NodeMD: Diagnosing Node-Level Faults in Remote Wireless Systems

Veljko Krunic, Eric Trumpler, Richard Han

Department of Computer Science
University of Colorado at Boulder

Technical Report CU-CS-1017-06

December 2006

NodeMD: Diagnosing Node-Level Faults in Remote Wireless Systems

Veljko Krunic, Eric Trumpler
Richard Han
University of Colorado, Boulder
krunic@ieee.org, Eric.Trumpler@colorado.edu, Richard.Han@colorado.edu

Abstract

Software failures in wireless sensor systems are notoriously difficult to debug. Resource constraints in wireless deployments substantially restrict visibility into the root causes of node-level system and application faults. At the same time, the high cost of deployment of wireless sensor systems often far exceeds the cumulative cost of all other sensor hardware, so that software failures that completely disable a node are prohibitively expensive to repair in real-world applications, e.g. by on-site visits to replace or reset nodes. We describe NodeMD, a deployment management system that successfully implements lightweight run-time detection, logging, and notification of software faults on wireless mote-class devices. NodeMD introduces a debug mode that catches a failure before *it completely disables a node and drops into a state that enables further diagnosis and correction, thus avoiding on-site redeployment. We present detailed analyses of NodeMD on real world applications of wireless sensor systems.*

1. Introduction

Wireless sensor networks (WSNs) are often deployed in distant rugged environments, e.g. Great Duck Island off the coast of Maine [3], around wildfires in the Bitterroot National Forest [4], and surrounding an active volcano in Ecuador [5]. These types of deployment are expensive and sometimes even risky to deployment personnel. For example, in the FireWxNet [4] deployment, a helicopter was used by fire personnel to deploy nodes on three different mountains, in some cases requiring the firefighters to climb down the mountain to place the nodes.

Compounding the expense and difficulty of WSN deployments concludes that software bugs are often encountered in the field. Software can reach buggy states initiated by data-driven sensing behavior in the wild that is not detected through ordinary lab testing. In addition, rigor in testing sensor networks is much smaller then testing in other regimes, e.g. space software, due to much more limited resources than are typically devoted to testing. Our own experience deploying a fire sensor network in the mountainous terrain of the Bitterroot National Forest in Idaho [4] suggests that software bugs will inevitably be experienced in the field.

The typical behavior after encountering a run-time software fault is for a remote node to enter a bad/unresponsive state that looks like a “black hole”. The fault is detected retroactively by what information we *don’t* receive. The node is completely disabled and needs to be redeployed. This is clearly a situation that we wish to minimize, not the least of which is explaining to a firefighter why they need to risk redeploying a remote node that has failed.

Our goal in this paper is to offer a diagnostic system, NodeMD, capable of (1) catching run-time software faults as they occur and before they completely disable a remote node, and (2) diagnosing the root cause of the fault, thereby substantially reducing the need for costly redeployment of nodes through on-site visits. The WSN research community has offered a variety of approaches towards reducing the cost of redeployment. SOS [9] is capable of propagating new code images to a remote system. Several systems, including Marionette [15] and Nucleus [14], are capable of requesting state information from a running system, and the recent approach taken by t-kernel [22] prevents some issues that can potentially disable a node in the field. Although these systems help alleviate the cost of redeployment, they still do not provide a complete solution. In particular, the critical functionality that alerts you to a problem with the node before the node is disabled, and a history at run-time about what led to that problem, giving programmers a good starting point for rapid-response debugging.

As an analogy, current methods of node debugging are similar to a doctor that needs to visit a very remote area to treat a patient. When the doctor “hears” that a patient is not well, he or she travels by wagon and goes to see and treat the patient. Similarly, just as a doctor’s in-home visit is expensive, so is the need for redeployment in WSN systems deployed in remote locations. The WSN community is in a situation where in-home visits are *almost* unnecessary, because we have a mail-order pharmacy (SOS), telephone (Marionette, Nucleus), vaccination from the most common diseases (t-kernel), and postmortem analysis (Nucleus). But with only these pieces of the puzzle, we cannot completely avoid a need for in-home visits because we are missing initial patient contact and timely diagnostic tools. There is no equivalent ability, in the suite of tools available to the WSN community, to a human patient that picks up the phone and reports "Doctor, I am not feeling well, these are the symptoms and this is what I did in the last few days".

NodeMD is the last piece of the puzzle that is necessary to bring the analogy of a "remote doctor" to the world of WSNs. With NodeMD providing the missing link, we can envision a complete system based on keeping the "human in the loop", in which problems with the software are brought immediately to the attention of the programmer, good diagnostic tools are provided for timely diagnosis of the problem, and once the problem is diagnosed and corrected, the capability to remotely update a sensor node with debugged code. Ultimately the goal of our system is to bring node debugging from its current archaic state in WSNs and embedded systems to the level that exists in modern desktop computing systems.

The main contributions of this paper comprise the following: building a fault management system for WSNs that is capable of detecting a broad spectrum of software faults at run-time; introducing a recovery/debug mode that catches those faults so as not to completely disable the afflicted node; timely notification of the fault along with a brief diagnostic history of the events that led up to the fault; continued interaction with the halted node to close the loop on the debugging cycle by including a human programmer; resource-constrained solutions to all of the above; and proof-of-concept implementation on several real-world sensor applications. The techniques proposed in this paper are generalizable across many different systems and most of them are not OS/application specific, but could be used in a wide context of embedded operating systems.

In Section 2, we discuss related work in fault management in WSNs. Section 3 presents the unified system architecture of NodeMD. Section 4 introduces our suite of algorithms for detecting faults at run-time, including stack overflow, deadlock, livelock, and application-specific faults. Section 5 discusses our solution for entering the recovery/debug mode upon a detected fault and providing notification via a compressed history trace of the events leading up to the fault. Section 6 closes the loop on fault management by allowing interactive debugging by a human of the remote node in the halted mode. Finally, section 7 provides a detailed analysis of the current implementation in Mantis OS [8] for several real-world sensor applications.

2. Related Work

Sensor network debugging today usually begins with staring at a set of blinking LEDs. JTAG interfaces on sensor boards provide increased visibility into faults, but only for nodes directly connected to a wired network. For wireless sensor nodes in a wireless deployment or testbed environment, some systems are emerging that provide limited visibility into fault behavior. The Sympathy system [13] focuses on debugging networking faults, providing periodic reporting of various networking metrics to diagnose the reason behind reduced network throughput. The approach is somewhat limited in its periodic reporting, though the period can be adjusted, and does not focus on detecting software failures at the node level.

Nucleus [14], a deployment debugging system, was developed to resolve a lack of information when live deployments fail. Its primary features are a robust logging system and on-demand requests for information from nodes in the network. One essential aspect we have in common is our debugging methods must persist even when the application fails. Nucleus stores "printf" style messages in a limited buffer within main memory, and also writes them to flash memory to act as a sensor node "black box". Such messages are inefficient to store in main memory considering storage size needed vs. amount of information logged, and the slow storage of messages in flash may affect timing in the program if log operations are called within timing sensitive code. Additionally, once a node has failed such information is only available after the node has been retrieved.

