local termination, a reporter sends a report message to the base station. We now prove the correctness of Reporter as follows.

Property 2: The safety and liveness requirements of termination detection are satisfied if all local termination reports from the set of reporters are received.

Proof. From Rule 2, we know that if all nodes in the network have terminated or are in the done state, all reporter nodes will detect termination locally as they will not receive any protocol messages for time T. Thus, when all reports are received, the manager can detect termination and liveness is satisfied.

To prove safety, we must show that the manager does not detect termination when some nodes are still active. Assume that the manager receives reports from all reporters and detects termination before the protocol has terminated. If protocol execution has not yet terminated, there must exist some active node p in the network. From our reactive protocol model, we know that for p to be active, it must have received a message from some node q within the last T time. From Property 1, the set of reporters is a Dominating Set of the set of nodes that send protocol messages. hence there must exist a reporter r within the one-hop neighborhood of q and r must

have also received the message sent by q within the last T time. However, according to Rule 2, r could not have detected local termination if it received a message in the last T time. This is a contradiction. □

We therefore conclude that the set of reporters, which is a Dominating Set of the set of nodes sending protocol messages, suffices for termination detection. An optimal solution would thus require the reporter nodes to form the Minimum Dominating Set (MDS) of nodes that send protocol messages. However, computing the MDS is NP-hard and would require exponential time. Existing approximation solutions either perform much worse than the optimal or require non-local computations or some knowledge about the network topology. Also, the set of nodes that send protocol messages could vary over different protocol executions. We therefore eschew computing the MDS of the network.

We experimentally validate, in Section 5.3.8, that our simple reporter selection rule, which does not assume any topology knowledge or require non-local computations, performs comparably to the MDS solution, for the particular networks considered in our experiments.

5.3.5 Routing Structure Creation

Reporter solves the second subproblem of autonomous operation by creating its own structure for collecting termination reports. Our approach is similar to that used in the classic Dijkstra and Scholten [26] termination detection protocol, and constructs a collection structure efficiently by once again exploiting protocol traffic, according to the following rule.

> *Rule 3: Parent selection*: Every node selects the first node from which it receives a protocol message as its parent.

This rule is implemented by action A_2 in Figure 5.5. Initially, all nodes have invalid parents. Upon receiving a message, a node checks whether it has an invalid parent. For the first message received, this check succeeds and the node selects the sender of the received message as its parent. For subsequent messages, the check will fail as it already has a valid parent. Based on Rule 3, we can prove the following property about the structure created by the selected parent links.

Property 3: Reporter constructs a spanning tree over the entire network rooted at the base station.

Proof. Rule 3 defines exactly one parent for every node. Since the network is connected, every node receives at least one protocol message and acquires a parent. There also cannot exist any cycles in our structure because if two nodes p and q select each other as parents then they must have received their first protocol message from each other, which is a contradiction. (A similar argument applies to cycles of arbitrary length). Since the base station invokes the underlying protocol by sending the first message, we obtain a spanning tree rooted at the base station. □

From our link model, we know that if a parent-child link exists, then the child-parent link has some non-trivial reliability. This reliability can be further improved using acknowledgements and retransmissions. We show in Section 5.3.8 that despite using the simple spanning tree construction in our algorithm, we obtain almost 100% reliability with very basic per-hop reliability mechanisms.

Of course, an application may already be maintaining a routing structure, perhaps one that selects better links using estimation or by exploiting topological knowledge. In such cases, the network manager is free to reuse the existing structure to collect

termination reports. Our simple algorithm however does not make any assumptions about the availability of application routing structures, is lightweight and designed to work with all applications and suffices to provide reliable results.

5.3.6 Detecting Global Termination from Local Reports

In Section 5.3.4, we proved that the network manager could safely conclude global termination of the underlying protocol if it received report messages from all reporters. However, the set of reporters is generally not fixed. We therefore describe two techniques whereby the manager can learn that the set of reporters from which reports are received is complete.

For the first technique, we make the additional assumption that the manager, through a localization service, knows the locations of all nodes in the network. Assuming a disk model for communication, the manager can identify regions that must have terminated from the source locations of received reports. Even if the communication range of nodes is not a unit disk, previous research by Zhao et al. [80] has identified the existence of a stable region, referred to as *inner band* within which communication is reliable. Of course, some nodes outside of the inner band may still be able to communicate with the node over what are referred to as *long links*. However, this is not a problem as our algorithm is conservative, so if a node that has not yet terminated has a long link to some reporter, then that reporter will not detect termination in its neighborhood. Thus, a non-terminated node may slow down one or more reporters, however once the protocol has terminated, our algorithm is guaranteed to detect it.

The second technique requires the manager to learn its reporter set. In this approach, when a node elects itself as reporter, it communicates this to the manager using the constructed spanning tree. During protocol execution, the manager thus learns about the reporter set and can then wait to receive termination reports from all nodes in this set before detecting termination. This approach does not assume any localization information, but it requires twice the number of messages.

To guarantee correctness deterministically, our algorithm requires messages from all reporters to be received at the base station. However, due to faults such as reporter failure or message loss, some reports may never be received. Even in the absence of such faults, some node executing the protocol might start misbehaving by sending arbitrary protocol messages. In this case, its reporter, although correct, will not detect and report local termination. In such scenarios, a network manager may be forced to live with only probabilistic guarantees about the correctness of termination detection and deal with faults through predefined network policies. One example of such a policy could be to wait for a fixed time to receive termination reports and declare termination of the underlying protocol if more than x% of the nodes (say 90%) could be inferred to have terminated. Another policy could be to send out explicit network queries using more expensive protocols such as SNMS or TinyDB to regions from which no termination reports are received.

5.3.7 Implementation Details

We implemented Reporter in TinyOS [37] as a reusable component that is easily integrated with different types of network protocols. The time bound T for protocol reactiveness and the message type(s) used by the protocol (referred to in TinyOS as

handler-id), assumed known to Reporter, are specified as input parameters by the manager in a header file. The Reporter component includes two modules, one for snooping and the other for detecting and reporting local termination.

We implemented our snooping module at the level of the generic communication module, known as GenericComm, which is responsible for delivering messages to and from TinyOS components to the radio layer, for two reasons. First, our algorithm needs to snoop on protocol messages to learn about reporters in its neighborhood and to detect local termination. Second, our algorithm piggybacks reporter information on protocol messages (action A_1), hence it needs access to these messages before they are sent out. Since all message communication for TinyOS components is handled by the GenericComm component, implementing the snooping module at this level is most efficient.

The detection and reporting module maintains a timer set to expire after the time T specified in the header file, and resets this timer every time the snooping module receives a protocol message. If this timer ever expires, local termination is detected and reporter nodes send a report message to their parent to be forwarded to the manager. Since reliable delivery of termination reports is critical, we implemented a message buffer at intermediate nodes. Report messages are enqueued in this buffer and retransmitted until they are acknowledged by the parent or until the maximum number of retransmissions is exceeded. Our implementation thus uses explicit acknowledgements provided by the TinyOS MAC along with retransmissions to improve message reliability on a per-hop basis.

5.3.8 Performance

We first analyze the computation, memory and messaging overheads for our implementation, and then present experimental results.

Overhead. As seen from action A_2 in Figure 5.5, Reporter performs two extra comparisons to check whether or not the node has a parent and whether or not the sender is a reporter, for every received protocol message. Similarly, for every send operation as shown in action A_1, we require one extra comparison. However, the actions contained within each of these comparisons only need to be executed once per execution of the protocol; once assigned, a parent is not changed. Reporter adds one extra bit to each protocol message to indicate whether the sender is a reporter or not. For TinyOS, we also had to modify the message structure as the sender-id was not included in a message by default. We also require one message when a reporter's timeout expires. Reporter thus has very low computation and message overhead.

Finally, our implementation of Reporter has a very light memory footprint, requiring only few hundred lines of code and around 50 bytes of additional RAM.

Reporter is thus easy to implement with very little computation, communication and memory overheads.

