

# Dustminer: Troubleshooting Interactive Complexity Bugs in Sensor Networks

Mohammad Maifi Hasan Khan, Hieu Khac Le, Hossein Ahmadi, Tarek F. Abdelzaher,  
and Jiawei Han

Department of Computer Science  
University of Illinois, Urbana-Champaign

mmkhan2@uiuc.edu, hieule2@cs.uiuc.edu, hahmadi2@uiuc.edu, zaher@cs.uiuc.edu  
hanj@cs.uiuc.edu

## ABSTRACT

This paper presents a tool for uncovering bugs due to interactive complexity in networked sensing applications. Such bugs are not localized to one component that is faulty, but rather result from complex and unexpected interactions between multiple often individually non-faulty components. Moreover, the manifestations of these bugs are often not repeatable, making them particularly hard to find, as the particular sequence of events that invokes the bug may not be easy to reconstruct. Because of the distributed nature of failure scenarios, our tool looks for *sequences* of events that may be responsible for faulty behavior, as opposed to localized bugs such as a bad pointer in a module. An extensible framework is developed where a front-end collects runtime data logs of the system being debugged and an off-line back-end uses frequent discriminative pattern mining to uncover likely causes of failure. We provide a case study of debugging a recent multichannel MAC protocol that was found to exhibit corner cases of poor performance (worse than single channel MAC). The tool helped uncover event sequences that lead to a highly degraded mode of operation. Fixing the problem significantly improved the performance of the protocol. We also provide a detailed analysis of tool overhead in terms of memory requirements and impact on the running application.

## Categories and Subject Descriptors

D.2.5 [Software Engineering]: [Testing and Debugging-Distributed Debugging]

## General Terms

Design, Reliability, Experimentation

## Keywords

Protocol debugging, Distributed automated debugging, Wireless sensor networks

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SenSys'08, November 5–7, 2008, Raleigh, North Carolina, USA.  
Copyright 2008 ACM 978-1-59593-990-6/08/11 ...\$5.00.

## 1. INTRODUCTION

*DustMiner* is a diagnostic tool that leverages an extensible framework for uncovering root causes of failures and performance anomalies in wireless sensor network applications in an automated way. This paper presents the design and implementation of *Dustminer* along with two case studies of real life failure diagnosis scenarios. The goal of this work is to further contribute to automating the process of debugging, instead of relying only on manual efforts, and hence reduce the development time and effort significantly.

Developing wireless sensor network applications still remains a significant challenge and a time consuming task. To make the development of wireless sensor network applications easier, much of the previous work focused on programming abstractions [27, 7, 25, 17, 11]. Most wireless sensor network application developers would agree, however, to the fact that most of the development time is spent on debugging and troubleshooting the current code, which greatly reduces productivity.

Early debugging and troubleshooting support revolved around testbeds [8, 36], simulation [18, 35, 34] and emulation environments [29, 12]. Recent source level debugging tools [38, 37] have greatly contributed to the convenience of the troubleshooting process. They make it easier to zoom in on sources of errors by offering more visibility into run-time state. Unfortunately wireless sensor network applications often fail not because of a single node coding error but as a result of improper interaction between components. Such interaction may be due to some protocol design flaw (e.g., missed corner cases that the protocol does not handle correctly) or unexpected artifacts of component integration. Interaction errors are often non-reproducible since repeating the experiment might not lead to the same corner-case again. Hence, in contrast to previous debugging tools, in this paper, we focus on finding (generally non-deterministically occurring) bugs that arise from interactions among seemingly individually sound components.

The main approach of *Dustminer* is to log many different types of events in the sensor network and then analyze the logs in an automated fashion to extract the sequences of events that lead to failure. These sequences shed light on what caused the bug to manifest making it easier to understand and fix the root cause of the problem.

Our work extends a somewhat sparse chain of prior attempts at diagnostic debugging in sensor networks. Sympathy [30] is one of the earliest tools in the wireless sensor networks domain that addresses the issue of diagnosing fail-

ures in deployed systems in an automated fashion. Specifically, it infers the identities of failed nodes or links based on reduced throughput at the base station. SNTS [15] provides a more general diagnostic tool that extracts conditions on current network state that are correlated with failure, in hopes that they may include the root cause. A diagnostic simulator (an extension to TOSSIM) is described in [14]. Being a simulator extension, it is not intended to troubleshoot deployed systems. It is based on frequent pattern mining (to find event patterns that occur frequently when the bugs manifest). Unfortunately, the cause of a problem is often an infrequent pattern; a single “bad” chain of events that leads to many problems.

We extend the diagnostic capability of the above tools by implementing a new automated discriminative sequence analysis technique that relies on two separate phases to find root causes of problems or performance anomalies in sensor networks; the first phase identifies frequent patterns correlated to failure as before. The second focuses on those patterns, correlating them with (infrequent) events that may have caused them, hence uncovering the true root of the problem. We apply this technique to identify the sequences of events that cause manifestations of interaction bugs. In turn, identifying these sequences helps the developer understand the nature of the bug. Our tool is based on a front-end that collects data from the run-time system and a back-end that analyses it. Both are plug-and-play modules that can be chosen from multiple alternatives. We identify the architectural requirements of building such an extensible framework and present a modular software architecture that addresses these requirements.

We provide two case studies of real life debugging using the new tool. The first case study shows how our tool isolates a kernel-level bug in the radio communication stack in the newly introduced LiteOS [6] operating system for sensor networks that offers a UNIX-like remote filesystem abstraction. The second case study shows how the tool is used to debug a performance problem in a recently published multichannel Media Access Control (MAC) protocol [16]. For both case studies we used MicaZ motes as target hardware.

It is important to stress what our tool is not. We specialize in uncovering problems that result from component interactions. Specifically, the proposed tool is not intended to look for local errors (*e.g.*, code errors that occur and manifest themselves on one node). Examples of the latter type of errors include infinite loops, dereferencing invalid pointers, or running out of memory. Current debugging tools are, for the most part, well-equipped to help find such errors. An integrated development environment can use previous tools for that purpose.

The rest of the paper is organized as follows. In Section 2, we describe recent work on debugging sensor network applications. In Section 3, we describe the main idea of our tool. In Section 4, we describe the design challenges with our proposed solution. Section 5 describes the system architecture of *Dustminer*. Section 6 describes the implementation details of the data collection front-end and data analysis back-end that are used in this paper along with their system overhead. To show the effectiveness of our tool, in Section 7, we provided two case studies. In the first, we uncover a kernel-level bug in LiteOS. In the second, we use our tool to identify a protocol design bug in a recent multi channel MAC protocol [16]. We show that when the bug is

fixed, the throughput of the system is improved by nearly 50%. Section 8 concludes the paper.

## 2. RELATED WORK

Current troubleshooting support for sensor networks (i) favors reproducible errors, and (ii) is generally geared towards finding *local* bugs such as an incorrectly written line of code, an erroneous pointer reference, or an infinite loop. Existing tools revolve around testing, measurements, or stepping through instruction execution. Marionette [37] and Clairvoyant [38] are examples of source debugging systems that allow the programmer to interact with the sensor network using breakpoints, watches, and line-by-line tracing. Source level debugger is more suitable to identify programming errors which is contained in a single node. It is difficult to find distributed bugs using source level debugger due to the fact that source level debugging interferes heavily with the normal operation of the code and may prevent the excitation of distributed bugs in the first place. It also involves manual checking of system states which is not scalable. SNMS [32] presents a sensor network measurement service that collects performance statistics such as packet loss and radio energy consumption. Testing-based systems include laboratory testbeds such as Motelab [36], Kansei [8], Emstar [12] etc. These systems are good at exposing manifestations of errors, but leave it to the programmer’s skill to guess the cause of the problem.

Simulation and emulation based systems include TOSSIM [18], DiSenS [35], S<sup>2</sup>DB [34], Atemu [29] etc. Atemu provides XATDB which is a GUI based debugger that provides interface to debug code at line level. S<sup>2</sup>DB is a simulation based debugger that provides debugging abstractions at different levels such as the node level and network level. It provides the concept of parallel debugging where a developer can set breakpoints across multiple devices to access internal system state. This remains a manual process, and it is very hard to debug a large system manually for design bugs. Moreover the simulated environment prevents the system from exciting bugs which arise due to peculiar characteristics of real hardware, and deployment scenarios such as clock skew, radio irregularities, and sensing failures, to name a few.

For offline parameter tuning and performance analysis, *record and replay* is a popular technique which is implemented by Envirolog [26] for sensor network applications. Envirolog stores module outputs (*e.g.*, outputs of sensing modules) locally and replays them to reproduce the original execution. It is good at testing performance of high-level protocols subjected to a given recorded set of environmental conditions. While this can also help with debugging such protocols, no automated diagnosis support is offered.

In [31], the authors pointed out that sensor data, such as humidity and temperature, may be used to predict network and node failure, although they do not propose an automated technique for correlating failures with sensor data. In contrast, we are focused on automating the troubleshooting of general protocol bugs which may not necessarily be revealed by analyzing sensor measurements.