Recent work done in t-kernel [22], a reliable OS kernel, takes an approach that ensures the scheduler is always able to retake control from a thread. At a low level, each branch instruction first jumps to the scheduler for verification before jumping back to the target address. In fact, this preemption technique would be useful to support some of the techniques proposed by NodeMD. t-kernel provides a "safe execution environment". However, t-kernel does not specify what algorithms to use for detecting faults nor how to efficiently provide information to diagnose a fault.

Marionette [15] provides a mechanism to query the memory in nodes for their state. It is specific to TinyOS, and does not focus on detection and notification of faults as they occur.

A variety of approaches for remote code updates in WSNs have been proposed, and are summarized in [7]. These approaches can be roughly divided into a networking component that achieves reliable code propagation, e.g. Deluge [10] and Aqueduct [11], and an operating system component that enables efficient update of code images on a sensor node, e.g. SOS [9] or the ELF loader [23]. Our fault management system is agnostic to the particular combination of mechanisms chosen for remote code updates. In theory any of them could be reused in the proposed architecture. For example, the ELF dynamic modules loader [23] was recently implemented inside of MOS to enable efficient code updates, the same platform upon which NodeMD is implemented. Our focus in this paper is not on these mechanisms, but instead is on our innovation in fault detection, notification, and diagnosis, the missing links in fault management for WSN systems.

3. System Architecture and Design Goals

NodeMD's fault management system consists of three main subsystems that correspond to the system shown in Figure 1. These subsystems are combined under a single unified architecture to provide an expansive solution to node-level fault diagnosis in deployed WSNs.

- The fault *detection* subsystem is designed for monitoring the health of the system and catching software faults such as stack overflow, livelock, deadlock, and application-defined faults as they occur, signified by the 'X' of the failed node in the figure.
- The fault *notification* or reporting subsystem is responsible for constant *system-oriented* logging, in a space and time-efficient manner, the sequence of events occurring in the system. This compressed event trace in the form of a circular bit vector is then conveyed in a notification message back to the human user.
- The fault *diagnosis* subsystem essentially closes the loop on the "debugging" cycle, halting the node and dropping it into a safe debug or error recovery mode wherein interactive queries can be accepted from a remote human user for more detailed diagnostic information, and remote code updates can also be accepted.

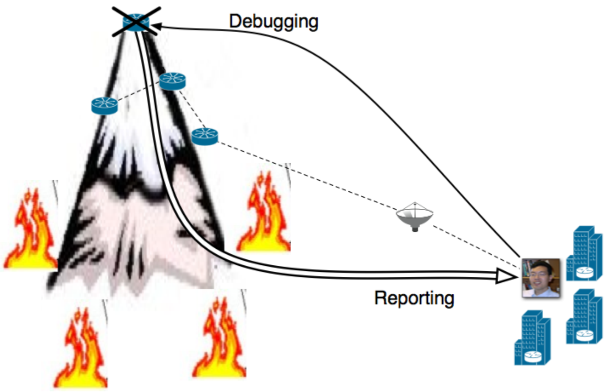


Figure 1. System architecture of NodeMD.

NodeMD must accomplish the above diagnostic features while achieving a variety of other design goals. First, it is important that fault detection and notification be memory-efficient and low overhead in terms of CPU and radio bandwidth, to fit within the resource-constraints of deployed sensor nodes. This has strong implications, such as the design to hold the event history in main memory versus external flash. Second, the design of NodeMD should afford the human user flexibility to extend and customize the diagnostic capabilities, e.g. in pursuit of a particular bug or class of bugs. For example, NodeMD allows a user to define their own application-specific conditions for triggering the detection of a "fault" and the subsequent halting of the node. NodeMD also allows users to request more detailed diagnostic information when a node is in a halted but functional debug mode. Third, our goal is to introduce algorithms and solutions that are generally applicable to a wide range of embedded systems. For example, the stack overflow detection algorithm is applicable not just on thread-based systems like MOS, but is also useful to detect aberrant behavior on event-driven single-stack systems like TinyOS.

4. Fault Detection

Detecting faults that can potentially disable a node is not a fully resolved problem in the context of WSNs. This section presents work towards identifying fault-prone conditions and implementing detection algorithms to prevent such conditions from paralyzing the node.

Our system currently identifies three generic classes of high-risk faults to applications that are of especial interest in *concurrent* sensor operating systems: stack overflow, livelock and deadlock, and out-of-bounds memory writes. Support for detection of both application-specific and OS-specific faults is also added in our implementation. Our design can expand to accommodate detection of other faults, but at present we have focused on effectively detecting these general classes of faults.

While many WSN operating systems follow event-driven models, some fault classes between event-driven and concurrent systems are mutually exclusive. Typical problems in event-driven programming concern the need for non-blocking concurrency and run-to-completion code segments, which are implcitly addressed by multithreaded scheduling. While our detection system is designed for the prominent issues in multithreaded systems, detection of some faults also applies to event-driven models, i.e. stack overflow.

4.1. Stack Overflow

Due to the extremely limited memory available, e.g. 4 KB of RAM on MICA class sensor motes, we have identified stack overflow as a key suspect in software failure. Although stack usage can be estimated by static analysis used in some approaches [20] [24], data dependency common in WSNs make it difficult to choose a stack size that is minimal yet guaranted never to be exceeded. In addition, errors in the code could make static analysis invalid. By comparison, if static analysis is useful for finding a "ballpark" stack size, stack overflow detection in NodeMD is a failsafe when the analysis results needs to be fine tuned.

Our challenge has been to design and build a lightweight detector that can catch stack overflow before it causes further damage. Our approach does not assume any hardware-based memory protection, such as an MMU, since such hardware support is frequently absent on the embedded microcontrollers typical of sensor nodes. Our implementation makes detection of stack overflow and heap exhaustion relatively inexpensive, so we can afford to call them frequently without using an excessive number of cycles. We are using an aspect-based [1] approach for this detection, and believe that this is a practical approach due to few assumptions about the code.

In order to understand what happens during a procedure call, we present an example of how the AVR-GCC [25] compiler handles initial entry point to a scope, usually a procedure. This is the compiler used for MICA-mote class sensor motes that have the AVR family of microcontrollers. When a procedure call is compiled using AVR-GCC, the compiler calculates the total stack requirements of a scope in the first pass, and then during the second pass it generates instructions to add this value to the stack pointer at the scope entry point. There are two conclusions we can draw from this behavior. First, when a procedure is called, the stack pointer is instantly set to the maxiumum stack depth of the procedure, and all stack locations are referenced at reverse offsets from this pointer. Second, the stack pointer will only increase at scope entry points. As a result, by checking stack overflow at these points the detection algorithm is both exhaustive and efficient.

Figure 2 shows the stack at the entry point of a procedure with 2 parameters and 3 local variables. Although locals 1, 2 and 3 are not yet defined, the compiler has determined that they will be defined within this scope and has reserved stack space for them. As shown by this example, we can tell whether a procedure will overflow its stack even before that stack space is actually used.

