Sympathy for the Sensor Network Debugger

Nithya Ramanathan, Kevin Chang, Rahul Kapur, Lewis Girod, Eddie Kohler, and Deborah Estrin
UCLA Center for Embedded Network Sensing
{nithya, kchang, rkapur, girod, kohler, destrin}@cs.ucla.edu

ABSTRACT

Being embedded in the physical world, sensor networks present a wide range of bugs and misbehavior qualitatively different from those in most distributed systems. Unfortunately, due to resource constraints, programmers must investigate these bugs with only limited visibility into the application. This paper presents the design and evaluation of Sympathy, a tool for detecting and debugging failures in sensor networks. Sympathy has selected metrics that enable efficient failure detection, and includes an algorithm that root-causes failures and localizes their sources in order to reduce overall failure notifications and point the user to a small number of probable causes. We describe Sympathy and evaluate its performance through fault injection and by debugging an active application, ESS, in simulation and deployment. We show that for a broad class of data gathering applications, it is possible to detect and diagnose failures by collecting and analyzing a minimal set of metrics at a centralized sink. We have found that there is a tradeoff between notification latency and detection accuracy; that additional metrics traffic does not always improve notification latency; and that Sympathy's process of failure localization reduces primary failure notifications by at least 50% in most cases.

Categories and Subject Descriptors: D.2.5 [**Software Engineering**]: Testing and Debugging—*Distributed debugging*; C.2.4 [**Computer Communication Networks**]: Distributed Systems

General Terms: Design, Reliability

Keywords: sensor networks, debugging, failure detection, failure localization, root causes

1 INTRODUCTION

Developing and debugging sensor network applications in a dynamic, distributed, and resource-constrained embedded environment is an iterative and sometimes laborious process. Initial application development can use a protected and interactive simulation. Once an application is physically deployed, however, interactivity and visibility are greatly reduced, and it becomes difficult to detect and pinpoint problems when they occur. For example, a gap in returned sample data may be caused by a critical node failure, a transient change in link connectivity, or some other unexpected combination of inputs. Responding to a failure can require physical access to a node; depending on the deployment scenario, even obtaining access can be expensive and difficult—or, worse, a cause of additional failures [17]. A sensor network system should therefore help narrow down failures and diagnose their causes, as much as possible, with minimal physical access and interactivity.

This problem is not new, of course. Many network software and hardware tools help IP network administrators diagnose various issues. Unfortunately, these tools often require network and node resources not available to embedded network sensors, or assume networks that are more stable than sensor networks, whose nodes frequently become inaccessible and have poor connectivity, limited power, and memory resources. Such tools may also ignore the cost of transmitting debugging information, another significant concern.

This paper presents *Sympathy*, a prototype tool for detecting and debugging failures in pre- and post-deployment sensor networks. More specifically, Sympathy is designed for data collection applications, which gather distributed data at a centralized sink location for analysis. (Most of today's deployed sensor networks fit this description, as will many networks deployed in future.) Nodes periodically send metrics back to a sink, which combines this information with passively-gathered metrics to detect failures and determine their causes.

Given sensor hardware and network limitations, these transmitted metrics must be minimized. Thus, Sympathy must find the debugging information that provides the maximum leverage: the information that allows the most precise and meaningful failure detection and localization for the lowest overhead. We chose metrics using a simple insight: in a data-collection network, there is a direct relationship between the amount of data collected at the sink and the existence of failures in the system. Insufficient data at the sink implies failure; sufficient data at the sink implies acceptable network behavior. Thus, Sympathy can limit its metrics collection to information about connectivity and data flow. Furthermore, when a failure occurs, the user's goal is to restore data collection. Sympathy's algorithms for localizing failures thus have the more specific and limited goal of telling the user which node or path is responsible for missing data. Of course, this excludes some kinds of network misbehavior, such as overenthusiastic transmission and resulting reduced battery lifetime. However, our experience with Sympathy—both when deployed alone and when deployed in concert with a real application [8]—indicates that many failures encountered in today's networks *do* fit our model, making Sympathy's debugging support largely sufficient as well as useful.

Sympathy gathers and analyzes general system metrics such as nodes' next hops and neighbors. Based on these metrics, it detects which nodes or components have not delivered sufficient data to the sink and infers the causes of these failures. Experimentation and actual deployments show that Sympathy can help detect internal failures whose causes in most cases would not be readily apparent.

Our deployment experience centers on the Extensible Sensing System (ESS), which gathers environmental data for user analysis [8]. ESS-Sympathy deployments are intended for 10 to 100 distributed sensor nodes routing data through a multihop tree back to a sink. During one deployment, Sympathy discovered data disruption due to excessive flux in the routing table. Intuitively, this is most likely due to poor link quality, due either to a bad radio or an ill-defined beacon period; Sympathy's analysis determined

SenSys'05, November 2–4, 2005, San Diego, California, USA.

that transmissions from most nodes to a shared next-hop neighbor were resulting in many CRC errors, and thus pinpointed congestion as the cause of data disruption. Since errors happened on multiple nodes, a bad radio was unlikely. We thus added jitter to query response transmissions, which, as expected, solved the problem.

In the rest of this paper, we discuss related work (Section 2), then describe Sympathy's definitions of failures and failure sources (Section 3). Section 4 then describes the Sympathy architecture and implementation, including the network metrics it collects and its algorithms for pinpointing failures. Section 5 evaluates Sympathy, based on repeated fault injection tests and anecdotes from actual deployments. Finally, we discuss future work and conclude.

2 Related Work

The current state of the art in sensor network debugging uses a combination of simulation, visualization tools, log files, passive monitoring, and tracing programs. Sympathy combines aspects of all these techniques. Simulation [13, 18] is critical for reducing the length of the development cycle and for repeatable experiments (which we make use of in our evaluation), but clearly doesn't replace debugging on the actual hardware; it is impossible to simulate real-time network dynamics, dynamic environments, and numerous timing, MAC, and hardware-related details. While log files can capture historical perspective and context, they contain excessive and unfiltered data that can obfuscate important events. Visualization tools can aid real-time debugging but often don't highlight events that may indicate a failure or capture historical context. For example, the Emstar visualizer [5, 6] shows *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. Sympathy runs and produces useful results in both simulation and deployment.

The Nucleus network management system (NMS) infrastructure helps sensor network applications export debugging and monitoring information [20]. Nucleus's support for exporting counters and statistics and recording application metrics is both easy to use and lightweight. With minimal modification, we could replace Sympathy's homegrown logging mechanism and application interface with Nucleus. NMS also exports specific metrics but does not provide infrastructure to analyze these metrics. Furthermore, these metrics consume more than double the RAM required for the rest of the stack [20]. Our contributions lie in determining the minimal metrics to export, consuming on the order of 50 bytes of RAM for all Sympathy mote code, and in defining algorithms that flag user-relevant failures based on these metrics.

Previous work on debugging in sensor networks [14, 15] proposes a pre-deployment architecture to monitor networks in simulation and potentially emulation. In this architecture, a processing node is directly connected with all other nodes over a simulated or Ethernet back channel to continually collect information from nodes. This architecture focuses on event detection and correlation, so the information collected also focuses on these events. We have found that event detection does not necessarily facilitate failure detection, while it does impose high overhead as events are frequent.

Periodic transmission of metrics from nodes to the sink is not a new idea. MintRoute [22] includes periodic transmission of neighbor tables to aid in debugging at the sink. However, it neither includes other metrics nor performs failure analysis at the sink.

Zhao et al. proposed an algorithm to continually compute aggregates (sum, average, and count) of loss rates, energy levels, and packet counts to aid in debugging and an energy-efficient way to calculate those aggregates with in-network processing [23, 24].

In terms of failure detection methods, Szewczyk et al. identify nodes that report sensor data exceeding a static threshold as being close to failure [19]. This work utilizes a simple model to specify expected values and infer imminent failure when returned data does not match the model. Others treat deviation from a calibration model as a failure indication [3], or observe that low battery power is correlated with wildly implausible data readings [21]. Sympathy detects failure based on data *quantity*, rather than data quality; this is sufficient to catch many important failures in practice. However, Sympathy's infrastructure could easily include data quality measurements.