*Sympathy* [30] presents an early step towards sensor network self-diagnosis. It specializes in attributing reduced communication throughput at a base-station to the failure of a node or link in the sensor network. Another example of automated diagnostic tools is [15] which analyzes pas-

sively logged radio communication messages using a classification algorithm [10] to identify states correlated with the occurrence of bugs. The diagnostic capability of [15] is constrained by its inability to identify event *sequences* that precipitate an interaction-related bug. The tool also does not offer an interface to the debugged system that allows logging internal events. The recently published diagnostic simulation effort [14] comes closest to our work. The present paper extends the diagnostic capability of the above tools by implementing an actual system (as opposed to using simulation) and presenting a better log analysis algorithm that is able to uncover *infrequent* sequences that lead to failures.

Machine learning techniques have previously been applied to failure diagnosis in other systems [5, 3, 22]. Extensive work in software bug mining and analysis studies includes a software behavior graph analysis method for back-tracing noncrashing bugs [22], a statistical model-based bug localization method [19], which explores the distinction of runtime execution distribution between buggy and nonbuggy programs, a control flow abnormality analysis method for logic error isolation [23], a fault localization-based approach for failure proximity [20], a Bayesian analysis approach for localization of bugs with a single buggy run, which is especially valuable for distributed environments where the bug can hardly be reproduced reliably [21], and a dynamic slicing-based approach for failure indexing which may cluster and index bugs based on their speculated root cause and locality [24], as well as discriminative frequent pattern analysis [9]. We extend the techniques of discriminative pattern analysis to a two-stage process for analyzing logs to pinpoint the cause of failure; the first stage identifies all the “symptoms” of the problem from the logs, while the second ties them to possible root causes.

Formal methods [33, 13, 4, 28] offer a different alternative based on verifying component correctness or, conversely, identifying which properties are violated. The approach is challenging to apply to large systems due to the high degree of concurrency and non-determinism in complex wireless sensor networks, which leads to an exponential state space explosion. Unreliable wireless communication and sensing pose additional challenges. Moreover, even a verified system can suffer poor performance due to a design flaw. Our tool automatically answers questions that help identify the cause of failure that manifests during run time.

### 3. DUSTMINER OVERVIEW

Most previous debugging approaches for sensor networks are geared at finding localized errors in code (with preference to those that are reproducible). In contrast, we focus on non-deterministically occurring errors that arise because of unexpected or adverse distributed interactions between multiple seemingly individually-correct components. The non-localized, hard-to-reproduce nature of such errors makes them especially hard to find.

*Dustminer* is based on the idea that, in a distributed wireless sensor network, nodes interact with each other in a manner defined by their distributed protocols to perform cooperative tasks. Unexpected sequences of events, subtle interactions between modules, or unintended design flaws in protocols may occasionally lead to an undesirable or invalid state, causing the system to fail or exhibit poor performance. Hence, in principle, if we log different types of events

in the network, we may be able to capture the unexpected sequence that leads to failure (along with thousands of other sequences of events). The challenge for the diagnostic tool is to automatically identify this culprit sequence. Our approach *exploits* both (i) non-determinism and (ii) interactive complexity to *improve* ability to diagnose distributed interaction bugs. This point is elaborated below:

- *Exploiting non-reproducible behavior*: We adapt data mining approaches that use examples of both “good” and “bad” system behavior to be able to classify the conditions correlated with good and bad. In particular, note that conditions that cause a problem to occur are correlated (by causality) with the resulting bad behavior. Root causes of non-reproducible bugs are thus inherently suited for discovery using such data mining approaches; the lack of reproducibility itself and the inherent system non-determinism improve the odds of occurrence of sufficiently diverse behavior examples to train the troubleshooting system to understand the relevant correlations and identify causes of problems.
- *Exploiting interactive complexity*: Interactive complexity describes a system where scale and complexity cause components to interact in unexpected ways. A failure that occurs due to such unexpected interactions is typically hard to “blame” on any single component. This fundamentally changes the objective of a troubleshooting tool from aiding in stepping through code (which is more suitable for finding a localized error in some line, such as an incorrect pointer reference), to aiding with diagnosing a *sequence of events* (component interactions) that leads to a failure state.

At a high level, our tool first uses a data collection front-end to collect runtime events for post-mortem analysis. Once the log of runtime events is available, the tool separates the collected sequence of events into two piles - a “good” pile, which contains the parts of the log when the system performs as expected, and a “bad” pile, which contains the parts of the log when the system fails or exhibits poor performance. This data separation phase is done based on a predicate that defines “good” versus “bad” behavior, provided by the application developer. For example the predicate, applied offline to logged data, might state that a sequence of more than 10 consecutive lost messages between a sender and receiver is bad behavior (hence return “bad” in this case). To increase diagnostic accuracy, experiments can be run multiple times before data analysis.

A discriminative frequent pattern mining algorithm then looks for patterns (sequences of events) that exist with very different frequencies in the two piles. These patterns are called *discriminative*. Later, such patterns are analyzed for correlations with preceding events in the logs, if any, that occur less frequently. Hence, it is possible to catch anomalies that cause problems as well as common roots of multiple error manifestations.

A well-known algorithm for finding frequent patterns in data is the Apriori algorithm [2]. This algorithm was used in previous work on sensor network debugging [14] to address a problem similar to ours where Apriori algorithm is used to determine the event sequences that lead to problems in sensor networks. We show that the approach has serious limitations and extend this algorithm to suite purposes of

sensor network debugging. The original Apriori algorithm, used in the aforementioned work, is an iterative algorithm that proceeds as follows. At the first iteration, it counts the number of occurrences (called *support*) of each distinct event in the data set (i.e., in the “good” or “bad” pile). Next, it discards all events that are infrequent (their support is less than some parameter *minSup*). The remaining events are frequent patterns of length 1. Assume the set of frequent patterns of length 1 is  $S_1$ . At the next iteration, the algorithm generates all the candidate patterns of length 2 which is  $S_1 \times S_1$ . Here ‘ $\times$ ’ represents the Cartesian product. It then computes the frequency of occurrence of each pattern in  $S_1 \times S_1$  and discards those with support less than *minSup* again. The remaining patterns are the frequent patterns of length 2. Let us call them set  $S_2$ . Similarly, the algorithm will generate all the candidate patterns of length 3 which is  $S_2 \times S_1$  and discard infrequent patterns (with support less than *minSup*) to generate  $S_3$  and so on. It continues this process until it cannot generate any more frequent patterns.

We show in this paper, how the previous work is extended for purposes of diagnosis. The problems with the algorithm and our proposed solutions are described in section 4.

## 4. CHALLENGES AND SOLUTIONS

Performing discriminative frequent pattern mining based on frequent patterns generated by Apriori algorithm poses several challenges that need to be addressed before we can apply discriminative frequent pattern mining for debugging. For the purposes of the discussion below, let us define an *event* to be the basic element in the log that is analyzed for failure diagnosis. The structure of an event in our log is as follows:  $\langle NodeId, EventType, attribute_1, attribute_2, \dots, attribute_n, Timestamp \rangle$

*NodeId* is used to identify the node that records the event. *EventType* is used to identify the event type (e.g., message dropped, flash write finished, etc). Based on the event type, it is possible to interpret the rest of the record (the list of attributes). The set of distinct *EventTypes* is often called the *alphabet* in an analogy with strings. In other words, if events were letters in an alphabet, we are looking for strings that cause errors to occur. These strings represent event *sequences* (ordered lists of events). The generated log can be thought of as a single sequence of logged events. For example,  $S_1 = (\langle a \rangle, \langle b \rangle, \langle b \rangle, \langle c \rangle, \langle d \rangle, \langle b \rangle, \langle b \rangle, \langle a \rangle, \langle c \rangle)$  is an event sequence. Elements  $\langle a \rangle, \langle b \rangle, \dots$ , are events. A discriminative pattern between two data sets is a subsequence of (not necessarily contiguous) events that occurs with a different count in the two sets. The larger the difference, the better the discrimination. With the above terminology in mind, we present how the algorithm is extended to apply to debugging.

### 4.1 Preventing False Frequent Patterns

The Apriori algorithm generates all possible combinations of frequent subsequences of the original sequence. As a result, it generates subsequences combining events that are “too far” apart to be causally correlated with high probability and thus reduces the chance of finding the “culprit sequence” that actually caused the failure. This strategy could negatively impact the ability to identify discriminative patterns in two ways; (i) it could lead to the generation

of discriminative patterns that are not causally related, and (ii) it could eliminate discriminative patterns by generating false patterns. Consider the following example.