NodeMD implements a compile time preprocessor to insert stack checking code at the entry point of every procedure in the application and supporting operating system (with a few exceptions, namely the scheduler). Our approach is inspired by features offered by the AspectC++ language [1] and AOP [27], although we have used a custom implementation for robustness and to avoid several limitations of AspectC++, including unnecessary overhead and language dependence. AspectC++ allows definition of the *aspect* that will execute code on a procedure entry and/or exit. On the backend it translates to standard C++ by nesting each called function within a wrapper function that executes entry and exit code. Unfortunately, the AspectC++ implementation roughly doubles the stack overhead due to additional variables that are put on the stack during the wrapper's call - these variables are not removed! We were unable to avoid this behavior without modifying the AspectC++ compiler, so NodeMD implements a parser for C files that inserts a procedure checking call within the target function itself.

```
SP ->  0
       <reserved for local 1>
       <reserved for local 2>
       <reserved for local 3>
       return addr
       old SP
       param 2
       param 1
       ....
```

Figure 2. Stack content after a procedure call, AVR-GCC assembly.

The stack checking algorithm itself compares the current thread's stack top to the current stack pointer (SP), taking the algorithm's 2 byte stack overhead into account. If the SP exceeds the thread stack top, calling the current function will result in a stack overflow. Interrupts are addressed in the same way; at each interrupt handler entry, the stack requirements are checked against the calling thread's stack top.

On the AVR (Mica2/Z) platform a red zone detection approach is not needed because the SP is only volatile during procedure calls, which we exhaustively check. It is also important to note that using the SIGNAL keyword to define AVR interrupt handlers avoids stack issues with nested interrupt processing. Future adaptations will require compiler-specific algorithms, but an open research issue is whether an efficient generic approach can be found.

Finally, since it is likely that several bytes of another thread's stack have already been corrupted, any further normal execution risks a memory error. Thus, when this case is detected, it's critical to immediately jump to error recovery code, i.e. a debug mode, and freeze the running state of the system. This is discussed further in section 5.

4.2. Deadlock and Livelock

Deadlock and livelock are cases where a node is still "alive" but is no longer responsive. Although the node hasn't experienced a fatal error and rebooted (as would be the case in a stack overflow) one or more application threads has entered a bad state. In a multithreaded application, it's assumed that the loss of even a single application thread will likely result in a useless node.

4.2.1. Deadlock Classical problems in concurrent programming arise from interdependency. When a piece of code blocks on a condition that will never be met, that part of the system is *deadlocked*.

Common cases for deadlocks are a collective dependence on semaphores, mutexes, timer interrupts, and data dependencies (the thread will unblock when the temperature has exceeded X degrees). Even something as simple as a thread that is explicitly suspended and is then forgotten to resume will deadlock itself and possibly other parts of the system. Taking all these possibilities into account an exhaustive algorithm to determine a fatal codependency would seem to be, if not impossible, quite difficult to achieve at run-time in resource-constrained nodes.

4.2.2. Livelock *Livelocked* code, a situation similar to deadlock, differs because the code is not specifically blocked but is unable to make forward progress. For example, a running thread will never pass a while(1); statement and is livelocked. However, since the thread is not blocked but instead executing compare-and-branch instructions repeatedly, the thread is not deadlocked. While this example is extreme, the while condition could just as easily be waiting for a state in a state machine that will never enter that state again.

Many of the conditions that cause deadlock can also result in a livelock by polling on a condition rather than blocking. In addition the dependency issues noted for deadlocks, detecting livelock becomes significantly more complex when livelocked code is within an interrupt disabled context. When an application livelocks within an interrupt handler or atomic section, the scheduler is no longer able to context switch, process timers, or have any control over system execution. Thus, even software solutions used to detect and recover from this condition may not be able to run. While deadlock can run into this same problem, the context switch initiated by a blocking call should leave the interrupt disabled context anyway.

4.2.3. General Solution: Thread Checkpoints Our key observation of deadlock and livelock is that they are two conditions with a common symptom: parts of the system are not running. Rather than addressing the causes of these conditions, our approach is to identify their *symptoms* and draw a diagnostic conclusion based on those symptoms.

- Some threads deadlocked (partial deadlock)
- All threads deadlocked
- At least one thread livelocked
- One thread livelocked in interrupt disabled context

In a multithreaded OS, the symptoms of all but the last condition can be identified when a persistent thread fails to repeat a sequence of code. In WSNs, applications are often duty-cycle driven due to sensing and/or power requirements, which leads to repeated segments of code, within a while() statement for example. When either a deadlock or livelock occurs in that thread, none of the statements within that loop will recur. Therefore the case we aim to detect is when a thread has noticeably stopped repeating.

Use of a hardware watchdog timer is the simplest way to detect this. If the watchdog is reset at every iteration of a while statement, the system recovers itself when that reset does not occur. Hoever, applying a watchdog to a multithreaded system presents a challenge: how can a single timer ensure that several threads are all executing properly?

.

Alternatively, the watchdog may be embedded within a low-level thread scheduler, but at a worst case it is not fine grained enough to catch every instance of livelock and deadlock. Even at the best case, a scheduler-based watchdog runs a serious risk of false positives because the scheduler must make assumptions about the application (how long can a thread be unresponsive before it's considered deadlocked/livelocked?). Yet another extensibility issue exists due to the logistics of watchdogs. On our target platform, the AVR (Mica2/Z) platform, the 8-bit hardware timers restrict the maximum watchdog length to 2 seconds. After encountering these constraints our approach aspires to integrate a viable software solution, while incorporating a hardware watchdog for additional reliabiltiy.

The solution proposed by NodeMD begins with an assumption that in a multithreaded OS we can estimate the period of the thread, e.g. the time it takes for a while() loop to iterate. Duty cycles in WSN applications are designed for relatively specific wakeup/sleep times. Combining the repetitive nature of threaded applications, and the time constraints needed for a correct duty cycle, our assumption is that we can base a thread "timeout" value on the approximate thread period. The application programmer effectively states some constraints about the program, and NodeMD's detection schemes determine if those application constraints have been violated. While this requires some manual insertion of code, it is a best-effort compromise that avoids assumptions about application timing.

Our implementation introduces the notion of a thread *checkpoint* to emulate the behavior of a hardware watchdog. Each thread *registers* one or more checkpoints to be expected, stored in either a hash table or linked list. During registration the programmer specifies the expected RTT of this checkpoint. Next, a set_checkpoint(&mycheckpoint) call is added to a repeated point in the thread, usually at the start of a while() statement. When a checkpoint is *set*, a checkpoint parameter is set to the current system real time, effectively timestamping the most recent thread period. As seen by the [#] indicators in Figure 3, this approach requires only 3 additional lines per checkpoint. Another point is that the value of C, the estimated time for the instructions not shown, can easily be overestimated, resulting only in a slightly longer detection time. As long as the thread period is not *under*estimated the algorithm will work correctly.

```
#define sleep_time_a 1000
#define C <approximate cost of ...>
checkpoint_t mycheckpoint;                [1]
void thread_a()
{
   register_checkpoint(&mycheckpoint,
                    sleep_time_a + C);  [2]
   while(1)
   {
      set_checkpoint(&mycheckpoint);    [3]
      ...
      thread_sleep(sleep_time_a);
   }
}
```

Figure 3. Example checkpoint code.

Verifying the timeout of each checkpoint is done at the kernel level. At a periodic interval (preferrably in a hardware timer), all registered checkpoints are compared to the current real time (CRT). Specifically, if the difference between the CRT and the thread's last timestamp exceeds the thread's timeout value, our algorithm assumes the thread has livelocked or deadlocked and enters error recovery code. Currently NodeMD enforces a default timeout equal to 2*period, but the multiplier can be set differently at compile time.