Experimental setup. We tested the correctness and performance of Reporter by performing a series of experiments using our TinyOS implementation composed with the Deluge and the Sprinkler reprogramming protocols as a case study. We used a network of 105 wireless sensor nodes deployed in a 15x7 grid pattern. We present here a brief overview of the two protocols, highlighting the key differences between the two.

• *Deluge*: Deluge is an unstructured, flooding-based protocol that is designed to work even when no information about the network topology is available. In Deluge, nodes periodically advertise their program version and the fraction of this latest program they possess. Whenever a node receives an advertisement for a higher version number or for a missing fraction of its current version, it sends a request message to the advertiser, which responds with the requested program fraction. Deluge specifies upper bounds on the time interval between advertisements, time within which a request for missing program fractions should be made and the time within which such a request should be responded to; it thus satisfies our reactive protocol model. To prevent network contention due to redundant transmissions, Deluge uses a suppression mechanism whereby nodes refrain from sending new request messages as long as they receive useful program fractions by overhearing requests and responses from others. However, the selection of which nodes send the request messages is done randomly, hence it varies across different protocol executions.

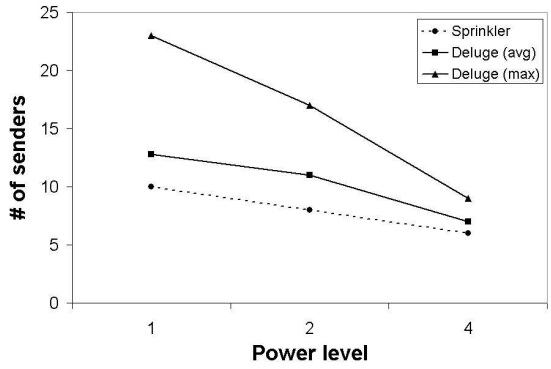
• *Sprinkler*: Sprinkler is a structured protocol that exploits topology information about a network to construct a backbone structure for program dissemination. The backbone constructed by Sprinkler is within O(1) of the Minimal Connected Dominating Set. Additionally, Sprinkler locally computes a TDMA schedule for disseminating new programs over the backbone node. Thus, Sprinkler preselects nodes that disseminate the new program and carefully schedules transmissions across these nodes to cut down message losses due to collisions and interference. The upper bound T for message forwarding can be derived from the TDMA schedule constructed by Sprinkler; it too thus satisfies our reactive model.

We selected Deluge and Sprinkler for our experiments because they are reliable, have been widely used in field deployments, and also use two completely different approaches for reprogramming.

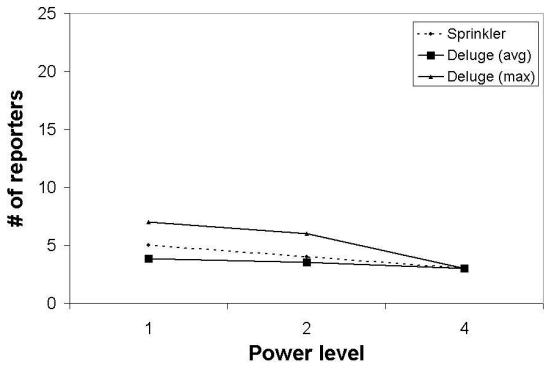
In each experiment run with each of the two protocols, we invoked a new reprogramming operation with a small, fixed new program. During a run, each node logged whether or not it sent any protocol messages and whether it became a reporter, the id of its parent, and the id(s) of reporter(s) in its one-hop neighborhood that it had overheard. The logged data from all nodes was collected reliably using the Ethernet backchannel in Kansei to construct the actual reporter set and the collection data structure. At the end of each run, reporters detected termination in their one-hop neighborhoods and sent report messages along their chosen parent links to the base station. Received reports were then compared with the actual reporter set to calculate reliability.

Results. Figure 5.6(a) plots the number of senders in the network for each reprogramming protocol and highlights the differences in their dissemination patterns. Since the inter-node separation was fixed for the testbed we used, we ran these experiments using different transmit power levels to create networks with different effective densities. The X-axis in the graph denotes power level while the Y-axis denotes the number of nodes that sent protocol messages.

As seen from Figure 5.6(a), the average number of senders in Deluge is more than that in Sprinkler, as expected. Also, while the Sprinkler backbone size remains fixed, the number of distinct senders in Deluge can vary quite a bit across different runs. For instance, at power level 1, we observed that as many as 23 out of 105 nodes sent



(a) Forwarding patterns for Deluge & Sprinkler.



(b) Reporter selection for our algorithm.

Figure 5.6: Efficiency of reporter selection in our algorithm.

a Deluge message in some execution while the number of senders in Sprinkler at the same power level was always 10.

However, even for these two protocols, that work quite differently, the number of reporters selected by our algorithm is comparable for the same power level (network density), as illustrated in Figure 5.6(b). Our experiments thus demonstrate the performance benefits of reporter selection in our algorithm. At the lowest power level (lowest node density), only **4-7%** of the total number of nodes became reporters while at higher power levels, i.e. highest node density, this number was even smaller. In most wireless sensor networks, network density is dictated by sensing range, as it is typically smaller than communication range. Reporter achieves substantial performance savings in such dense networks.

Figure 5.7 shows the spatial distribution of senders and reporters in the 15x7 network from one execution from both protocols at power level 2. The arrows indicate the routing paths chosen by Reporter for collecting termination reports. We see

that the dissemination pattern of Sprinkler is regular and deterministic while that of Deluge is random. However, the net performance of Reporter is quite similar for both protocols. It should be noted that in a small number of executions with the Deluge protocol, our algorithm did end up with reporters that were within one-hop of each other. This occurred due to message collisions during transmission as a result of which the piggybacked reporter selection was not heard by the second node that also chose itself to be a reporter. This does not affect the correctness of Reporter, and only slightly affects its performance.

In the baseline algorithm described in Section 5.3.1, every node sends a termination report to the base station which creates a burst of network traffic. Reliable end-to-end delivery of such traffic bursts is especially challenging, with even the best known solutions only achieving about 90% reliability. As shown in Figure 5.7, our algorithm selects only about 5% of nodes as reporters that are somewhat evenly spread across the network. For this less severe traffic load, even our simple, per-hop reliability mechanisms gave us 98.4% routing reliability on average for report collection at the base, with 92% of the runs yielding 100% reliability.

For the network setup and power level shown in the Figure 5.7, the size of the Minimum Dominating Set is 3 nodes, whereas our algorithm selects 4 nodes as reporters in the average case. However, our algorithm is forced to select the base station, which for this setup is a corner node, as a reporter since it sends the first protocol message. These results are reproduced at other power levels too where the number of reporters selected by our algorithm matches the calculated size of the Minimum Dominating Set.

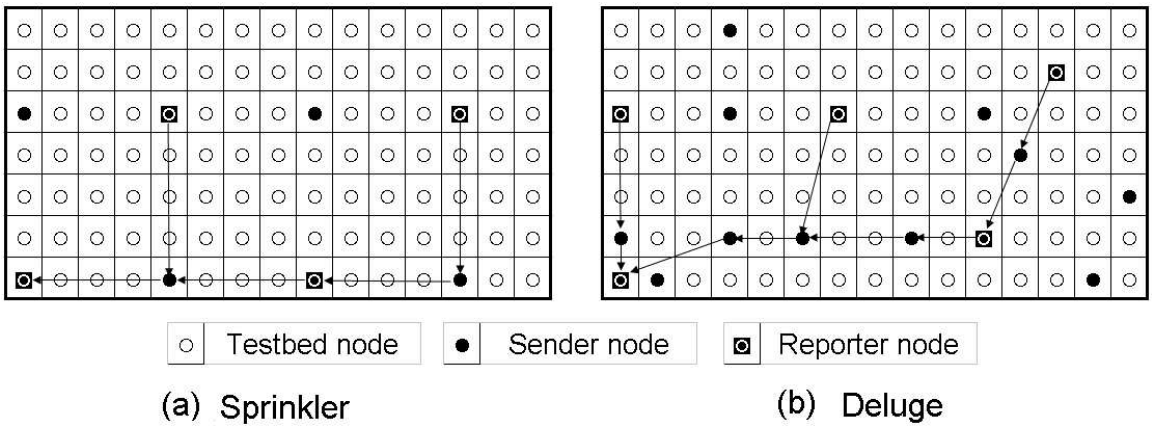


Figure 5.7: Spatial distribution of reporters selected by our algorithm.