Many Internet network management systems have insights we can apply. The Simple Network Management Protocol (SNMP) manages the exchange of network statistics between a centralized server and *agent* nodes, which respond to queries and asynchronously signal events; our architecture (like that of Nucleus) is similar. Management by Delegation moves some network management responsibility off the centralized server and onto distributed nodes, using mobile code [7]; this technique may become important in sensor networks as tiny virtual machines become more widely deployed [12].

Kiciman et al. collect low-level network metrics and use statistical analysis to identify application-level anomalous behavior; once an anomaly is detected, the faulty node is rebooted [11]. Fox et al. extend this idea to suggest using statistical learning techniques based on pre-existing models and "well-understood external indicators" to identify anomalous behavior [4].

Ruan and Pai's DeBox system [16] motivated Sympathy's initial design. DeBox suggests that exposing minimal internal state to applications in real time affords better performance analysis and tuning than passive profilers that provide information post facto. 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.

3 Failures and Failure Sources

Sympathy aims to detect a large class of sensor network failures and *localize* each failure to simple, actionable information about its likely source. Code running on a non-resource-constrained network node called a *sink*—often a data sink, such as a Stargate-class system—continuously monitors normal network traffic and Sympathy-generated traffic for failure conditions. When a failure is detected, Sympathy triggers failure localization and reporting so users can take appropriate action.

Before describing the algorithms used to detect and localize failures, we discuss what detected and localized failures are.

Failures Sympathy expects all live network nodes to generate traffic of some kind, whether routing updates, time synchronization beacons, or data periodically transmitted to the sink. We call this traffic *monitored traffic* to distinguish it from Sympathy's own *metrics traffic* (statistics packets generated by nodes and transmitted to the sink). Sympathy detects a failure and triggers localization *when a node generates less monitored traffic than expected.*

For example, in a network running the Surge application, every node normally generates a packet containing a sensor reading every n seconds. Sympathy might detect a failure, and trigger further localization, if a node in the network generated no reading for $3n$ seconds. The extra factor of three reduces false positives in the event of packet loss but delays failure notification as well. Our evaluation measures the consequences of this tradeoff.

Surge and ESS, the two systems we evaluate, resemble many currently deployed sensor networks in that they periodically transmit sensor data to the sink. For these systems, Sympathy monitors the sensed data, as well as routing beacons and other expected communication. This has the advantage of making failure detection almost end-to-end. Any failure in the sensor data path will trigger Sympathy, including failures in sensor boards that don't affect routing or other node software. However, it is not a requirement; for example, Sympathy could track only routing beacons from nodes in its broadcast domain in a system with no regularly-transmitted data. Future work will address entirely event-driven systems with no regular communication of any kind; however, we expect that most networks will feature some regular communication, however infrequent, if only to verify connectivity.

Sympathy itself generates additional *metrics traffic* from each node; this in-depth information helps localize failures. The absence of metrics traffic can indicate a problem, but Sympathy considers the absence of monitored traffic more significant.

Localized Sources Sympathy's algorithms assign each detected failure a *localized source*, an actionable description of the most likely cause of the failure. We aim to choose the simplest localized source that explains the failure. After experimenting with larger sets of more specific sources, we decided that a small set of general sources is better: users must take the same actions for general and specific sources, such as going out into the field and moving a node, yet more specific sources are more likely to be wrong. The more specific source identification, and any information used to calculate it, is still available as part of Sympathy's output, if desired. There are three localized sources for a node's failure to transmit enough monitored traffic:

- **Self**. The node's failure has been localized to the node itself. The node may have crashed or rebooted, there may be another local bug preventing data transmission, or there may be a connectivity issue (the node does not have a route to the sink). Remedial action will probably involve moving or interacting with the node itself (e.g. changing its batteries).
- **Path.** The node's failure is due to a failure along the path from the node to the sink, such as a different node's failure or excessive collisions along the path. Sympathy identifies a node along the path and the problem potentially causing the packet loss in order to focus the user's search. Remedial action will probably involve moving or fixing a node or the network in its vicinity.
- **Sink.** Often when the whole network appears to be failing, the simplest explanation is a failure at the sink, such as bad sink placement or changes in the environment since deployment. Clues such as no node being able to hear the sink but hearing other nodes point Sympathy to issues localized at the sink. Remedial action will probably involve changing the sink placement or examining sink metrics for bugs or other connectivity issues.

4 ARCHITECTURE

Sympathy detects and localizes failures using information from all network nodes, including both resource-constrained TinyOS-based motes and resource-rich Linux-based sinks. Sympathy code on motes simply transmits metrics and responds to occasional queries; all failure detection and localization runs on sinks. The actual localization process has four stages. In the first, ongoing stage, the sink *collects metrics* from other nodes in the system. Metrics include

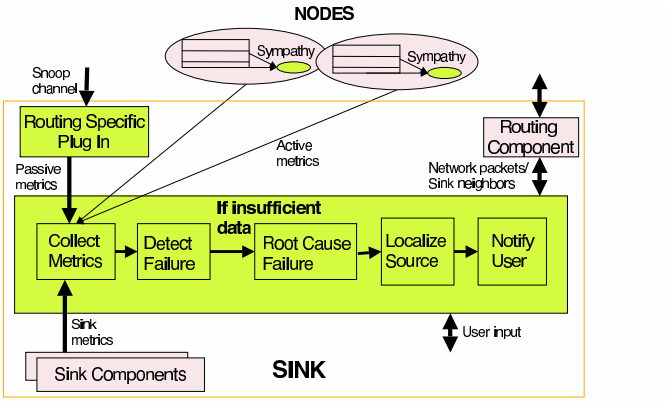

Figure 1: Sympathy system, including code running at the nodes and sink, high-level versions of interfaces between components, and an overview of Sympathy's failure localization algorithm.

information transmitted as part of normal communication, such as sampled data, and information transmitted actively by Sympathy code running on motes. Upon any packet reception Sympathy looks for failures by analyzing this metric stream, looking for *insufficient data* received from any node. When a failure is detected, Sympathy *root-causes* the failure by analyzing metrics and running tests as necessary to determine the cause. Common root causes include a node crash or reboot, no node has a route to the sink because the sink has no active neighbors, or the sink is not receiving data because the node is not receiving requests. Finally, after determining the root causes for all existing failures in the system at that time, Sympathy assigns each failure a *localized source*, which is one of the three sources (Self, Path, and Sink) identified above. These sources are reported to the user with supporting documentation. The goal is to produce a minimal list of failures that the user must definitely fix. Thus, if one failure potentially explains another—for example, an upstream node's (node closer to the sink) crash might explain why a node downstream of it seemingly disappeared—we mark the latter failure as *secondary*, and report it with less urgency.

The metrics Sympathy collects and the root causes to which it assigns failures are both based on a simple insight: that in the absence of failure, network flows are conserved [9]. Thus, when data is lost, it must be getting lost somewhere in the network, and finding the location of loss is a good approximation to finding the true cause of the failure. For example, perhaps a sink request for data got lost; or, following the data path forward, the node's response got lost due to collisions. Sympathy's *flow model* systematizes this data path. It specifies that nodes need: neighbors in order to have routes; routes in order to send data; to receive requests in order to send responses (sometimes); and to send data in order for the sink to receive the data. Metrics and root causes are chosen so that, in the event of insufficient data, Sympathy is able to trace the data flow through the network to narrow down the cause and location of data loss.

Figure 1 shows a component overview of the Sympathy system.

4.1 Metrics

The Sympathy sink collects a variety of metrics to aid in both failure detection and localization. We added metrics as needed to distinguish different logical locations in our flow model, such as failures local to a node vs. due to collisions along a path, and have arrived at a simple stable set. Metrics are collected in three ways: Sympathy code running on every network node *actively* transmits packets containing metrics back to the sink; the sink snoops nearby

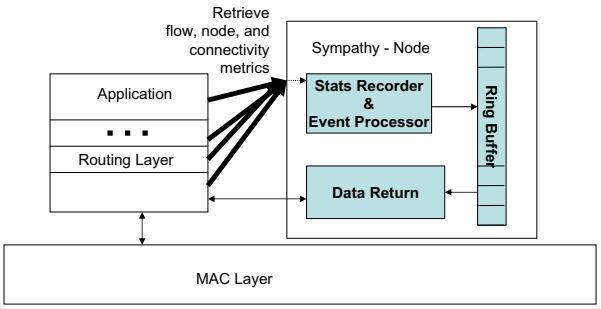

Figure 2: Sympathy code on a resource-constrained TinyOS node. Sympathy contains interfaces to collect statistics from all layers in the stack, including the routing layer and individual application components.

application and transport traffic to discover metrics *passively*; and Sympathy code running on the sink extracts sink metrics from *the sink application itself*.