Notice that this solution does not account for the final detection case, in which a thread is livelocked with interrupts disabled. In this situation control flow is never released from the running thread. Our hardware timers are crippled, the scheduler cannot initiate a context switch or process any software timers, both of which prohibit the detection algorithm from running.

To solve this problem, NodeMD incorporates a hardware watchdog as a second tier in a *hierarchical* protection scheme. While checkpoints in software ensure the correctness of each thread, the watchdog is enabled and then reset each time the kernel detection algorithm executes. If the detection algorithm is ever unable to run, such as when an interrupt disabled livelock occurs, the watchdog acts as a safety mechanism and enters recovery code once the node has reset. One of the limiting factors of the AVR watchdog is its 2 second maximum timeout, so the detection algorithm needs to have a more frequent period than the watchdog limit.

Unfortunately, part of our diagnosis is based on the preservation of main memory, which is lost when the hardware resets. An area we're still exploring is whether references to main memory can be saved to non-volatile storage and used to access the old data. If the memory on a platform is not zeroed after a watchdog reset, and we provide static heap memory for separate recovery components in the system, it may be possible to save the volatile areas we're interested in (as that static memory would always be at the same place and would not overwrite volatile memory). Implementation success will likely vary on a platform-by-platform basis, so this is proposed as a best effort solution.

Finally, it should be noted that NodeMD's detection method is not a time-critical approach. The deadlock or livelock has already occurred when it is caught. We do not believe that this is a big limitation, as the system catches the fault soon thereafter and is able to drop into a debug mode that enables continued interaction with the node, i.e. deadlock and livelock do not paralyze our NodeMD system.

4.3. Illegal memory access

A recent approach to software memory protection has used memory maps and permissions to ensure the validity of memory writes [2]. Validating a memory access in this implementation was determined to cost 66 cycles, which translates to a fairly modest cost in memory unintensive applications, i.e. Surge. Although the memory protection cost is high when applied to memory intensive applications, we believe that this approaches is viable because they could be enabled only for debugging, and then disabled once when code causing the memory fault is fixed. NodeMD is capable of supporting detection of illegal memory writes, though we have currently focused our efforts elsewhere on more unaddressed aspects of fault detection.

4.4. Application-specific faults

Many data integrity rules to WSN applications are domain specific. An example is temperature in a weather observation system, which should not report values outside of a logical range, or report rates of change that are too rapid. Incorrect data typically indicates a sensor hardware fault.

Our system supports an API that the application programmer can call when custom code detects that domain-specific constraints are violated. Our implementation currently introduces the ASSERT(condition) macro to validate that certain application constraints are not untrue. This is similar to the approach introduced by Design by Contract

[12], but would not kill a program. Instead, since WSN applications are single process, the application jumps directly to error recovery code.

Although on the surface this looks like “just plain asserts”, there are proposed methods for designing software in a way that uses assertions to the maximum effect. One example of such work is Design by Contract [12] mentioned above, which uses assertions to verify method preconditions, postconditions and/or invariants.

As an example of assertions in the application specific domain, a weather observation system could check that gradients in temperature change are within expected limits, and that the behavior of a particular node is consistent with the network (e.g. if a single node among 10 nodes in the space of 1 square mile is detecting a temperature that is 30 degrees centigrade lower than other sensors, the sensor is probably broken).

We believe that this custom detection and its interaction with the system is one of the areas where significant additional research can be done.

5. Fault Notification

For many complex problems that arise in debugging, human interaction is often the only reliable way to address many software issues. Therefore, when a fault is detected we desire to relay a diagnostic profile of the faulty node to the application programmer in order to help diagnose the cause of the fault.

Retrieving fault information poses perhaps the most difficult challenge to any WSN debugging system. With a wired interface, JTAG debugger units provide a multitude of information to any connected node. This solution is practical for a handful of nodes, but to debug an entire deployment we need a JTAG unit for each node. At a cost about three times that of the actual sensor node [21], using a JTAG adapter for every node in a testbed is simply not practical. In addition, the use of JTAG units does not expand to purely wireless environments.

Conventional string logging is a more commonly used approach for wired devices. While string logging is viable for some debugging, there are several significant limitations. Each character in a printed string is sent sequentially over the serial line. At the maximum speed of a serial line, 57600 baud, sending each byte takes 2.8 ms, not including the overhead in software [17]. In general, sending strings on the serial line is likely to change timing of the program and as a result mask or alter timing dependent problems. It’s not uncommon for an embedded program to run correctly with several “printf” statements, only to fail when they are taken out.

Additionally, serial transmission is an interrupt driven operation not compatible with much of the interrupt disabled code we have in WSN operating systems. Although polling implementations can be used, this introduces its own subset of problems. While messages can be buffered and sent at a later time, the buffer space needed to store string messages in main memory conflicts with the memory conservation WSN applications require. Frequent propogation of messages over a multihop network substantially increases the cumulative costs in both time and energy consumption.

We present a solution instead that is minimally intrusive to the running application yet offers a rich set of diagnostic information designed to identify how and why an application faulted.

5.1. Maintaining a streamlined diagnostic profile

Once a fault is detected, a key design issue is what information to send in the error report. Should only a summary of the information be presented to the human? If so, which information should be included in the summary? Another observation is that a snapshot of the current state of memory may be insufficient to diagnose certain software faults. The history or profile of behavior leading up to the fault may also need to be preserved, e.g. the sequence of function calls that resulted in the software fault, not just the current call stack. This opens up a variety of issues, such as how much recorded history to store and where (in RAM, in-chip flash, external flash), how to compress that history in memory-limited systems, and what historical information and events will be most useful to which types of faults.

The solution NodeMD implements is to keep an execution trace of recent system events within a circular bitmap, similar to work found in ARTS [18] and the Wind River System Viewer [19]. Each defined event can be described as a unique order of bits, and compressed to a length dependent on the number of combinations needed to express all recorded events. Events are encoded in compressed form and entered into a circular buffer in main memory. When memory allocated to the buffer is exhausted we begin overwriting the oldest events first. One important thing to note is NodeMD avoids using flash memory because the expensive write instructions do not facilitate frequent log messages.

Which events in the system are recorded depends to some extent on the application domain. We have identified a set of 15 significant events that we have found to paint a fairly accurate picture of execution history. These include procedure entry/exit, thread behavior (context switches, blocking, sleeping), timer behavior, and interrupts. Most events are logged at various levels in the operating system, however our preprocessor discussed in Section 4.1 also adds debugging code to the application when necessary.

In addition to the system defined events, some application specific events can be added to help diagnosis. While in the system domain it makes sense to log a semaphore operation, in the application domain it may make sense to log particular events related to application behavior, e.g. "I think the fire is starting" in the case of a fire control system. Due to that, the system should have the ability to log custom user events. Saving of user events in a circular buffer should be more configurable than the underlying system level logging.

There is a memory tradeoff between the detail of events logged and the length of logging that is possible. Long event traces (e.g. last 5 minutes of running) are useful when trying to determine at what time a fault occurred, but if there are not enough details in them to know exactly what happened, they are not useful enough to resolve the fault. Although we do not write in flash memory due to the performance impact for deployment mode, debug mode could be modified to allow for writing the event queue in flash memory, allowing for much larger buffer sizes at the expense of execution times.

