UCLA
Papers

Title
Towards a Debugging System for Sensor Networks

Permalink
https://escholarship.org/uc/item/20j234gg

Authors
Ramanathan, Nithya
Kohler, Eddie
Estrin, D

Publication Date
2005-05-05

Peer reviewed

eScholarship.org Powered by the California Digital Library
University of California

NEM570_online 4/29/05 8:28 PM Page 1

INTERNATIONAL JOURNAL OF NETWORK MANAGEMENT
Int. J. Network Mgmt 2005; **15**: 000–000
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/nem.570

Towards a debugging system for sensor networks

By Nithya Ramanathan*,†, Eddie Kohler and Deborah Estrin

Due to their resource constraints and tight physical coupling, sensor networks afford limited visibility into an application's behavior. As a result it is often difficult to debug issues that arise during development and deployment. Existing techniques for fault management focus on fault tolerance or detection; before we can detect anomalous behavior in sensor networks, we need first to identify what simple metrics can be used to infer system health and correct behavior. We propose metrics and events that enable system health inferences, and present a preliminary design of Sympathy, a debugging tool for pre- and post-deployment sensor networks. Sympathy will contain mechanisms for collecting system performance metrics with minimal memory overhead; mechanisms for recognizing application-defined events based on these metrics; and a system for collecting events in their spatiotemporal context. The Sympathy system will help programmers draw correlations between seemingly unrelated, distributed events, and produce graphs that highlight those correlations. As an example, we describe how we used a preliminary version of Sympathy to help debug a complex application, Tiny Diffusion. Copyright © 2005 John Wiley & Sons, Ltd.

Introduction

Sensor networks—networks of small, resource-constrained wireless devices embedded in a dynamic physical environment—have led to new algorithms, protocols, and operating system designs.[1,2] In sensor networks, interactions between sensor hardware, protocols, and environmental characteristics are hard to predict, making application design an iterative process between debugging and deployment.[3] For example, owing to flaky or variable link connectivity, post-deployment environments can present unexpected combinations of inputs, or stimulate untested control paths in routing and transport code, uncovering new bugs and necessitating different application designs. Furthermore, because sensor networks are at an early stage of development, debugging needs are more fundamental than for the Internet (e.g., rebooting a router often solves issues seen in the Internet, but rebooting a node rarely fixes a bug in a sensor network).

1 *Nithya Ramanathan* ■■

Eddie Kohler ■■

Deborah Estrin ■■

Department of Computer Science, UCLA, Los Angeles, Califormia, USA

**Correspondence to: N. Ramanathan, Department of Computer Science, UCLA, Los Angeles, CA 90025, USA.*
†E-mail: nithya@cs.ucla.edu

UNCORRECTED PROOF

Copyright © 2005 John Wiley & Sons, Ltd.

NEM570_online 4/29/05 8:28 PM Page 2

—Why Do We Need New Debugging Tools?—

We distinguish *debugging* from performance analysis and fault tolerance, and define debugging as *the process of root-causing a high-level failure*. This is an iterative process that begins by detecting a fault, isolating it, and then root-causing it—which often results in the identification of another fault, starting another iteration of this debugging loop. This process differs from traditional fault tolerance methods that aim to provide user-level transparency to system failures. Sensor network users do not primarily need fault tolerance; it is more important to know when and where a failure is occurring and have tools that will enhance visibility and aid in detecting, root-causing and fixing the fault causing the failure.

We distinguish debugging from performance analysis and fault tolerance, and define debugging as the process of root-causing a high-level failure.

Sensor networks are difficult to debug primarily owing to lack of visibility into the nodes and the dearth of effective debugging tools. Nodes' limited memory, communication, network and power resources prevent them from freely storing and transmitting debugging information, as this quickly depletes energy and network lifetime. As a result, once a sensor network is deployed, visibility into the network drops dramatically.

Envision deploying a dynamically taskable environmental monitoring sensor network, and not receiving queried data at the sink. Is this data loss caused by a mote tasking failure, failed sensors that are not returning samples, or packet loss along the path? Without more information, this failure is virtually impossible to track down and debug. With better tools and more observations, we may determine that data is not reaching the sink due to wildly varying link qualities at an individual or group of nodes. However, we cannot stop here as this still does not elucidate a fix: we must perform another iteration and determine why the link qualities are changing. This could lead to uncovering yet another fault which would result in another iteration in the debugging process. All these iterations must often take place in the field, where visibility is lowest, since environmental conditions may trigger bugs.

Sensor networks are not only hard to debug due to the lack of visibility; they contain bugs characteristic of both distributed, embedded *and* wireless systems, which are notoriously hard to detect and root-cause. Such bugs can be multicausal and timing-sensitive; often they are triggered by ephemeral events such as race conditions, asynchronous changes in distributed state, and interactions with the physical environment, making them hard to reproduce. Failures can also be caused by the *interactions between* nodes, regardless of whether independent nodes and modules are functioning correctly. A final hurdle in debugging is having too much information, which can be as ineffective as not enough information; this is especially true for a system that may scale to hundreds or thousands of nodes. These issues are not solved by removing power and memory constraints, and also occur during pre-deployment debugging (debugging that occurs during simulation and emulation).

