Detection and Repair of Software Errors in Hierarchical Sensor Networks

Douglas Herbert*, Yung-Hsiang Lu*, Saurabh Bagchi* and Zhiyuan Li†
*Center for Wireless Systems and Applications
*School of Electrical and Computer Engineering, †Department of Computer Science
Purdue University, West Lafayette, Indiana 47907
{herbertd, yunglu, sbagchi}@purdue.edu, li@cs.purdue.edu

Abstract— **Sensor networks are being increasingly deployed for collecting critical data in various applications. Once deployed, a sensor network may experience faults at the individual node level or at an aggregate network level due to design errors in the protocol, implementation errors, or deployment conditions that are significantly different from the target environment. In many applications, the deployed system may fail to collect data in an accurate, complete, and timely manner due to such errors. If the network produces incorrect data, the resulting decisions on the data may be incorrect, and negatively impact the application. Hence, it is important to detect and diagnose these faults through run-time observation. Existing technologies face difficulty with wireless sensor networks due to the large scale of the networks, the resource constraints of bandwidth and energy on the sensing nodes, and the unreliability of the observation channels for recording the behavior. This paper presents a semi-automatic approach named H-SEND (Hierarchical SEnsor Network Debugging) to observe the health of a sensor network and to remotely repair errors by reprogramming through the wireless network. In H-SEND, a programmer specifies correctness properties of the protocol ("invariants"). These invariants are associated with conditions (the "observed variables") of individual nodes or the network. The compiler automatically inserts checking code to ensure that the observed variables satisfy the invariants. The checking can be done locally or remotely, depending on the nature of the invariant. In the latter case, messages are generated automatically. If an error is detected at run-time, the logs of the observed variables are examined to analyze and correct the error. After errors are corrected, new programs or patches can be uploaded to the nodes through the wireless network. We construct a prototype to demonstrate the benefit of run-time detection and correction.**

I. Introduction

Sensor networks enable continuous data collection or rare event detection in large, hazardous or remote areas. The data being collected can be critical. Detecting forest fire or tracking tank movement are two examples from civilian and military domains. Sensor network protocols are distributed protocols designed to be scalable in terms of the number of nodes and the sensor field sizes. Distributed protocols are widely recognized as being difficult to design [1]. Sensor network protocols are even harder due to the additional constraints and requirements: scarce resources, large scale, intermittent connectivity, event-driven environment, heterogeneous nodes in the network, and likelihood of failures. Even with a correct design, errors may still occur in implementation. Sensor networks present unique challenges because of the lack of sophisticated debugging tools and the difficulty of testing after deployment. Even after extensive testing, errors may still occur due to environment conditions, such as high temperatures. Implementations that have been certified correct in a developer's test environment may be deployed in a condition drastically different. While this is true of many systems, this is especially true with sensor networks as they are *in situ* in physical environments that may be changing over the period of deployment. A "fault" is defined as the underlying defect in the software or the hardware (for example a bug). If the fault is exercised (i.e. the line of the program is executed), the fault becomes an "error". If the error causes some manifestation that makes the program behavior differ from the specification, a "failure" has occurred.

Run-time techniques are required to detect errors in order to maintain high-fidelity data in the presence of possible errors from design, implementation, or a hostile environment. The H-SEND approach observes node conditions and network traffic to detect symptoms of errors. Earlier work for run-time observation in wired networks [2], [3], [4] does not directly apply to sensor networks. The detection algorithms may execute at a location far away from nodes where data is collected. For example, the network's base station may use computationally intensive algorithms to detect whether a sensor malfunctions by comparing one sensor's data with the data from surrounding sensors. Sensor networks are resource-limited; hence, it is essential to minimize the overhead in observation and detection. There are three types of overhead: storage, computation, and network. Additional storage is needed because the program size increases after inserting additional checking code. Observed variables also require storage space. Executing the checking code incur computation overhead. Finally, some detection must be conducted by aggregating information from multiple nodes; this creates additional network traffic. H-SEND differs from existing work in that it is specialized for large scale sensor networks. H-SEND has four key features:

(a) During program development, a programmer can specify important properties as "invariants" that should never be violated in the network's operation. An invariant may be associated with a particular execution state and is checked only during this state.
(b) When the program is compiled, the code for checking invariants is automatically inserted. As ex-

Proceedings of the IEEE International Conference on Sensor Networks, Ubiquitous, and Trustworthy Computing (SUTC'06)
0-7695-2553-9/06 $20.00 © 2006 IEEE

plained earlier, an invariant may be checked locally or remotely. Consequently, the compiler may also generate code to send messages to a remote location to detect errors that cannot be determined by a single sensor node.

(c) After deployment, the inserted code is used to detect abnormal behavior of the network. If an anomaly is detected, several actions may be triggered, such as increasing logging details or reporting errors to the base station. The errors will be analyzed at the base station by a human programmer to determine the faulty nodes and create fixes to the problem.

(d) After a failure is detected, a new program is uploaded to the relevant nodes through multi-hop wireless reprogramming. The mechanism disseminates the code update to the required nodes without needing physical access to the nodes in a scalable fashion.

When implementing this approach, special consideration of certain details is important.