At the moment, our implementation supports event logging in RAM. Initial experiences with this system suggest that our design choices provide enough detail and that the event buffer is large enough to record enough information about errors. A more detailed analysis of NodeMD's event logging and its effectiveness in conjunction with other system components is described in section 7.

In addition to what is implemented so far, we predict that a useful function would be to allow the equivalent of a request to "preserve the buffer at the moment when this pattern is encountered, and stop logging once when buffer space is exhausted". This would allow us to create *snapshots* of situations in which the error occurs a long time before the node enters debugging mode (e.g. error manifests as crash 10 minutes later), and would allow us the maximum usable data in the event buffer, at the expense of debugging information before and long after the set time.

5.2. Entering a debug mode

Our system is designed to enter a "debug mode" that will take effect when a fault is detected. Before a node enters a faulty state, it jumps to a sequence of methods responsible for stabilizing and preserving the state of the system. This mode could alternatively be initiated at any other time with a specific network command. For the the system faults addressed in this paper, we believe we have solutions to the previously identified faults that ensure that the notification is properly sent.

In addition to this, any memory location (including complete memory dump useful for debugging on simulators) could be sent on user request. However, as the complete memory picture is expensive to transmit over a wireless network, this information will be sent only at the request of the human operator.

At the time of the fault, a set of initial error recovery code freezes critical parts of the system to avoid issues that might arise from the fault, such as a context switch after a stack overflow. Certain application modules are then reinitialized in software to ensure critical operations such as networking needed for notification will be possible even if an error occurred in that module. For example if the application failed inside a call to the radio driver, it's likely that the mutex held by that call would not be released until the driver was essentially "reset". NodeMD takes a software solution to resetting OS components in order to preserve the main memory as much as possible.

After the initial code, NodeMD enters a debugging state with bi-directional communication. A faulty node uses the wireless network to inform the human that the system is in a faulty state and upload the available crash information. Given that the event trace is large enough to span several packets, the initial content of this information is limited to the direct cause of error and the event trace itself. Following the first upload, the node will remain in a duty-cycled standby state waiting for instructions. While NodeMD has a limited implementation of this debugging mode, open research issues include whether jumping to this mode could cause parts of the system at the time of the fault to be lost, and whether certain faulty states could interfere with the correct operation of the debugging code. Additionally, there is a great deal of post-analysis research still to be done regarding reliable network communication between the programmer and the debugging mode.

6. Fault Diagnosis - Closing the Loop

The final piece of the architecture is closing the loop to enable interaction between the human user and faulty nodes in the system. Our system permits two forms of data to be sent to the faulty node, namely queries for more detailed information and updates containing new code. For example, our system is open to retrieving an entire memory dump from a faulty sensor node along with any logged diagnostic information. In terms of remote code updates, our intent is to choose a reasonable combination of reliable code propagation and degree of operating system modularity to enable dynamic reprogramming. The prior work in this area [9, 10, 11] offers many options for closing the loop in fault management systems for WSNs. .

At this point the node is effectively in a remote debugging state and the human is able to perform several actions.

6.1. Remote Debugging

By tweaking the monitoring parameters more information about the fault can be collected (e.g. increasing the size of the event bitmap, and amount of info collected). The node can be restarted to replicate the error and take the new parameters into effect. If more information is still needed, an entire memory dump can be transferred from the sensor node to the human. However, this is an expensive operation because a Mica2/Z node has total of 4 KB of RAM [6], and packet sizes are typically fewer than 50 bytes. This becomes significantly slower and more costly if a faulty node is multiple hops away from the base station.

Our controls allow the human to obtain all available fault information on a node, and at the same time avoids unnecessarily straining power consumption of the node and network (as would be a case if an entire memory dump were initiated). At the same time, it allows on-demand transfer of all information to the human, allowing the human to balance how usable the information is versus how much strain its transfer puts on system resources.

6.2. Code Updates

The Mantis research group is currently working on an implementation that modifies MOS to support dynamic loading of modules as a means of efficient code updates. The MOS system has been supplemented with a thread whose task is to act as an ELF loader [23]. This work is an ongoing collaboration with the Swedish Institute of Computer Science (SICS). Once this is completed, our implementation of NodeMD in MOS will be able to leverage this mechanism for integrating a method for remote code updates.

7. Implementation and Experimental Analysis

To evaluate the effectiveness of NodeMD, we present our implementation results from the use of NodeMD in the Mantis OS (MOS). All of our experimental results are based on this MOS implementation; however the system is not inherently tied to any OS. Notification and diagnostic schemes proposed in this paper could be implemented in any operating system, and although fault detection schemes proposed are tailored towards multithreaded OS's, some of the general techniques are applicable to event driven models as well.

7.1. Detection of Discussed Faults

With respect to the detection of deadlock, livelock, stack overflow and application-specific faults, the implementation of NodeMD is able to successfully detect the target conditions presented in earlier sections.

In our experiments we are able to implement several cases of stack overflow, all of which are immediately detected with an accurate event history leading to that stack overflow. In an ironic twist, while testing the system for deadlock recovery a bug in the recovery code caused a stack overflow. Although the recovery code was not expected to analyze itself and this scenario was unintentionally encountered, NodeMD's stack overflow detection correctly identified the problem.

The exhaustive approach in our stack overflow detection should always detect cases of stack overflow without the risk of false positives. That said, part of our detection does take the 2 byte calling overhead for the detection function into account, and therefore will detect the overflow 2 bytes early.

Section 4.2.3 identifies four specific cases all classified under the general terms deadlock and livelock: complete deadlock, partial deadlock, livelock, and interrupt-disabled livelock. For each of these conditions we evaluated NodeMD's checkpoint-based algorithm on a binary scale: either the deadlock/livelock occurance was caught, or it was not. Our tests use a simple application to reproduce conditions leading to these cases.

The test application starts a set of threads programmed to either run correctly, or encounter one of the problems above. The checkpoint-based approach was able to accurately detect the presence of all cases with 1, 2 and more than 2 threads in combinations of complete deadlock, partial deadlock and livelock.

As for the false positives, due to how algorithm works, if correct thread periods are specified, the algorithm will not incorrectly report a deadlock or livelock. However, the responsibility for estimating this value is left entirely to the application programmer.

Keeping in mind that this algorithm is an extension of the standard watchdog timer approach to detection of deadlock/livelock (by allowing the equivalent of multiple watchdog timers with only one hardware timer), it is not suprising that it has characteristics similar to watchdog timers. Namely, avoidance of false positives, and capturing the occurance of deadlock or livelock within a time bound equal to the sum of the failed thread's period and the periodic interval of the checking algorithm.

Our current implementation was able to detect that interrupt-disabled livelock case occured. This is probably the least common deadlock state, as systems are spending majority of their time with the interrupts enabled. At the moment, implementation is limited to entering debug mode, so signaling of this situation is possible but previous state of the memory is not preserved.

Correctly diagnosing an individual case of deadlock/livelock has actually occurred has also proven to be dependent on the event trace. Checkpoint-based detection algorithm is solution providing information to the human, it expects an programmer to correctly interpret the data in order to diagnose the problem. Experience from the hard-real time community that is using similar tools [19], [26] indicates that similar systems provide significant help in understanding system behavior, and that if anything there is not as much question of is this usefull approach as would logging of additional event categories be usefull.