Suppose we have the following two sequences:

$$S_1 = (\langle a \rangle, \langle b \rangle, \langle c \rangle, \langle d \rangle, \langle a \rangle, \langle b \rangle, \langle c \rangle, \langle d \rangle)$$

$$S_2 = (\langle a \rangle, \langle b \rangle, \langle c \rangle, \langle d \rangle, \langle a \rangle, \langle c \rangle, \langle b \rangle, \langle d \rangle)$$

Suppose the system fails when  $\langle a \rangle$  is followed by  $\langle c \rangle$  before  $\langle b \rangle$ . As this condition is violated in sequence  $S_2$ , ideally, we would like our algorithm to be able to detect  $(\langle a \rangle, \langle c \rangle, \langle b \rangle)$  as a discriminative pattern that distinguishes these two sequences.

Now, if we apply the Apriori technique, it will generate  $(\langle a \rangle, \langle c \rangle, \langle b \rangle)$  as an equally likely pattern for both  $S_1$ , and  $S_2$ . As in both  $S_1$  and  $S_2$ , it will combine the first occurrence of  $\langle a \rangle$  and the first occurrence of  $\langle c \rangle$  with the second occurrence of  $\langle b \rangle$ . So it will get canceled out at the differential analysis phase.

To address this issue, the key observation here is that the first occurrence of  $\langle a \rangle$  should not be allowed to combine with the second occurrence of  $\langle b \rangle$  as there is another event  $\langle a \rangle$  after the first occurrence of  $\langle a \rangle$  but before the second occurrence of  $\langle b \rangle$  and the second occurrence of  $\langle b \rangle$  is correlated with second occurrence of  $\langle a \rangle$  with higher probability.

To prevent such erroneous combinations, we use a *dynamic search window* scheme where the first item of any candidate sequence is used to determine the search window. In this case, for any pattern starting with  $\langle a \rangle$ , the search window is  $[1, 4]$  and  $[5, 8]$  in  $S_1$  and  $S_2$ . With this search window, the algorithm will search for pattern  $(\langle a \rangle, \langle c \rangle, \langle b \rangle)$  in window  $[1, 4]$  and  $[5, 8]$  and will fail to find it in  $S_1$  but will find it in sequence  $S_2$  only. As a result, the algorithm will be able to report pattern  $(\langle a \rangle, \langle c \rangle, \langle b \rangle)$  as a discriminative pattern.

This *dynamic search window* scheme also speeds up the search significantly. In this scheme, the original pattern (of size 8 events) was reduced to windows of size 4 making the search for patterns in those windows more efficient.

### 4.2 Suppressing Redundant Subsequences

At the frequent pattern generation stage, if two patterns,  $S_i$  and  $S_j$ , have *support*  $\geq minSup$ , the Apriori algorithm keeps both sequences as frequent patterns even if one is a subsequence of the other and both have equal support. This makes perfect sense in data mining but not in debugging. For example, when mining the “good” data set, the above strategy assumes that any subset of a “good” pattern is also a good pattern. In real-life, this is not true. Forgetting a step in a multi-step procedure may well cause failure. Hence, subsequences of good sequences are not necessarily good. Keeping these subsequences as examples of “good” behavior leads to a major problem at the differential analysis stage when discriminative patterns are generated since they may incorrectly cancel out similar subsequences found frequent in the other (i.e., “bad” behavior) data pile. For example, consider two sequences below:

$$S_1 = (\langle a \rangle, \langle b \rangle, \langle c \rangle, \langle d \rangle, \langle a \rangle, \langle b \rangle, \langle c \rangle, \langle d \rangle)$$

$$S_2 = (\langle a \rangle, \langle b \rangle, \langle c \rangle, \langle d \rangle, \langle a \rangle, \langle b \rangle, \langle d \rangle, \langle c \rangle)$$

Suppose, for correct operation of the protocol, event  $\langle a \rangle$  has to be followed by event  $\langle c \rangle$  before event  $\langle d \rangle$  can happen. In sequence  $S_2$  this condition is violated. Ideally, we would like our algorithm to report sequence:

$S_3 = \langle a \rangle, \langle b \rangle, \langle d \rangle$  as the “culprit” sequence. However, if we apply Apriori algorithm, it will fail to catch this sequence. This is because it will generate  $S_3$  as a frequent pattern both for  $S_1$  and  $S_2$  with support 2 and will get canceled out at the differential analysis phase. As expected,  $S_3$  will never show up as a “discriminative pattern”. Note that with the *dynamic search window* scheme alone, we cannot prevent this.

To illustrate, suppose a successful message transmission involves the following sequence of events:

$\langle enableRadio \rangle, \langle messageSent \rangle, \langle ackReceived \rangle, \langle disableRadio \rangle$

Now although sequence:

$\langle enableRadio \rangle, \langle messageSent \rangle, \langle disableRadio \rangle$

is a subsequence of the original “good” sequence, it does not represent a successful scenario as it disables radio before receiving the “ACK” message.

To solve this problem, we need an extra step (which we call *sequenceCompression*) before we perform differential analysis to identify discriminative patterns. At this step, we remove the sequence  $S_i$  if it is a subsequence of  $S_j$  with the same *support*<sup>1</sup>. This will remove all the redundant subsequences from the frequent pattern list. Subsequences with a (sufficiently) different support, will be retained and will show up after discriminative pattern mining.

In the above example, pattern  $\langle a \rangle, \langle b \rangle, \langle c \rangle, \langle d \rangle$  has *support* 2 in  $S_1$  and *support* 1 in  $S_2$ . Pattern  $\langle a \rangle, \langle b \rangle, \langle d \rangle$  has *support* 2 in both  $S_1$  and  $S_2$ . Fortunately, at the *sequenceCompression* step, pattern  $\langle a \rangle, \langle b \rangle, \langle d \rangle$  will be removed from the frequent pattern list generated for  $S_1$  because it is a subsequence of a larger frequent pattern of the same support. It will therefore remain only on the frequent pattern list generated for  $S_2$  and will show up as a discriminative pattern.

### 4.3 Two Stage Mining for Infrequent Events

In debugging, sometimes less frequent patterns could be more indicative of the cause of failure than the most frequent patterns. A single mistake can cause a damaging sequence of events. For example, a single node reboot event can cause a large number of message losses. In such cases, if frequent patterns are generated that are commonly found in failure cases, the most frequent patterns may not include the real cause of the problem. For example, in case of node reboot, manifestation of the bug (message loss event) will be reported as the most frequent pattern and the real cause of the problem (the node reboot event) may be overlooked.

Fortunately, in the case of sensor network debugging, a solution may be inspired by the nature of the problem domain. The fundamental issue to observe is that much computation in sensor networks is *recurrent*. Code repeatedly visits the same states (perhaps not strictly periodically), repeating the same actions over time. Hence, a single problem, such as a node reboot or a race condition that pollutes a data structure, often results in multiple manifestations of the same unusual symptom (like multiple subsequent message losses or multiple subsequent false alarms). Catching these recurrent symptoms by an algorithm such as Apriori is much easier due to their larger frequency. With such symptoms identified, the search space can be narrowed and it becomes

<sup>1</sup>This mechanism can be extended to remove subsequences of a similar but not identical support.

easier to correlate them with other less frequent preceding event occurrences. To address this challenge, we developed a two stage pattern mining scheme.

At the first stage, the Apriori algorithm generates the usual frequent discriminative patterns that have support larger than *minSup*. For the first stage, *minSup* is set larger than 1. It is expected that the patterns involving manifestations of bugs will survive at the end of this stage but infrequent events like a node reboot will be dropped due to their low support.

At the second stage, at first, the algorithm splits the log into fixed width segments (default width is 50 events in our implementation). Next, the algorithm counts the number of discriminative frequent patterns found in each segment and ranks each segment of the log based on the count (the higher the number of discriminative patterns in a segment, the higher the rank). If discriminative patterns occurred consecutively in multiple segments, those segments are merged into a larger segment. Next, the algorithm generates frequent patterns with *minSup* reduced to 1 on the  $K$  highest-ranked segments separately (default  $K$  is 5 in our implementation) and extracts the patterns that are common in these regions. Note that the initial value of  $K$  is set conservatively. The optimum value of  $K$  depends on the application. If with the initial value of  $K$ , the tool failed to catch the real cause, the value of  $K$  is increased iteratively. In this scheme, we have a higher chance of reporting single events such as race conditions that cause multiple problematic symptoms. Observe that the algorithm is applied on data that is the total logs from several experimental runs. The race condition may have occurred once at different points of some of these runs.

This scheme has a significant impact on the performance of the frequent pattern mining algorithm. Scalability is one of the biggest challenges in applying discriminative frequent pattern analysis to debugging. For example, if the total number of logged events is of the order of thousands (more than 40000 in one of our later examples), it is computationally impossible to generate frequent patterns of non-trivial length for this whole sequence. Using two stage mining, we can dramatically reduce the search space and make it feasible to mine for longer frequent patterns which are more indicative of the cause of failure than shorter sequences.