5.3.9 Related Work

The problem of distributed termination detection was introduced by Dijkstra and Scholten [26] in 1980 and has been extensively studied since [17, 18, 26, 59]. While our approach for tree construction for collecting reports is similar to the one proposed in [26], our ideas for detecting termination differ from existing approaches in several ways. Existing termination detection algorithms require explicit signalling of control information between neighboring nodes and to the base station to detect termination whereas we exploit the reactive nature of wireless sensor networks and the broadcast communication model to reduce global termination detection to locally detecting neighborhood termination at a small fraction of nodes in the network. We also do not require all nodes to signal termination unlike previous approaches. Rather, our Reporter algorithm once again exploits the underlying protocol to select a small faction of nodes as reporters and is thereby well suited to sensor networks where message efficiency is highly desirable.

The concept of a Dominating Set has been used in protocols for bulk dissemination [61, 63] and clustering [19, 20]. However, these protocols typically use location information or introduce their own control messages to create a clustering structure in the network. Our algorithm does not assume any network information and imposes minimal control overhead for selecting reporters. Clustering algorithms also typically incur control overhead for maintaining clusters in the presence of node joins and failures. Unlike such static clustering schemes, our algorithm performs a one-shot, on-demand selection of reporters for every execution of the underlying protocol. Also, as shown by our experiments with Sprinkler, Reporter automatically exploits any existing structure of the underlying protocol, however it can work equally well with unstructured protocols, as demonstrated by or results for Deluge.

5.3.10 Conclusions

Termination detection is a critical service for management of multi-phase sensor network applications. In this paper, we reformulated the classical termination detection problem by identifying a reactive model that is commonly used in low-power wireless sensor network protocols, and proposed Reporter, an algorithm for efficient termination detection in this model. Reporter imposes minimal computation, communication and memory overheads while achieving significant message efficiency over existing solutions. This improved efficiency also results in an improvement in reliability, which would otherwise be hard to achieve. We implemented Reporter as a TinyOS component that is readily integrated with most sensor network protocols to detect their termination. Finally, we validated the correctness and performance of

Reporter through experiments on a real testbed for two popular reprogramming protocols. Our experiments show that even though the two protocols are quite different, Reporter achieves comparable performance for both, and requires only about 5% of the number of nodes required by existing protocols to send termination reports.

CHAPTER 6

EXPERIMENT ORCHESTRATION

6.1 Introduction

Existing network management scenarios and services available for the wireless sensor network context are to a large extent dependent on some form of human involvement. In some cases, this involvement is needed because the faults that management seeks to address are not clearly defined or may be unanticipated, hence management only helps in detecting that something has gone wrong and not in responding to it. In other cases, even though the nature of faults is known a-priori, completely autonomous solutions may not be possible, as we showed for the stabilizing reconfiguration problem in Chapter 4. Finally, autonomous management may be hard to achieve because the management protocols used might themselves be prone to faults in the unreliable wireless sensor network environment.

Although automated network management is an especially hard problem, we identify in this chapter, several commonly used patterns in wireless sensor network experimentation for which automation can be achieved. We then present a framework for orchestrating wireless sensor network experimentation that is part of our MASE

architecture. Our experiment orchestration is made up of the following three components — a library of standard components that implements the various orchestration functions, tools for instrumenting user applications with these library components, and a scripting environment for specifying and executing user applications.

We first describe two experimentation patterns along with application-specific case studies that are well suited for automated network management and form the use cases for our orchestration framework. We then describe the design of our network orchestration framework and highlight the roles of its different components. Finally, we present the implementation of this framework for our Kansei testbed as a case study.

6.2 Experimentation Patterns for Automation

In this section, we identify two commonly used patterns in sensor network experimentation that lend themselves easily to automated management.

6.2.1 Iterative Execution Pattern

In the *iterative execution pattern*, an experiment is defined as a series of iterations for the same application, each using different parameter values as inputs. In traditional software systems, the iterative execution pattern is often used in testing to verify that the application works correctly for a large set of inputs. However, in wireless sensor networks, iterative execution of experiments is also critical for selecting the optimal protocol parameters at design time. This is because unlike the Internet, wireless sensor networks do not use a standardized, layered communication architecture. In fact, the performance of network protocols is often highly sensitive to the type of application and its generated traffic. Designing a network protocol that

provides a certain quality of service thus typically involves an iterative tuning process in which different application and network parameters are tested to determine the ones that perform best. We now describe, using the influence field based classification application as a case study, how experiments that follow the iterative execution pattern can be automated.

Case Study: Influence Field Based Classification

The *influence field* of an object j with respect to a given sensing modality is the region surrounding j where it can be "detected" by a sensor of that modality. This region depends on both the characteristics of the object, such as its size and shape, as well as the sensing modality being used. Differences in the area and/or shape of the influence fields of different objects can often be used to distinguish between them. Figure 6.1 illustrates the differences between magnetic influence fields for two objects, a person carrying a metal rod and a vehicle. The size and shape of the influence fields shown in this figure depend on the amount and distribution of metallic content in each object type and the orientation of the object (e.g., the dumbell shape of the vehicle influence field is attributed to the positions of its axles). The influence field feature is thus useful in sensor network applications for surveillance, where typical tasks include detection, classification, and tracking of various types of objects.

As a matter of fact, the influence field feature was the primary basis for classification and tracking of people, people carrying a significant amount of metal on them (aka "soldiers") and cars, via a dense, wireless sensor network in both *A Line In The Sand* and *ExScal* deployments for intrusion detection. To estimate the influence field, each node merely has to detect a binary "presence" of an object; network-based aggregation of these bits yields the influence field without substantial or complex node

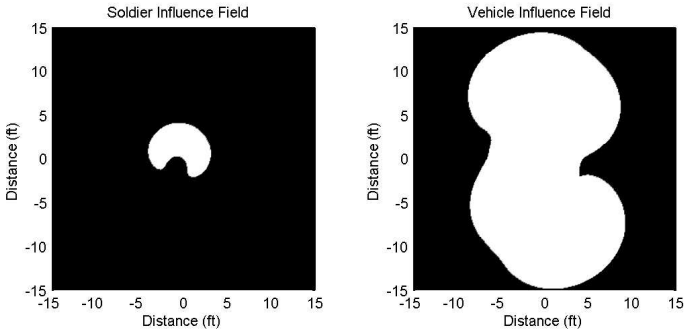


Figure 6.1: Magnetometer based influence fields for two object types.

operation. The influence field feature is thus well suited for wireless sensor network applications where individual nodes are constrained due to limited processing, sensing and communication capabilities. The key challenge in realizing the influence field is the unreliability of wireless sensor networks. Event losses –both in nodes and in the network– are fundamental to wireless sensor node platforms and their impact on the application can be substantial. Thus, both node and network unreliability have to be dealt with while estimating the influence field.

In [12], we analytically derived necessary conditions such as minimum sensor node density and network reliability required for reliably estimating influence fields of objects to achieve classification for basic fault models such as node fail-stops, false positives and false negatives, channel fading and channel contention. However, no well-defined fault model exists for end-to-end routing in multi-hop wireless sensor networks where the extent and distribution of faults not only depends on the particular network protocols used but also on the amount of traffic generated at various points in the network. Consequently, to validate the influence field approach in real

settings with multi-hop routing, we had to rely on experimentally measured performance of the various algorithms. This experimental validation process is concretely illustrated below.

The classification accuracy specified by the user dictates the minimum separation, in terms of the number of detection messages, that should be achieved between the estimated influence fields of different object types. However, end-to-end network reliability is an unknown, complex function of different parameters of both the application, viz. the type of object being detected (the number of detections), the transmission probability for each detection (the total network traffic), its location (distance from the base station) and routing protocol, viz. the number of retransmissions, transmit power level, transmission latency, etc.