Sympathy's current metrics fall into three main categories, connectivity, flow, and node metrics. The set is extensible, however; applications can create and use arbitrary metric types.

Connectivity Metrics Network connectivity information, collected from every node in the network, helps Sympathy detect and localize routing failures. Thus, Sympathy collects every node's current **ROUTING TABLE**—its sink, associated next hop, and path quality—and **NEIGHBOR LIST**, which lists all of its neighbors, whether involved in routing or not. It also collects the sink's NEIGHBOR LIST. This information is collected both passively and actively. In the passive implementation, a plug-in on the sink snoops nodes' broadcast routing control packets, parses them and passes their metrics such as neighbor and routing advertisements to Sympathy. Our plug-in for MintRoute [22] is 85 lines of nesC, and for ESS [8] 120 lines. Passive collection cannot, however, obtain connectivity information for distant nodes whose broadcasts aren't heard by the sink. Thus, active collection code runs on each node in the network. A Sympathy TinyOS module periodically collects metrics from various layers of the networking stack using existing interfaces, packages them into packets, and transmits them via the application's routing layer; Figure 2 shows this architecture. A similar module on the sink collects its connectivity information as well, using Emstar [5] IPC mechanisms.

Flow Metrics Sympathy monitors the network's traffic load as well as its connectivity. Traffic is measured independently for an extensible set of *components*, which are entities with somewhat independent flow. For example, different applications will have different components, and within a single application, queries, responses, and routing advertisements might have one component each. Our ESS and Surge evaluations use one component each, in each case measuring application data packets. For each component, Sympathy collects **PACKETS TRANSMITTED** and **PACKETS RECEIVED** from each node. Existing routing layers do not transmit this information, so it must be collected actively. In addition, for each component, Sympathy maintains per-node counters of **SINK PACKETS TRANSMITTED** (i.e., packets transmitted from the sink to the node, including broadcasts) and **SINK PACKETS RECEIVED** (packets received by the sink from the node), and a **SINK LAST TIMESTAMP** (the most recent time that the sink received a packet from the node). Sympathy uses these counters to track data flow from the sink to the node.

Node Metrics Each node actively transmits its **UPTIME** to the sink, allowing Sympathy to detect reboots. In addition, each node transmits **BAD PACKETS RECEIVED** and **GOOD PACKETS RECEIVED** counters, for packets received with and without CRC errors. The sink maintains its own BAD PACKETS RECEIVED and GOOD PACKETS RECEIVED. These counters are used to detect congestion.

Active metrics collection is triggered by a TinyOS timer that goes off once per **metric period**. Larger metric periods minimize overhead relative to other application traffic (and do not necessarily impact failure detection latencies, as we discuss in Section 5). When the timer goes off, Sympathy code on the node collects all relevant metrics and transmits them back to the sink, using the application's routing layer (rather than an independent routing layer or a back channel). Using the existing routing layer can make it more difficult to debug routing-related failures, but in our experience we have always accumulated sufficient data from the regular transmission of Sympathy metrics before a routing layer failure to root-cause that failure. In addition, if the routing layer provides support for flooding (which is generally simpler than tree routing, and thus might work even when tree routing is broken), nodes can flood metrics in response to requests from the sink.

Passively collected metrics are preferable to actively collected metrics, since they add no resource cost to operating the network. Some simple localization is possible in a one-hop Sympathy network using passive metrics and sink metrics alone. However, failure localization generally depends on information, such as the number of packets each component tried to transmit (whether or not it was received), currently available only through active collection. Once active collection is available, passive metrics are somewhat redundant (although they can provide useful supplementary information between metrics periods, or when active metrics packets are dropped). Thus, Sympathy can work without passive metrics or the routing plug-in that provides them.

Nodes store and transmit metric counters using units of packets since node boot, modulo the counter size. This is robust to metrics packet loss; the sink can calculate the number of packets sent or received between any two metrics packets by subtracting their counter values. Rollover must be accounted for, of course: if a counter value is lower than the previous counter value, Sympathy checks the node's UPTIME to see if the node has rebooted; if not, it assumes the counter has rolled over and calculates differences accordingly. This implicitly assumes that counters will roll over at most once between successfully received metrics packets. When a metrics packet is lost, Sympathy linearly interpolates counter values. For instance, if a single metrics packet at time t is not received, then counter increases in the metrics packet at time $t + 1$ are attributed half to time t and half to time $t + 1$.

All metrics time out after a period called an **epoch**. For instance, if Sympathy does not receive a node's neighbor list for an epoch or more, then it invalidates any cached neighbor list. The epoch is used as a common timeout to determine if something is wrong with a node or its delivery path. Longer epochs delay failure detection, but also make detection more robust to short-term packet loss. The smallest sensible epoch is one metric period, and we set the epoch to a multiple of the metric period. The best setting would be small enough for Sympathy to quickly notify the user of failures in the network, yet large enough to allow the sink time to collect packets from nodes and time to differentiate transient packet loss from failure. Section 5 explores the consequences of different epochs.

Nodes that intentionally or unintentionally report incorrect metrics can utterly confuse Sympathy. We leave this for future work.

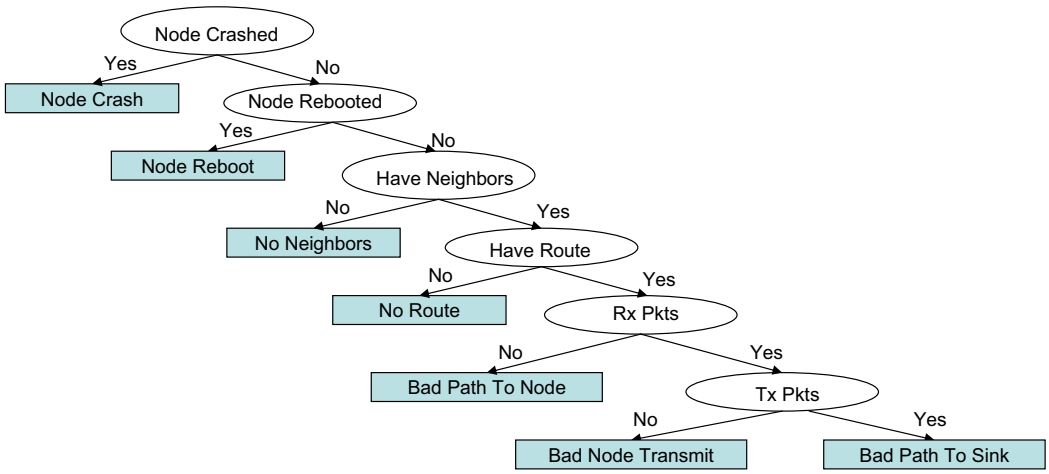

Figure 3: Decision tree Sympathy uses to root-cause node failures once they have been detected.

4.2 Failure Detection

Sympathy triggers failure detection every time it receives a packet from a node, and once a metric period at the very minimum. Failures are detected using flow metrics. Specifically, Sympathy determines whether the sink has received sufficient data from every component on every node over the past epoch. Insufficient data indicates a failure. Thresholds for sufficient data are specified per component and per node, in units of packets per epoch, and may be modified by the application (using an update interface) or the user (using a command-line interface). In the simple periodic-collection applications we measure, these thresholds are the same for all nodes; per-node thresholds allow Sympathy to monitor other applications, regardless of differential transmission rates or in-network aggregation, as long as the sink knows the quantity and/or quality of data it should expect.

Thus, the sink has received sufficient packets from a node component if its SINK PACKETS RECEIVED counter at least equals the corresponding threshold.

Sympathy detects a failure if and only if some sink component does not receive sufficient packets.

Sympathy also maintains analogous thresholds that detect when a node's component doesn't receive sufficient data from the sink (using the PACKETS RECEIVED counter), and when a node's component doesn't transmit sufficient data, regardless of how much data is received (using the PACKETS TRANSMITTED counter). These thresholds are used during the root-causing process; violations do not signal failure in and of themselves.