### 4.4 Other Challenges

Several other changes need to be made to standard data mining techniques. For example, the amount of logged events and the corresponding frequency of patterns can be different from run to run depending on factors such as length of execution and system load. A higher sampling rate at sensors, for example, may generate more messages and cause more events to be logged. Many logged event patterns in this case will appear to be more frequent. This is problematic when it is desired to compare the frequency of patterns found in “good” and “bad” data piles for purposes of identifying those correlated with bad behavior. To address this issue, we need to normalize the frequency count of events in the log. In the case of single events (i.e., patterns of length 1), we use the ratio of occurrence of the event instead of absolute counts. In other words, the *support* of any particular event,  $\langle e \rangle$  in the event log is divided by the total number of events logged, yielding in essence the probability of finding that event in the log,  $P(e)$ . For patterns of length more than

1, we extend the scheme to compute the probability of the pattern given recursively by  $P(e1).P(e2|e1).P(e3|e1.e2), \dots$ . The individual terms above are easy to compute. For example,  $P(e2|e1)$  is obtained by dividing the support of the pattern  $\langle e1 \rangle, \langle e2 \rangle$  by the total support of patterns starting with  $\langle e1 \rangle$ .

Finally, there are issues with handling event parameters. Logged events may have parameters (e.g., identity of the receiver for a message transmission event). Since event parameter lists may be different, calling each variation a different event will cause a combinatorial explosion of the alphabet. For example, an event with 10 parameters, each of 10 possible values will generate a space of  $10^{10}$  possible combinations. To address the problem, continuous or fine-grained parameters need to be discretized into a smaller number of ranges. Multi-parameter events need to be converted into sequences of single-parameter events each listing one parameter at a time. Hence, the exponential explosion is reduced to linear growth in the alphabet, proportional to the number of discrete categories a single parameter can take and the average number of parameters per event. Techniques for dealing with event parameter lists were introduced in [14] and are not discussed further in this paper.

## 5. DUSTMINER ARCHITECTURE

We realize that the types of debugging algorithms needed are different for different applications, and are going to evolve over time with the evolution of hardware and software platforms. Hence, we aim to develop a modular tool architecture that facilitates evolution and reuse. Keeping that in mind, we developed a software architecture that provides the necessary functionality and flexibility for future development. The goal of our architecture is to facilitate easy use and experimentation with different debugging techniques and foster future development. As there are numerous different types of hardware, programming abstractions, and operating systems in use for wireless sensor networks, the architecture must be able to accommodate different combinations of hardware and software. Different ways of data collection should not affect the way the data analysis layer works. Similarly we realize that for different types of bugs, we may need different types of techniques to identify the bug and we want to provide a flexible framework to experiment with different data analysis algorithms. Based on the above requirements, we designed a layered, modular architecture as shown in Figure 1. We separate the whole system into three subsystems; (i) a data collection front-end, (ii) data preprocessing middleware and (iii) a data analysis back-end.

### 5.1 Data Collection Front-End

The role of data collection front-end is to provide the debug information (i.e., log files) that can be analyzed for diagnosing failures. The source of this debug log is irrelevant to the data analysis subsystem. As shown in Figure 1, the developer may choose to analyze the recorded radio communication messages obtained using a passive listening tool, or the execution traces obtained from simulation runs, or the run-time sequences of events obtained by logging on actual application notes and so on. With this separation of concerns, the front-end developer could design and implement the data collection subsystem more efficiently and indepen-

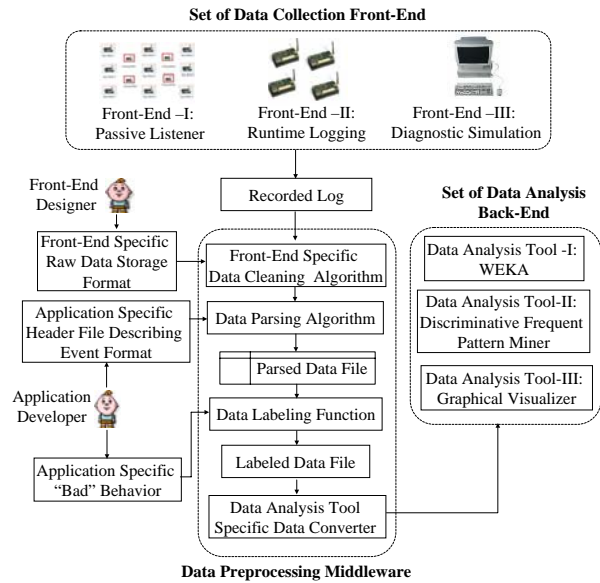


Figure 1: Debugging framework

dently. The data collection front-end developer merely needs to provide the format of the recorded data. These data are used by the data preprocessing middleware to parse the raw recorded byte streams.

### 5.2 Data Preprocessing Middleware

This middleware that sits between the data collection front-end and the data analysis back-end provides the necessary functionality to change or modify one subsystem without affecting the other. The interface between the data collection front-end and the data analysis back-end is further divided into the following layers:

- *Data cleaning layer:* This layer is front-end specific. Each supported front-end will have one instance of it. The layer is the interface between the particular data collection front-end and the data preprocessing middleware. It ensures that the recorded events are compliant with format requirements.
- *Data parsing layer:* This layer is provided by our framework and is responsible for extracting meaningful records from the recorded raw byte stream. To parse the recorded byte stream, this layer requires a header file describing the recorded message format. This information is provided by the application developer (i.e., the user of the data collection front-end).
- *Data labeling layer:* To be able to identify the probable causes of failure, the data analysis subsystem needs samples of logged events representing both “good” and “bad” behavior. As “good” or “bad” behavior semantics are an application specific criterion, the application developer needs to implement a predicate (a small module) whose interface is already provided by us in the framework. The predicate, presented with an ordered event log, decides whether behavior is good or bad.

- *Data conversion layer*: This layer provides the interface between the data preprocessing middleware and the data analysis subsystem. One instance of this layer exists for each different analysis back-end. This layer is responsible for converting the labeled data into appropriate format for the data analysis algorithm. The interface of this data conversion layer is provided by the framework. As different data analysis algorithms and techniques can be used for analysis, each may have different input format requirements. This layer provides the necessary functionality to accommodate supported data analysis techniques.

### 5.3 Data Analysis Back-End

At present, we implement the data analysis algorithm and its modifications presented earlier in Section 4. It is responsible for identifying the causes of failures. The approach is extensible. As newer analysis algorithms are developed that catch more or different types of bugs, they can be easily incorporated into the tool as alternative back-ends. Such algorithms can be applied in parallel to analyze the same set of logs to find different problems with them.

## 6. DUSTMINER IMPLEMENTATION

In this section, we describe the implementation of the data collection front-end and the data analysis back-end that are used for failure diagnosis in this paper. We used two different data collection front-ends for two different case studies. The first one is implemented by us and used for real time logging of user defined events on flash memory in MicaZ motes, and the second front-end was a built-in logging support functionality provided by LiteOS operating system for MicaZ motes. At the data analysis back-end, we used discriminative frequent pattern analysis for failure diagnosis. We describe the implementation of each of these next.

### 6.1 The Front-End: Acquiring System State

We used two different data collection front-ends to collect data: (i) event logging system implemented for MicaZ platform in TinyOS 2.0 and (ii) kernel event logger for MicaZ platform provided by LiteOS. The format of the event logged by the two subsystems are completely different. We were able to use our framework to easily integrate the two different front-ends and use the same back-end to analyze the cause of failures, which shows modularity. We briefly describe each of these front-ends below.

#### 6.1.1 Data Collection Front-End for TinyOS

*Implementation:*

The event logger for MicaZ hardware is implemented using the TinyOS 2.0 BlockRead and BlockWrite interfaces to perform read and write operations respectively on flash. BlockRead and BlockWrite interfaces allow accessing the flash memory at a larger granularity which minimizes the recording time to flash.

To minimize the number of flash accesses we used a global buffer to accumulate events temporarily before writing to flash. Two identical buffers (buffer A and B) are used alternately to minimize the interference between event buffering

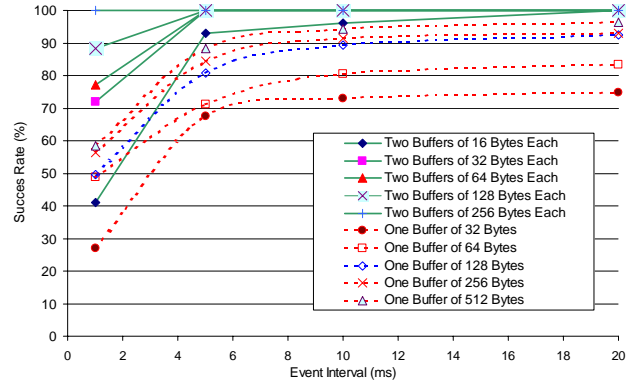


Figure 2: Impact of buffer size and event rate on logging performance

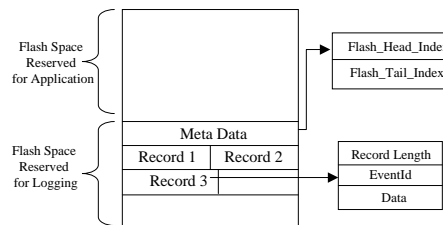


Figure 3: Flash space layout

and writing to flash. When buffer A gets filled up, buffer B is used for temporary buffering and buffer A is written to flash and vice versa. In Figure 2 we show the effect of buffer size on logging performance for single buffer and double buffer respectively. Using two buffers increases the logging performance substantially. As shown in figure, for event rate of 1000 events/second, using one buffer of 512 bytes has a success ratio (measured as the ratio of successfully logged events to the total number of generated events) of only 60% whereas using two buffers of 256 bytes each (512 bytes in total) can give almost 100% success ratio. For a rate of 200 events/second, two buffers of 32 bytes each is enough for 100% success ratio.