Figure 6.2 shows how the estimated influence fields and the classification accuracy varies as a function of these different parameters for two of the objects, viz. a soldier and a car, being distinguished in the *A Line In The Sand* system.

As seen from this figure, the parameter settings of (9,3,15) gave us the best classification performance for the target object types. However, to collect the data needed to make this determination, we ran a total of 280 experiments using 16 different parameter configurations which is a small subset of the possible parameter space. Such an experimentation procedure is extremely repetitive and requires extensive human involvement for relatively simple tasks, thereby making it an excellent choice for automation.

Now consider the same scenario in which the user specifies a list of parameter choices in a configuration file associated with an application. Also assume that the patterns of event detections for different object types have been captured as traces

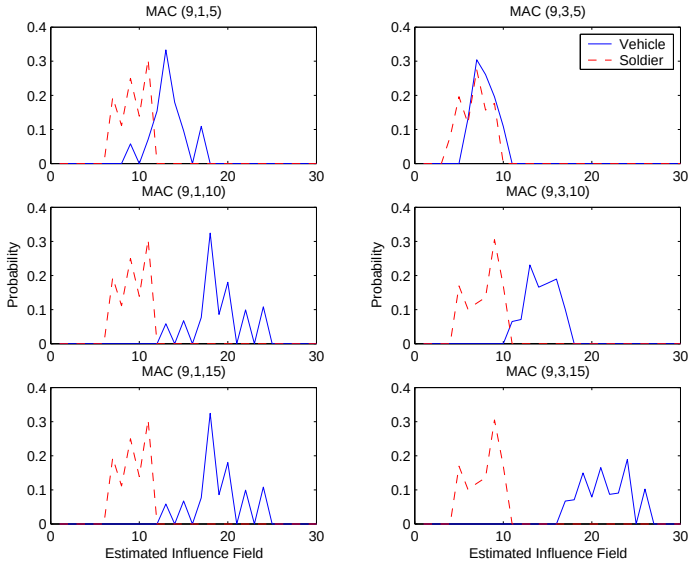


Figure 6.2: Probability distribution of the estimated influence field as a function of media access control (MAC) power, transmissions, and latency. MAC(P,T,L), where P is the power setting, T is the total number of transmissions, and L is the latency in seconds.

during a one-time field experiment. The Kansei testbed already provides support for injecting sensor data traces into user applications to emulate real world sensing behaviors. An orchestrated sensor network experiment would thus iterate through the set of parameters specified by the user, configuring it with one set at a time, inject the corresponding object traces in the network, and log the output of each run. A human user could then look at the collected data at the end of all runs, or provide an evaluation function to the framework, to determine the best choice.

6.2.2 Multi-phase Execution Pattern

In the *multi-phase execution pattern*, an experiment is defined as a sequence of multiple phases, each of which is invoked and executed synchronously. As described

in Chapter 5, sequential execution is common for experiments in which the current phase depends on some functionality provided by the previous phase and therefore on the successful completion of that phase. We also motivated in Chapter 5, the need for synchronized execution of application phases to optimize network performance. In the multi-phase execution pattern, the network manager decides whether or not to switch to the next phase based on a pre-defined policy. One type of policy is to switch to the next phase when a certain number of nodes – 1, >90%, all – have finished executing the current phase. We now describe, using the *ExScal* application as a case study, how experiments that follow the multi-phase execution pattern can be automated.

Case Study: The *ExScal* application

In Chapter 2, we described *ExScal*, which is a typical intrusion detection application well-suited for wireless sensor networks. The *ExScal* system consists of several components, each of which performs specific tasks such as reprogramming, localization, routing, sensing etc. Due to resource constraints on wireless sensor devices and limitations on network bandwidth and other resources, the execution of these components is broken down into multiple successive phases.

Such synchronized, multi-phase execution of sensor network applications is a suitable candidate for automation using experiment orchestration. Using the fault models for various application phases as derived in Chapter 2, a manager can specify the minimum number of nodes that need to complete a certain phase before switching to the next one. For example, consider the reprogramming phase in the *ExScal* application which downloads the other application components in the network. Before invoking the reprogramming operation, a manager might need to verify that a certain fraction

of nodes are actually in the reprogramming phase and have not failed due to deployment or configuration faults. Similarly, it is critical for a manager to ascertain that the reprogramming phase has finished before invoking the next phase. In both these cases, the manager can use the reprogramming fault model to determine the number of nodes expected to complete successfully.

We have already shown through our design of services such as Chowkidar or Reporter how a network can be instrumented to reliably report node health or reprogramming termination status. These services could thus be used by an automated manager to collect this information from the network. We also know from the derived fault models the fraction of network nodes expected to survive deployment and enter reprogramming and the fraction of these nodes expected to successfully complete reprogramming. Based on these fault models or on application quality requirements, *ExScal* users could specify a threshold value for the minimum number of nodes required to complete the reprogramming phase and the expected time in which this operation should complete. The automated manager could then wait for the user-specified amount of time to collect node health or termination reports following which it could automatically invoke the next phase if the threshold was exceeded. For scenarios where the threshold is not exceeded, users can specify policy actions that the manager can automatically invoke. One policy could be to simply retry this phase with a new invocation. Another policy could be to suspect the reprogramming service as bad and reboot to a default or "safe" application. Finally, a simple policy might be to simply halt execution and notify a human manager via email.

Although the responsibility of specifying the policies and intervening if none of them work ultimately lies with the human network manager, the execution of the

application, which in the common case either respects the known fault models or can be managed using the specified policies, can be largely automated if the network was instrumented to provide appropriate feedback to a manager that could use it to orchestrate network actions. Our personal experiences about the human efforts required to execute these relatively mundane tasks while managing the *ExScal* network convince us that synchronized, multi-phase execution is a feasible candidate for orchestrated network execution.

The patterns described above indicate that sensor network experimentation can be automated in some cases to reduce human involvement by automating simple tasks such as experiment configuration, iterative data collection and sequential execution, based on user specified parameters and policies. In the following section, we describe the different elements of our experiment orchestration framework that enable such automation.

6.3 Experiment Orchestration Framework

The management schema presented in Chapter 3 in which we identified the key steps performed by a manager during a sensor network experiment. Based on this schema and the experimentation patterns described in the previous section, we identify the three elements in our experiment orchestration framework as: (i) software library (ii) instrumentation and (iii) execution control logic. In this section, we describe the roles of each of these elements in the overall orchestration framework.

1. **Software library.** Recall from the network management schema that the manager receives as input from the user, an experiment specification that includes the different user programs to be executed on different platforms and the parameters for each of these programs. However, before configuring this experiment in the network, the manager includes certain additional components in this specification to enable network management functions.

   For instance, health monitoring components may be included by the manager for monitoring the status of user programs in an ongoing manner, termination detection components may be included to detect termination of program execution, time synchronization and data logging components may be included to timestamp and log error messages. Library components provide and use well-defined, standard interfaces that are known to user programs so that they can be easily integrated with and can interact with user programs.

   The role of the software library in our orchestration framework is to provide components that implement standard functions that are required for network management.

2. **Instrumentation.** Modern, sophisticated operating systems and execution environments provide dynamic, runtime component instantiation and linking capabilities during user experimentation. However, constrained memory and processing resources force sensor network operating systems such as TinyOS to sacrifice dynamism and runtime flexibility in favor of efficiency. TinyOS applications are thus loaded and executed as a single, monolithic program with multiple components linked statically to each other at runtime.

Also, large scale sensor networks and testbeds such as *ExScal* or Kansei are often heterogenous in nature. In such heterogenous networks, it is not sufficient to include a management function at one level but not at others. For example, to perform health monitoring for the mote network, the corresponding health monitoring components on Stargates or other network platforms may also need to be loaded.

The two tasks in the instrumentation process for experiment orchestration are: (i) inclusion and correct wiring of library management components with user programs and (ii) configuration of experiment specification to include additional components required for these management functions and their appropriate startup parameters.

While these tasks can and typically are performed manually by users for wireless sensor network experiments, we provide tools for code rewriting and dependency evaluation that automate the instrumentation process.