It is very difficult to design an experiment that measures effectiveness in a general case, but the fact that the combination of hardware watchdogs and event traces have been used for a long time in the hard real time community [19] attests to their usefulness in practice. In addition, NodeMD provides more information than the programmer had earlier, including how deadlock detection location corresponds to the code.

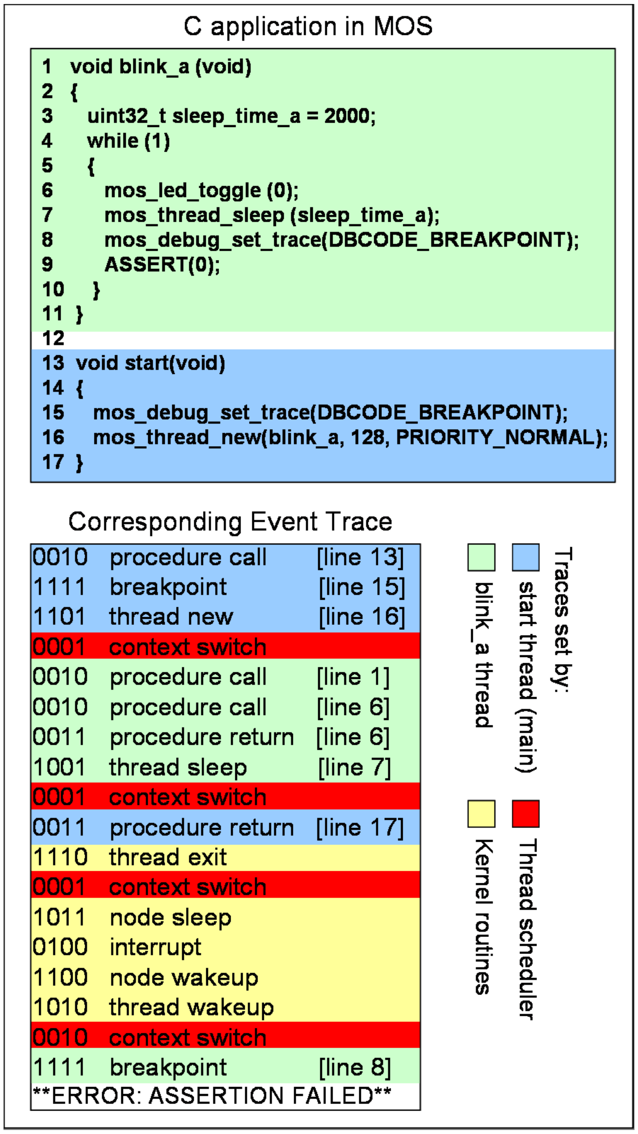
7.2. Event Logging - Detailed Analysis

Using the compressed trace described in section 5.1, our implementation of NodeMD uses 4 bits for each logged event in the system. Using binary bit patterns we have 24̂ (16) possible events. Application behavior is modeled by the following set of events:

- Context switches
- Procedure calls/returns
- Hardware interrupts
- Thread blocks/unblocks, both explicit and OS directed, i.e. interrupt driven devices
- Software timer sets/fires
- Thread sleep/wakeup behavior
- Creating and exiting threads

In Figure 4 we can see how the C code in a simple mos application corresponds to an event trace received when a fault occurs. In the code we see two threads, start and blink_a, respectively identified by blue and green color coding. The trace below has also been color coded to represent the high level picture of system behavior, and line numbers have been added to traces where we can approximate the application-level cursor. We can clearly identify different running thread contexts, context switchs, and kernel routines for thread scheduling and power management. Note that in this example the ASSERT(0) statement simulates the failure of an application-specific fault. We will describe a more complex example in detail further in this section.

At line 8 the application code sets a BREAKPOINT trace code in order to help identify key locations in the application code. “Breakpoint” can be inserted anywhere in code as a “find me” for the programmer, which helps to provide correspondence between code and event traces. In Figure 4 we see this breakpoint appear as the last trace before the error, so since we know where the breakpoint was inserted (which in other cases will likely be compounded with any following events) we can conclude where the error occurred.



C application in MOS

```
1   void blink_a (void)
2   {
3     uint32_t sleep_time_a = 2000;
4     while (1)
5     {
6       mos_led_toggle (0);
7       mos_thread_sleep (sleep_time_a);
8       mos_debug_set_trace(DBCODE_BREAKPOINT);
9       ASSERT(0);
10    }
11  }
12
13  void start(void)
14  {
15    mos_debug_set_trace(DBCODE_BREAKPOINT);
16    mos_thread_new(blink_a, 128, PRIORITY_NORMAL);
17  }
```

Corresponding Event Trace

```
0010  procedure call     [line 13]
1111  breakpoint         [line 15]
1101  thread new         [line 16]
0001  context switch
0010  procedure call     [line 1]
0010  procedure call     [line 6]
0011  procedure return   [line 6]
1001  thread sleep       [line 7]
0001  context switch
0011  procedure return   [line 17]
1110  thread exit
0001  context switch
1011  node sleep
0100  interrupt
1100  node wakeup
1010  thread wakeup
0010  context switch
1111  breakpoint         [line 8]
**ERROR: ASSERTION FAILED**
```

Figure 4. Example application and corrosponding trace data.

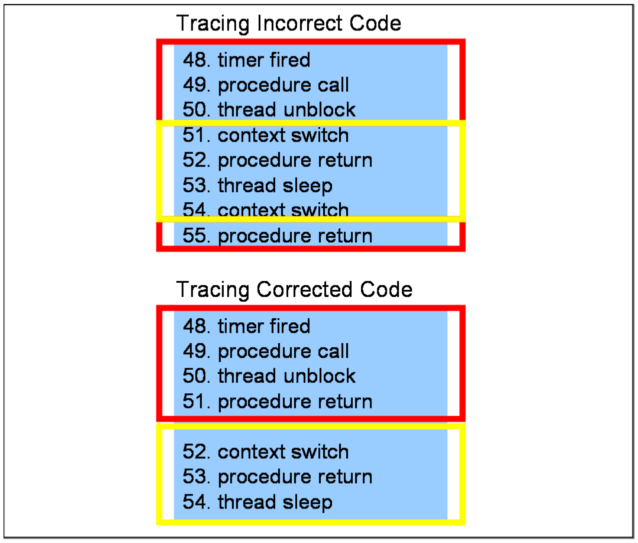
One issue we’ve encountered is the ambiguity when changing contexts with several threads, as opposed to this case which has only two threads. When a context switch occurs, it’s often very difficult to tell which thread is currently running in the trace, especially when those threads have very similar behavior. Part of the optimization needed by our run-time logging is a tradeoff between the amount of history to record and the detail of each record. The requirements will vary widely on an application-specific basis, so we allow the application to configure such parameters as buffer size and bits per trace code. A flag at compile time could allow the scheduler to set a trace at each context switch indicating which thread is now running. Alternatively, application programmers can increase the trace code size to explicitly create their own trace codes on top of the default 16.

