NodeMD: Diagnosing Node-Level Faults in Remote Wireless Sensor Systems

Veljko Krunic, Eric Trumpler, Richard Han
Department of Computer Science
University of Colorado at 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 costs of deployment of wireless sensor systems often far exceed the cumulative costs 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 the node into a stable state that enables further diagnosis and correction, thus avoiding on-site redeployment. We analyze the performance of NodeMD on a real world application of wireless sensor systems.*

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*diagnostics, distributed debugging, error handling and recovery, tracing*; C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design—*wireless communication*

General Terms

Design, Experimentation, Management, Performance, Reliability

Keywords

Diagnosis, Software Fault, Wireless Sensor Networks, Deployment

1. INTRODUCTION

The vision of wireless sensor networks (WSNs) typically consists of a large number of very low cost sensor nodes that can be spread over a wide area to collect environmental data and relay that data back to a remote database or server via a self-organizing wireless mesh network. WSNs are often deployed in distant rugged environments, e.g. Great Duck Island off the coast of Maine [2], around wildfires in the Bitterroot National Forest in Idaho [3], and surrounding an active volcano in Ecuador [4]. These types of deployments are expensive and sometimes even dangerous to deployment personnel. For example, in the FireWxNet [3] 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 difficulty of WSN deployments is that software bugs are inevitably encountered in the field, following a familiar theme that has been experienced all too commonly in other deployed software systems. Commercial applications and operating systems typically have large quality-control resources devoted to testing of software prior to deployment, yet still encounter software bugs in the field that require frequent patching. Despite exhaustive testing, commercial handheld devices with embedded software such as cell phones and wireless PDAs continue to suffer from software glitches during operation. As some well publicized software failures during space missions are showing (e.g. Mars PathFinder [15, 28]), software errors are a fact of life even for NASA, which has considerable resources at its disposal for testing prior to launching a mission. Our expectation is that WSN applications will face similar difficulties with software bugs that occur in the field. Moreover, we expect these problems to be exacerbated in WSNs by two factors: WSN systems typically are limited by having much scarcer resources available for testing than commercial and NASA-funded systems; and the data-driven nature of WSNs can create an unexpected fault-inducing combination of inputs that is difficult to forecast during limited lab testing. Indeed, our own experiences deploying FireWxNet confirmed that software bugs arose during our deployment despite our best efforts to eliminate errors through lab testing.

The cost of repairing a node that has been crippled due to a software failure is especially high in WSN applications, due to the time, money, and effort required to revisit a node deployed in such remote rugged terrain. Solutions available in other domains to address software failures do not easily apply to the case of WSNs, due to these extreme conditions of deployment as well as the extreme resource constraints characteristic of WSNs. For example, to achieve the vision of many low cost sensor nodes, today's sensor motes typically have extremely limited memory available, e.g. 4 KB of RAM and 128 KB of flash on MICA2 [5] class sensor motes. The embedded controllers characteristic of these sensor nodes also typically lack hardware memory protection and MMU units. Given these substantial hardware limitations, i.e. up to six orders of magnitude less RAM for WSN systems than for PC systems, we expect desktop-class solutions for detecting and repairing software faults to be too expensive to directly apply to the resource-constrained domain of

MobiSys'07, June 11–13, 2007, San Juan, Puerto Rico, USA.

WSNs. Embedded systems such as cell phones are closer in resources to WSN systems, e.g. tens of MBs of RAM, but even here their solutions do not necessarily apply. For example, when cell phones/PDAs become unresponsive due to faulty embedded software, their owners can often fix the problem by manually resetting and/or power cycling the device. Manual reset is a prohibitively expensive option in remote wireless sensor deployments, requiring on-site visitation.

A system that could catch a software fault before it completely disables a remote sensor node, and can provide diagnostic information to remotely troubleshoot the root cause of the fault, would be invaluable to in situ WSN deployments. 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. Even if this situation occurs in the lab during testing, the ability to provide more information than just a "black hole of silence" is clearly beneficial. Such a diagnostic system would be useful not only for in situ applications but also for troubleshooting errors during the testing phase.

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) remotely diagnosing the root cause of the fault, thereby substantially reducing the need for costly redeployment of nodes through on-site visits. Our solution must be tailored for WSNs, i.e. it must be lightweight and have a small footprint appropriate for the sensor network environment.