3. **Execution control logic.** The software library and instrumentation tools in our framework help automate the important step of experiment specification in the network management schema. Once the experiment is specified however, it needs to be executed in the network. This execution is governed by user specified rules that are typically enforced by a human manager.

   To automate this process, our framework provides the following two components: (i) scripting constructs using which users can specify execution rules for an experiment and (ii) an execution control runtime that executes the experiment as per the user-specified rules.

The scripting constructs for specifying execution rules are based on the operations identified in the experimentation patterns described in Section 6.2. For example, in the iterative experiment pattern, we need to execute experiments one after the other, which implies a ';' operation. For multi-phase executions, it may be required to verify that the previous experiment terminated before the next one is started. The instrumentation tools could then include a termination detection component along with the user programs as part of each experiment. The execution runtime is programmed to receive management messages of certain types that are generated by the instrumented management components, hence in the example described above the runtime can determine termination of an experiment if it receives termination messages from each instrumented component in the network, implying a 'conditional' operation. Termination messages could be lost however, hence the runtime could blocked forever waiting for a lost message. To solve this problem, users may specify a 'wait' time during which the runtime waits to receive management messages, after which it can decide whether or not to proceed based on a threshold value for received messages.

Management of wireless sensor network experiments that follow the patterns identified in Section 6.2 can thus be automated using our orchestration framework described above. Clearly, additional scripting constructs will be needed and the runtime logic will need to evolve to automate more and more experiment patterns. However, we believe that this can be accomplished by defining appropriate management components that can be instrumented to provide the required management information in each experiment.

6.4 Kansei Implementation

The orchestration framework design, presented above, is independent of network specifics and can be instantiated for different hardware and software platforms. As a case study, we present the Kansei implementation of our framework which includes the XSM and TelosB mote platforms running TinyOS, the Stargate platform running Linux and a server platform running Linux. We now describe the different pieces that we have implemented on the various Kansei platforms to realize the above framework.

6.4.1 Software library

The orchestration library for Kansei provides the following standard components and interfaces for common functions needed for management of testbed experiments. It should be noted that some of these components have been implemented by modifying existing components that were already available as part of the Kansei experimentation toolkit.

1. *Data logging.* Logging data about reliability or faults for later analysis is a key management operation for most sensor network experiments. We therefore provide a TinyOS logging component that can be used by experiments to easily log management data by sending it to a serial port such as the UART or USB connector on the XSM or TelosB platforms respectively. Our logging component also includes the standard SerialForwarder implementation for Linux which needs to be executed on Stargates to receive messages on the UART or USB ports.

2. *Data injection.* As stated earlier, the Kansei testbed supports the injection of trace data, collected offline, into a running experiment to emulate real world phenomena realistically in an indoor environment. We exploit the injection facility provided by Kansei to provide a standard component that allows multiple user components to share the same injection interface. Again, our injection component includes a TinyOS-based implementation for motes and a Linux-based implementation for Stargates.

3. *Synchronized time.* Experiments often require network nodes to have access to a common time base. This synchronized time can be used for time-stamping logged management data or for coordinating experiment actions. We provide a simple time synchronization component that provides a common network time across the entire mote network. We exploit the NTP protocol, which runs in server mode at Kansei and in client mode at all Stargates to provide synchronized time for all Stargates.

   Our time synchronization module on a Stargate simply reads this synchronized time value provided by NTP periodically and forwards it over the serial port to the attached mote(s). Our TinyOS time synchronization component running on the mote thus periodically receives synchronized network time which it keeps updating in the refresh interval using its local clock. We argue that the drift in local clock values over the interval between NTP refreshes is tolerable for network management.

4. *Synchronized execution.* A TinyOS program typically begins to execute on a node as soon as the node has been programmed. However, due to the large scale

of Kansei, the time at which nodes across different parts of the network finish programming could differ by several minutes. However, for certain experiments, network managers may wish to synchronize execution across the network.

Our synchronized execution component for motes allows a network manager to initialize, start and stop experiment execution on a mote through a network based interface similar to the `StdControl` interface in TinyOS. A user experiment that needs network based control thus simply has to subscribe to this component interface, which includes the init, start and stop functions. The init and start functions are used to initialize and start components in a synchronized manner using network based commands while the stop function is used to indicate termination of the experiment on that node and results in a message being sent to the execution runtime at the manager. Once again, the standard SerialForwarder component is used to communicate commands and notifications to and from motes.

6.4.2 Instrumentation

As described earlier, instrumentation consists of two parts - adding management components to user programs and configuring experiment specifications to include these management components.

Recall that due to efficiency considerations, the TinyOS compiler combines all of the system and user components into a single executable. Our code instrumentation tools for the TinyOS platform therefore require access to user source code prior to compilation. We provide command-line tools that accept annotated user source code written in TinyOS as input and automatically include the appropriate management

components, generate the necessary wirings for these components and produce new TinyOS code which can then be compiled using the standard TinyOS compiler.

To illustrate this process, we describe how users can access the time synchronization component in our Kansei implementation. The TinyOS time synchronization component maintains time as a 32-bit unsigned integer which can be accessed within a user program using the `@KanseiTime` operator. A sample user program, called `SampleM.nc` could thus use time as follows:

```
uint32_t currentTime;
.  .  .
currentTime = @KanseiTime;
```

Our code instrumentation tool which accepts this user program as input, replaces the @KanseiTime term with the appropriate function call as follows:

```
currentTime = call KanseiTimeSync.getTime();
```

It also wires the `KanseiTimeSyncC` component which provides the KanseiTimeSync interface used in the above example to the user component `Sample.nc` as shown below:

```
SampleM.StdControl -> KanseiTimeSyncC;
SampleM.KanseiTimeSync -> KanseiTimeSyncC;
```

The resulting code would form a valid TinyOS program and be compiled as usual to get a standard TinyOS executable.

The second part of our instrumentation process involves configuring the experiment specification. We assume that experiments in the Kansei testbed are executed by providing a specification file as input which lists the different mote, Stargate and server programs that need to be started in this experiment along with their parameters. Thus, in the example described above, for an experiment to access network

time, the NTP server and client components need to be started on the Kansei server and Stargates respectively. This information is made available through a dependency database to our instrumentation tools which then modify the specification file for Kansei to include these programs for execution.

It should be noted that at present, Kansei supports experiment scheduling only through an interactive user interface; adapting the scheduler interface to accept specification files as input is part of the ongoing Kansei development following which our experiment instrumentation can be fully integrated with Kansei.

Related work. Instrumenting application code has been studied extensively for traditional software systems and also to some extent for wireless sensor networks. We present a brief overview of some of the existing approaches for experiment instrumentation, highlighting similarities and differences with respect to ours.

The SNMS [72] system allows application developers to expose certain attributes of interest to a management subsystem so that they can be queried over the network. The SNMS approach is similar to ours in that both use user-annotations in user provided source code to appropriately insert and wire the desired management functions. SNMS does not however address configuration for multi-platform experiments as we do.

Both SNMS and our approach are based on pre-compilation processing of user code. Alternatively, this functionality could be built into the compiler as is the case with TinyOS attributes. We chose our current approach over this to avoid frequent compiler changes as more and more management functions are identified and added and also because we would need additional tools to instrument across platforms in any case.

6.4.3 Execution Control Logic

In the previous subsection, we showed how an individual experiment can be automatically instrumented to generate the correct input specification for Kansei. In this subsection, we describe the proposed constructs and the execution runtime which enable automatic control of multiple executions of experiments that follow the patterns described in Section 6.2. The execution control environment is designed for the Kansei server platform and it can execute operations on Kansei and receive messages from it.

1. *Basic Kansei operations.* Our orchestration framework is intended to provide support for higher level constructs using basic experimentation primitives. It therefore follows that all the basic experimentation primitives supported by Kansei are also supported in our framework. For example, users may specify operations such as `execute(exptconfig1.cfg)` or `inject(tracedata.txt)` as part of the experiment control script.

2. *Sequential operations.* Sequential operations are specified by the user using the semicolon operator. Iterative experiments can be executed as multiple sequential operations. Thus, the iterative pattern in *A Line In The Sand* can be automated using multiple `execute` operations, the configurations for each of which differ only in the parameter values used while compiling the application.

