

# Sympathy: A Debugging System for Sensor Networks

Nithya Ramanathan, Eddie Kohler, Lewis Girod, and Deborah Estrin

*UCLA Center for Embedded Network Sensing*

{*nithya, kohler, girod, destrin*}@cs.ucla.edu

## I. INTRODUCTION

Sensor networks—networks of small, resource-constrained wireless devices embedded in a dynamic physical environment—have led to new algorithm, protocol, and operating system designs [1], [2]. Interactions between sensor hardware, protocols, and environmental characteristics are impossible to predict, so sensor network application design is an iterative process between debugging and deployment [3].

Current debugging techniques fall short for systems which contain bugs characteristic of both distributed *and* embedded systems. Such bugs can be difficult to track because they are often multicausal, non-repeatable, timing-sensitive and have ephemeral triggers such as race conditions, decisions based on asynchronous changes in distributed state, or interactions with the physical environment. Furthermore, it is a challenge to extract debugging information from a running system without introducing the probing effect (alteration of normal behavior due to instrumentation) or draining excessive energy.

This paper presents a preliminary design and evaluation of *Sympathy*, a debugging tool for pre-deployment sensor networks and motivated by Ruan and Pai’s DeBox system [4]. *Sympathy* consists of mechanisms for collecting system performance *metrics* with minimal memory overhead; mechanisms for recognizing *events* based on these metrics; and a system for collecting events and their *spatio-temporal context*.

*Sympathy* introduces the idea of correlating seemingly unrelated events, and providing context for these events, in order to track down bugs and find their root causes. Using *Sympathy* we have begun to distill out the important metrics, events and generic correlators that help find bugs quickly, and to transmit this data in ways that minimize energy consumption and probing effects. This process is ongoing. Our current contribution, then, is a tool that can be used for pre-deployment debugging, and for analysis on the role of a debugging tool in the entire design process. 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*.

In related work, [5] and [6] address the data collection aspects of post-deployment debugging, but focus on the mechanism to *gather* statistics instead of their *content*. Our work is complementary, since *Sympathy* is so far mostly concerned with content: discovering the most useful metrics to collect. Simulations and visualization tools are also helpful, but do not capture historical context or aid in determining the cause

of a failure. While log files can provide context to a failure, they often contain excessive data which can obfuscate important events. *Sympathy* distinguishes itself from passive data logging approaches by proactively collecting and highlighting potentially relevant events and their context in order to aid in isolating their causes.

## II. ARCHITECTURE

*Sympathy*’s general architecture is as follows: *Sympathy* collects *metrics* from all nodes and watches the metrics for indications of *events*, which are metric changes that often indicate important changes in application state. On inferring an event, *Sympathy*:

- 1) Stores all metrics it has collected from the past 200 time units for the node causing the trigger, providing temporal context.
- 2) Stores all metrics it has collected from the past 200 time units for the nodes neighboring the node where the event was detected, providing spatial context.
- 3) Prints event and context information to a log file, which can aid in correlating events.
- 4) Calls applications interested in the event.

The version of *Sympathy* described here collects four metrics: neighbor lists, link quality, nodes’ top two choices for next hop, and associated next-hop path loss. It watches for two types of events based on these metrics, namely missing or isolated nodes and changes in route selection, neighbor lists, or link quality.

## III. EVALUATION

To demonstrate *Sympathy*’s potential as a debugging tool, we ran it with a nesC implementation of *tiny diffusion* [7], a routing algorithm based on directed diffusion [8]. 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 debugged this system pre-deployment, using simulations on a 14-node network that ran for two hours. 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 seconds. *Sympathy* aided over traditional debugging techniques by highlighting the frequent changes in next-hop selection and providing spatial correlation, which revealed that

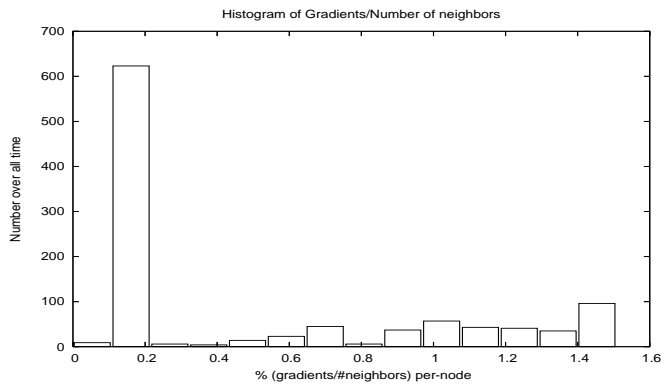


Fig. 1. Histogram of number of gradients received by a node that changed its next hop, as a percentage of the number of neighbors in that node’s neighbor list. Each node should receive roughly as many gradients as it has neighbors, but the graph shows that most nodes received gradients from only 10% 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

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 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 1 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 due 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 spatio-temporal 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 events can be tracked). In addition to highlighting correlations, Sympathy avoids several iterations of debugging and re-running 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 most of the metrics collected by Sympathy are *not* application-specific, ongoing work will include a comprehensive analysis of more generic metrics, events and correlators.

#### IV. CONCLUSION/FUTURE WORK

This work is one step on the road to a debugging tool that will cover both pre- and post-deployment debugging. Future work will focus on developing better methods for identifying significant correlations and porting the tool to enable post-deployment debugging.

Post-deployment debugging will rely more on inferences of system state based on externally observable metrics, such as messages, and will not be as precise as the pre-deployment techniques discussed here. We plan to deploy strategically placed Linux-based microsensors that could shift power-heavy debugging, logging and transmission operations off the low-power sensor nodes, while preserving timing. Interestingly, a third-party snooper observing a network running tiny-diffusion could collect all of the metrics utilized by Sympathy today, entirely avoiding extra broadcasts containing debugging information.

#### REFERENCES

- [1] R. Szweczyk, J. Polastre, and A. Mainwaring, “Lessons from a sensor network expedition,” in *First European Workshop on Wireless Sensor Networks*, Berlin, Germany, January 2004.
- [2] M. Hamilton, “Hummercams, robots, and the virtual reserve,” Feb. 2000.
- [3] L. Girod, J. Elson, A. Cerpa, T. Stathopoulos, N. Ramanathan, and D. Estrin, “EmStar: a Software Environment for Developing and Deploying Wireless Sensor Networks,” in *Proceedings of the 2004 USENIX Technical Conference*, Boston, MA, 2004, USENIX, To appear.
- [4] Y. Ruan and V. Pai, “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.
- [5] J. Zhao, R. Govindan, and D. Estrin, “Computing aggregates for monitoring wireless sensor networks,” in *Proceedings of the IEEE ICC Workshop on Sensor Network Protocols and Applications*, Anchorage, AK, 2003, IEEE.
- [6] J. Zhao, R. Govindan, and D. Estrin, “Residual energy scans for monitoring wireless sensor networks,” in *Proceedings of the IEEE Wireless Communications and Networking Conference*, Florida, 2002, IEEE.
- [7] J. Heidemann, F. Silva, and D. Estrin, “Matching Data Dissemination Algorithms to Application Requirements,” in *Sensys*, Los Angeles, 2003.
- [8] C. Intanagonwiwat, R. Govindan, and D. Estrin, “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, Aug. 2000, pp. 56–67, ACM Press.