During our implementation we uncovered an event trace of an actual legacy bug in MOS, previously only detected by unpredictable behavior and code analysis. Recently, several MOS programmers had identified a bug where certain thread behavior would unknowningly initiate a context switch while within an interrupt handler. Specifically, when an interrupt handler posted a semaphore that unblocked a thread, the kernel would initiate a thread dispatch to immediately process the unblocked thread (if that thread was at the front of the ready queue). In most cases, this would not pose a problem because a blocking operation in the other thread would immediately context switch back to the handler, which would then exit. However, under certain conditions, MOS programmers reported a visible *1 second* delay would occur between the entry and return from an interrupt handler. While that specific example is not available, we identifed the occurance of this phenomena while testing this system. The before-and-after traces from the bugged code and then the corrected code are shown in Figure 5.

Notice the highlighted traces in the first trace section. Areas in red are execution within the interrupt handler, areas in yellow are outside of the handler. When a timer fires [48], it's handler procedure is called [49] and the semaphore is posted [50] (unblocking a thread waiting for that semaphore). Immediately we recognize the system context switch out of the handler [51] *before* the trace reports a procedure return. This indicates our handler has not returned which results in several unpredicted conditions, one of which is the new running thread remains in the interrupt disabled context initiated by the handler. Fortunately within a few instructions the other thread goes to sleep [53] and context returns to the handler [54], which then returns [55]. Clearly there could have been a serious context error if the external thread did not block immediately.

In the second trace section, the same set of code is run with the OS bug fixed. Since the section highlighted in red is the entire interrupt handler routine without interruptions, we have verified that the bug has been fixed.

Finally, while the shown traces are true to the actual execution in most cases, the events in this model are implemented within MOS on a best-effort basis. One thing we do not include are scheduler interrupts (occuring every 1 ms in MOS), and instead record behavior occasionally resulting from that interrupt, such as context switching or software timer processing. Recording a trace every 1 ms may be realistic, but by overwriting useful data in the buffer that rapidly it defies the practical use of this system. If an error were to occur directly related to that interrupt, we would be unable to visualize that error in the trace.


Tracing Incorrect Code

48. timer fired
49. procedure call
50. thread unblock
51. context switch
52. procedure return
53. thread sleep
54. context switch
55. procedure return

Tracing Corrected Code

48. timer fired
49. procedure call
50. thread unblock
51. procedure return
52. context switch
53. procedure return
54. thread sleep

Figure 5. Before-and-after traces from a bug in MOS, where an application could unknowningly context switch out of an interrupt handler.

7.3. Event Trace Evaluation

One of the most difficult questions posed by our system is the optimal event trace size. How can we most efficiently use our limited memory to log only useful data? In some cases simply covering all events within the period of each thread is acceptable, in others more extensive information is necessary. In general, the factors that influence our buffer "burn rate" are entirely application specific: the number of threads and software timers, the number of function calls within that concurrent code, and even the types of functions called determine the required size for a certain time window.

Table 1 identifies the number of traces logged in MOS routines commonly called by sensor applications.

As a case study, let's describe the sensor networking application used in the FireWxNet deployment. This is a very complex application encompassing nearly all of the features in MOS. Within the application, two threads are spawned. In the first, every 1 second data is read from 4 different sensors into a packet buffer and sent over the radio. The other

Routine	Traces Required
mos_led_blink	2
printf	13 + n chars
dev_read	18
com_send (CC2420)	23
com_recv	31
com_recv_timed	32 (success)
com_recv_timed	12 (timed out)

Table 1. Trace requirements for common application-called routines in MOS.

thread repeats a blocking receive on a 5 second timeout. During the execution of these threads, a wind sensor hardware interrupt fires periodically, and three software timers retrieve data from the wind sensor, updates the neighbor table, and handles state transitions for power management. After 1 minute awake time, the node changes to a low power sleep state for the following 14 minutes.

In our experiments, processing a single iteration of the sending and receiving threads, plus all concurrent timers, requires approximately 250 traces. Even one iteration of just the sending thread, without timers, requires 98 traces, resulting from several dev_read() calls followed by a com_send(). Given that the scheduled awake time for this duty cycle leads to at least 60 iterations of the sending thread, logging the entire awake period is outrageously expensive - around 6000 traces, or 3000 bytes using our approach.

However, many conclusions can be drawn from only the most recent events. A stack overflow trace showing a series of function calls without returns may be due to an unintentional recursion. On the other hand, a stack overflow showing relatively normal behavior indicates that the allocated stack size was probably too small, and certain conditions lead to a slightly higher requirement than expected.

Alternative methods for determining buffer sizes presents an interesting topic for future work in this area. A static analyzer could be incorporated into the system for optimal buffer size allocation. Using a database of trace sizes expanded from the limited example in Figure **??** and a set of basic requirements for application programs, a rough estimate for buffer space used per time could be determined. Additionally, allowing for selective logging of only particular events and for “freezing” the buffer content after particular patterns of events has occurred (as suggested in section 5.1.) would allow for the available buffer space to be utilized only for the time period with the most useful debugging information.

7.4. General Overhead

How do our algorithms actually impact system performance? One of NodeMD’s primary objectives was to remain lightweight and unintrusive as possible to the underlying application and OS. Evaluating the simplest blink_led application against the FireWxNet application from our case study, the requirements of NodeMD are quite reasonable. For reference, the blink_led application uses a single thread to frequently toggle an LED (the FireWxNet code is described in the above section). Table 7.4 shows a comparison between the original data in MOS followed by the compounded overhead with NodeMD included.

Application	Original MOS	NodeMD Included
blink_led RAM	585	887
blink_led ROM	25212	28768
FireWxNet RAM	780	1072
FireWxNet ROM	30204	34470

Table 2. Overhead analysis in MOS, see text for details.

Using these results we see an increase in main memory requirements corresponding to 92 bytes + trace buffer size + 10 bytes per checkpoint). These experiments assume 1 checkpoint per application thread, which translates to an extra 10 bytes in the blink_led application and 20 bytes in the FireWxNet application. The 92 bytes of static overhead is accrued from necessary globals in the implementation, including a 67 byte packet buffer. We also see an additional 1̃4% increase in program memory, a result of the preprocessor-added debugging code, the added amount being dependent on the complexity of the application. Given the features added by NodeMD, and the flexiblity to tailor several trace buffer details to minimize or expand overhead, the requirement costs of NodeMD are far outweighed by its contributions.

One of the important qualities of the logging system is that it does not impact program timing in substantial way. In effect, each log operation takes either 43 or 79 cycles, depending on whether the log crosses a byte boundary or not. Although it is possible that this would be enough to change timing of the program, this is a fairly small number. Calling mos_led_on() costs 35 cycles, so writing a trace is equivalent to turning on between one and two LEDs, which is essentially invisible to the application.

The algorithm itself is a straightforward sequence of instructions and is unlikely to dramatically change the order of execution, as opposed to a printf statement that initiates several blocking operations, context switches, and hardware interrupts.

Likewise, the detection of stack overflow uses only 32 cycles to check at each function call, which is even less likely to noticeably impact timing.

Although exhaustively checking procedure entry points and frequently calling traces can become expensive, generally WSN applications are considered to have a relative abundance of CPU cycles [22]. The FireWxNet application spends the majority of its awake cycle idling, waiting for timers, data available on the radio, data available on the ADC, and other system events. Therefore the additional cycles added by NodeMD should not introduce substantial timing delays due to CPU processing latency.