The sizes of these buffers are configurable as different applications need different amounts of runtime memory. It is to be noted that if the system crashes while some data are still in the RAM buffer, those events will be lost. The flash space layout is given in Figure 3.

A separate MicaZ mote (*LogManager*) is used to communicate with the logging subsystem to start and stop logging. Until the logging subsystem receives the “StartLogging” command, it will not log anything and after receiving “StopLogging” command it will flush the remaining data that is in the buffer to flash and stop logging. This gives the user the flexibility to start and stop logging whenever they want. It also lets the user to run their application without enabling logging, when needed, to avoid the runtime overhead of logging functionality without recompiling the code.

We realize that occasional event reordering can occur due to preemption, interrupts, or task scheduling delays. An

occasional invalid log entry is not a problem. An occasional incorrect logging sequence is fine too as long as the same occasional wrong sequence does not occur consistently. This is because common sequences do not have to occur every time, but only often enough to be noticed. Hence, they can be occasionally mis-logged without affecting the diagnostic accuracy.

*Time Synchronization:*

We need to timestamp the recorded events so that events recorded on different nodes can be serialized later during offline analysis. To avoid the overhead of running a time synchronization protocol on the application mote, we used an offline time synchronization scheme. A separate node (*TimeManager*) is used to broadcast its local clock periodically. The event logging component will receive the message and log it in flash with a local timestamp. From this information we can calculate the clock skew on different nodes in reference to *TimeManager* node, adjust the timestamp of the logged events and serialize the logs. We realize that the serialized log may not be exact but it is good enough for pattern mining.

*Interface:*

The only part of the data collection front-end that is exposed to the user is the interface for logging user defined events. Our design goal was to have an easy-to-use interface and efficient implementation to reduce the runtime overhead as much as possible. One critical issue with distributed logging was to timestamp the recorded events so that events on different nodes can be serialized later during offline analysis. To make event logging functionality simpler, we defined the interface to the logging component as follows:

```
log(EventId, (void *)buffer, unit8_t size)
```

*log(EventId, (void\*)buffer, unit8\_tsize)* is the key interface between application developers and the logging subsystem. To log an event, the user has to call the *log()* function with appropriate parameters. For example, if the user wants to log the event that a radio message was sent and also wants to log the receiverId along with the event, he/she needs to define the appropriate record structure in a header file (this file will also be used to parse the data) with these fields, initialize the record with appropriate values and call the *log* function with that record as the parameter. This simple function call will log the event. The rest is taken care of by the logging system underneath. The logging system will pad the timestamp with the recorded event and log as a single event. Note that *NodeId* is not recorded during logging. This information is added when data is uploaded to PC for offline analysis.

*System Overhead:*

The event logging support requires 14670 bytes of program memory (this includes the code size for BlockRead and BlockWrite interface provided by TinyOS 2.0) and 830 bytes of data memory when 400 bytes are used for buffering (two buffers of 200 bytes each) data before writing to flash. User can choose to use less buffer space if the expected event rate is low. To instrument code, the program size increase is minimal. To log an event with no attributes, it needs a single line of code. To log an event with *n* attributes, it takes *n + 1* lines of code, *n* lines are to initialize the record and 1 line to call the *log()* function.

### 6.1.2 Data Collection Front-End for LiteOS

LiteOS [6] provides the required functionality to log kernel events on MicaZ platforms. Specifically, the kernel logs events including system calls, radio activities, context switches and so on. An event log entry is a single 8-bit code without attributes. In Figure 4, we present a subset of events logged for the LiteOS case study presented in our paper. We used an experimental set up of a debugging testbed with all motes connected to a PC via serial interfaces. In pre-deployment testing on our indoor testbed, logs can thus be transmitted in real-time through a programming board via serial communication with a base-station. When a system call is invoked or a radio packet is received on a node, the corresponding code for that specific event is transmitted through the serial port to the base station (PC). The base station collects event codes from the serial port and records it in a globally ordered file.

## 6.2 The Data Analysis Back-End

At the back-end, we implement the data preprocessing and discriminative frequent pattern mining algorithm. To integrate the data collection front-end with the data preprocessing middleware, we provided a simple text file describing the storage format of the raw byte stream stored on flash for each of the front-ends. This file was used to parse the recorded events. The user supplied different predicates as a Java function. These were used to annotate data into good and bad segments. The rest of the system is a collection of data analysis algorithms such as discriminative frequent pattern mining, or any other tool such as Weka [1].

## 7. EVALUATION

To test the effectiveness of the tool, we applied our tool to troubleshoot two real life applications. We present the case studies where we have used our tool successfully. The first was a kernel level bug in the LiteOS operating system. The second was to debug a multichannel Media Access Control(MAC) protocol [16] implemented in TinyOS 2.0 for MicaZ platform with only one half-duplex radio interface.

### 7.1 Case Study - I: LiteOS Bug

In this case study, we troubleshoot a simple data collection application where several sensors monitor light and report it to a sink node. The communication is performed in a single-hop environment. In this scenario, sensors transmit packets to the receiver, and the receiver records received packets and sends an "ACK" back. The sending rate that sensors use is variable and depends on the variations in their readings. After receiving each message, depending on its sequence number, the receiver decides to record the value or not. If the sequence number is older than the last sequence number it has received, the packet is dropped.

This application is implemented using MicaZ motes on LiteOS operating system and is tested on an experimental testbed. Each of the nodes is connected to a desktop computer via an MIB520 programming board and a serial cable. The PC acts as the base station. In this experiment, there was one receiver (the base node) and a set of 5 senders (monitoring sensors). This experiment illustrates a typical experimental debugging set up. Prior to deployment, pro-

grammers would typically test the protocol on target hardware in the lab. This is how such a test might proceed.

### 7.1.1 Failure Scenario

When this simple application was stress tested, some of the nodes would crash occasionally and non-deterministically. Each time different nodes would crash and at different times. Perplexed by the situation, the developer (a first-year graduate student with no prior experience with sensor networks) decided to log different types of events using LiteOS support and use our debugging tool. These were mostly kernel-level events along with a few application-level events. The built-in logging functionality provided by LiteOS was used to log the events. A subset of the different types of logged events are listed in Figure 4.

Recorded Events	Attribute List
Context_Switch_To_User_Thread	Null
Packet_Received	Null
Packet_Sent	Null
Yield_To_System_Thread	Null
Get_Current_Thread_Address	Null
Get_Radio_Mutex	Null
Get_Radio_Send_Function	Null
Mutex_Unlock_Function	Null
Get_Current_Thread_Index	Null
Get_Current_Radio_Info_Address	Null
Get_Current_Radio_Handle_Address	Null
Post_Thread_Task	Null
Get_Serial_Mutex	Null
Get_Current_Serial_Info_Address	Null
Get_Serial_Send_Function	Null
Disable_Radio_State	Null
Get_Current_Radio_Handle	Null

Figure 4: Logged events for diagnosing LiteOS application bug

### 7.1.2 Failure Diagnosis

After running the experiment, “good” logs were collected from the nodes that did not crash during the experiment and “bad” logs were collected from nodes that crashed at some point in time. After applying our discriminative frequent pattern mining algorithm to the logs, we provided two sets of patterns to the developer, one set includes the highest ranked discriminative patterns that are found only in “good” logs as shown in Figure 5, and the other set includes the highest ranked discriminative patterns that are found only in “bad” logs as shown in Figure 6.

Based on the discriminative frequent pattern, it is clear that in “good” pile,  $\langle Packet\_Received \rangle$  event is highly correlated with the  $\langle Get\_Current\_Radio\_Handle \rangle$  event. On the other hand, in the “bad” pile, though  $\langle Packet\_Received \rangle$  event is present, the other event is missing. In the “bad” pile,  $\langle Packet\_Received \rangle$  is highly correlated with  $\langle Get\_serial\_Send\_Function \rangle$  event. From these observations, it is clear that proceeding with a  $\langle Get\_serial\_Send\_Function \rangle$  when  $\langle Get\_Current\_Radio\_Handle \rangle$  is missing is the most likely cause of failure.