A medical analogy can provide some insight into the state of the art with respect to current methods of sensor node debugging. Visiting a failed node in the field is similar to a country doctor that needs to visit a remote area to treat a sick patient. For both a doctor's in-home visit and on-site repair of failed remote sensor nodes, the cost of the visit is prohibitively expensive. The WSN community has proposed a variety of approaches to mitigate these costs. SOS [8] provides an ability to remotely patch a sensor OS, and can be seen as analogous to a mail-order pharmacy that remotely provides medicine to alleviate a sickness. Marionnete [14] and Nucleus [13] provide the ability to remotely query a node for run-time state information, and is analogous to a doctor using the telephone to query a sick patient as to their health. t-kernel [23] provides a general framework that seeks to prevent certain software faults like livelock, but not others such as stack overflow, and can be seen as vaccinating a patient against certain diseases but not others. Nucleus also provides an event log in flash that can be recovered after a node has died, and is analogous to providing post-mortem analysis.

Given all these pieces of the puzzle, we are still missing effective tools that are equivalent to a patient proactively reporting the rapid onset and current symptoms of an illness, as well as their history of behavior that led up to that illness, before that illness completely incapacitates that patient. 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". Given today's WSN debugging tools, a node can still fail without reporting any information about the failure at the time of the failure. As a result, today's WSN community still cannot completely avoid a need for the equivalent of in-home visits.

NodeMD is the last piece of the puzzle that is necessary to realize the equivalent of a fully capable "remote doctor" in the world of WSNs and thereby drastically reducing the need for on-site visits. 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 before they disable a node, good diagnostic tools are provided for timely diagnosis of the problem, and the appropriate remedy can be applied by remotely updating a sensor node with debugged code. Ultimately the goal of our system is to bring node debugging in these challenging, resource-constrained, remote wireless environments to a level that is as useful as what 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 a real world sensor application. The techniques proposed in this paper are designed to be generalizable across many different systems, and we foresee future implementations of NodeMD being 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 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 [7] 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 either an in situ wireless deployment or testbed environment, some systems are emerging that provide limited visibility into fault behavior. The Sympathy system [12] 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 application and OS software failures on a node.

Nucleus [13], 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 theme we share is that 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 because the information logged vs. storage size is sparse. Also, 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 [23], a reliable OS kernel, takes an approach that ensures the system is always able to retake control from an application. At a low level, each branch instruction first jumps to the system 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" that allows the system to recover from problems such as deadlock or livelock. However, t-kernel is designed for reliability rather than debugging, and only ensures that the system can always execute. It does not react to the onset of such faults it may circumvent, i.e. deadlock and livelock. Nor does it address how to detect other types of faults, such as stack overflow, or how to efficiently provide useful information for fault diagnosis.

Marionette [14] provides a mechanism to query the memory in nodes for their state. It is specific to TinyOS, and does not focus on detection, preemption, and notification of faults as they occur.

A variety of approaches for remote code updates in WSNs have been proposed, and are summarized in [6]. These approaches can be roughly divided into a networking component that achieves reliable code propagation, e.g. Deluge [9] and Aqueduct [10], and an operating system component that enables efficient update of code images on a sensor node, e.g. SOS [8] or the ELF loader [24]. 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 NodeMD's architecture. For example, the ELF dynamic modules loader [24] 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 automated 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 immediately after a fault.
- 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.

NodeMD must accomplish the above diagnostic features while achieving a variety of other design goals. First, it is essential that fault detection and notification be extremely memory-efficient and low overhead in terms of CPU and radio bandwidth, to fit within the extreme resource constraints demanded by deployed sensor nodes. This has strong implications, for example on streamlining the design of the event logging in main memory. Second, the design of NodeMD should afford the human user flexibility to extend and customize its diagnostic capabilities, i.e. 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. Users can further request more detailed diagnostic information when a node is in the halted but stable/responsive debug mode. NodeMD also allows programmers to customize event logging by adding custom events to the history trace. 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 also to event-driven single-stack systems like TinyOS.

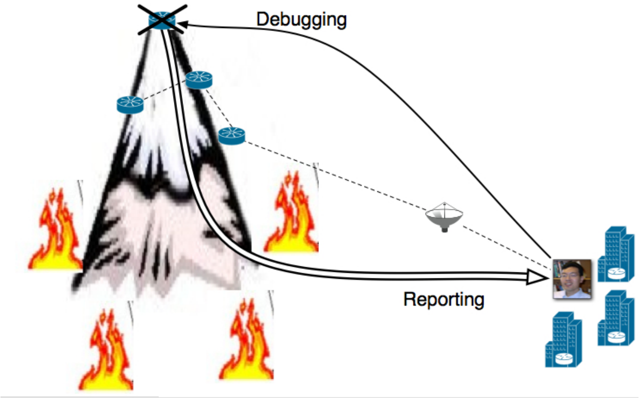


Figure 1: System architecture of NodeMD.