(a) The solution should have small overhead in storage, computation, and network. H-SEND has very small overhead; we present the analysis of the overhead in Section IV-D.

(b) The solution should not add substantial burden to programmers. H-SEND assists programmers by (semi-)automatically determining where to insert invariant checking code and when to send messages that include observed variables.

(c) In some sensor networks, sensing activities involve multiple nodes working together. A desirable solution needs to handle such an environments for error detection.

(d) Sensor networks frequently include heterogeneous nodes and are organized as hierarchies. Error detection must be able to operate in these configurations.

H-SEND addresses all four important issues as follows. (a) It checks invariants through a hierarchy. H-SEND does not send all observed variables to a central location for detection. Instead, invariants are checked at the closest nodes where the requisite information is available. (b) H-SEND uses a compiler to determine the locations to insert code to check invariants and send observed information. A programmer only needs to specify the invariants and the variables to be observed. Thus, the programmer's burden is minimal. (c) H-SEND collects information from multiple nodes and invariants are verified at the collection point. (d) H-SEND naturally handles heterogeneity by allowing different nodes to check different types of invariants and also by performing remote checking when observed information is aggregated.

We present a prototype to demonstrate H-SEND through a data gathering protocol in a hierarchical configuration with a leader election algorithm. This is a hierarchy of three levels — sensing nodes, cluster head (one for each cluster), and the base station. Leader election serves as a fundamental building block and involves many exemplary invariants. Some invariants are local to a node but others are collective to a cluster or the entire network. We choose a representative leader election protocol called LEACH (Low-Energy Adaptive Clustering Hierarchy) [5], [6]. LEACH assigns cluster heads in a "near round-robin manner" to evenly distribute energy drain. A set of invariants is inserted into the application code. We use simulations to measure the overhead of the augmented code in our approach.

II. Related Work

The main contribution of this paper is a framework for the detection of software errors in distributed sensor networks. Our work is built upon the progress in recent years on sensor network prototyping, error detection, and recovery. This paper presents a new approach for efficient, prompt, and accurate detection of errors in sensor networks without specialized nodes. H-SEND inserts code at appropriate granularity to detect erroneous behavior and verifies the invariants in a resource-conserving manner.

A. Distributed Algorithms for Organization

Sensor networks are distributed systems. Many distributed algorithms have been studied [7]. Sensor networks have stringent resource constraints, including energy, storage, and computation capability. To conserve energy, some routing protocols use hierarchies among sensor nodes [8], [9]. Sensor nodes are divided into clusters and a special node in each cluster relays messages between clusters. This special node, called the cluster head, can be chosen in several ways. If sensor nodes are heterogeneous, the nodes that have more resources (battery capacity, faster processor, long-range antenna, etc) are selected as cluster heads. If all nodes are the same, they take turns playing the role of a leader through a leader election protocol [10], [11], [12].

B. Error Detection and Recovery

Error detection in sensor networks has been studied by many researchers. The predominant technique is local observation whereby nodes oversee traffic passing through the neighbor nodes [13], [14], [15], [16], [17]. Existing work does not separate the network into a payload and an observation system. Each node can potentially play a role in both systems. Previous work uses local observation to build trust relationships among nodes [16], [14], detect attacks [15], [17], or discover routes with certain properties, such as a node becoming disconnected [13]. In our previous work [18], we analyze the capabilities and the limitations of local observation as a primitive in sensor networks. The paper also presents a method to enable neighbor observation in resource-constrained environments and to lay out the fundamental structures and the state to be maintained at each node.

Local observation detects deviations from correct behavior at a local level. Correlation-based detection systems have been proposed for wired networks [19], [20], [21], without considering the resource constraints. Some studies provide solutions for detecting problems in specific ad-hoc network protocols [22]. H-SEND is more flexible and efficient because it uses a hierarchical approach. If an invariant can be checked by an individual node, this node checks the invariant without sending any

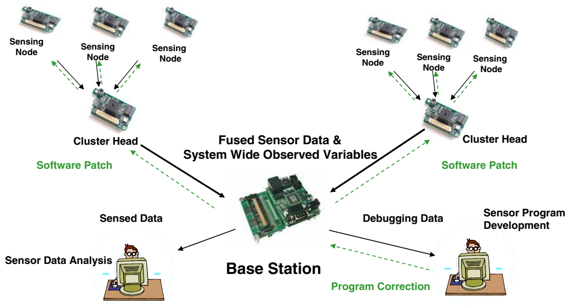


Fig. 1. Overview of the framework for error detection, propagation, diagnosis, and repair.

network messages. Some invariants are checked within a group of nodes (or *cluster*), or at the base station, where a global view of the whole system is available.

III. Techniques for Error Detection, Diagnosis, and Repair

A. Overview

Our system determines the health of sensor networks by detecting software errors, propagating the information to the base station, assisting a programmer to diagnose the errors, and then distributing correct software after the programmer fixes the errors. Our approach addresses five issues: "Who performs the observation?", "What is observed?", "How is a failure detected?", and "What actions are taken when an error is detected?"