These characteristics of sensor networks motivate the need for a debugging tool that can enhance visibility while preserving resources, non-intrusively observe the network, and provide contextual information for failures; in addition, the tool must transition seamlessly between both pre- and post-deployment environments. Furthermore, the tool must extract debugging information from a running system without introducing the probing effect (alteration of normal behavior due to instrumentation). Standard debugging approaches that only provide passive infrastructure, such as running a debugger or continual logging, fall short for sensor networks; this will be discussed further under 'Current State of the Art'.

—Debugging Goals—

An ideal debugging system may:

1. Detect a problem or unusual behavior by monitoring simple system metrics.
2. Aid in debugging the problem by collecting 'useful' information to provide context.
3. Proactively verify a hypothesis by injecting tests and go back to step 1 as needed.
4. Notify the user.

UNCORRECTED PROOF

Copyright © 2005 John Wiley & Sons, Ltd.

NEM570_online 4/29/05 8:28 PM Page 3

5. Fully debug the problem.
6. Finally, attempt to repair the problem.

This paper presents a preliminary design and evaluation of *Sympathy*, a debugging tool for pre- and post-deployment sensor networks that is designed to address the first three steps listed above. Our goal is to identify metrics that serve as accurate indicators of system health in order to determine what is needed for an autonomous system to monitor itself. Sympathy's primary goals are to enhance a user's confidence in the system and aid in debugging. Sympathy will consist of mechanisms for collecting system performance *metrics* with minimal memory overhead; mechanisms for recognizing application-defined *events* based on these metrics; and a system for collecting events in their *spatiotemporal context*. We define a *metric* as directly observable system state based on externally visible behavior, and an *event* as a notable change in state of a metric. The key, then, is to define what a *notable change* is, and thereby determine when events are important to note. The Sympathy system will help programmers draw correlations by collecting distributed, time-stamped events. Sympathy will impose minimum storage requirements on each mote, be non-intrusive with respect to the protocol and timing of the application, and monitor events within a sensor node as well as interactions between nodes.

Our current contribution is a preliminary design and implementation of a tool that can be used for pre-deployment debugging, an initial analysis of metrics useful for debugging, and the role of a debugging tool in the entire design process. Using Sympathy we have begun to distill the important metrics, events, and generic correlators that indicate system health and help find bugs quickly, and to transmit this data in ways that minimize energy consumption and probing effects. We found that by logging specific metrics and events, a system can perceive potential issues and enable quick discovery of their root cause.

Using Sympathy we have begun to distill the important metrics, events, and generic correlators that indicate system health and help find bugs quickly.

Sympathy's approach of correlating seemingly unrelated events has proven useful in detecting and debugging failures involving interactions between multiple nodes. To continue our previous example, imagine our sink stops getting data only from nodes A and B, and all data routed to the sink passes through node X. A system that reports that both node A *and* node B stopped sending data at approximately the same time, that node C *still* considers both nodes to be neighbors, and that link quality to node X from the previous hop suddenly dropped helps a user to isolate potential causes. In this scenario, the user can speculate that nodes A and B are probably still alive and that the dropped data is more likely due to the link to node X. All of these conclusions can be drawn simply by identifying correlated events based on nodes' neighbor lists. However, log files containing megabytes of unrelated data make it difficult to find and correlate events, especially those that are seemingly unrelated at first glance.

It is important to note that Sympathy entails user participation; it is not meant to be a generic bug-finder or 'black-box' technique. During pre-deployment, Sympathy is most effective once initial bugs have been fixed, and the harder-to-find coding and algorithmic bugs remain. Sympathy employs both traditional network management metrics and indicators of system health (e.g., route flapping and packet loss) in conjunction with sensor network-specific metrics and events (e.g., neighbor-list changes and next-hop selections), chosen as a result of sensor networks' unpredictable and highly varying links.

The Sympathy tool is new—we have used it so far only in simulation and emulation. Nevertheless, our experiences have been positive enough to validate the approach. Eventually, Sympathy will be part of a system that can aid in debugging sensor networks both pre- and post-deployment. Below we present a useful case study that demonstrates our current contributions by showing how Sympathy was used to debug a failure in Tiny Diffusion.[4]

Current State of the Art

Standard debugging and fault detection approaches are often based on the assumption that all nodes: (1) are accessible, either by a human system administrator or another node with reli-

UNCORRECTED PROOF

NEM570_online 4/29/05 8:28 PM Page 4

able connectivity; (2) have unlimited power; and (3) fail due to local causes, as opposed to interactions between several nodes. Furthermore, these techniques assume there is minimal cost associated with continually transmitting debug information to a centralized server. Such techniques fall short for sensor networks, which contain bugs characteristic of distributed, embedded, *and* wireless systems, and which need algorithms that minimize power, processing, and memory usage.