In our code, some of these thresholds are specified as fractions of other metrics; the node-receive threshold, for example, is a fraction of the corresponding SINK PACKETS TRANSMITTED counter.[1]

4.3 Root Causes

Once Sympathy has detected a failure, it uses an empirically-developed decision tree to determine the most likely cause of packet loss in the network. The decision tree uses metrics to track through Sympathy's flow model and determine where data is lost. Sympathy root-causes a failed node N by performing the following tests in order, and reporting the first root cause that fits:

1. If N's SINK LAST TIMESTAMP metric is at least one epoch old, and no other node has N listed in its NEIGHBOR LIST, then the node has crashed or otherwise completely fallen off the network. The root cause is **Node Crash**. Crashes are the hardest failure to distinguish, since their signature—the lack of traffic—is common to other failure modes, such as high congestion; thus, the relatively long timetable for reporting a Node Crash root cause.
2. Otherwise, if N's UPTIME metric has unexpectedly rolled over, then the node has rebooted. The root cause is **Node Reboot**, and Sympathy resets all of the node's counter metrics (since the node's versions of the counters have been reset to zero). To distinguish reboot from regular counter rollover, Sympathy checks whether the counter could feasibly have rolled over from the previous metrics packet.
3. If N's NEIGHBOR LIST is empty or expired (more than one epoch old), the node cannot hear from its neighbors, and the root cause is **No Neighbors**.
4. If N's ROUTING TABLE is empty or expired, the node cannot route to the sink, and the root cause is **No Route**.
5. At this point, N is alive, has not rebooted, and has what appears to be a good route back to the sink. The failure is either entirely node-local, or concerns the path back to the sink. If N's PACKETS RECEIVED metric is below the corresponding threshold, then the node isn't receiving the packets it should from the sink; the root cause **Bad Path To Node**.
6. If N's PACKETS TRANSMITTED metric is below the corresponding threshold, then the node's software isn't correctly transmitting data; the root cause is **Bad Node Transmit**.
7. Otherwise, N is both receiving and transmitting sufficient data. The data must be getting lost on its way to the sink; the root cause is **Bad Path To Sink**.

This decision process proceeds from general to specific causes: from basic health (steps 1–2) through connectivity (steps 3–4) to communication (steps 5–7). Within the last three steps, the order is determined by network flow: for the sink to receive sufficient data, the node must receive sufficient requests (step 5) and transmit responses (step 6) that aren't dropped en route (step 7). Sympathy determines independent root causes for each failed node–component pair, although in many cases multiple failing components will have the same root cause.

[1]This choice assumes that the component's PACKETS RECEIVED counter only counts packets from the sink. Other threshold setting methodologies are possible, of course.

Root Cause	Metrics	Localized Source
Node Crash	SINK LAST TIMESTAMP, NEIGHBOR LIST	Self
Node Reboot	UPTIME	Self
No Neighbors	NEIGHBOR LIST	Self
No Route	ROUTING TABLE	Self
Bad Path To Node	PACKETS RECEIVED	Path/Self
Bad Node Transmit	PACKETS TRANSMITTED	Self
Bad Path To Sink	SINK PACKETS RECEIVED	Path/Self
No Sink Route	ROUTING TABLE	Sink
No Neighbors (at sink)	NEIGHBOR LIST	Sink
Congestion	GOOD PACKETS RECEIVED, BAD PACKETS RECEIVED	N/A

Figure 4: Root causes with associated metrics and localized sources.

Simultaneously, Sympathy checks for other root causes that may explain the failure. These include congestion-related causes and causes involving the sink.

- If the sink's NEIGHBOR LIST is empty, then the sink is isolated. A **No Neighbors** root cause is recorded for the sink.
- If no node has a valid ROUTING TABLE with a route to the sink, then no one can route to the sink. A **No Sink Route** root cause is recorded.
- Sympathy checks each node in the network (failed or not) for excessive collisions. If a node's BAD PACKETS RECEIVED metric is more than 75% of its GOOD PACKETS RECEIVED metric (an experimentally derived threshold), a **Congestion** root cause is recorded for that node.

The decision tree is summarized in Figure 3; tests are in circles, and the resulting root causes are the terminating boxes. Root causes, associated metrics, and potential localized sources are summarized in Table 4.

4.4 Source Localization

Sympathy then analyzes the resulting root causes and assigns each failure a localized source: either Self (the node itself is broken), Path (the path is broken), or Sink (the sink is broken). Failures localized to Self are called *primary* failures, since they cannot be traced to any other cause in the network; all other failures are *secondary*. Sympathy logs all failures, but highlights primary failures as potential causes for all other failures, encouraging the user to prioritize fixing those failures.

The algorithm is relatively simple. If Sympathy recorded a No Neighbors or No Sink Route failure for the sink, the sink is broken; this trumps all other failure conditions. The sink failure is assigned a localized source of Self, and every other failure is assigned a localized source of Sink. Otherwise, for communication root causes (Bad Path To Node, Bad Node Transmit, and Bad Path To Sink: steps 5–7 in Section 4.3's decision tests), Sympathy looks along the path to the sink for another failure that might take precedence. Specifically, given a failure at node *N* with such a root cause, Sympathy tracks the path from *N* to the sink via ROUTING TABLE metrics. If a node *O* on this path—but closer to the sink—has any root cause recorded, including Congestion, then *N*'s failure is assigned a localized source of Path with *O* as the primary cause. This search applies to the sink as well: If the sink has Congestion recorded, then *N*'s failure is assigned a localized source of Path with the sink as the primary cause. (While we experimented with allowing upstream failures to explain No Neighbors and other basic-health and connectivity root causes, this mischaracterized too many failures as secondary.) If neither of these exceptions applies, then the failure is primary, and its localized source is Self.

Thus, if a node *N* is experiencing a failure with root cause Bad Node Transmit, but a node upstream has Crashed, then *N*'s failure will be reported as secondary; but if *N* actually Rebooted, its failure is primary, despite the upstream Node Crash.

This source localization procedure may mischaracterize some failures as secondary; for example, a Bad Node Transmit failure is unlikely to be caused by an upstream Bad Path To Node failure. Our goal, however, is to report the smallest number of failures that *must* be fixed to return the network to health. It would be convenient to create a more precise model of flow through the network based on collected topology information, and thus determine exactly where data is lost in the network. This might also allow us to limit failure notifications in case of network partition; currently Sympathy would report this with one primary Node Crash failure per partitioned node, instead of a primary failure at the partition's border with secondary failures beyond that border. We leave this as future work; the difficulty, of course, is that Sympathy never knows how accurate its view of the network is.

4.5 Failure Notification

Once a failure is localized, Sympathy notifies the user through a log file containing the detected failure (and whether it is primary or secondary), its root cause(s), its localization, relevant metrics, and calculated statistics and events. The ROUTING TABLE and NEIGHBOR LIST metrics are annotated with statistics such as neighbor age and number of times a route has been changed. Sympathy also checks basic invariants, such as that the selected next-hop is also in the neighbor table; it identified a bug in ESS with this simple rule. Nodes are allowed to transmit specific statistics that Sympathy does not understand; if the sink can translate them into ASCII, Sympathy will include these statistics in its reports as well. Finally, if the user sent a ping request about a node (see below), all the resulting responses will also be included in the log file.

Due to a quirk of our current implementation, the user is only immediately notified of failures resulting in zero packets received from a node to the sink in the past epoch: that is, Node Crash and Node Reboot. Other failures are reported up to one metric period after they happen.

4.6 Command Interface

Sympathy provides several command line and Web-accessible mechanisms for users to query and control the system. In order to minimize the impact on system lifetime, Sympathy provides a knob at the Sympathy sink for user control of the metric period after deployment. This knob is useful during deployment when users may initially require frequent transmission of metrics in order to obtain immediate feedback on configuration and topology changes, but want to scale back the metrics to save network resources once the deployment is complete. The user can also set the metric transmission period to 0 which disables metric transmission from nodes, and moves Sympathy to only collecting metrics passively and upon user request. Users can set thresholds and read logs.

In addition, Sympathy provides a *ping-about* command that floods the network with a query for information about a specific node. If the node receives the flood, it responds by transmitting its

standard metrics packets. Additionally, any other node that has specific information about the queried node, such as number of packets received from it or the time of last communication, transmits this data back. Our Web-based user interface graphically displays the network of nodes, allows users to retrieve all information gathered about the node, and highlights any failures; we have also integrated Sympathy with a graphical interface for visualizing metrics and environmental data [2].