1) Who performs observation?: There has been substantial work on observing run-time behavior in software and in hardware in the wired domain [2], [3], [4]. In most cases, the observed node and the observer form separate sub-systems. The observer has several advantageous characteristics: it may be a monolithic entity with perfect knowledge of the observed, it may be failure proof or may only fail in constrained ways, such as fail-silence, or it may not have any resource constraints.

Our target environment contains no central authority to perform observation. Rather, as much behavior as possible is observed locally. Hence, in H-SEND observation is performed in a distributed manner across all nodes. Nodes in the network play a dual role of observer and observee concurrently.

It is feasible to design an observation framework in stages of increasing complexity. The first option sends all incoming and outgoing events to the base station. This has the entire rule base for the network and verifies the events according to the rule base. The communication path as well as the base station are failure free. This design is not attractive due to large overhead. In the next stage, the observer system is placed in the vicinity of the observed system, such as a configuration where nearby nodes form a group, called a cluster. Each cluster has an observer performing the detection according to a rule base for the cluster. This assumes that the observers are specialized, failure free and the communication between the observed system and the observer is also failure free. To relieve the individual nodes in the observed system from the responsibility of pro-actively forwarding packets to the observer system, the observer passively listens to the messages on the wireless channels. However, this interferes with sleep protocols that are designed to put nodes in a low power mode when they are not actively transmitting or receiving packets.

In H-SEND, the end points of communication are the only possible entities that may verify a behavior manifested through the communication. A cluster head may learn through a message received from a node in its cluster that the node's network distance from the head is too far. The conditions to be observed are classified as (a) *local conditions* and (b) *remote conditions*. Local conditions are based on program variables, which are available at the same node. Remote conditions are based on variables collected from remote nodes (henceforth referred to as remote variables), corresponding to variables at the end point of the communication, either directly or transitively. Thus, if communication takes place from node A to B and from node B to C, a behavior of node A may be checked at node A (local condition checking) or at nodes B or C (remote condition checking). This points to an architecture for each node having a *local observation component* and a *global observation component*. The former is responsible for observing the internal state changes within the node in response to different events, the latter is responsible for observing behavior communicated through a network message.

2) What is observed and when?: Two types of invariants are checked: *local invariants* and *remote invariants*. The first is formed from variables resident on the same node (henceforth referred to as local variables, not to be confused with local variables within a function) only and the second from a mix of local and non-local variables. The local invariants can be checked at any point where the constituent variables are in scope, while the remote invariants can be checked when the set of network messages carrying all the non-local variables have been successfully received and the local variables are in scope.

There exists another dimension to classify invariants: *stateless invariants* and *stateful invariants*. For the invariants on a single node, stateless invariants are always true for the node, irrespective of the node's operation states. In contrast, stateful invariants are true only when the node is in a particular execution state. Naturally stateful invariants put a further constraint when the invariants can be verified. An example of stateless invariants is "A node belongs to at most one cluster at any moment." A stateful invariant is "A node can send a message to its cluster head only after a head has acknowledged the node's join message." A third dimension of invariants is *single node invariants* and *multi-node invariants*. The former includes invariants which involve variables from a single node. The latter combines variables from multiple nodes before the invariants can be verified. An example of single-node invariants is that a node must be within a threshold h hops away from its cluster head. An example of multi-node invariants is that the received signal strengths at a cluster head must not have a variance greater than another threshold.

3) How is a failure detected?: A failure is detected when one or multiple invariants are violated. The verification of a local invariant involves some computation without additional communication. The verification of a remote invariant involves additional communication. An optimization is to piggyback the variables required for remote checking with a payload message. Sensor networks are energy bound so nodes are often put to sleep for conserving energy. After a period of sleeping, nodes wake up, sense data, forward the data to the cluster head, and then return to sleep. Furthermore, some nodes may have only portions of hardware awake, such as their wireless receivers. Thus, sending debug information separately can be costly in terms of energy. An alternative is to piggy-back debugging information onto data messages that contain sensed data. This reduces the cost of communication — the fixed cost is amortized. Additionally, this removes interference with any existing node sleep-awake protocol. However, this implies that the error can be detected only when a data message is generated. Such delay, fortunately, is bounded and an analysis is presented in Section IV-D.

4) What actions are taken when an error is detected?: Errors can be classified into multiple degrees of severity. The most severe errors, once detected, will be sent to the base station through the cluster heads immediately. Less severe errors are sent to cluster heads for future diagnosis. The least severe errors may be stored in a local buffer of the node and sent to the cluster head or the base station upon request. In the current design of H-SEND, the errors are examined by a human programmer to diagnose the cause. In the future, diagnosis may be be partially automated, for example, through a diagnosis system for distributed systems suggested in [23].

B. Example

In this section, we describe how to write invariants, specify observed variables, and detect errors in H-SEND. We explore two widely used distributed protocols as examples. The first is *cluster formation* and the second is *cluster head election*. Table I shows the messages among the nodes. Table II are the invariants used for the two protocols. Some invariants are detected at the node level ("N-level"); some others are at cluster level ("C-level"). Some of the invariants are single node and some are multi-node.

Once an error is detected and diagnosed, repair actions are taken. The repair actions are of two types - uploading a new version of the program or changing the parameters of the currently executing program, thereby making it execute correctly for the deployed environment. A new program can be uploaded onto the node through the wireless network. Several protocols exist to provide remote code uploading, for example [24] [25]. Currently this process is not automated in H-SEND.