3. *Wait operations.* Wait operations suspend the execution of the control logic for the time specified by the user. Wait times may be specified by the user in order to allow an operation to complete before invoking another one during sequential execution. Thus a user may specify

`execute(exptconfig1.cfg); wait(600); execute(exptconfig2.cfg)`

to allow experiment 1 to run for 10 minutes before experiment 2 is started.

4. *Conditional operations.* Conditional operations allow users to specify multiple execution paths from which one can be chosen at runtime. As described earlier, the instrumentation process includes management components such as termination detection or health monitoring that send certain types of notification messages to the manager. We therefore allow users to specify conditions based on these received management messages as shown below:

   `if (numRcvdMsgs[terminationtype] > 90) execute(exptconfig2.cfg)`

   `else wait(600)`

   This operation states that the experiment 2 can be executed if termination messages have been received from at least 90 nodes, else the manager needs to wait for 10 more minutes.

5. *Runtime.* The experiment control runtime is implemented on the Kansei server and performs two functions. First, it sequentially processes the user specified experiment control script to execute the listed operations. Second, the runtime maintains a communication channel with each experiment node over which it can send or receive messages to and from the different management components that have been instrumented as part of the experiment. The runtime logs the different management messages received from these components that are instrumented as part of the experiment so that they can be used to perform conditional checks as described above.

Again, note that some of these constructs cannot be fully implemented and integrated until Kansei is redesigned to allow scheduling through specification files. However, in its current state, the runtime does log the intended actions that it would have executed to a file as well as all of the management messages that are received.

6.5 Conclusions

In this section, we described our framework for network orchestration that allows the automation of simple experimentation patterns that commonly occur in wireless sensor networks. We also described the implementation of our framework for the Kansei wireless sensor network testbed. Our framework can be extended by providing more library components, adding more complex constructs for experiment control and providing better instrumentation tools, including graphical user interfaces based on the Eclipse [32] architecture, where library components can be added to user programs using an instrumentation plugin. Finally, we plan to fully integrate our framework with the Kansei environment to make it available to all testbed users.

CHAPTER 7

CONCLUDING REMARKS

Given the sophistication achieved by several research and industrial teams in designing and deploying wireless sensor network applications over the last few years, it is only a matter of time before these networks become pervasive in our daily lives.

Our experiences in designing and fielding large scale wireless sensor networks such as *A Line In The Sand*, *ExScal* and *Kansei* have shown that in the resource constrained sensor network environment, failures are bound to occur. Moreover, it is extremely hard to design applications that perfectly tolerate all occurring faults, especially if they are unanticipated.

The large scale at which wireless sensor networks are deployed also makes zero or one-touch network operation critical. This implies that even when faults occur, nodes must still be able to communicate enough information about the state of the network and have the ability to be recovered remotely through network based actions.

Our thesis therefore is that reliable and scalable network management is the key enabler for these wireless sensor network applications that are limited by resource constraints and prone to different types of faults. In this dissertation, we therefore studied this reliable and scalable network management problem for large scale deployments and testbeds of wireless sensor devices.

7.1 Summary of Contributions

We proposed a fault model for large scale wireless sensor networks based on empirical fault measurements collected from several outdoor deployments such as *A Line In The Sand* and *ExScal*, and in our indoor testbed Kansei. Our fault model identifies different types of faults that can occur in a wireless sensor network and the impact of these faults on the performance or the *yield* of the system. Our findings provide key insights for network designers and managers to design the best overall system configuration that meets application quality requirements and to determine the management services such as health monitoring or *Human-In-The-Loop* stabilization that are needed to deal with these faults.

We designed and implemented the MASE architecture for the reliable and scalable management of wireless sensor networks and as part of MASE, developed components for key management services such as network reconfiguration, health monitoring, termination detection and network orchestration.

Our stabilizing reconfiguration protocol solves the critical problem of version number cycling that we identified to exist in existing reconfiguration services. Using a novel approach called *Human-In-The-Loop* stabilization, our protocol guarantees convergence of configuration versions via a local detection algorithm that requires low computation and communication overheads.

The Chowkidar protocol enables a network manager to obtain reliable node and link health information from all nodes in the network. Chowkidar uses low cost tree construction and PIF wave protocols and can tolerate ongoing node and link failures.

Chowkidar is especially useful because of its masking fault-tolerance property in monitoring scenarios such as testbed experimentation or network control and actuation where accuracy is critical.

The Reporter protocol enables highly efficient monitoring of the termination status of application protocols or phases. In contrast to Chowkidar, Reporter computes a small subset of network nodes whose collective termination status is sufficient to indicate the termination status of the whole network. Reporter is thus well-suited for field deployments of battery powered devices where energy efficiency is critical.

Having designed reliable building blocks for network configuration and observation, we then proposed a network orchestration framework that aims to reduce human involvement in wireless sensor network management by automating commonly found experimentation and execution patterns. Our implementation of this framework for the Kansei testbed platform provides a rich set of library components, tool support for instrumenting user programs and experiments with these components and an execution environment for the instrumented user experiments.

7.2 Future work

7.2.1 Extensions to Proposed Ideas

As more and more data is obtained from long-lived wireless sensor network deployments and testbeds, for different hardware and software platforms, we need to keep refining our existing fault models in the hope of designing systems that will produce predictable behaviors once they are deployed in the real world.

We believe that the *Human-In-The-Loop* approach to stabilization, used to achieve stabilizing reconfiguration in this dissertation, is particularly useful in wireless sensor

networks where self-stabilization may be impossible or extremely expensive to achieve due to resource constraints. We plan to explore other problems where using our approach would either be necessary or more efficient than its existing, self-stabilizing counterparts.

In this dissertation we proposed two approaches for monitoring network state, both of which are guaranteed to cover the whole network state. An alternative management approach assumes that wireless sensor networks are inherently probabilistic in nature and therefore achieves efficient monitoring by sampling the network, instead of collecting full information, to provide probabilistic guarantees about network state. We intend to study sampling-oriented solutions for various network management tasks and to compare their correctness and performance advantages over their deterministic counterparts.

Our network orchestration framework enables certain types of user experiments to be automated. In addition to fully integrating our implementation with the Kansei testbed, we intend to extend network orchestration to other experimentation patterns and management tasks. We also plan to integrate orchestration with protocols such as Chowkidar and Reporter so that it can be used to automate execution of outdoor deployments.

7.2.2 Other Relevant Management Problems

In this dissertation, we have addressed some of the key issues in wireless sensor network management. However, there are other problems that are also relevant in the context of making wireless sensor networks more practical and easy to deploy, use and manage.

A number of software faults in wireless sensor networks are caused because the total processing or memory requirements for concurrently handling multiple events exceed the resources available on a node. Scheduling is a well known mechanism for resolving conflicts in resource access, however scheduling for event-based sensor networks requires careful design so that it does not force applications to become overly conservative. An interesting resource management problem in this area is determining whether a given application, which is a set of components, each with an expected processing, memory and real-time requirement, can be safely executed on a given sensor device. Conversely, one could determine the critical conditions such as the maximum rates of certain events, at which the application would enter unstable states. This is similar to a counterexample generated by a model checking algorithm. Additionally, the manager could also determine a safe schedule by reallocating tasks within a single node or a group of nodes.

Another important resource management problem is that of managing the total power consumption of the network to maximize network lifetime. Existing research has addressed power management problems such as determining the right amount of redundancy to deploy in a network for it to last a given lifetime and determining appropriate scheduled or randomized sleep and wake-up algorithms that maximize lifetime. However, given that faults may occur at any time and that application quality needs to be maintained despite faults, the existing schemes for power management need to be integrated with online fault monitoring and diagnosis to help network managers always maintain the correct network configuration. Exploring this relationship between fault monitoring and power management is an interesting management problem.

Security is an important requirement for many sensor network applications such as intrusion detection or actuation and control, where an adversary could influence the network to avoid detection or perform incorrect actions to destabilize the system. In joint previous work with Naik et al. [60, 62], we considered the problem of key management using a secure pairwise key update protocol called Whisper. For large scale networks, pairwise key updates may not scale well; secure, managed updates of security keys in a wireless sensor networks is therefore a challenging problem of interest to us.