5 EVALUATION

We evaluate Sympathy in simulation by varying system parameters, traffic conditions, failure injection, and instrumenting two different network stacks (Surge running on MintRoute [22], and ESS [8] with multihop), and anecdotally in several deployed systems instrumented with Sympathy. Through simulation and deployment we validate all of Sympathy's possible source localizations and exercise its failure detection. We validate the Self source localization by injecting node crashes and evaluating Sympathy's detection of the correct failure and source, and the Path source localization by showing that Sympathy localizes data loss to congestion without sacrificing detection accuracy.

Our performance hypothesis is that Sympathy accurately detects injected failures with relatively low network overhead and low latency. To evaluate this hypothesis, we inject failures into a 20-node simulated network and measure latency and accuracy of failure detection. We find that Sympathy detects every failure we inject. There is a tradeoff between detection latency (time from when the failure is injected to when Sympathy notifies the user of the failure) and accuracy (number of primary failure notifications). As the epoch increases, detection latency gets worse (Figures 6 and 13) while accuracy gets better (Figures 10 and 11). Although the packet overhead is reduced by a factor of three by tripling the metric period from 3 to 9 minutes, Sympathy's failure detection latency is not significantly impacted for Node Crash failures. By performing path analysis to determine if a node's failure is caused by a failure upstream, Sympathy is able to reduce excessive failure notifications by at least 50% for most cases; and using collision detection in the network as an indication for congestion and a source of packet loss, Sympathy can reduce failure notifications even further.

5.1 Methodology

We evaluate Sympathy by injecting node crashes into the network in simulation using Emstar [5]. Since node crashes are detected conservatively, they are the hardest failure to detect accurately and quickly. Most other root causes have been observed and validated in deployment and simulation.

Using Emstar's contention-based channel model to simulate collisions, we vary traffic conditions in order to increase packet loss and congestion. The traffic conditions are: (1) no application traffic, just routing traffic (*no-app*); (2) one 10-byte application packet sent every 60 seconds (*traffic60*); and (3) one 10-byte application packet sent every 30 seconds (*traffic30*); and (4) one 10-byte application packet sent every 10 seconds (*traffic10*). In each case, thresholds were set accordingly; in the no-app case, Sympathy tracked Sympathy metrics instead of application data.

Unless otherwise specified, each set of experiments consists of injecting single node crashes into a 20-node network with a 7-hop diameter. Different trials within an experiment fail each node in the network to check for topology effects, and each experiment is repeated twice. The epoch setting defaults to 540 seconds, the metric period to 3 minutes, and the traffic scenario to traffic30. Tests run for about an hour, and end once Sympathy detects that the node has crashed with a source localization of Self.

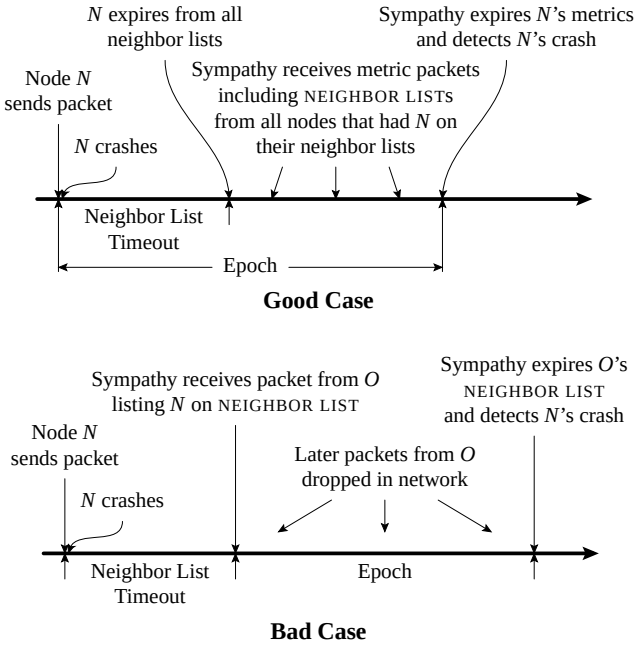

Figure 5: Sympathy can detect node *N*'s crash within one epoch of its injection when all other nodes send updated information removing *N* from their neighbor lists. Otherwise, Sympathy must wait for all neighbor list metrics that mention *N* to expire. This takes at most an additional neighbor list timeout, unless a node incorrectly continues to mention *N*.

We use empirical cumulative distribution functions or ECDFs to present the distribution of our data for several graphs.

5.2 Failure Detection Latency

Sympathy is quite conservative before reporting a node crash, since the only symptom of a crash—that is, the absence of any data—can indicate so many other kinds of failure (such as persistent congestion or a path failure between the node and the sink). Specifically, Sympathy will not report a node crash for node *N* until (1) it has not received any data from *N* for an epoch, and (2) no other node lists *N* as a neighbor in its NEIGHBOR LIST metric (which can either occur if the node explicitly sends a NEIGHBOR LIST update without *N*, or if the sink flushes its cached copy of the NEIGHBOR LIST because the node has not sent an update during the epoch). Thus, the failure notification latency will depend on the order of events. A notification latency of one epoch is possible if, during that epoch, Sympathy hears updated NEIGHBOR LISTs from *N*'s former neighbors, or, alternatively, if all *N*'s neighbors' NEIGHBOR LISTs expire. (If the epoch duration is less than the time it takes for a node to time out from a neighbor list, then this "good case" can only occur if the relevant NEIGHBOR LISTs expire.) Latencies of less than one epoch are possible when *N* was quiet before it crashed, or its pre-crash packets were dropped; this can fool Sympathy into detecting a crash early. Greater notification latencies are possible, too, if another node sends an updated neighbor list that *includes N*, as might occur if its metric period ends immediately before *N* expires from its neighbor list. Assuming nodes transmit correct metrics, the latency will be extended by at most one neighbor list timeout. Figure 5 demonstrates the cases.

As the epoch increases relative to the metric period, the probability of observing a latency longer than an epoch will *decrease*. This is because the "bad case" in Figure 5 requires that all of *O*'s metrics

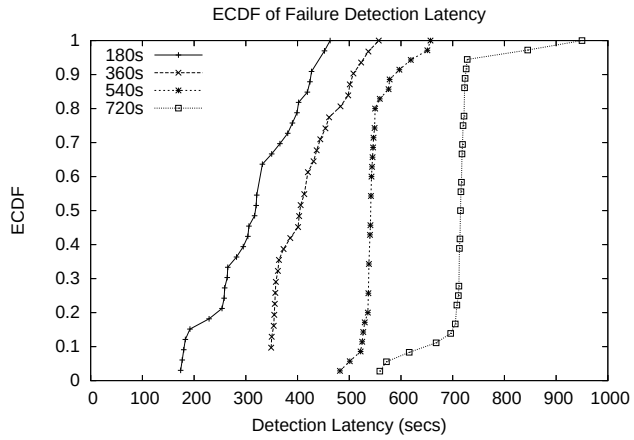

Figure 6: Failure detection latency for different epoch sizes. As the epoch increases, the sink is more likely to detect the failure within one epoch.

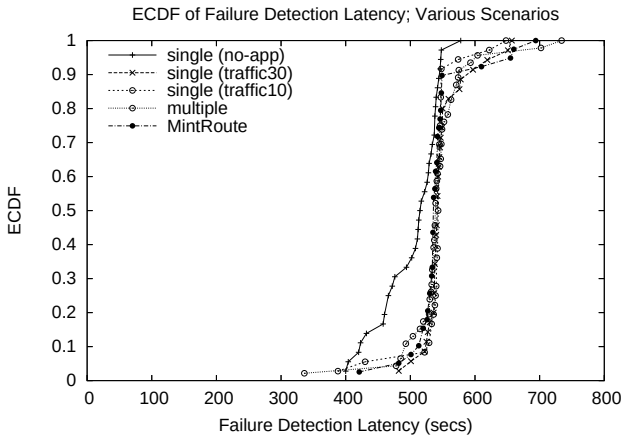

Figure 7: Failure detection latency for varied scenarios, including different levels of background traffic, multiple simultaneous failures, and a different routing protocol and application. Epoch is 540 sec throughout; all results are similar.

are dropped for a period of one epoch; as the epoch increases, more metrics are transmitted per epoch and this becomes less likely.