While some sensor network faults, such as node failures and bad route selections, are similar to those seen in common distributed architectures such as the Internet, the approaches to detecting these failures are necessarily different owing to the embedded, wireless, and resource-constrained platform. Moreover, common sensor network failures, such as data not arriving at the sink, nodes not receiving tasking or query packets from a sink, or even performance-based issues such as nodes consuming too much power, are not as prominent in Internet debugging; and Internet debugging may focus more on user latency, high availability, and application-level failures—issues not necessarily pertinent for sensor networks.

Current techniques for distributed systems can fall into the *debugging infrastructure/passive monitoring* and *fault detection* categories.

—Infrastructure/Passive Monitoring—

Current debugging infrastructure and techniques include the use of passive monitoring, tracing programs, simulation, visualization tools, and debug log files. While simulations are useful, clearly they are not a replacement for debugging on the actual hardware; it is impossible to simulate real-time network dynamics, dynamic environments, and numerous timing, MAC, and hardware-related details.

Visualization tools are helpful for real-time debugging when running on actual hardware. Figure 1 is a screen capture of Emview, a visual-

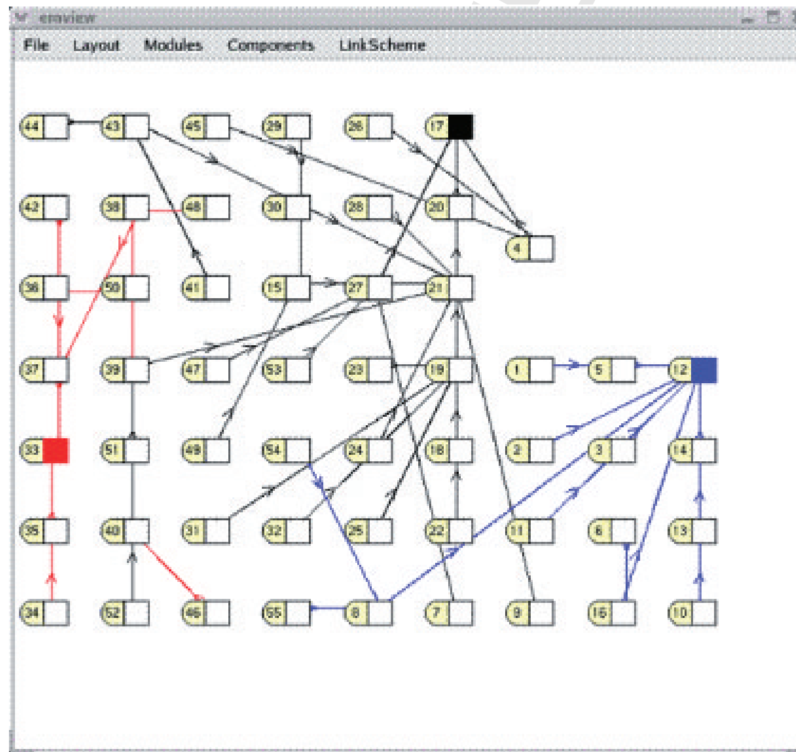

Figure 1. Screen capture of Emview, the visualizer available with Emstar.[3] This screen shows nodes, neighbor-list connectivity and link qualities associated with each link. The image changes in real time with changes in the network, capturing no historical context and performing no data analysis

NEM570_online 4/29/05 8:28 PM Page 5

izer tool designed for debugging distributed embedded applications. However, visualizers often don't highlight events that may indicate a failure, nor are they meant to capture historical context. For example, Emview-like visualizers show *either* link quality *or* neighbor-level connectivity; conflicts in these properties—a node that has no neighbors despite relatively high-quality links, for example—are difficult to see.

While log files can capture historical perspective and context, they contain excessive and unfiltered data that can obfuscate important events.

Tracing tools such as the Gnu Debugger (GDB) are highly utilized in order to understand real-time code dynamics, but such tools ignore the platform constraints of sensor network nodes, assuming that users can access a node through some sort of shell in order to launch and run the tool.

While traditional network management does not specifically address characteristics specific to sensor networks, such as low power and communication needs, there are many insights that we can apply.

The simple network management protocol (SNMP) implements a protocol to manage the exchange of network statistics between a centralized server, the network management system (NMS), and the agent nodes that record and transmit the metrics. The NMS queries agents and receives network statistics as well as asynchronous events from agents and can set variables within agents. Agents receive and store management data, and can asynchronously signal events to the NMS. SNMP focuses all processing at the NMS, expecting nodes to continually transmit all metrics back to this centralized server. This places an undue load on sensor network nodes that must minimize transmission in order to extend network lifetime.

Management by delegation[5] begins to address this issue of centralized processing of network statistics by moving some of the responsibilities of network management from a centralized server to distributed nodes. This responsibility transfer is done using mobile code: i.e., downloading scripts that can perform management tasks or even dynamic tasking to nodes. These scripts, or *delegation agents*, empower individual nodes to take action based on observed behavior, instead of consuming network bandwidth to convey metrics and commands between the centralized server and the nodes. The node is then able to monitor its own behavior, detect any problems, diagnose these issues, and even repair the problems.