Message	Function
M1: Election	Initiate the election process for a CH
M2: Data	Send sensed data from a node to a CH
M3: Aggregate Data	Aggregate data in a CH and send to base station
M4: I'm a new CH	Inform the nodes that the sender is a new CH
M5: I'm a CH	Send periodic "keep-alive" to nodes in the cluster
M6: My CH is unavailable	Realize my CH is unreachable and send to the base station
M7: Relieve CH	Inform the other nodes that the CH intends to relinquish its role due to, for example, impending energy exhaustion

TABLE I

MESSAGES USED FOR CLUSTER FORMATION AND CLUSTER HEAD (CH) ELECTION

Correct Behavior	Error Detection
Cluster formation: A node is no more than α hops away from a cluster head.	M2 to the CH has hop count in the header and is $>$ (C-level, α. single node detection)
Cluster head election: There should be a single CH at any point in time.	M5 from more than one node. (C-level, multi-node detection) M5 from multiple CH incoming into a node. (N-level, single node detection)
A cluster head wishing to relinquish it's role should be able to do this within β time units.	M7 from the CH followed by M4 coming out the same node after β time units. (C-level, single node detection)
A cluster head election should not happen more often than once every γ time units.	M1 generated more often than once every γ time units in the entire cluster. (C-level, multi-node detection)

TABLE II

INVARIANTS AND DETECTION METHODS FOR CLUSTERING AND ELECTION PROTOCOLS

C. Automatic Insertion of Code for Invariant Verification

This section explains how insertion can be automated. The programmer specifies invariants as predicates defined over the observed program variables. In this paper, we use a C like language to explain such predicates. The underlying principles, however, also apply to other commonly-used programming languages.

An invariant may be specified for a function, a set of functions, a block of statements, or a single statement only. If an invariant is specified for a single statement, then the specification is placed immediately after that statement. For example, immediately after the statement `a = message_hops;`, the programmer may add a invariant `/*( a < MAX_HOPS )*/`. It requires the inequality to hold immediately after this particular assignment. If an invariant is specified for an entire function, then the specification can be placed at the beginning of the function body. In this case, `a < MAX_HOPS`, should hold throughout the entire function. If an invariant must be satisfied no matter what function is being executed, then the specification must be placed either at the beginning of a program module (a source file in C) or in a specification file that is accessible to the invariant insertion tool. An invariant specification may contain global variables, local variables, or both. In the latter two cases, the specification may contain the local variables from a single function or multiple functions. We use a *scope modifier* to clarify the function which contains a local variable. For example, the notation $f :: var$ refers to the variable var declared in function f. An invariant applies to all sensor nodes unless a *node modifier* is present. If the node modifier is present, then the invariant applies to only a subset of nodes. For example, $n\_ID :: f :: nhops < 3$ indicates that for the sensor n_ID the local variable *nhops* in function f should be less than 3.

The "forall" quantifier $\forall$ can be applied to functions. The entire set of functions is denoted by F. The predicate $\forall f \in \{f_1, f_2, \ldots, f_n\} P(f)$ states that the predicate P must be true on all sensor nodes whenever any of the listed functions, f_1, f_2, $\ldots$, f_n, are executed. For example, $\forall f \in \{f_1, f_2, \ldots, f_n\} f :: current\_head = f :: m.sender$ indicates that when executing any of the listed functions, the local variable `current_head` should equal to the `sender` field in the message m. This means that a node should expect to receive the message m from the current cluster head only. Receiving m from any other node indicates an error. When giving examples of specifications, we identify the most recently received message by the subscript *in* such as in m_{in}, and the most recently sent message by the subscript *out*. We distinguish several types of messages, including the messages M1 through M7 which are listed in Table I. The $\forall$ and "exist" $\exists$ quantifiers can be applied to both messages and node IDs. For example, $\forall m_{in} \mid m.type =$ M5 reads "For all received messages of the M5 type". We denote the entire set of sensor nodes by N and the entire set of messages by M.

The invariants listed in Table II can be specified in the program using the format shown in Table III. For example, to make sure that a node is no more than α hops away from a cluster head, the programmer encodes the first error-detection rule listed in Table II and inserts the following specification in the C like program: /*($\forall\ n \mid n.ID == current\_head \ \ m_{in}.hops \leq \alpha$)*/. We assume that the variable m_{in} stores a message newly received by a current cluster head (whose node ID matches the local variable *current_head*). The message has been propagated through a number of network hops which is recorded in the field *hops* in the message. The specification states that the number of hops should not exceed the value α. The compiler-based insertion tool analyzes the invariant specifications and converts them into executable program statements. The previous example can be converted to the following C statements: `if ((ID == current_head) && !(`$m_{in}.hops <= \alpha$`)) { /* error handing */}`. This approach uses the compiler to determine which pieces of data are available locally (*current_head* in this example), and which pieces of data need to be obtained from other nodes (*hops* in this example). Source code is then inserted on remote nodes to send the data required to evaluate invariants over the network.