We plot the failure detection time as an ECDF over an entire set of experiments in Figure 6. In our system, the neighbor list timeout is 320 seconds, so the upper bound on detecting a failure given correct node operation is 320 seconds plus an epoch. The failure was detected as a primary failure in every case. The observations made above for failure detection latency are consistent with this graph: failure detection latency increases with epoch, but as epoch increases the sink is increasingly likely to detect the failure within one epoch. This explains why the latencies for an epoch of 180 seconds are pretty evenly distributed between 180 seconds (the lower bound) and 460 seconds (close to the upper bound of 500 seconds), while the latencies for an epoch of 720 seconds are predominantly between 710 and 720 seconds.

We then measured Sympathy in a variety of different scenarios and traffic conditions; Figure 7 contains a line for each scenario. To demonstrate Sympathy's ability to detect multiple simultaneous failures, we inject two failures within one metric period, choosing every possible combination of two nodes in the network; detection latency was measured from when the first failure is injected into the network. To demonstrate Sympathy's modularity with respect to routing protocols and applications, we instrumented the routing protocol MintRoute [22] and the data delivery application Surge, and re-ran our experiments. Finally, to demonstrate Sympathy's relative indifference to background traffic, we ran with 3 different traffic scenarios, no-app, traffic30, and traffic10. The latency ECDFs are largely similar in all cases. In most scenarios, Sympathy detects the failure within one epoch, but in many instances of the no-application-traffic scenario, Sympathy detects the failure before a complete epoch has elapsed. This is because the Sympathy traffic on which no-app relies is sent much less frequently than application traffic; thus, failures are more likely to be injected in the middle of a long quiet period. Sympathy mistakenly assumes that the failure occurred at the beginning of the quiet period, before the failure was even injected. We conclude that Sympathy's failure detection latency is not notably impacted by running with a different routing protocol (MintRoute), by varying traffic, or by multiple simultaneous failures.

In Figure 8 we compare a system that performs failure detection exactly once per metric period (timer-triggered) with a system that performs failure detection whenever a packet arrives at the sink

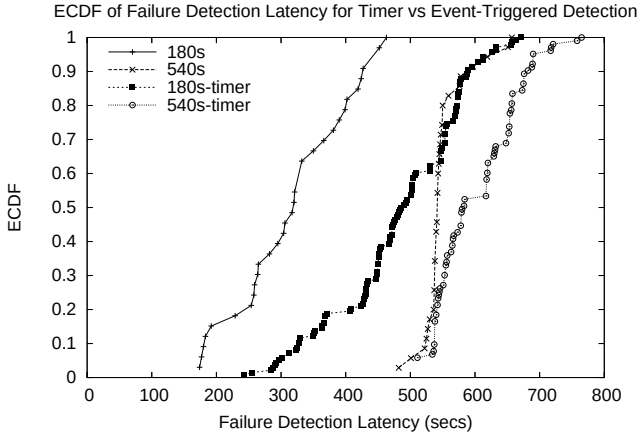

Figure 8: Failure detection latency for Sympathy-style event-triggered failure detection and for timer-triggered failure detection. Event-triggered failure detection has a lower average latency, as the latency upper bound for timer-triggered is one metric period greater.

(Sympathy's default mode). We would expect timer-triggered systems to have higher latency, as failure detection may be delayed by up to one metric period. This is exactly what we see: timer-triggered failure detection latencies for 180-second epochs are up to one metric period (180 seconds) greater. Sympathy-style event-triggered failure detection reduces the time to detect a failure, and is more likely to detect a failure within an epoch. The advantage of an event-triggered system is more pronounced with an epoch size of 180 seconds because, as mentioned before, any system's detection latency will approach the minimum as the epoch increases.

5.3 Failure Detection Accuracy

To evaluate Sympathy's failure detection accuracy, we measure the number of primary failure notifications (failures that are localized to Self as opposed to failures that are localized elsewhere in the network) produced in various scenarios. Sympathy aims to reduce primary failure notifications caused by temporary network conditions, such as transient congestion, thereby focusing the user's time on the most important problems in the network. Since our experiments inject one failure each, the theoretically optimal number of primary failures per experiment is one, although in reality the in-

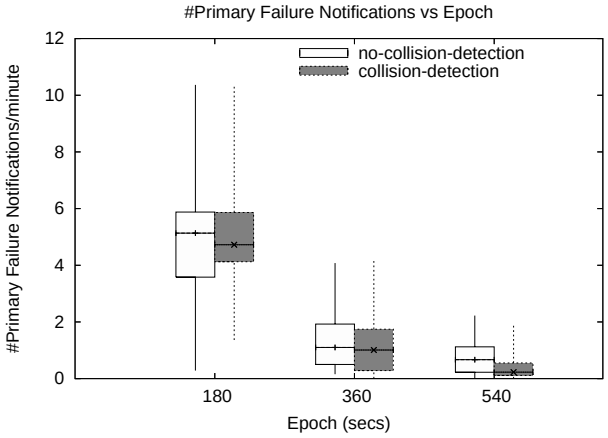

Figure 9: Number of primary failure notifications vs. epoch with and without collision detection. The number of primary failure notifications decreases as the epoch increases for both cases. In addition, enabling collision detection further reduces primary failures, as Sympathy is able to more accurately identify congestion-induced failures as secondary.

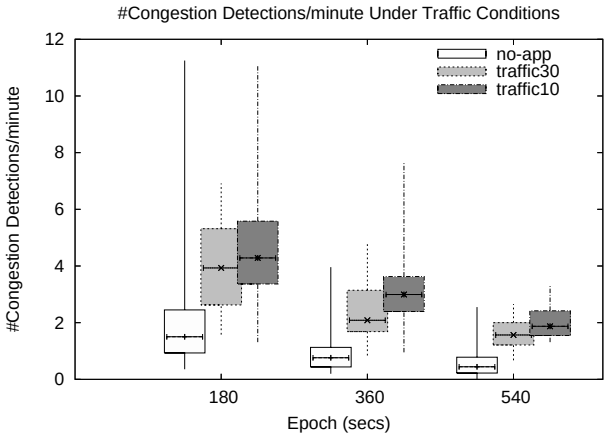

Figure 10: Number of collision detections vs. epoch. Congestion root causes increase with traffic, making collision notifications a good indicator for congestion detection.

jected failure can cause other real network problems that Sympathy will currently report. Thus, we evaluate failure detection accuracy by testing how few primary failure notifications are produced. Again, every experiment reported the injected failure as primary.

First, we consider the epoch setting. Figure 9 shows the number of primary failure notifications for varying epoch lengths; ignore the distinction between collision detection and no collision detection for now. In this graph the top and bottom of the box represent the 25th and 75th quartiles, a horizontal line is placed at the median, and the whiskers represent the minimum and maximum values. We report notifications per minute to compensate for different test lengths, and notifications are aggregated over all nodes in the network. Repeated failure notifications are included (Sympathy continues to notify the user of a failure every metric period until it is fixed). In this figure we see that the accuracy improves—the number of primary notifications decreases—as the epoch increases. This occurs for two reasons. The probability that Sympathy will have a valid route for a node (i.e. has heard a route update from the node in the past epoch) increases as the epoch increases, as there is simply more time to receive a packet from the node. So, for shorter epochs, Sympathy is more likely not to have a valid

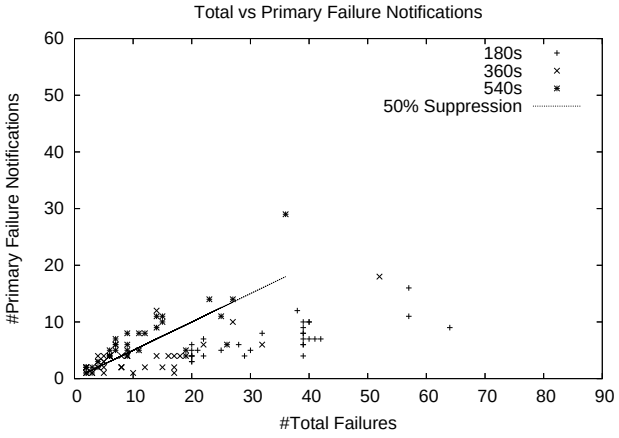

Figure 11: Number of primary compared to total failure notifications for epochs of 180, 360, and 540 seconds; suppression of failures is a result of failure localization. Points below the line indicate greater than 50% suppression of failures; lower points on the graph are better. Failure notifications are aggregated for a 20-node network.

route for a node, which is by necessity a primary failure. Second, as we explained above, the probability of Sympathy detecting failures caused by short-term packet loss decreases as the epoch increases.