Debugging tools designed for sensor networks are in their nascent stages. Although no common practices exist yet, Zhao *et al*.[6,7] make several recommendations for post-deployment debugging. Zhao *et al*.[6] present an algorithm to continually compute aggregates (sum, average, and count) of loss rates, energy levels, and packet counts to aid in debugging. Zhao *et al*.[7] argue that, while it is important to continuously gather node state in order to monitor the health of the network, it is not feasible to do so for each node due to energy limitations. The authors propose an energy-efficient algorithm based on in-network aggregation. The authors focused solely on the process of efficiently transmitting collected metrics and do not specify what metrics should be collected or how to process this data.

There are several TinyOS-based tools that support debugging. For example, SNMS is an infrastructure to enable system monitoring and fault detection.[8] This tool allows programmers to export counters and statistics and record application metrics. We have focused instead on determining the right metrics to export. With minimal modification, we could leverage the infrastructure provided by SNMS and replace Sympathy's logging mechanism and application interface.

While debugging tools such as SNMP, SNMS, and management by delegation provide monitoring infrastructure, we could find no work that examined or proposed specific metrics and data analysis techniques specifically applicable for sensor networks. Furthermore, Sympathy distinguishes itself from such passive data logging approaches by proactively collecting only potentially relevant events and their context, in order to highlight failures and aid in isolating their causes. In addition, Sympathy specifies generic low-level metrics and events that specifically examine *inter-node* dynamics in addition to internal node metrics.

—Fault Detection—

Another group of work related to Sympathy is those tools that utilize models for fault detection.

UNCORRECTED PROOF

NEM570_online 4/29/05 8:28 PM Page 6

These systems use models to infer internal state based on externally visible statistics. Szewczyk *et al*.[9] identify nodes that report sensor data exceeding a static threshold as being close to failure. This work utilizes a simple model to specify expected values and infer the node health when returned data does not match the model.

Model-based calibration techniques for sensor networks are similar to, though more refined than, the simple thresholding techniques used by Szewczyk *et al*. Feng *et al*. define calibration as 'the process of mapping raw sensor readings into corrected values'.[10] While calibration takes the process of fault detection one step further by attempting to fix the values, Feng *et al*. use their error model to interpret the sensor data and determine which sensors may be faulty.

Kiciman *et al*. collect low-level network metrics and use statistical analysis in order to identify application-level anomalous behavior.[11] This approach solely focuses on identifying faults, and is based on their stated assumption that identifying a fault consumes a majority of the time involved in handling a failure for Internet service providers. The authors postulate that once a fault has been detected, the fix involves simple techniques such as rebooting a node. Rebooting can work well for occasional transient faults on an essentially solid infrastructure. It can help for sensor networks as well (on networks that support remote reboot), but most sensor network infrastructure—network protocols, node operating systems, and so forth—is still under active development, and far from Internet-level stability and robustness.

Fox *et al*. extend this idea to suggest using statistical learning techniques to identify anomalous behavior.[12] Their first step in this process is to: 'Ensure the system is in a state in which it is mostly doing the right thing most of the time, according to simple and well-understood external indicators.' However, these 'simple and well-understood external indicators' do not exist for sensor networks. Before we can move on to apply increasingly complex tools to monitoring sensor network behavior, we must first identify such indicators.

Tools that analyze effects of configuration changes and predict anomalous behavior come close to the needs of sensor networks, but do not often take into account interactions between nodes, node constraints, and the unpredictable communication of wireless radios. These tools often assume perfect knowledge, expecting that nodes can reliably and continuously transmit system metrics to a server. This non-determinism in the environment is a key difference, even between pre-deployment and post-deployment debugging of sensor networks, as will be discussed below under 'Post-Deployment Architecture'.

Ruan and Pai's DeBox system[13] motivated the initial design of Sympathy and may be the most similar to it. DeBox suggests that exposing minimal internal state in real time to applications affords better performance analysis and tuning than passive profilers that provide information post facto. This transparency allows applications to get immediate feedback on the impacts of their actions on kernel performance and behavior. While Sympathy is not as concerned with performance, and focuses on fault detection and debugging, this approach of enhancing system visibility and transparency by exposing minimal internal state forms the basis of our work.

Architecture

We propose a generalizable architecture, called Sympathy, which continually monitors a network while an application is running, enhances visibility by identifying and collecting generic, low-level system metrics and events used to infer system health, and highlights unexpected correlations between these events in order to detect and help debug failures. The second-tier goal is to inform applications of events in case they can modify their behavior. The design of Sympathy should be considered preliminary; as we move to post-deployment debugging, architectural details may change.

Sympathy logs metrics as they change. These metrics are directly collected from the application, not independently calculated by Sympathy. We aim to define a minimal set of metrics sufficient for inferring system health and postulate that the list in Table 1 fulfills this criterion.

The metrics are then analyzed to detect *events*, which are essentially notable changes in the metric state. Once the network has established initial neighbor lists and routing configurations, any occurrence of the events listed in Table 2 are considered notable.

UNCORRECTED PROOF

Copyright © 2005 John Wiley & Sons, Ltd.