When all the observed variables in an invariant are accessible from a single function f, then the verification statements for the invariant are inserted in f. The statements are inserted at every point where an observed variable is modified. However, if the observed variables are not accessible from a single function, then new global variables are created to shadow the local variables, allowing the predicate components to be evaluated. For example v_1 and v_2 are accessible from f only, and v_3 is accessible from g only, then the invariant $v_1 + v_2 < v_3$ is verified by writing $v_1 + v_2$ to a new global variable x, allowing the verification statement $x < v_3$ to be executed in g. If the observed variables are not accessible from a single sensor node, then additional messages must be generated to forward information to an aggregation point where the invariant can be checked, such as at a cluster head.

Correctness Behavior	**Invariant Specification**
Cluster formation:	
A node is no more than α hops from a cluster head.	$\forall n \mid n.ID = current\_head$ $m_{in}.hops \leq \alpha$
Cluster head election:	
A node belongs to one and only one cluster.	$\forall\ m_{in} \mid m_{in}.type =$ M5 $m_{in}.sender = current\_head$
A cluster head wishing to relinquish its role should be able to do this within β time units.	$\forall m_{out} \mid m_{out}.type =$M7 $\exists\ m_{in} \mid m_{in}.type =$ M4 $\wedge\ 0 < m_{in}.time -$ $m_{out}.time \leq \beta$
A cluster head election should not happen more often than once every γ time units.	$\forall (n1, n2) \in \mathsf{N} \times \mathsf{N}$ $\forall\ (m^{(1)}_{out}, m^{(2)}_{out})$ $\mid m^{(1)}_{out}.type =$ M1 $\wedge m^{(2)}_{out}.type =$ M1 $\mid m^{(1)}_{out}.time - m^{(2)}_{out}.time \mid > \gamma$
Cluster head election:	
If a node detects unavailability of a cluster head, a backup should take over within δ time units.	$\forall\ m_{out} \mid m_{out}.type =$M6 $\exists\ m_{in} \mid m_{in}.type =$ M4 $\wedge$ $0 < m_{in}.time - m_{out}.time \leq \delta$

TABLE III
EXAMPLES OF INVARIANT SPECIFICATIONS

IV. CASE-STUDY: DEBUGGING A DISTRIBUTED LEADER ELECTION PROTOCOL

A. LEACH

We implemented the LEACH (Low-Energy Adaptive Clustering Hierarchy) cluster based leader election protocol for wireless sensor networks [5], [6]. In LEACH, the nodes organize themselves into clusters, with one node acting as the head in each cluster. LEACH randomizes which node is selected as the head in order to evenly distribute the responsibility among nodes and to prevent draining the battery of one node too quickly. A cluster head compresses data (also called *data fusion*) before sending the data to the base station. LEACH assumes that all nodes are synchronized and divides election into rounds. Nodes can be added or removed at the beginning of each round. In each round, a node decides whether to become a head using the following probability. Suppose p is the desired percentage of cluster heads (5% is suggested in [5]). If a node has not been a head in the last $\frac{1}{p}$ rounds, the node chooses to become a head with probability $\frac{p}{1 - p \times (r \bmod \frac{1}{p})}$, where r is the current round. After $\frac{1}{p}$ rounds, all nodes are eligible to become cluster heads again. If a node decides to become a head, the node broadcasts a message to the other nodes. The other nodes joins a cluster whose leader's broadcast message

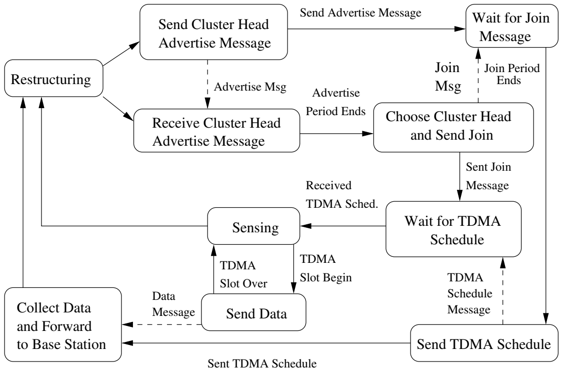


Fig. 2. State Diagram of the LEACH Protocol

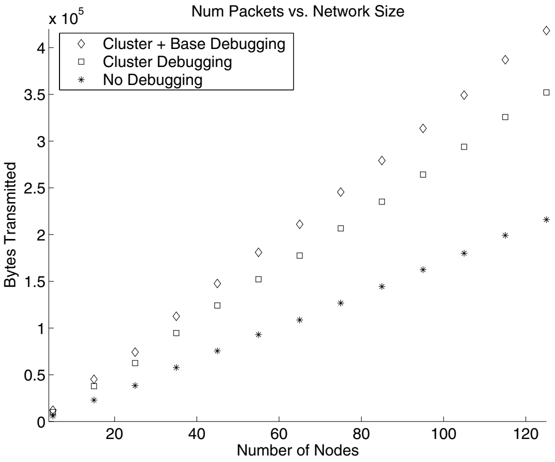


Fig. 3. Network Traffic vs. Network Size

has the greatest signal strength. In the case of a tie, a random cluster is chosen. LEACH is used in many other studies, such as [26], [27], [28] because LEACH is efficient, simple to implement, and resilient to node failures.