To explain the error we will briefly describe the way a

$\langle Packet\_Received \rangle, \langle Packet\_Sent \rangle, \langle Get\_Current\_Radio\_Handle \rangle$
$\langle Packet\_Received \rangle, \langle Get\_Current\_Radio\_Handle\_Address \rangle, \langle Get\_Current\_Radio\_Handle \rangle$
$\langle Packet\_Received \rangle, \langle Mutex\_Unlock\_Function \rangle, \langle Get\_Current\_Radio\_Handle \rangle$
$\langle Packet\_Received \rangle, \langle Disabale\_Radio\_State \rangle, \langle Get\_Current\_Radio\_Handle \rangle$
$\langle Packet\_Received \rangle, \langle Post\_Thread\_Task \rangle, \langle Get\_Current\_Radio\_Handle \rangle$

Figure 5: Discriminative frequent patterns found only in “good” log for LiteOS bug

$\langle Context\_Switch\_to\_User\_Thread \rangle, \langle Get\_Current\_Thread\_Address \rangle, \langle Get\_Serial\_Send\_Function \rangle$
$\langle Packet\_Received \rangle, \langle Context\_Switch\_to\_User\_Thread \rangle, \langle Get\_Serial\_Send\_Function \rangle$
$\langle Packet\_Received \rangle, \langle Post\_Thread\_Task \rangle, \langle Get\_Serial\_Send\_Function \rangle$
$\langle Packet\_Received \rangle, \langle Get\_Current\_Thread\_Index \rangle, \langle Get\_Serial\_Send\_Function \rangle$
$\langle Packet\_Received \rangle, \langle Get\_Current\_Thread\_Address \rangle, \langle Get\_Serial\_Send\_Function \rangle$

Figure 6: Discriminative frequent patterns found only in “bad” log for LiteOS bug

received packet is handled in LiteOS. In the application, receiver always registers for receiving packets, then waits until a packet arrives. At that time, the kernel switches back to the user thread with appropriate packet information. The packet is then processed in the application. However, at very high data rates, another packet can come when the processing of the previous packet has not yet been done. In that case, LiteOS kernel overwrites the radio receive buffer with new information even if the user is still using the old packet data to process the previous packet.

Indeed, for correct operation,  $\langle Packet\_Received \rangle$  event always has to be followed by  $\langle Get\_Current\_Radio\_Handle \rangle$  event before  $\langle Get\_Serial\_Send\_Function \rangle$  event. Otherwise it crashes the system. Over-writing a receive buffer for some reason is a very typical bug in sensor networks. This example is presented to illustrate the use of the tool. In section 7.2 we present a more complex example that explores more of the interactive complexity this tool was truly designed to uncover.

### 7.1.3 Comparison with Previous Work

To compare the performance of our discriminative pattern mining algorithm with previous work on diagnosing sensor network bugs [15, 14], we implemented the pure Apriori algorithm, used in [14], to generate frequent patterns and perform differential analysis to extract discriminative patterns. We did not compare with [15] because that work did not look for *sequences* of events that cause problem but rather looked for *current state* that correlates with imminent failure. Hence, it addressed a different problem. For this case study, when we applied the Apriori algorithm to the “good” log and the “bad” log, the list of discriminative patterns missed the  $\langle Packet\_Received \rangle$  event completely and failed to identify the fact that the problem was correlated with the timing of packet reception. Moreover, when we applied the Apriori algorithm to multiple instances of “good” logs and “bad” logs together, the list of discriminative patterns returned was empty. All the frequent patterns generated by Apriori algorithm were canceled at the differential phase. This shows the necessity of our extensions as described in section 4.

## 7.2 Case Study - II: Multichannel MAC Protocol

In this case study we debug a multichannel MAC protocol. The objective of the protocol used in our study is to assign a home channel to each node in the network dynamically in such a way that the throughput is maximized. The design of the protocol exploits the fact that in most wireless sensor networks, the communication rate among different nodes is not uniform (e.g., in a data aggregation network). Hence, the problem was formulated in such a way that nodes communicating frequently are clustered together and assigned the same home channel whereas nodes that communicate less frequently are clustered into different channels. This minimizes overhead of channel switching when nodes need to communicate. This protocol was recently published in [16].

During experimentation with the protocol, it was noticed that when data rates between different internally closely-communicating clusters is low, the multi-channel protocol outperforms a single channel MAC protocol comfortably as it should. However, when the data rate between clusters was increased, while the throughput near the base station still outperformed a single channel MAC significantly, nodes further from the base station were performing worse than in the single channel MAC. This should not have happened in a well-designed protocol as the multichannel MAC protocol should utilize the communication spectrum better than a single channel MAC. The author of the protocol initially concluded that the performance degradation was due to the overhead associated with communication across clusters assigned to different channels. Such communication entails frequent channel switching as the sender node, according to the protocol, must switch the frequency of the receiver before transmission, then return to its home channel. This incurs overhead that increases with the transmission rate across clusters. We decided to verify this conjecture.

As a stress test of our tool, we instrumented the protocol to log events related to the MAC layer (such as message transmission and reception as well as channel switching) and used our tool to determine the discriminative patterns generated from different runs with different message rates, some of which performing better than others. For better understanding of the failure scenario detected, we briefly describe the operation of the multichannel MAC protocol below.

### 7.2.1 Multichannel MAC Protocol Overview

In the multichannel MAC protocol, each node initially starts at channel 0 as its home channel. To communicate with others, every node maintains a data structure called “neighbor table” that stores the neighbor home channel for each of its neighboring nodes. Channels are organized as a ladder, numbered from lowest (0) to highest (12). When a node decides to change its home channel, it sends out a “Bye” message in its current home channel which includes its new home channel number. Receiving a “Bye” message, each other node updates its neighbor table to reflect the new home channel number for the sender of the “Bye” message. After changing its home channel, a node sends out a “Hello” message in the new home channel which includes its nodeID. All neighboring nodes on that channel add this node as a new neighbor and update their neighbor tables accordingly.

To increase robustness to message loss, the protocol also includes a mechanism for discovering the home channel of

a neighbor when its current entry in the neighbor table becomes stale. When a node sends a message to a receiver on that receiver’s home channel (as listed in the neighbor table) but does not receive an “ACK” after ‘n’ (n is set to 5) tries, it assumes that the destination node is not on its home channel. The reason may be that the destination node has changed its home channel permanently but the notification was lost. Instead of wasting more time on retransmissions on the same channel, the sender starts scanning all channels, asking if the receiver is there. The purpose is to find the receiver’s new home channel and update the neighbor table accordingly. The destination node will eventually hear this data message and reply when it is on its home channel.

Since the above mechanism is expensive, as an optimization, overhearing is used to reduce staleness of the neighbor table. Namely, a node updates the home channel of a neighbor in its neighbor table when the node overhears an acknowledgement (“ACK”) from that neighbor sent on that channel. Since the “ACK”s are used as a mechanism to infer home channel information, whenever a node switches channels temporarily (e.g., to send to a different node on the home channel of the latter), it delays sending out “ACK” messages until it comes back to its home channel in order to prevent incorrect updates of neighbor tables by recipients of such ACKs.

Finally, to estimate channel conditions, each node periodically broadcasts a “channelUpdate” message which contains the information about successfully received and sent messages during the last measurement period (where the period is set at compile time). Based on that information, each node calculates the channel quality (i.e., probability of successfully accessing the medium), and uses that measure to probabilistically decide whether to change its home channel or not. Nodes that sink a lot of traffic (e.g., aggregation hubs or cluster heads) switch first. Others that communicate heavily with them follow. This typically results into a natural separation of node clusters into different frequencies so they do not interfere.

### 7.2.2 Performance Problem

This protocol was executed on 16 MicaZ nodes implementing an aggregation tree where several aggregation cluster-heads filter data received from their children, significantly reducing the amount forwarded, then send that reduced data to a base-station. When the data rate across clusters was low, the protocol outperformed the single channel MAC. However, when the data rate among clusters was increased, the performance of the protocol deteriorated significantly, performing worse than a single channel MAC in some cases. The developer of the protocol assumed that this was due to the overhead associated with the channel change mechanism which is incurred when communication happens among different clusters heavily. Much debugging effort was spent on that direction with no result.

### 7.2.3 Failure Diagnosis

To diagnose the cause of the performance problem, we logged different types of MAC events as listed in Figure 7.