NEM570_online 4/29/05 8:28 PM Page 7

Metric name	Metric description
Neighbor lists	List of neighbors. Neighbors are identified by ID.
Link ingress/egress	Link quality from and to each neighbor. Link quality is calculated as a delivery rates between 0 (100% loss) and 100 (100% delivery).
Byte counts	The number of bytes transmitted and received by this node.
Next hop (Routing table)	The next hop chosen by this node.
Path loss (Routing table)	Whole-path loss rates calculated over an entire path from a node to the sink, using pair-wise link qualities at each hop. Each node calculates a separate path loss for each (next hop, sink) pair in its routing table, then chooses the next hop with the lowest path loss. Path loss is the inverse of link quality: lower values mean lower packet loss and thus provide better quality of delivery.

Table 1. Metrics gathered at each time step

Event name	Description	Metrics used to recognize event
Missing node	No node reports a node *n* as a neighbor. Logs *n*	All neighbor lists
Isolated node	Node *n* has no neighbors. Logs *n*	*n*'s neighbor list
Route change	*n*'s next hop changes at least once. Logs the previous and current next hop, the associated path loss for the top two choices for next hop, and the number of gradient messages received in this round	*n*'s routing table information
Neighbor list change	Node *n*2 joins or is dropped from *n*1's neighbor list. Logs *n*1, *n*2, and current and previous link qualities	*n*1's neighbor list
Link quality change	Node *n*2's link quality to *n*1 drops below a statically defined threshold. Logs *n*1, *n*2, and current and previous link qualities	*n*1's neighbor list

Table 2. Events that are detected based on the gathered metrics, and the metrics needed to discern them

Sympathy consists of two types of nodes: a *Sympathy-sink* and a *Sympathy-node*, shown in Figure 2. In general, the Sympathy-sink performs most of the event processing, and receives updated metrics from Sympathy-node processes using the IPC framework provided by Emstar. The Sympathy-sink contains components to record metric data from any node, identify events by analyzing metrics, and record event context.

Once an event has been detected, the Sympathy-sink updates its data structures and notifies clients interested in the specific event. These clients are passed event-specific data structures. A current client application exists that uses this information

NEM570_online 4/29/05 8:28 PM Page 8

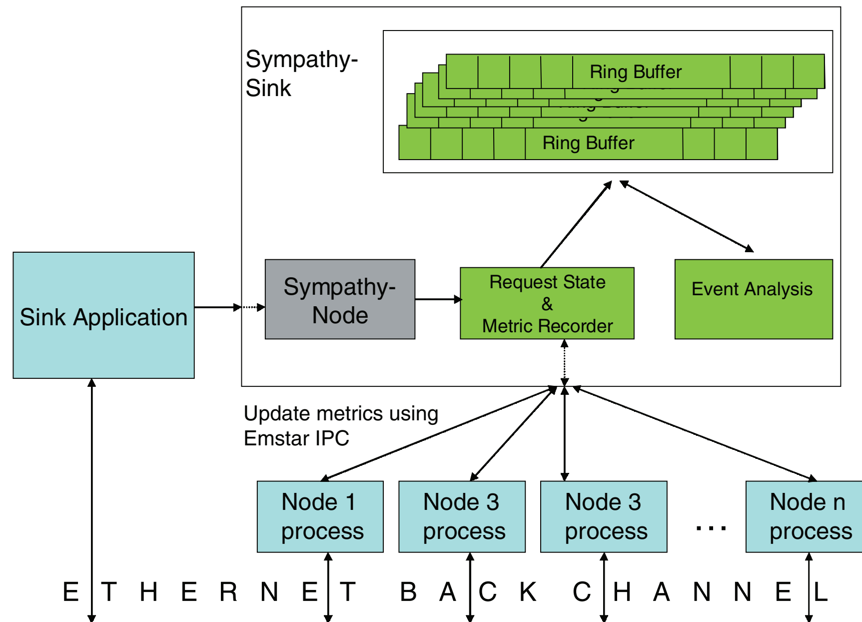

Figure 2. General architecture of a simulated system running with Sympathy. The Sympathy-sink receives updated metrics from Sympathy-node processes using the IPC framework provided by Emstar. The Sympathy-sink contains components to record metric data from any node, process metrics to identify events, and record event context

to determine the frequency of next-hop changes for each node.

—Implementation—

During application development the pre-deployment Sympathy implementation runs in *emulation mode* on the *ceiling array* as shown in Figure 2.

The ceiling array is a test-bed of mica2 motes connected over an Ethernet back channel to a Linux server running Emstar.[3,4] Emstar is an event-based application framework that facilitates heterogeneous networks of TinyOS-based motes and Linux-based micro-servers. Each node is run as a separate process, communicating using the IPC mechanisms provided by Emstar. Simulation is used in the traditional sense; nesC code is simulated using the EmTos[14] component of Emstar.

The ceiling array is employed for emulation mode which uses real mote radios for communication, but handles all processing on a centralized server running Emstar. The server communicates with each mote over the back channel in order to get and receive packets from the network. However, because all node processing occurs on the same server, state and debugging information between nodes is communicated using Emstar's IPC.