BIBLIOGRAPHY

[1] A. Arora et al. ExScal: Elements of an extreme scale wireless sensor network. In *11th IEEE Intl. Conf. on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 102–108, 2005.

[2] A. Arora, P. Dutta, S. Bapat, and V. Kulathumani et al. A line in the sand: A wireless sensor network for target detection, classification, and tracking. *Computer Networks, Special Issue on Military Communications Systems and Technologies*, 46(5):605–634, July 2004.

[3] A. Arora, E. Ertin, R. Ramnath, M. Nesterenko, and W. Leal. Kansei: A high-fidelity sensing testbed. *IEEE Internet Computing*, 10(2):35–47, March/April 2006.

[4] A. Arora and M. Gouda. Distributed Reset. *IEEE Transactions on Computers*, 43(9):1026–1038, 1994.

[5] A. Arora, P. Sinha, E. Ertin V. Naik, H. Zhang, M. Sridharan, and S. Bapat. ExScal Backbone Network Architecture. Appeared as poster in MobiSys'05.

[6] B. Awerbuch, B. Patt-Shamir, and G. Varghese. Self-stabilization by local checking and correction. In *32nd Symp. on Foundations of computer science*, 1991.

[7] B. Awerbuch, B. Patt-Shamir, and G. Varghese. Self-stabilization by local checking and correction (extended abstract). In *Proceedings of 32nd Annual IEEE Symposium on Foundations of Computer Science*, pages 268–277, 1991.

[8] B. Awerbuch, B. Patt-Shamir, G. Varghese, and S. Dolev. Self-Stabilization by Local Checking and Global Reset. In *8th Intl. Work. on Distributed Algorithms (WDAG)*, 1994.

[9] S. Bapat and A. Arora. Message efficient termination detection in wireless sensor networks. Technical Report OSU-CISRC-6/06-TR75, Department of Computer Science and Engineering, The Ohio State University, 2006.

[10] S. Bapat and A. Arora. Stabilizing Reconfiguration in Wireless Sensor Networks. In *SUTC '06: Proceedings of the IEEE International Conference on Sensor Networks, Ubiquitous and Trustworthy Computing, to appear*, 2006.

[11] S. Bapat, V. Kulathumani, and A. Arora. Analyzing the yield of ExScal, a large-scale wireless sensor network experiment. In *13th IEEE Intl. Conf. on Network Protocols (ICNP)*, pages 53–62, 2005.

[12] S. Bapat, V. Kulathumani, and A. Arora. Reliable estimation of influence fields for classification and tracking in unreliable sensor networks. *24th Symposium on Reliable Distributed Systems (SRDS)*, 2005.

[13] S. Bapat, W.Leal, T. Kwon, P. Wei, and A. Arora. Chowkidar: A health monitor for wireless sensor networks. Technical Report OSU-CISRC-6/06-TR76, Department of Computer Science and Engineering, The Ohio State University, 2006.

[14] P. Buonadonna, D. Gay, J. Hellerstein, W. Hong, and S. Madden. Task: Sensor network in a box. In *European Workshop on Sensor Networks (EWSN)*, 2005.

[15] Q. Cao, T. Abdelzaher, T. He, and J. Stankovic. Towards optimal sleep scheduling in sensor networks for rare event detection. In *Proceedings of IPSN*, 2005.

[16] A. Cerpa, J. Elson, D. Estrin, L. Girod, M. Hamilton, and J. Zhao. Habitat monitoring: Application driver for wireless communications technology, 2001.

[17] S. Chandrasekharan and S.Venkatesan. A message-optimal algorithm for distributed termination detection. In *Journal of Parallel and Distributed Computing, 8:245252*, 1990.

[18] K. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. In *ACM Trans. on Computer Systems, 8(3):326343*, 1985.

[19] Y. Chen and A. Liestman. Approximating minimum size weakly connected dominating sets for clustering mobile ad hoc networks. In *3rd ACM Intl. Symp. on Mobile Ad Hoc Networking and Computing (MobiHoc)*, 2002.

[20] I. Chlamtac and A. Farago. A new approach to the design and analysis of peer-to-peer mobile networks. In *Wireless Networks, 5:149-156*, 1999.

[21] Y. Choi, M. Gouda, H. Zhang, and A. Arora. Routing on a logical grid in sensor networks. Technical Report TR04-49, The University of Texas at Austin, 2004.

[22] Moteiv Corporation. TMote Sky Datasheet . `http://www.moteiv.com/products/docs/tmote-sky-datasheet.pdf`.

[23] A. Cournier, A. K. Datta, F. Petit, and V. Villain. Enabling snap-stabilization. In *Proceedings of ICDCS 2003*, 2003.

[24] A. Cournier, F. S. Devismes, and V. Villain. Snap-stabilizing PIF and useless computations. In *Proceedings of 12th International Conference on Parallel and Distributed Systems - Volume 1 (ICPADS'06)*, pages 39–48, 2006.

[25] E. Dijkstra. Self-stabilizing systems in spite of distributed control. *Comm. of the ACM*, 17(11):643–644, 1974.

[26] E. Dijkstra and C. Scholten. Termination detection for diffusing computations. In *Information Processing Letters 11(1):1-4*, 1980.

[27] A. Dunkels, B. Gronvall, and T. Voigt. Contiki - A Lightweight and Flexible Operating System for Tiny Networked Sensors. In *29th Annual IEEE Intl. Conf. on Local Computer Networks (LCN)*, pages 455–462, 2004.

[28] P. Dutta and J. Hui et al. Trio: enabling sustainable and scalable outdoor wireless sensor network deployments. In *5th Intl. Conf. on Information processing in Sensor Networks (IPSN)*, 2006.

[29] 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 *3rd Symp. on Information Processing in Sensor Networks (IPSN)*, 2005.

[30] Embedded Networks Laboratory, USC. Tutornet: A Tiered Wireless Sensor Network Testbed. `http://enl.usc.edu/projects/tutornet/index.html`.

[31] E. Ertin, A. Arora, R. Ramnath, V. Naik, and S. Bapat et al. Kansei: A Testbed for Sensing at Scale. In *5th Intl. Conf. on Information Processing in Sensor Networks (IPSN)*, pages 399–406, 2006.

[32] The Eclipse Foundation. Eclipse plugin architecture. `http://www.eclipse.org`.

[33] M. Gouda, Y. Choi, and A. Arora. Sentries and sleepers in sensor networks. In *OPODIS*, pages 384–399, 2004.

[34] C. Han, R. Kumar, R. Shea, E. Kohler, and M. Srivastava. A Dynamic Operating System for Sensor Nodes. In *3rd Intl. Conf. on Mobile systems, applications, and services (MobiSys)*, pages 163–176, 2005.

[35] R. Hao, D. Lee, J. Ma, and J. Yang. Fault management for networks with link-state routing protocols. In *IEEE/IFIP Network Operations and Management Symposium (NOMS)*, 2004.

[36] T. He, S. Krishnamurthy, J. Stankovic, T. Abdelzaher, L. Luo, R. Stoleru, T. Yan, L. Gu, J. Hui, and B. Krogh. Energy-efficient surveillance system using wireless sensor networks. In *MobiSys '04: Proceedings of the 2nd international conference on Mobile systems, applications, and services*, pages 270–283, 2004.

[37] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. In *Architectural Support for Programming Languages and Operating Systems*, pages 93–104, 2000.

[38] J. Hui and D. Culler. The dynamic behavior of a data dissemination protocol for network programming at scale. In *2nd Intl. Conf. on Embedded networked sensor systems (SenSys)*, pages 81–94, 2004.

[39] IETF. RFC 1157. `www.ietf.org/rfc/rfc1157.txt`.

[40] IETF. RFC 2328. `www.ietf.org/rfc/rfc2328.txt`.

[41] IETF. RFCs 791 and 793. `www.ietf.org/rfc`.

[42] Crossbow Technology Inc. MPR400/410/420 MICA2 Mote. `http://www.xbow.com/Products/productsdetails.aspx?sid=72`.