The question posed to our tool was “Why is the performance bad at higher data rate?”. To answer this question, we first executed the protocol at low data rates (when the performance is better than single channel MAC) to collect

Recorded Events	Attribute List
Ack_Received	Null
Home_Channel_Changed	oldChannel, newChannel
TimeSyncMsg	referenceTime, localTime
Channel_Update_Msg_Sent	homeChannel
Data_Msg_Sent_On_Same_Channel	destId, homeChannel
Data_Msg_Sent_On_Different_Channel	destId, homeChannel, destChannel
Channel_Update_Msg_Received	homeChannel, neighborId, neighborChannel
Retry_Transmission	oldChannelTried, nextChannelToTry
No_Ack_Received	Null

Figure 7: Logged events for diagnosing multichannel MAC protocol

<No_Ack_Received>,<Retry_Transmission>
<Retry_Transmission>,<No_Ack_Received>
<Data_Msg_Sent_On_Same_Channel: homechannel:0>,<No_Ack_Received>,<Retry_Transmission>,<Retry_Transmission: nextchanneltotry:1>,<Retry_Transmission>,<Retry_Transmission: oldchanneltried:1>,<No_Ack_Received>
<Data_Msg_Sent_On_Same_Channel: homechannel:0>,<No_Ack_Received>,<Retry_Transmission>,<Retry_Transmission: nextchanneltotry:1>,<Retry_Transmission: nextchanneltotry:2>
<Data_Msg_Sent_On_Same_Channel: homechannel:0>,<No_Ack_Received>,<Retry_Transmission>,<Retry_Transmission: nextchanneltotry:1>,<Retry_Transmission: oldchanneltried:2>,<Retry_Transmission: nextchanneltotry:3>,<No_Ack_Received>,<Retry_Transmission: oldchanneltried:3>

Figure 8: Discriminative frequent patterns for multichannel MAC protocol

logs representing “good” behavior. We then again executed the protocol with a high data rate (when the performance is worse than single channel MAC) to collect logs representing “bad” behavior.

After performing discriminative pattern analysis, the list of top 5 discriminative patterns that were produced by our tool is shown in Figure 8.

The sequences indicate that, in all cases, there seems to be a problem with not receiving acknowledgements. Lack of acknowledgements causes a channel scanning pattern to unfold. This is shown as the *< Retry\_Transmission >* event on different channels, as a result of not receiving acknowledgements. Hence, the problem does not lie in the frequent overhead of senders changing their channel to that of their receiver in order to send a message across clusters. The problem lied in the frequent lack of response (an ACK) from a receiver. At the first stage of frequent pattern mining *< No\_Ack\_Received >* is identified as the most frequent event. At the second stage, the algorithm searched for frequent patterns in top *K* (e.g., top 5) segments of the logs where *< No\_Ack\_Received >* event occurred with highest frequency. The second stage of the log analysis (correlat-

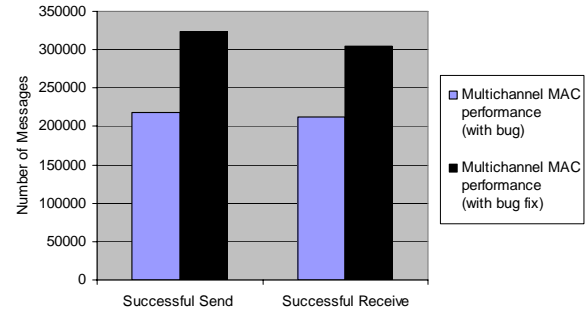


Figure 9: Performance improvement after the bug fix

ing frequent events to preceding ones) then uncovered that the lack of an ACK from the receiver is preceded by a temporary channel change. This gave away the bug. As we described earlier, whenever a node changes its channel temporarily, it disables “ACK” until it comes back to its home channel. In a high intercluster communication scenario, disabling the “ACK” is a bad decision for a node that spends a significant amount of time communicating with other clusters on channels other than its own home channel. As a side effect, nodes which are trying to communicate with it fail to receive an “ACK” for a long time and start scanning channels frequently looking for the missing receiver. Another interesting aspect of the problem that was discovered is the cascading effect of the problem. When we look at generated discriminative patterns across multiple nodes (not shown for space limitations), we see that the scanning patterns revealed in the logs shown in fact cascades. Channel scanning at the destination node often triggers channel scanning at the sender node and this interesting cascaded effect was also captured by our tool.

As a quick fix, we stopped disabling “ACK” when a node is outside its home channel. This may appear to violate some correctness semantics because a node may now send an ACK while temporarily being on a channel other than its home. This, one would think, will pollute neighbor tables of nodes that overhear the ACK because they will update their tables to indicate an incorrect home channel. In reality, the performance of the MAC layer improved significantly (up to 50%), as shown in Figure 9. In retrospect, this is not unexpected. As intercluster communication increases, the distinction between one’s home channel and the home channel of another with whom one communicates a lot becomes fuzzy, as one spends more and more time on that other node’s home channel (to send messages to it). When ACKs are never disabled, the neighbor tables of nodes will tend to record with a higher probability the channel on which each neighbor spend most of its time. This could be the neighbor’s home channel or the channel of a node downstream with which the neighbor communicates a lot. The distinction becomes immaterial as long as the neighbor can be found on that channel with a high probability. Indeed, complex interaction problems often seem simple when explained but are sometimes hard to think of at design time. Dustminer was successful at uncovering the aforementioned interaction and significantly improve the performance of the MAC protocol in question.

### 7.2.4 Comparison with Prior Work

As before, we compare our results with the result of the algorithm in [14], which uses the Apriori algorithm. As we mentioned earlier, one problem with the previous approach is scalability. Due to huge numbers of events logged for this case study (about 40000 for “good” logs and 40000 for “bad” logs), we could not generate frequent patterns of length more than 2 using the approach in [14]. To generate frequent patterns of length 2 for 40000 events in the “good” log, it took 1683.02 seconds (28 minutes) and to finish the whole computation including differential analysis it took 4323 seconds (72 minutes). With our two-stage mining scheme, it took 5.547 seconds to finish the first stage and finishing the whole computation including differential analysis took 332.924 seconds (6 minutes). In terms of quality of the generated sequences (which is often correlated with the length of the sequence), our algorithm returned discriminative sequences of length upto 8, that was enough to understand the chain of events causing the problem as illustrated above. We tried to generate frequent patterns of length 3 with the approach in [14] but terminated the process after one day of computation that remained in progress. We used a machine of 2.53 GHz speed and 512 MB RAM. The generated patterns of length 2 were insufficient to give insight into the problem.

### 7.3 Debugging Overhead

To test the impact of logging on application behavior, we ran the multichannel MAC protocol with logging enabled and without logging enabled with both moderate data rate and high data rate. The network was set as a data aggregation network.

For moderate data rate experiment, the source nodes (node that only sends messages) were set to transmit data at a rate of 10 messages/sec, the intermediate nodes were set to transmit data at a rate of 2 messages/sec and one node was acting as the base station (which only receives messages). We tested this on a 8 nodes network with 5 source nodes, 2 intermediate nodes and one base station. Over multiple runs, after we take the average to get a reliable estimate, average number of successfully transmitted messages was increased by 9.57% and average number of successfully received messages was increased by 2.32%. The most likely reason is writing to flash was creating a randomization effect which probably helped to reduce interference at the MAC layer.

At high data rate, source nodes were set to transmit data at a rate of 100 messages/sec and intermediate nodes were set to transmit data at a rate of 20 messages/sec. Over multiple runs, after we take the average to get a reliable estimate, average number of successfully transmitted messages was reduced by 1.09% and average number of successfully received messages was dropped by 1.62%. The most likely reason is the overhead of writing to flash kicked in at a such high data rate and eventually reduced the advantage experienced at a low data rate.

The performance improvement of the multichannel MAC protocol reported in this paper is obtained by running the protocol at the high data rate to prevent over estimation.

We realize that this effect on application may change the behavior of the original application slightly, but that effect seems to be negligible from our experience and did not affect the diagnostic capability of the discriminative pattern mining algorithm which is inherently robust against minor

statistical variance.

As multichannel MAC protocol did not use flash memory to store any data, we were able to use the whole flash for logging events. To test the relation between quality of generated discriminative patterns and the logging space used, we used 100KB, 200KB and 400KB of flash space in three different experiments. The generated discriminative patterns were similar. We realize that different application has different amount of flash space requirements and the amount of logging space may affect the diagnostic capability. To help in severe space constraints, we provide the radio interface so users can choose to log at different times instead of logging continuously. User can also choose to log events at different resolutions (e.g., instead of logging every message transmitted, log only every 50<sup>th</sup> message transmitted).

For LiteOS case study, we did not use flash space at all as the events were transmitted to basestation (PC) directly using serial connection and eliminate the flash space overhead completely which makes our tool easily usable for testbeds which often provides serial connections.

## 8. CONCLUSION

In this paper, we presented a sensor network troubleshooting tool that helps the developer diagnose root causes of errors. The tool is geared towards finding interaction bugs. Very successful examples of debugging tools that hunt for localized errors in code have been produced in previous literature. The point of departure in this paper lies in focusing on errors that are not localized (such as a bad pointer or an incorrect assignment statement) but rather arise because of adverse interactions among multiple components each of which appears to be correctly designed. The cascading channel-scanning example that occurred due to disabling acknowledgements in the MAC Protocol illustrates the subtlety of interaction problems in sensor networks. With increased distribution and resource constraints, the interactive complexity of sensor networks applications will remain high, motivating tools such as the one we described. Future development of *Dustminer* will focus on scalability and user interface to reduce the time and effort needed to understand and use the new tool.

## Acknowledgments

The authors thank the shepherd, Kay Römer, and the anonymous reviewers for providing valuable feedback and improving the paper. This work was supported in part by NSF grants DNS 05–54759, CNS 06–26342, and CNS 06–13665.