The Sympathy-sink is virtual and omniscient, using Emstar's IPC to continuously and reliably receive metrics from all Sympathy-nodes without impacting timing or performance, as shown in Figure 2. Interestingly, for identification of most events collected by Sympathy, nodes only need transmit their metrics to a local one-hop neighbor; a centralized node with global knowledge is only needed in order to identify the *missing-node* event. While the Sympathy-sink leverages the Emstar infrastructure to collect metrics from each node, Sympathy has no implicit dependence on Emstar and can be implemented independently.

Sympathy has no implicit dependence on Emstar and can be implemented independently.

Copyright © 2005 John Wiley & Sons, Ltd.

NEM570_online 4/29/05 8:28 PM Page 9

The Sympathy-sink analyzes the metrics transmitted from all Sympathy-nodes, and triggers on events. Upon triggering, the Sympathy-sink:

- provides *temporal context* by storing all metrics it has collected from the past 200 time units for the node causing the trigger;
- provides spatial context by storing all metrics it has collected from the past 200 time units for the nodes neighboring the node where the event was detected;
- aids in correlating seemingly unrelated events by printing event and context information to a log file; and calls applications interested in the event.

For example, route-flapping (frequent changes in the routing table, used to detect badly configured routers or system instability) can be identified by examining the next-hop metric collected at each node. Currently a route-flapping event is defined as a change in next hop. Once this event is identified, Sympathy logs the event and its spatiotemporal context. This information is logged to the same file so that temporally and spatially correlated events are easily discernible.

Post-Deployment Architecture

The post-deployment implementation of Sympathy would differ from the pre-deployment architecture in that nodes must consume valuable network bandwidth and power in order to transmit information to a Sympathy-sink. In post-deployment, no IPC is available to convey debugging information between nodes, and code is run directly on the motes, so the implementation must also curtail memory usage in order to conform to the mote platform. Since the metric collection and initial processing occur directly on a Sympathy-node, and metrics cannot be continually transmitted to the Sympathy-sink, only limited events and recent metric values would be stored at each Sympathy-node.

Owing to the sparser resources available during post-deployment, the Sympathy-sink will have incomplete knowledge about the state of each node due to flaky links and heavy transmission costs. As a result, post-deployment debugging relies more on inferences of system state based on externally observable metrics, and will not be as precise as the pre-deployment techniques discussed here.

Because of the power limitations that necessitate minimal communication, nodes must decide which events are most important to transmit to the Sympathy-sink. In addition, precisely defining which events and metrics are important, and when they should be transmitted, becomes even more critical.

The Sympathy-sink would have to run on a non-resource-constrained node—such as a Linux-based stargate—which can accommodate the additional storage and processing requirements required at the sink. Periodically, the Sympathy-sink could flood a request for nodes to send their event data and current metric state in order to ensure the health of the system.

Evaluation

To demonstrate Sympathy's potential as a debugging tool, we ran it with a nesC implementation of *Tiny Diffusion*,[4] a routing algorithm based on directed diffusion.[15] In Tiny Diffusion nodes periodically flood neighbor beacons (to calculate link quality), neighbor lists and associated link qualities (to identify asymmetric links), and *gradients* which carry a node's next hop and projected path loss (to determine a node's next hop). We ran Sympathy with Tiny Diffusion in simulation, using a 14-node network. Each simulation ran for 2 h at a time. Our goal was to determine why Tiny Diffusion had been experiencing loss rates an order of magnitude higher than expected in data delivery to the sink.

After the first run, using the events triggered in Sympathy, we saw nodes change their next-hop selection approximately every 170 s. Sympathy aided us over traditional debugging techniques by highlighting the frequent changes in next-hop selection and providing spatial correlation, which revealed that during each period on average 39% of nodes changed their next hop. While we would expect some churn in next-hop selection, the continuous flux appeared suspicious.

We then investigated the temporal context provided for each event by Sympathy: that is, the metrics and events that occurred close in time to the unusual changes in next hop. Surprisingly, we found that most nodes that changed their next hop

UNCORRECTED PROOF

Copyright © 2005 John Wiley & Sons, Ltd.