Figure 2 shows the states of the LEACH protocol. Each solid arrow indicates an event that causes a state change, and each dashed arrow indicates a communication message. Invariants can be easily created from this state diagram. If a node is in a certain state, and any event occurs for which the state diagram is not defined, an error has occurred. Possible invariants for the LEACH protocol include "only in the 'Wait for Join Message State' should a 'Join Message' be received" or "A node should only receive a 'TDMA schedule' in the 'wait for TDMA schedule state' ". The compiler can then insert code to check the health of a node or the network.

B. Sensor Programming Environment and Simulation

A typical hierarchical sensor network is shown in Figure 1. Once sensor network software is created by a developer, it may be uploaded to individual sensors by utilizing distributed propagation techniques over the radio link [29] as illustrated in Figure 1. Berkeley Mica Motes [30] are widely used sensor nodes for experiments. Mica nodes use TinyOS as the run-time environment. TinyOS provides an event-based simulator, TOSSIM, that can be used to simulate a network of varying node size [31]. TOSSIM compiles from the same source code as the Mica platform, and simulates communication with bit-level granularity. Our experiments use TOSSIM because it scales to large numbers of nodes easily. TOSSIM provides deterministic results so it is a better test bed in contrast to the non-deterministic results provided by real-life execution. Finally, TOSSIM allows us to separate instrumentation code from the actual code running on each node so we can measure the nodes' behavior without perturbing the network's normal operations.

C. Examples of Invariant Violation

At present, all invariants are manually inserted but insertion can be done by a compiler as explained in Section III-C. While developing the H-SEND framework, we originally intended to write "correct" code first and then intentionally inject errors later. However, we encountered unexpected behavior by the nodes and decided to insert invariants first to help us isolate the error (or errors). We observed that some nodes entered the "Cluster Head Advertise" state at wrong time. The error was a state-transition violation. An invariant required that "Restructuring State" be the previous state before the "Send Cluster Head Advertise Message" state. This is a binary example: there is only one correct previous state. If the previous state is incorrect, the invariant is violated. After this invariant was inserted, we discovered an error in our LEACH implementation. When the invariant was violated, an error was reported at the node level. Without this distributed debugging system, a simple error would have been difficult to diagnose. This shows that a binary invariant can be very helpful. An invariant can also include numeric quantities. For example, we can observe the signal strength received by each node in order to analyze the health of the network. An invariant can be written to ensure that the signal strength from a cluster head does not vary above 50%. If this invariant is violated, an error is reported. This report can assist the protocol designer to decide whether a more robust (and higher overhead) protocol should be chosen.

D. Analysis

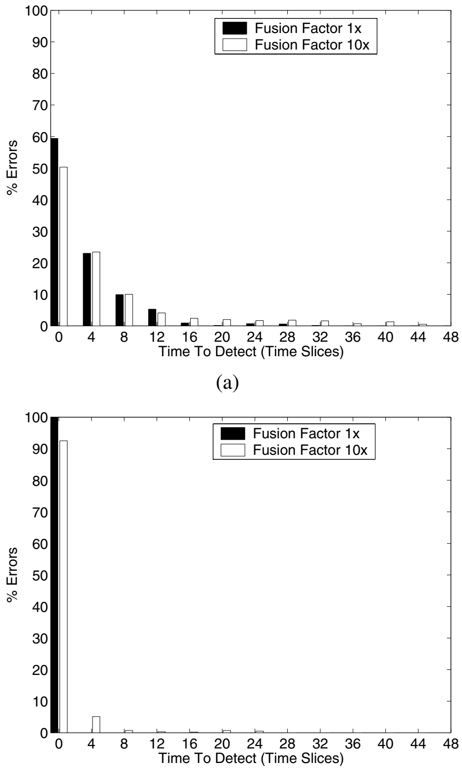
This section analyzes the overhead, time to detect errors, and code size.

1) Network Traffic Scaling: Since sensor nodes have limited energy, they should send as little information as possible to conserve energy. LEACH uses data fusion to reduce the amount of network traffic. We can analyze the network overhead of H-SEND as follows. Let m_c and m_b represent the size of a message sent from a node to its cluster head and the base station. Let f be the fusion factor. For example, f is 10 if the cluster head summarizes 10 samples and sends the average to the base station. Let δ be the additional amount of information sent by each node for failure detection.