[43] Crossbow Technology Inc. Stargate Gateway (SPB400). `http://www.xbow.com/Products/productsdetails.aspx?sid=85`.

[44] S. Katz and K. Perry. Self-stabilizing extensions for message-passing systems. In *9th ACM Symp. on Principles of distributed computing (PODC)*, pages 91–101, 1990.

[45] Y. Kim, A. Arora, V. Kulathumani, U. Arumugam, and S. Kulkarni. On the effect of faults in vibration control of fairing structures. In *Fifth ASME Intl. Conf. on Multibody Systems, Nonlinear Dynamics and Controls (MSNDC)*, 2005.

[46] V. Kulathumani, P. Shankar, Y. Kim, A. Arora, and R. Yedavalli. Reliable control system design despite byzantine actuators. In *Fifth ASME Intl. Conf. on Multibody Systems, Nonlinear Dynamics and Controls (MSNDC)*, 2005.

[47] S. Kulkarni and A. Arora. Multitolerance in distributed reset. *Chicago Journal of Computer Science*, 4, 1998.

[48] S. Kulkarni and M. Arumugam. TDMA Service for Sensor Networks. In *24th Intl. Conf. on Distributed Computing Systems Workshops (ICDCSW)*, pages 604–609, 2004.

[49] S. Kulkarni and L. Wang. Mnp: Multihop network reprogramming service for sensor networks. In *25th International Conference on Distributed Computing Systems (ICDCS)*, 2005.

[50] S. Kumar, T. Lai, and J. Balogh. On k-coverage in a mostly sleeping sensor network. In *Proceedings of MobiComm*, pages 144–158, 2004.

[51] W. Leal, S. Bapat, T. Kwon, P. Wei, and A. Arora. Stabilizing health monitoring for wireless sensor networks. In *8th Intl. Symp. on Stabilization, Safety, and Security of Distributed Systems (SSS)*, 2006.

[52] W. Leal, S. Bapat, T. Kwon, P. Wei, and A. Arora. Stabilizing Health Monitoring for Wireless Sensor Networks. In *8th Intl Symp on Stabilization, Safety, and Security of Distributed Systems (SSS)*, page to appear, 2006.

[53] W. Leal, S. Bapat, T. Kwon, P. Wei, and A. Arora. Stabilizing health monitoring for wireless sensor networks. Technical Report OSU-CISRC-6/06-TR62, Department of Computer Science and Engineering, The Ohio State University, 2006.

[54] P. Levis and D. Culler. Mate: a tiny virtual machine for sensor networks. In *10th Intl. Conf. on Arch. Support for Prog. Lang. and Operating Systems (ASPLOS-X)*, 2002.

[55] P. Levis, N. Lee, M. Welsh, and D. Culler. TOSSIM: Accurate and Scalable Simulation of entire TinyOS applications. In *1st Intl. Conf. on Embedded networked sensor systems (SenSys)*, pages 126–137, 2003.

[56] P. Levis, N. Patel, D. Culler, and S. Shenker. Trickle: A Self-Regulating Algorithm for Code Propagation and Maintenance in Wireless Sensor Networks. In *1st Symp. on Networked Systems Design and Implementation (NSDI)*, 2004.

[57] S. Madden, M. Franklin, J. Hellerstein, and W. Hong. Tinydb: an acquisitional query processing system for sensor networks. *ACM Trans. Database Syst.*, 30(1):122–173, 2005.

[58] A. Mainwaring, J. Polastre, R. Szewczyk, D. Culler, and J. Anderson. Wireless sensor networks for habitat monitoring. In *Proceedings of WSNA'02*, Atlanta, GA, September 2002.

[59] F. Mattern. Global quiescence detection based on credit distribution and recovery. In *Information Processing Letters 30*, pages 195–200, 1989.

[60] V. Naik, A. Arora, S. Bapat, and M. Gouda. Whisper: Local secret maintenance in sensor networks. *IEEE Distributed Systems Online*, 4(9), 2003.

[61] V. Naik, A. Arora, P. Sinha, and H. Zhang. Sprinkler: A reliable and energy efficient data dissemination service for wireless embedded devices. In *26th IEEE Real-Time Systems Symposium (RTSS)*, 2005.

[62] V. Naik, S. Bapat, A. Arora, and M. Gouda. Whisper: Local secret maintenance in sensor networks. In *Principles of Dependable Systems (PDS)*, 2003.

[63] S. Parthasarathy and R. Gandhi. Fast distributed well connected dominating sets for ad hoc networks. Technical Report CS-TR-4559, Univ. of Maryland, 2004.

[64] J. Polastre, J. Hill, and D. Culler. Versatile low power media access for wireless sensor networks. In *2nd Intl. Conf. on Embedded networked sensor systems (SenSys)*, pages 95–107, 2004.

[65] N. Ramanathan, K. Chang, R. Kapur, L. Girod, E. Kohler, and D. Estrin. Sympathy for the sensor network debugger. In *SenSys '05: Proceedings of the 3rd international conference on Embedded networked sensor systems*, pages 255–267, 2005.

[66] D. Raychaudhuri and I. Seskar et al. Overview of the ORBIT Radio Grid Testbed for Evaluation of Next-Generation Wireless Network Protocols. In *IEEE Wireless Communications and Networking Conference (WCNC)*, 2005.

[67] L. Ruiz, I. Siqueira, L. e Oliveira, H. Wong, M. Nogueira, and A. Loureiro. Fault management in event-driven wireless sensor networks. In *MSWiM '04: Proceedings of the 7th ACM international symposium on Modeling, analysis and simulation of wireless and mobile systems*, pages 149–156, 2004.

[68] G. Simon, M. Maroti, A. Ledeczi, G. Balogh, B. Kusy, A. Nadas, G. Pap, J. Sallai, and K. Frampton. Sensor network-based countersniper system. In *Proceedings of ACM SenSys*, 2004.

[69] J. Staddon, D. Balfanz, and G. Durfee. Efficient tracing of failed nodes in sensor networks. In *WSNA '02: Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications*, pages 122–130, 2002.

[70] T. Stathopoulos, J. Heidemann, and D. Estrin. A remote code update mechanism for wireless sensor networks. Technical Report CENS-TR-30, UCLA, 2003.

[71] R. Szewczyk, J. Polastre, A. Mainwaring, and D. Culler. Lessons from a sensor network expedition. In *Proceedings of EWSN'04*, January 2004.

[72] G. Tolle and D. Culler. Design of an application-cooperative management system for wireless sensor networks. In *Proceedings of the EWSN*, 2004.

[73] International Telecommunication Union. Telecommunications Management Network. `http://www.itu.int/rec/T-REC-M/en`.

[74] The Ohio State University. Chowkidar Node Status Webpage. `http://exscal.nullcode.org/kansei/chowkidar/nodestatus.php`.

[75] G. Werner-Allen, P. Swieskowski, and M. Welsh. MoteLab: A Wireless Sensor Network Testbed. In *4th Intl. Conf. on Information Processing in Sensor Networks (IPSN)*, 2005.

[76] N. Xu, S. Rangwala, K. Chintalapudi, D. Ganesan, A. Broad, R. Govindan, and D. Estrin. A wireless sensor network for structural monitoring. In *Proceedings of SenSys*, pages 13–24. ACM Press, 2004.

[77] Y. Yao and J. Gehrke. The cougar approach to in-network query processing in sensor networks. *SIGMOD Rec.*, 31(3):9–18, 2002.

[78] H. Zhang, A. Arora, Y. Choi, and M. Gouda. Reliable bursty convergecast in wireless sensor networks. In *6th ACM Intl. Symp. on Mobile ad hoc networking and computing (MobiHoc)*, pages 266–276, 2005.

[79] H. Zhang, A. Arora, and P. Sinha. Learn on the fly: data-driven link estimation and routing in sensor network backbones. In *25th IEEE International Conference on Computer Communications (INFOCOM)*, 2006.

[80] J. Zhao and R. Govindan. Understanding packet delivery performance in dense wireless sensor networks. In *1st Intl. Conf. on Embedded networked sensor systems*, 2003.