## 9. REFERENCES

- [1] <http://www.cs.waikato.ac.nz/ml/weka/>.
- [2] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proceedings of the Twentieth International Conference on Very Large Data Bases (VLDB'94)*, pages 487–499, 1994.
- [3] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance debugging for distributed systems of black boxes. In *Proceedings of the nineteenth ACM symposium on Operating systems principles (SOSP'03)*, pages 74–89, 2003. Bolton Landing, NY, USA.
- [4] P. Ballarini and A. Miller. Model checking medium access control for sensor networks. In *Proceedings of*

- the 2nd International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISOLA'06)*, pages 255–262, Paphos, Cyprus, November 2006.
- [5] P. Bodík, G. Friedman, L. Biewald, H. Levine, G. Candea, K. Patel, G. Tolle, J. Hui, A. Fox, M. I. Jordan, and D. Patterson. Combining visualization and statistical analysis to improve operator confidence and efficiency for failure detection and localization. In *Proceedings of the 2nd International Conference on Autonomic Computing (ICAC'05)*, 2005.
  - [6] Q. Cao, T. Abdelzaher, J. Stankovic, and T. He. The liteos operating system: Towards unix-like abstractions for wireless sensor networks. In *Proceedings of the Seventh International Conference on Information Processing in Sensor Networks (IPSN'08)*, April 2008.
  - [7] E. Cheong, J. Liebman, J. Liu, and F. Zhao. Tinygals: a programming model for event-driven embedded systems. In *Proceedings of the 2003 ACM symposium on Applied computing (SAC'03)*, pages 698–704, 2003. Melbourne, Florida.
  - [8] E. Ertin, A. Arora, R. Ramnath, and M. Nesterenko. Kansei: A testbed for sensing at scale. In *Proceedings of the 4th Symposium on Information Processing in Sensor Networks (IPSN/SPOTS track)*, 2006.
  - [9] G. D. Fatta, S. Leue, and E. Stegantova. Discriminative pattern mining in software fault detection. In *Proceedings of the 3rd international workshop on Software quality assurance (SOQUA '06)*, pages 62–69, 2006.
  - [10] E. Frank and I. H. Witten. Generating accurate rule sets without global optimization. In *Proceedings of the Fifteenth International Conference on Machine Learning (ICML'98)*, pages 144–151, 1998.
  - [11] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesc language: A holistic approach to networked embedded systems. In *Proceedings of Programming Language Design and Implementation (PLDI'03)*, pages 1–11, June 2003.
  - [12] 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 annual conference on USENIX Annual Technical Conference (ATEC'04)*, pages 24–24, Boston, MA, 2004.
  - [13] Y. Hanna, H. Rajan, and W. Zhang. Slede: Lightweight specification and formal verification of sensor networks protocols. In *Proceedings of the First ACM Conference on Wireless Network Security (WiSec)*, Alexandria, VA, March-April 2008.
  - [14] M. M. H. Khan, T. Abdelzaher, and K. K. Gupta. Towards diagnostic simulation in sensor networks. In *Proceedings of International Conference on Distributed Computing in Sensor Systems (DCOSS)*, 2008. Greece.
  - [15] M. M. H. Khan, L. Luo, C. Huang, and T. Abdelzaher. Snts: sensor network troubleshooting suite. In *Proceedings of International Conference on Distributed Computing in Sensor Systems (DCOSS)*, 2007. Santa Fe, New Mexico, USA.
  - [16] H. K. Lee, D. Henriksson, and T. Abdelzaher. A practical multi-channel medium access control protocol for wireless sensor networks. In *Proceedings of International Conference on Information Processing in Sensor Networks (IPSN'08)*, St. Louis, Missouri, April 2008.
  - [17] P. Levis and D. Culler. Mate: a tiny virtual machine for sensor networks. In *Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, San Jose, California, October 2002.
  - [18] P. Levis, N. Lee, M. Welsh, and D. Culler. Tossim: accurate and scalable simulation of entire tinys applications. In *Proceedings of the 1st international conference on Embedded networked sensor systems (SenSys'03)*, pages 126–137, Los Angeles, California, USA, 2003.
  - [19] C. Liu, L. Fei, X. Yan, J. Han, and S. P. Midkiff. Statistical debugging: A hypothesis testing-based approach. *IEEE Transactions on Software Engineering*, 32:831–848, 2006.
  - [20] C. Liu and J. Han. Failure proximity: a fault localization-based approach. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering (SIGSOFT '06/FSE-14)*, pages 46–56, 2006.
  - [21] C. Liu, Z. Lian, and J. Han. How bayesians debug. In *Proceedings of the Sixth International Conference on Data Mining (ICDM'06)*, pages 382–393, December 2006.
  - [22] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff. Sober: statistical model-based bug localization. In *Proceedings of the 13th ACM SIGSOFT international symposium on Foundations of software engineering (FSE-13)*, 2005. Lisbon, Portugal.
  - [23] C. Liu, X. Yan, and J. Han. Mining control flow abnormality for logic error isolation. In *Proceedings of 2006 SIAM International Conference on Data Mining (SDM'06)*, Bethesda, MD, April 2006.
  - [24] C. Liu, X. Zhang, J. Han, Y. Zhang, and B. K. Bhargava. Failure indexing: A dynamic slicing based approach. In *Proceedings of the 2007 IEEE International Conference on Software Maintenance (ICSM'07)*, Paris, France, October 2007.
  - [25] L. Luo, T. F. Abdelzaher, T. He, and J. A. Stankovic. Envirosuite: An environmentally immersive programming framework for sensor networks. *ACM Transactions on Embedded Computing Systems*, 5(3):543–576, 2006.
  - [26] L. Luo, T. He, G. Zhou, L. Gu, T. Abdelzaher, and J. Stankovic. Achieving Repeatability of Asynchronous Events in Wireless Sensor Networks with EnviroLog. In *Proceedings of the 25th IEEE International Conference on Computer Communications (INFOCOM'06)*, pages 1–14, 2006.
  - [27] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. Tinydb: an acquisitional query processing system for sensor networks. *ACM Transactions on Database Systems*, 30(1):122–173, 2005.
  - [28] P. Olveczky and S. Thorvaldsen. Formal modeling and analysis of wireless sensor network algorithms in real-time maude. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, Rhodes Island, Greece, April 2006.

- [29] J. Polley, D. Blazakis, J. McGee, D. Rusk, and J. S. Baras. Atemu: A fine-grained sensor network simulator. In *Proceedings of the First International Conference on Sensor and Ad Hoc Communications and Networks (SECON'04)*, pages 145–152, Santa Clara, CA, October 2004.
- [30] N. Ramanathan, K. Chang, R. Kapur, L. Girod, E. Kohler, and D. Estrin. Sympathy for the sensor network debugger. In *Proceedings of the 3rd international conference on Embedded networked sensor systems (SenSys'05)*, pages 255–267, 2005.
- [31] R. Szcwcyk, J. Polastre, A. Mainwaring, and D. Culler. Lessons from a sensor network expedition. In *Proceedings of the First European Workshop on Sensor Networks (EWSN)*, 2004.
- [32] G. Tolle and D. Culler. Design of an application-cooperative management system for wireless sensor networks. In *Proceedings of the Second European Workshop on Wireless Sensor Networks (EWSN'05)*, pages 121–132, Istanbul, Turkey, February 2005.
- [33] P. Volgyesi, M. Maroti, S. Dora, E. Osses, and A. Ledeczi. Software composition and verification for sensor networks. *Science of Computer Programming*, 56(1-2):191–210, 2005.
- [34] Y. Wen and R. Wolski. *s<sup>2</sup>db*: A novel simulation-based debugger for sensor network applications. UCSB 2006, 2006-01.
- [35] Y. Wen, R. Wolski, and G. Moore. Disens: scalable distributed sensor network simulation. In *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP'07)*, pages 24–34, 2007. San Jose, California, USA.
- [36] G. Werner-Allen, P. Swieskowski, and M. Welsh. Motelab: A wireless sensor network testbed. In *Proceedings of the Fourth International Conference on Information Processing in Sensor Networks (IPSN'05), Special Track on Platform Tools and Design Methods for Network Embedded Sensors (SPOTS)*, pages 483–488, April 2005.
- [37] K. Whitehouse, G. Tolle, J. Taneja, C. Sharp, S. Kim, J. Jeong, J. Hui, P. Dutta, and D. Culler. Marionette: Using rpc for interactive development and debugging of wireless embedded networks. In *Proceedings of the Fifth International Conference on Information Processing in Sensor Networks: Special Track on Sensor Platform, Tools, and Design Methods for Network Embedded Systems (IPSN/SPOTS)*, pages 416–423, Nashville, TN, April 2006.
- [38] J. Yang, M. L. Soffa, L. Selavo, and K. Whitehouse. Clairvoyant: a comprehensive source-level debugger for wireless sensor networks. In *Proceedings of the 5th international conference on Embedded networked sensor systems (SenSys'07)*, pages 189–203, 2007.