We also evaluate the efficacy of using distributed collision detection to localize the source of a node's failure to network congestion. Collision detection, and the associated Congestion root cause, reduces primary failure notifications by localizing some transient failures to a Path source when there is downstream congestion. Figure 9 also shows a drop in the number of primary failure notifications when collision detection is on (as compared to the case when collision detection is off) of up to 50% for larger epochs. We verified that this is because some of the remaining congestion-induced failures are correctly identified as secondary. Congestion detection is irrelevant for shorter epochs because the sink is less likely to have a valid route for a node, as previously explained. Figure 10 directly shows the number of Congestion root causes reported under different traffic scenarios; as traffic increases, so does the number of congestion notifications.

Finally, the scatterplot in Figure 11 shows that failure localization in general, which includes both congestion-related failure suppression and non-congestion-related failure suppression (i.e., a Bad Path To Sink failure marked secondary because of a downstream Node Crash failure), is quite effective. The total number of failures detected by Sympathy is shown on the *x* axis, and the number of primary failure notifications on the *y* axis. The experiment is run for epoch periods of 180, 360, and 540 seconds; lower points are better. For a majority of tests, Sympathy is able to categorize more than 50% of failures as secondary, thus minimizing the number of failures the user must attend to.

5.4 Overhead

Deploying Sympathy with ESS creates overhead both in terms of extra network load and binary code size.

The network load added by Sympathy primarily depends on the metric period. Figure 12 shows that using a smaller metric period also puts much more load on the network. This graph shows ECDFs of the fraction of network traffic taken by Sympathy; smaller ratios represent less overhead, so lines towards the left are better. We consider both a conservative case, in which there is only Sympathy and routing-layer traffic, and a more realistic scenario, in which we add application traffic. This application traffic consists of one

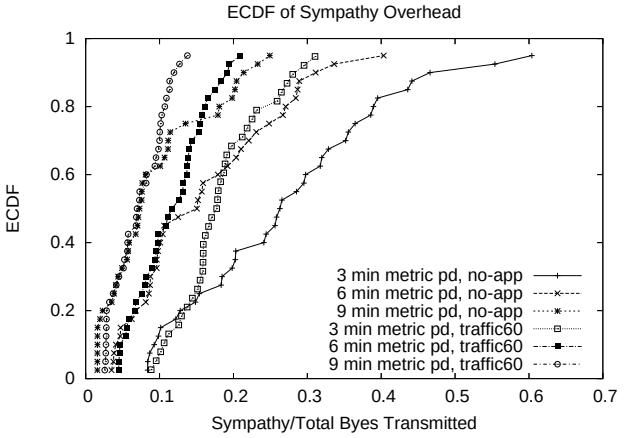

Figure 12: Sympathy overhead correlates with how often nodes send metrics. Each line is run under a specific scenario; each point represents the ratio of Sympathy bytes transmitted over total bytes transmitted, from one particular node.

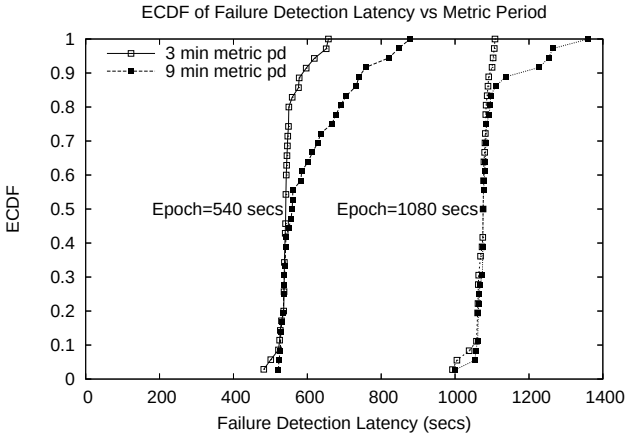

Figure 13: Failure detection latency for two different metric periods (3 and 9 minutes), each at epochs of 540 and 1080 seconds. As the metric period increases (reducing Sympathy traffic overhead), node crash failure detection latency is not significantly impacted.

10-byte data packet sent every minute; this is conservative as most data collection applications would send more data. Data points on top represent nodes that are 1 to 2 hops to the sink, and thus must forward a lot of traffic. As expected, the 9-minute period has less overhead than the 3-minute period; but in all cases with application traffic, the overhead is less than or equal to 31% of data traffic.

Figure 13 shows the time to detect a node crash given metric periods of 3 and 9 minutes, and an epoch period of 540 and 1080 seconds. Although the packet overhead is reduced by a factor of three by tripling the metric period from 3 to 9 minutes, Sympathy's failure detection latency is not significantly impacted. This is because notification latency for this particular failure is primarily impacted by epoch, not redundant metric updates from nodes; other types of failures would show somewhat different results. Experiments with more frequent metric transmission are more likely to have detection latencies of exactly an epoch, as the sink is more likely to receive at least one metrics packet from a live node during any given epoch.

Sympathy consumes minimal RAM and code space on the TinyOS platform. Table 14 contains the increase when instrumenting an application with Sympathy. Using the default mode of Sympathy results in a 47-byte increase in RAM; if the *ping-about* functionality is added the RAM consumption increases to 97 bytes (in order to store 5 bytes of additional information for each of 10 neighbors—the neighbor table size is configurable).

Binary	**RAM**	**ROM**
Sympathy	47 bytes	1558 bytes
Sympathy with *ping-about*	97 bytes	1900 bytes

Figure 14: Sympathy memory footprint for TinyOS on mica2.

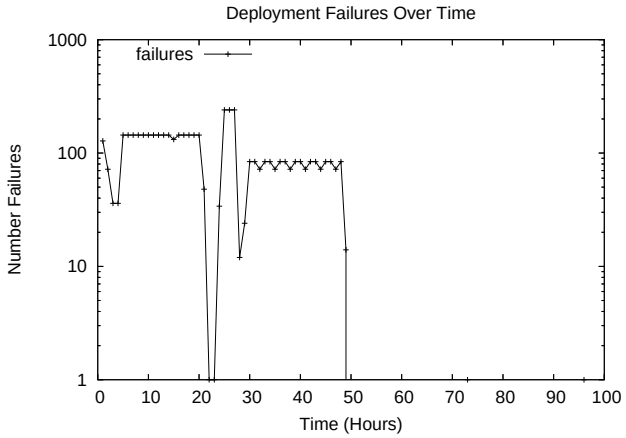

Figure 15: Failures detected during a deployment. The initial deployment fails with excessive failure notifications; the problem is completely fixed after the 49th hour.

5.5 Anecdotal Experience

Sympathy has been used to deploy and monitor two ongoing deployments at James Reserve [10] and several short-term local deployments. We briefly discuss our debugging experiences.

To verify our system, we physically deployed and tested in Malibu, California. We ran for 100 hours with 15 nodes randomly spread around a backyard. The deployment was relatively smooth. The graph of the number of failures reported by Sympathy is shown in Figure 15 (initial failures are due to network setup and initialization). However, one hour after leaving the site we stopped getting data from nodes. Sympathy had reported that all of the nodes had failed and root-caused the failure as the sink did not have any neighbors, localizing the source to the sink. This had resulted from a last minute move of the sink, which prevented the rest of the network from hearing it. Moving the sink back at the 22nd hour immediately fixed the system. At that time however we also updated the sink software to record more data. The new sink software had a bug, and so the sink stopped advertising routes to the network; within a short period of time the nodes routes timed out and they again stopped routing data to the sink. After the 49th hour we finally fixed the problem and Sympathy no longer detected faults.

In another sensor network deployment of 25 nodes at James Reserve, gradually over a short period of time the sink stopped receiving data from all nodes. Sympathy notified us that starting at 9AM that morning no node had a route to the correct sink, which explained why the sink had stopped receiving data. On examining the metrics we saw that nodes *did* have a route to another node ID, which happened to correspond with a node in the network, that was not a sink. After some time, nodes lost even that "route", and remained in a state without a route. Given the data, we hypothesized that the network was exposed temporarily to an extraneous node with an ID that matched that of a node already in the network. That

node advertised itself as a sink, confusing the network. We were able to validate this hypothesis and determine that, due to a bug in the routing code, the network never recovered.