The value of δ is zero if no information is transmitted for detecting failures. The total amount of data sent in the whole wireless network can be expressed as $\sum_{\forall x \in \text{ nodes}} \sum_{\substack{\text{messages}\\ \text{from } x}} (m_c + \frac{m_b}{f} + \delta)$. One goal of the H-SEND approach is to minimize the communication overhead. Suppose m_1 is the total amount of information transmitted in the network without any detection messages (i.e. $\delta = 0$). Let m_2 be the amount of information with detection messages. The overhead is defined as $\frac{m_2 - m_1}{m_1}$. In H-SEND, nodes only forward debugging data to cluster heads, and cluster heads only forward debugging data to the base station (i.e. upwards). No debugging data is sent back down to nodes from higher levels of the hierarchy. The rationale is that diagnosis needs to aggregate information only. Therefore, adding nodes results in a linear increase in network traffic. The case study presented here observed three variables at the cluster level, and six variables at the network level. Figure 3 shows that the traffic grows linearly for network sizes between 5 and 125 nodes. This figure shows three lines: (a) no error detection. This has the same amount of traffic as node-level detection. (b) cluster-level detection, and (c) cluster and base-level detection. The vertical axis shows the number of bytes transmitted. The actual amount depends the duration of the simulated network. Regardless of the duration, the ratio of $\frac{(b)}{(a)}$ and $\frac{(c)}{(a)}$ is approximately 1.64 and 1.95, respectively. In other words, the percentage of the network overhead is nearly a constant. Detecting errors as close to the source as possible allows H-SEND to reduce the amount of traffic sent over the network. The worst case scenario is to send all data to the base-station, and perform data-analysis at the base station. Through simulation, it was found that the H-SEND method resulted in a 7% message reduction size vs. sending all data needed to evaluate invariants to the base station.

2) Detection Time: To further reduce network traffic, observed detection data is piggy-backed onto data messages through the network as part of normal operation. This saves the fixed cost of communicating a new packet, such as the cost of the header bytes accompanying each packet (7 bytes out of 36 bytes for the Mica2 platform). Piggy-backing data adds a bounded latency to detection, as data is held at the node or cluster level until a data message is sent to the next level. Due to bounded detection time, all errors are reported, and there are no losses. If piggy backing is not used, error propagation delay is of the order of communication delay. If the error is delay sensitive, an additional strategy that can be used in addition to piggy-backing is generating an explicit control message if the delay goes above a threshold.

Piggy-backing error messages causes bounded delays. Detection time is defined as the time period between when a node detects an error, and the base station receives the message indicating an error. The worst-case detection time occurs when a node transmits data in the first transmit slot and detects an error in the very next slot, and must wait for all nodes in it's cluster to transmit (n-1 slots). It must then wait for the network to restructure, and then the same node must be assigned to the last transmit slot (n-1 slots). Analytically, we can define the worst case detection time as: 2×(Number of Transmit Slots-1)+Number of Slots to Restructure. This equation was confirmed by simulation. The LEACH protocol has 4 slots of administrative overhead. In [6] it is found that 5% of nodes acting as Cluster Head is ideal, yielding an average cluster size of 20 nodes, and therefore 20 time slots to broadcast results. Using these parameters, the worst-case detection time is 42 time slots. The data fusion factor will affect the detection time, as higher fusion factors result in fewer messages. As a result, detection time increases when the fusion factor increases. Figure 4 (a) shows a histogram of node-level detection time at fusion factors of 1 and 10. As the figure shows, most errors can be detected within 4 time slots. When the fusion factor is higher, the figure shows that detection time increases. Figure 4 (b) shows the detection time for cluster level error detection. The detection time is significantly less than at the node level, because cluster heads communicate with the base station much more often.



(a)

(b)

Fig. 4. Simulated Results for Detection Time. (a) Node-Level (b) Cluster-Level

3) Code Size: When implementing the LEACH protocol, all nodes except the base station must use the same binary image because all nodes can be cluster heads at

Components	ROM Size	RAM Size
LEACH without observation	11744	1466
LEACH with node level observation	12838	1470
LEACH with node, and cluster level observation	12906	1530
LEACH with node, cluster, and base station level observation	13040	1639

TABLE IV
CODE SIZE OF H-SEND IN BYTES

some point. The data reported in Table IV was collected with -O1 optimization, based on binary images for the Mica2 platform. The column for ROM indicates the code size written to the flash memory. The column for RAM indicates the memory requirement at run-time. The baseline includes the program that performs the basic sensor functionality and LEACH leader election. Adding node level observation increases the code size by 9% $(\frac{12838}{11744} - 1)$. Adding all levels of observation increases the code size by 11% $(\frac{13040}{11744} - 1)$. The increased RAM size comes from the additional bytes in the buffers for each packet.

V. CONCLUSION AND FUTURE WORK

This paper presents a hierarchical approach for detecting software errors for sensor networks. The detection is divided into multiple levels: node, cluster, and base station. Programmers specify the conditions (called invariants) that have to be satisfied. These invariants can be inserted by a compiler automatically. Our method is distributed and has low overhead in code size and network traffic. We use a leader election protocol as a case study but our method applies to a wide range of protocols.The H-SEND approach is designed to be tied into other existing technologies. For example, model-based mechanisms could be used in addition to programmer specified invariants to insert monitoring code. For future work, we would like to implement error-masking, and provide a tiered classification of errors based upon severity. We plan to implement automatic invariant insertion by a compiler and consider other types of errors, such as detecting malicious nodes injected into the network.

VI. ACKNOWLEDGMENTS

Doug Herbert is supported by the Tellabs Fellowship from Purdue's Center for Wireless Systems and Applications. This project is supported in part by the National Science Foundation grant CNS 0509394 and by Purdue Research Foundation. Any opinions, findings, and conclusions or recommendations in the projects are those of the authors and do not necessarily reflect the views of the sponsors.