8. Future Work

We have identified several key areas for future work as we have presented the paper. In addition, NodeMD needs more in situ testing, in order to prove its capabilities in deployed environments. We plan to instrument a WSN field application in the coming months with NodeMD. We would like to be able to assess the accuracy of such a NodeMD deployment in capturing bugs that occur in the field. We would also like to demonstrate the generalizability of the proposed detection algorithms and notification architecture of NodeMD to other embedded OS's such as micro-C OS.

9. Conclusions

This paper has described NodeMD, a comprehensive system implementation for detection, notification, and diagnosis of sofware failures in remote wireless sensor nodes, thereby minimizing the need for on-site redeployment of failed nodes. NodeMD is capable of detecting a broad spectrum of software faults as they occur and before the completely disable a node, including stack overflow, deadlock, livelock, and application-specific faults. We present specific novel detection algorithms: aspect-based stack overflow detection; and application-defined thread *checkpoints* that act as custom watchdog timers within each thread. We introduce a debug mode that halts the embedded system upon detection of a failure, and notifies a remote user via a summarized event trace in the form of a bit vector. Our system closes the loop by permitting interactive queries from the remote human user for more diagnostic state. We present detailed implementations and experimental analysis of all of our fault detection algorithms on real-world applications such as the FireWxNet. We observe that NodeMD has already captured two real-world bugs in the Mantis OS.

References

[1] O. Spinczyk, A. Gal, W. Schrder-Preikschat, “AspectC++: An Aspect-Oriented Extension to C++”, Proceedings of the 40th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific 2002), Sydney, Australia, 2002

[2] R. Kumar Rengaswamy, E. Kohler, M. Svrivastava, “Software Based Memory Protection In Sensor Nodes”. Proceedings of the Third Workshop on Embedded Sensor Networks (EMNETS), May 2006.

[3] A. Mainwaring, J. Polastre, R. Szewczyk, D. Culler, and J. Anderson. Wireless sensor networks for habitat monitoring. In 1st ACM International Workshop on Wireless Sensor Networks and Applicatoins (WSNA 2002), Atlanta, GA, September 2002.

[4] C. Hartung, C. Seielstad, S. Holbrook and R. Han, “FireWxNet: A Multi- Tiered Portable Wireless System for Monitoring Weather Conditions in Wildland Fire Environments”, Proceedings of the Fourth International Conference on Mobile Systems, Applications, and Services (MobiSys), 2006, pp. 28-41.

[5] G. Werner-Allen, K. Lorincz, M. Ruiz, O. Marcillo, J. Johnson, J. Lees, and M. Welsh. “Deploying a Wireless Sensor Network on an Active Volcano”, IEEE Internet Computing, Special Issue on Data-Driven Applications in Sensor Networks, vol. 10, no. 2, March/April 2006, pp. 18-25.

[6] Crossbow Technologies: “Mica2 Series (MPR4x0)”, available at http://www.xbow.com.

[7] Q. Wang, Y. Zhu, L. Cheng, ”Reprogramming Wireless Sensor Networks: Challenges and Approaches”, IEEE Network, vol. 20, no. 3, May/June 2006, pp. 48-55.

[8] S. Bhatti, J. Carlson, H. Dai, J. Deng, J. Rose, A. Sheth, B. Shucker, C. Gruenwald, A. Torgerson, R. Han, ”MANTIS OS: An Embedded Multithreaded Operating System for Wireless Micro Sensor Platforms, ” ACM/Kluwer Mobile Networks & Applications (MONET), Special Issue on Wireless Sensor Networks, vol. 10, no. 4, August 2005, guest co-editors P. Ramanathan, R. Govindan and K. Sivalingam, pp. 563-579.

[9] C. Han, R. Kumar, R. Shea, E. Kohler, M. Srivastava, “SOS: A dynamic operating system for sensor networks”. Proceedings of the Third International Conference on Mobile Systems, Applications, And Services (Mobisys), 2005.

[10] J. Hui, D. Culler. “The Dynamic Behavior of a Data Dissemination Protocol for Network Programming at Scale”. Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems, SenSys, 2004.

[11] L. A. Phillips, “Aqueduct: Robust and Efficient Code Propagation in Heterogeneous Wireless Sensor Networks,” Master's thesis, Univ. CO, 2005.

[12] B. Meyer: Applying “Design by Contract”, in Computer (IEEE), vol. 25, no. 10, October 1992, pages 40-51.

[13] N. Ramanathan, K. Chang, R. Kapur, L. Girod, E. Kohler, and D. Estrin. “Sympathy for the Sensor Network Debugger”. In the Proceedings of 3rd ACM Conference on Embedded Networked Sensor Systems (SenSys 05), Nov.

2005. San Diego, California.

[14] G. Tolle and D. Culler. “Design of an Application-Cooperative Management System for Wireless Sensor Networks.” Proceedings of the Second European Workshop on Wireless Sensor Networks (EWSN), 2005.

[15] K. Whitehouse, G. Tolle, J. Taneja, C. Sharp, S. Kim, J. Jeong, J. Hui, P. Dutta, D. Culler. “Marionette: Providing an Interactive Environment for Wireless Debugging and Development”. In The Fifth International Conference on Information Processing in Sensor Networks (IPSN’06).

[16] R. Wilson, “Shedding light on the Mars rover malfunction”, EE Times, 02/20/04.

[17] Serial line. http://www.pa.msu.edu/hep/d0/ftp/run2b/l1cal/hardware/channel_link_tester/channel_link_tester.txt

[18] H. Tokuda, C. Mercer, “ARTS: A Distributed Real-Time Kernel”, ACM SIGOPS Operating Systems Review, vol. 23, issue 3, July 1989, pp. 29-53

[19] D. Wilner, “WindView: a tool for understanding real-time embedded software through system vizualization”, Proceedings of the ACM SIGPLAN 1995 workshop on Languages, compilers and tools for real-time systems, 1995, pp. 117-123

[20] J. Regehr, A. Reid, K. Webb, “Eliminating stack overflow by abstract interpretation”, ACM Transactions on Embedded Computing Systems (TECS) vol. 4 , Issue 4, November 2005, pp. 751-778

[21] JTAG distributor. www.digikey.com

[22] L. Gu, J. Stankovic, “t-kernel: A Naturalizing OS Kernel for Low-Power Cost-Effective Computers”. In Proceedings of 4th ACM Conference on Embedded Networked Sensor Systems (SenSys 06), Nov. 2006. Boulder, Colorado.

[23] A. Dunkels, N. Finne, J. Eriksson, and T. Voigt. “Run-Time Dynamic Linking for Reprogramming Wireless Sensor Networks.” ACM SenSys, 2006.

[24] W. McCartney, N. Sridhar, “Abstractions For Safe Concurrent Programming In Networked Embedded Systems”. In Proceedings of 4th ACM Conference on Embedded Networked Sensor Systems (SenSys 06), Nov. 2006. Boulder, Colorado.

[25] GCC, the GNU Compiler Collection, ported to the AVR platform. http://gcc.gnu.org/

[26] K. Bradley, J.K. Strosnider, “An application of complex task modeling”, Real-Time Technology and Applications Symposium, 1998. Proceedings. Fourth IEEE, Jun 1998, pp. 85-90

[27] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, J. Irwin, “Aspect-Oriented Programming, Proceedings of the European Conference on Object-Oriented Programming, 1997, vol.1241, pp.220242.