NEM570_online 4/29/05 8:28 PM Page 10

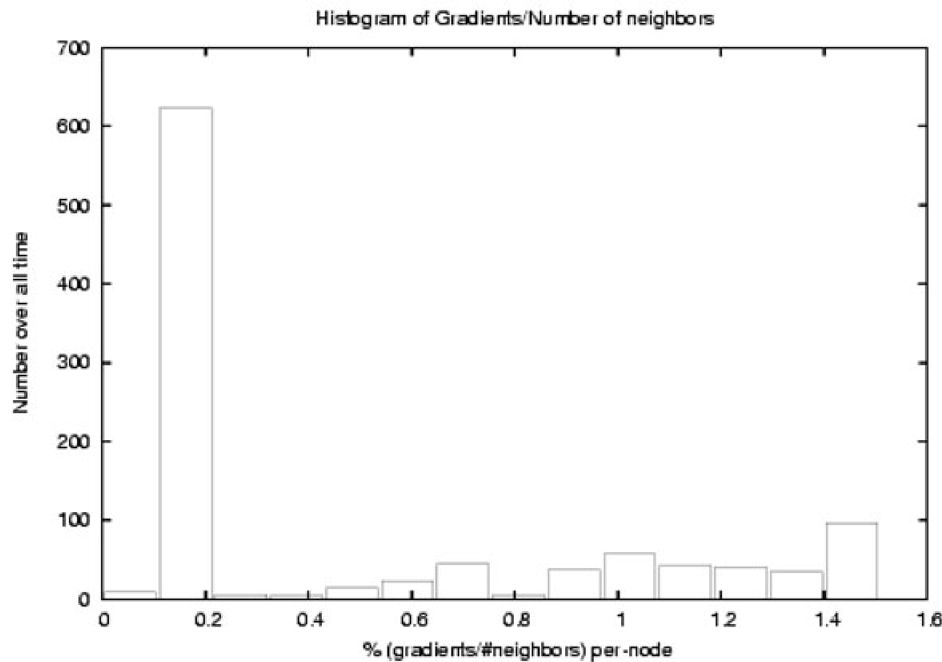

Figure 3. Histogram of number of gradients received by a node that changed its next hop, as a percentage of the number of neighbors in that node's neighbor list. Each node should receive roughly as many gradients as it has neighbors, but the graph shows that most nodes received gradients from only 10% of their neighbors (a minority of nodes may send multiple gradients, resulting in greater than 100%). The final bar represents nodes who heard at least one gradient, but had 0 neighbors recorded

did so because *they had received only one gradient message* and thus had only one choice for a next hop. Clearly, this was the cause for the frequent changes in next-hop selection. Furthermore, there was a high probability that nodes frequently selected high-loss paths, as they were given only one choice for next hop: had they received more than one gradient message, nodes could have chosen a better next hop with lower path loss. This in turn was a probable cause for the high loss rates observed at the sink.

To quantify our findings, we graphed the ratio of gradients received vs. number of neighbors. Figure 3 presents the results in a histogram: the vast majority of next-hop changes took place when the node received gradients from 10% or less of its neighbors. This is particularly strange because neighbor lists are recalculated each period from neighbor beacons that are flooded out immediately before the gradient messages. So, on an ideal, minimally varying, 0-loss link, a node should receive 100% of the gradient messages sent by the nodes on its neighbor list. Yet an order of magnitude fewer gradient messages than neighbor beacons were received.

We theorize that many nodes received such a small percentage of their intended gradient messages owing to collisions caused by synchronization of nodes' gradient floods. Code examination corroborated this theory, revealing that while jitter was added to the transmission of neighbor beacons, no jitter had been added to the transmission of gradient floods.

Sympathy's strength lies in its support for highlighting events and correlating them with metrics in their spatiotemporal context. This is an improvement over traditional debugging techniques in three ways: it facilitates discovery of correlations by associating context with a specific event; it provides event tracking, which involves maintaining state; and it determines which events are important to track (only a finite number of

UNCORRECTED

Copyright © 2005 John Wiley & Sons, Ltd.

NEM570_online 4/29/05 8:28 PM Page 11

events can be tracked). In addition to highlighting correlations, Sympathy avoids several iterations of debugging and rerunning that would otherwise be needed to capture and analyze metrics in order to find events.

However, Sympathy cannot be used in a vacuum, nor can it be used to find bugs automatically. We used our knowledge of Tiny Diffusion to dismiss extraneous correlations, and to add the second-best gradient to the final list of metrics collected. While the metrics currently collected by Sympathy are *not* application specific, ongoing work will include a comprehensive analysis of generic metrics, events, and correlators.

Future Work

Our goal is to identify generic metrics that are useful to collect for a broad class of applications in order to make inferences about system health and highlight other interesting configuration and performance-related properties. Currently we have identified several metrics that have proven useful for debugging Tiny Diffusion. While we began with a TinyOS and Emstar-based implementation running with Tiny Diffusion, Sympathy is by no means limited to this environment. Instead we are using this as a starting platform with which to develop our ideas. In the near future, we hope to generalize our initial metrics, develop better methods for identifying correlations, and include combining sensor data with system metrics and results from self-tests and injected probes in order to ensure expected behavior.

Once we determine metrics that can serve as accurate indicators of system health for a broad class of applications, nodes can also send back periodic maintenance reports in order to increase users' confidence in the system health. In addition, these metrics can be used to provide insight into performance and system characteristics as well as understanding—in real time—how a configuration change impacts functionality and performance.

We also plan to task motes and inject probes based on observed metrics. Ideally this can be done even at individual nodes; however, we will begin by implementing a centralized node (e.g. a Sympathy-sink) which receives metrics and based on its analysis decides it needs further information from a certain region. At this point, it could either inject probes or command nodes to perform self-tests.

We could also deploy third-party snoopers running on a non-resource-constrained, Linux-based micro-server. Strategically placed snoopers could shift power-heavy debugging, logging, and transmission operations off the low-power sensor nodes and lengthen overall network lifetime. Snoopers also do not interfere with timing and protocol issues, making them attractively modular.