REFERENCES

[1] G. Tel, "Topics in Distributed Algorithms," in *Chapter 3: Assertional Verification*. Cambridge University Press, 1991.

[2] M. Diaz, et al., "Observer-A Concept for Formal On-Line Validation of Distributed Systems," *IEEE Transactions on Software Engineering*, 1994.

[3] G. Khanna, et al., "Self Checking Network Protocols: A Monitor Based Approach," in *International Symposium on Reliable Distributed Systems*, 2004.

[4] M. Zulkernine, et al., "A Compositional Approach to Monitoring Distributed Systems," in *IEEE International Conference on Dependable Systems and Networks*, 2002.

[5] W. B. Heinzelman, et al., "An Application-Specific Protocol Architecture for Wireless Microsensor Networks," *IEEE Transactions on Wireless Communications*, 2002.

[6] W. R. Heinzelman, et al., "Energy-Efficient Communication Protocol for Wireless Microsensor Networks," in *Hawaii International Conference on System Sciences* 2000.

[7] N. A. Lynch, *Distributed Algorithms*. Morgan Kaufmann, 1996.

[8] S. Soro, et al., "Prolonging the Lifetime of Wireless Sensor Networks via Unequal Clustering," in *IEEE International Parallel and Distributed Processing*, 2005.

[9] M. Younis, et al., "Energy-Aware Routing in Cluster-based Sensor Networks," in *IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunications Systems*, 2002.

[10] S. Dolev, et al., "Uniform Dynamic Self-Stabilizing Leader Election," *IEEE Transactions on Parallel and Distributed Systems*, 1997.

[11] K. Nakano, et al., "A Survey on Leader Election Protocols for Radio Networks," in *International Symposium on Parallel Architectures, Algorithms and Networks*, 2002.

[12] G. Singh, "Leader Election in the Presence of Link Failures," *IEEE Transactions on Parallel and Distributed Systems*, 1996.

[13] A. Nasipuri, et al., "Performance of Multipath Routing for On-Demand Protocols in Mobile Ad Hoc Networks," *Mobile Networking Applications*, 2001.

[14] A. A. Pirzada, et al., "Establishing Trust in Pure Ad-Aoc Networks," in *Conference on Australasian computer science*, 2004.

[15] S. Marti, et al., "Mitigating Routing Misbehavior in Mobile Ad Hoc Networks," in *International Conference on Mobile computing and networking*, 2000.

[16] S. Buchegger, et al.,"Performance Analysis of the Confidant Protocol," in *ACM International Symposium on Mobile Ad Hoc Networking & Computing*, 2002.

[17] Y. an Huang, et al.,"A Cooperative Intrusion Detection System for Ad Hoc Networks," in *ACM workshop on Security of ad hoc and sensor networks*, 2003.

[18] I. Khalil, et al., "Liteworp: A Lightweight Countermeasure for the Wormhole Attack in Multihop Wireless Networks," in *International Conference on Dependable Systems and Networks*, 2005.

[19] P. Ning, et al., "Techniques and Tools for Analyzing Intrusion Alerts," *ACM Transactions on Information. Systems. Security*, 2004.

[20] F. Valeur, et al., "A Comprehensive Approach to Intrusion Detection Alert Correlation," *IEEE Transactions on Dependable and Secure Computing*, 2004.

[21] Y.-S. Wu, et al., "Collaborative Intrusion Detection System (CIDS): A Framework for Accurate and Efficient Ids," in *Computer Security Applications*, 2003.

[22] B. R. Smith, et al., "Securing Distance-Vector Routing Protocols," in *Symposium on Network and Distributed System Security*, 1997.

[23] G. Khanna, et al., "Automated Monitor Based Diagnosis in Distributed Systems," *Purdue University, School of ECE, TR# 05-13*, 2005.

[24] J. W. Hui, et al., "The Dynamic bBhavior of a Data Dissemination Protocol for Network programming at Scale," in *International Conference on Embedded Networked Sensor Systems*, 2004.

[25] S. S. Kulkarni, et al., "MNP: Multihop Network Reprogramming Service for Sensor Networks," in *IEEE International Conference on Distributed Computing Systems*, 2005.

[26] S. Lindsey, et al., "Data Gathering Algorithms in Sensor Networks Using Energy Metrics," *IEEE Transactions on Parallel and Distributed Systems*, 2002.

[27] R. Min, et al., "Low-Power Wireless Sensor Networks," in *International Conference on VLSI Design*, 2001.

[28] S. D. Muruganathan, et al., "A Centralized Energy-Efficient Routing Protocol for Wireless Sensor Networks," *IEEE Communications Magazine*, 2005.

[29] J. W. Hui, et al., "The Dynamic Behavior of a Data Dissemination Protocol for Network Programming at Scale," in *International Conference on Embedded Networked Sensor Systems*, 2004.

[30] J. L. Hill, et al., "Mica: A Wireless Platform for Deeply Embedded Networks," *IEEE Micro*, 2002.

[31] P. Levis, et al., "TOSSIM: Accurate and Scalable Simulation of Entire TinyOS Applications," in *International Conference on Embedded Networked Sensor Systems*, 2003.

0-7695-2553-9/06 $20.00 © 2006 IEEE