6 GENERALIZABILITY

Sympathy is designed for a specific but very broad class of applications that gather data over a single or multihop tree-based routing structure from many nodes at a sink. However, Sympathy can be modified to apply to different applications and routing layers. For example, an application that does not require regular data exchange between nodes, but has a notion of the amount of data it expects at a sink, would work with Sympathy's current implementation. For an event-based application with no regular data exchange, the user has two options: use Sympathy in its current implementation and rely on the transport of Sympathy metrics to ensure the continual health of the system (requiring pre-emptive as opposed to the more ideal on-demand routing), or turn off regular metric transmissions and only receive metrics when the user triggers metric collection.

Sympathy is not intimately tied to any specific routing layer and can use any routing protocol that provides a route from a node to the sink. Its path analysis and some root causes, such as Node Crash, are based on the assumption that neighbor and routing tables are maintained by each node. However, many non-tree based routing algorithms such as rumor routing [1] still maintain routing and neighbor tables, which is all that Sympathy requires in order to detect and root-cause failures and analyze paths for source localization.

Adding an additional metric requires approximately 10 extra lines of code at the node and another 10 at the sink. Any additional RAM consumption depends solely on the size of the metric.

7 FUTURE WORK

One of the interesting challenges that has arisen through the design of Sympathy is handling unknown network state. In other words, how much can Sympathy infer based on the information it does have?

For example, during path analysis, if a node's route has timed out at Sympathy (because Sympathy has not gotten any information on the node's route in the last epoch), then can Sympathy use an outdated route in determining failure dependencies? One possible scenario could be that Sympathy is not receiving updated route information from the node due to a failure along the path; in this case, Sympathy should use the outdated route information to trace the path to the root cause. However, perhaps the route really is outdated and in the meantime the node has switched routes: another failure on a different path is actually preventing the data delivery. Similarly, Sympathy determines a node has failed if it has fallen off of all nodes' neighbor lists, but it is simply guessing that if no node has heard from a node, that node must dead. It may just be that the node has dropped off of neighbor lists but is still periodically transmitting packets.

In such instances, it would be helpful to have some mechanism to specify confidence in the root cause analysis provided by Sympathy (i.e. using a Bayes network). In the case above, Sympathy could then provide two scenarios, each with a confidence estimate (possibly depending on the duration of time elapsed since Sympathy had last heard from the node; as time passed, the confidence in the first diagnosis would decrease).

Another approach we plan to explore is placing more emphasis on decentralized processing of *some* of the metrics, so that the sink does not always have to have the most updated state of the network. Decentralized processing would also allow nodes to pro-actively send event notifications or updated metrics back to the sink, allowing for less frequent transmission of metrics packets. Future work will include examining which events should trigger a metric delivery to the sink in order to trigger failure detection.

Finally, giving the sink a more proactive role in the debugging process would also aid in handling insufficient or outdated information. For example, if the sink is unsure of the current state of a node, and needs to know what the node's route is, the sink should be able to request this information from the node in order to complete its path analysis.

8 CONCLUSION

Debugging sensor networks can be a long and laborious process. We have therefore developed Sympathy, a tool for automatically diagnosing and aiding in the debugging of sensor network systems. Even with a small code footprint that sends out minimal metric data, we can still deduce the status of the system and in many cases narrow down or even pinpoint the exact cause of a system failure. Our prototype uses straightforward data collection nodes to send and analyze simple metrics, and is capable of helping debuggers find failures more quickly and easily compared to current approaches.

Documentation and download links for this software are freely available at http://lecs.cs.ucla.edu/~nithya/sympathy.

ACKNOWLEDGMENTS

This material is based in part upon work supported by the National Science Foundation under Grant No. 0435497. The authors would like to thank Jennifer Gillenwater for her work on the web interface for Sympathy and the reviewers for their comments, many of which improved the design and performance of Sympathy.

REFERENCES

[1] D. Braginsky and D. Estrin. Rumor routing algorithm for sensor networks. In *Proc. 1st ACM Workshop on Sensor Networks*, 2002.

[2] K. Chang, N. Ramanathan, J. Palsberg, and D. Estrin. Deployment Analysis System. In *SenSys 2005 Demo*, November 2005.

[3] J. Feng, S. Megerian, and M. Potkonjak. Model-based calibration for sensor networks. In *IEEE International Conference on Sensors*, October 2003.

[4] A. Fox, E. Kiciman, D. Patterson, M. Jordan, and R. Katz. Combining statistical monitoring and predictable recovery for self-management. In *Proc. Workshop on Self-Managed Systems*, October 2004.

[5] 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 *Proc. USENIX*, Boston, MA, 2004. USENIX. To appear.

[6] L. Girod, T. Stathopoulos, N. Ramanathan, and D. Estrin. Tools for deployment and simulation of heterogeneous sensor networks. Technical report, April 2004.

[7] G. Goldszmidt and Y. Yemini. Distributed management by delegating mobile agents. In *The 15th International Conference on Distributed Computing Systems*, June 1995.

[8] B. Greenstein, T. Schoellhammer, N. Xu, L. Girod, T. Stathopoulos, M. Wimborow, and et al. M. Taggart. The Extensible Sensing System testbed, 2004.

[9] SAS Institute, editor. *SAS/OR Software: Changes and Enhancements*. SAS Institute, Inc, 2005.

[10] R. Kapur, T. Schoelhammer, and N. Ramanathan. Lessons from a James Reserve deployment, March 2005.

[11] E. Kiciman and A. Fox. Detecting application-level failures in component-based Internet services. In *IEEE Transactions on Neural Networks*, Spring 2005.

[12] P. Levis, D. Gay, and D. Culler. Active sensor networks. In *Proc. NSDI*, 2005.

[13] P. Levis, N. Lee, M. Welsh, and D. Culler. TOSSIM: Accurate and scalable simulation of entire TinyOS applications. In *Proc. SenSys*, 2003.

[14] N. Ramanathan, E. Kohler, and D. Estrin. Towards a debugging system for sensor networks. *International Journal for Network Management*, 2005.

[15] N. Ramanathan, E. Kohler, L. Girod, and D. Estrin. Sympathy: A debugging system for sensor networks. In *Proc. EmNets-I*, 2004.

[16] Y. Ruan and V. Pai. Making the "box" transparent: System call performance as a first-class result. In *Proc. USENIX*, Boston, MA, 2004. USENIX.

[17] C. Sharp, S. Shaffert, A. Woo, N. Sastry, C. Karlof, S. Sastry, and D. Culler. Design and implementation of a sensor network system for vehicle tracking and autonomous interception. In *Proc. EWSN*, 2005.

[18] V. Shnayder, M. Hempstead, B. Chen, and M. Welsh. PowerTOSSIM: Efficient power simulation for TinyOS applications. In *Proc. SenSys*, 2004.

[19] R. Szewczyk, J. Polastre, A. Mainwaring, and D. Culler. Lessons from a sensor network expedition. In *Proc. EWSN*, January 2004.

[20] G. Tolle and D. Culler. SNMS: Application-cooperative management for wireless sensor networks. In *Proc. SenSys*. ACM, 2004.

[21] G. Tolle, J. Polastre, R. Szewczyk, D. Culler, N. Turner, K. Tu, S. Burgess, T. Dawson, P. Buonadonna, D. Gay, and W. Hong. A macroscope in the redwoods. In *Proc. SenSys*, November 2005.

[22] A. Woo, T. Tong, and D. Culler. Taming the underlying challenges of reliable multihop routing in sensor networks. In *Proc. SenSys*, 2003.

[23] J. Zhao, R. Govindan, and D. Estrin. Residual energy scans for monitoring wireless sensor networks. In *Proc. IEEE Wireless Communications and Networking Conference*, Florida, 2002. IEEE.

[24] J. Zhao, R. Govindan, and D. Estrin. Computing aggregates for monitoring wireless sensor networks. In *Proc. IEEE ICC Workshop on Sensor Network Protocols and Applications*, Anchorage, AK, 2003. IEEE.