Conclusion

Standard debugging methods, applications, and infrastructure do not directly apply to sensor networks, as most sensor nodes have extremely limited storage and energy capabilities. It is not a question of simply porting debugging tools like GDB over to a mote: debugging strategies and best practices must be reformulated to accommodate sensor networks' limited visibility and inter-nodal dynamics. A debugging solution that can seamlessly move between development and post-deployment debugging can also facilitate the natural sensor network design process.

In this paper we presented the preliminary design of Sympathy, a tool that enables the debugging of sensor networks during the development phase. It is based on a triggering system that identifies a priori events, provides spatiotemporal context to aid in isolating the source, and calls applications interested in an event. Using this tool, we demonstrate that it is possible to draw interesting conclusions based on collecting metrics, detecting seemingly unrelated events, and drawing even very simple correlations between them. Without Sympathy or a similar tool, it would have been difficult to quickly determine that Tiny Diffusion nodes were switching their next hops, and realize the periodicity of this behavior. Furthermore, it is not necessarily obvious that this behavior could have been correlated with the number of gradient messages received by each node. While Sympathy cannot yet be used for post-deployment debugging, it is already a useful stepping stone for analysis of event correlation methods, and for determining what metrics are most useful for health monitoring and fault detection in deployed systems.

UNCORRECTED PROOF

NEM570_online 4/29/05 8:28 PM Page 12

Acknowledgements

The authors thank Lewis Girod for his valuable input through the design and debugging process. They are also indebted to Thanassis Boulis and Sanjay Jha for their guidance and suggestions, which greatly improved the paper.

References

1. Szewczyk R, Polastre J, Mainwaring A. Lessons from a sensor network expedition. In *First European Workshop on Wireless Sensor Networks*, Berlin, Germany, January 2004.
2. Cerpa A, Elson J, Estrin D, Girod L, Hamilton M, Zhao J. Habitat monitoring: application driver for wireless communications technology. In *ACM SIGCOMM*, April 2001.
3. Girod L, Elson J, Cerpa A, Stathopoulos T, Ramanathan N, Estrin D. EmStar: a software environment for developing and deploying wireless sensor networks. In *Proceedings of the 2004 USENIX Technical Conference*, Boston, MA, 2004 (to appear).
4. Heidemann J, Silva F, Estrin D. Matching data dissemination algorithms to application requirements. In *Sensys*, Los Angeles, 2003.
5. Goldszmidt G, Yemini Y. Distributed management by delegating mobile agents. In *15th International Conference on Distributed Computing Systems*, June 1995.
6. Zhao J, Govindan R, Estrin D. Computing aggregates for monitoring wireless sensor networks. In *Proceedings of the IEEE ICC Workshop on Sensor Network Protocols and Applications*, Anchorage, AK, 2003.
7. Zhao J, Govindan R, Estrin D. Residual energy scans for monitoring wireless sensor networks. In *Proceedings of the IEEE Wireless Communications and Networking Conference*, Florida, 2002.
8. Tolle G, Culler D. Snms: application-cooperative management for wireless sensor networks. In *Proceedings of the Second ACM Conference on Embedded Networked Sensor Systems*, ACM, 2004.
9. Szewczyk R, Polastre J, Mainwaring A, Culler D. Lessons from a sensor network expedition. In *1st European Workshop on Wireless Sensor Networks*, January 2004.
10. Feng J, Megerian S, Potkonjak M. Model-based calibration for sensor networks. In *IEEE International Conference on Sensors*, October 2003.
11. Kiciman E, Fox A. Detecting application-level failures in component-based internet services. In *IEEE Transactions on Neural Networks* 2005; Spring.
12. Fox A, Kiciman E, Patterson D, Jordan M, Katz R. Combining statistical monitoring and predictable recovery for self-management. In *Proceedings of Workshop on Self-Managed Systems*, October 2004.
13. Ruan Y, Pai V. Making the 'box' transparent: system call performance as a first-class result. In *Proceedings of the 2004 USENIX Technical Conference*, Boston, MA, 2004 (to appear).
14. Girod L, Stathopoulos T, Ramanathan T, Estrin D. Tools for deployment and simulation of heterogeneous sensor networks. *Tech. Rep.*, CENS-TR-37, April 2004.
15. Intanagonwiwat C, Govindan R, Estrin D. Directed diffusion: a scalable and robust communication paradigm for sensor networks. In *Proceedings of the Sixth Annual International Conference on Mobile Computing and Networking*, Boston, MA. ACM Press, August 2000; 56–67. ■

If you wish to order reprints for this or any other articles in the *International Journal of Network Management*, please see the Special Reprint instructions inside the front cover.

UNCORRECTED PROOF

Copyright © 2005 John Wiley & Sons, Ltd.

AUTHOR QUERY FORM

Dear Author,

During the preparation of your manuscript for publication, the questions listed below have arisen. Please attend to these matters and return this form with your proof.

Many thanks for your assistance.

Query References	Query	Remarks
1.	*Author* Please provide brief biographies for each author.	
2.	*Author* Published yet?	
3	*Author* Page numbers?	
4	*Author* Published yet?	