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 application-specific faults. NodeMD is architected so that other detectors can be added to our system, such as detection of out-of-bounds memory writes, but at present we have focused first on detecting these three 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 implicitly 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 [5] 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 [21, 25], data dependencies common in WSNs make it difficult to choose a stack size that is minimal yet guaranteed never to be exceeded. In addition, errors in the code can 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. The detector needs to be lightweight as it may have to run often in the really resource constrained environment of the sensor node. As explained below, we employ an aspect-like [19] approach for detection, which checks for stack overflow at each procedure call. This approach 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 believe that this is a lightweight practical approach that also makes few assumptions about the code. For example, our stack overflow detector 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. In addition, the software running on typical sensor nodes today is usually available in the source form. As a result, we opted to base our stack overflow detection approach on a source code instrumentation approach.

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 [19] and AOP [20], although we have used a custom implementation 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! 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.

The stack checking algorithm itself compares the current thread's stack top to the stack pointer (SP) just after a procedure is called. 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. To address stack increases due to parameter passing, we protect a small amount of "padding" next to the top of the stack, and report a stack overflow if this memory is at risk. Insulating the overflow from other memory ensures that the soon-to-be-executed recovery code has not been corrupted.

As a concrete example, consider how the AVR-GCC [26] compiler handles initial entry point to a scope, e.g. a procedure. This is the compiler used for MICA class sensor motes that use the AVR family of microcontrollers. In general, stack space is consumed by three components to a procedure call: (1) the call overhead (return address, old stack pointer), (2) the passed parameters to the call, and (3) the local data defined in the new function. Given that the call overhead is constant, this component of the problem is easily solved by enforcing an extra 4 bytes of "red zone" to always be available above the SP.

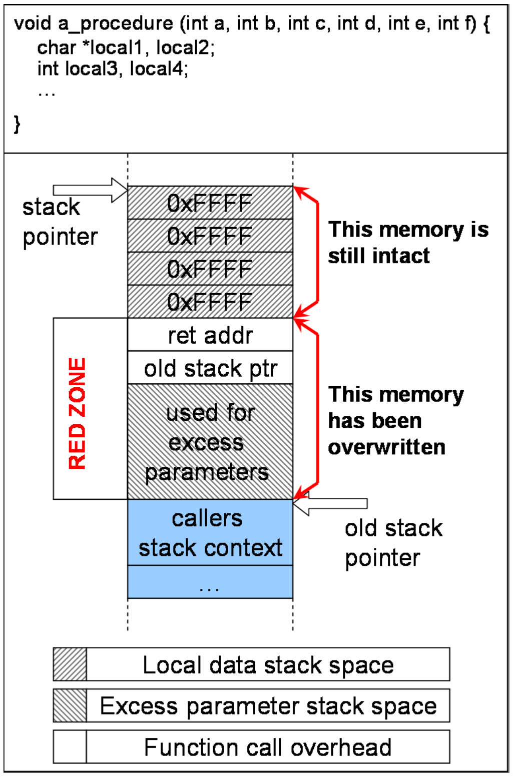
Designing the stack checking algorithm to exist within the called function is actually the result of some initial analysis of how AVR-GCC initializes local variables. When a procedure call is compiled using AVR-GCC, the compiler calculates the stack requirements for all local variables in a procedure in its first pass. During the second compiler pass it uses this calculated value to increment the stack pointer by *the total* stack usage due to local data - all before the first line in the procedure is run. The conclusion this implies is that the stack pointer is already pre-incremented when the procedure's code begins to execute. Local variables are written to the stack using STD (Store indirect with displacement) instructions offset from the new stack pointer rather than with PUSH instructions. The advantageous result is any stack space reserved for local data will not be overwritten until those local variables are defined. Inserting a stack checking algorithm in the procedure's first line of C code, before any local definitions, ensures that we can read the pre-incremented SP and identify a future stack overflow *before* any memory has been corrupted. However, this behavior only holds true for stack overflows due to locally defined data.

Parameter pushing poses the most difficult challenge to our stack overflow detection method. Writing a parameter onto the stack uses the PUSH instruction, leading to much more unpredictable fluctuations in the stack pointer at run time. The stack consumed by parameters cannot be determined inside the function call, as we do with local data, without some stack space having already been corrupted before our detection algorithm is able to run. To counteract potential memory corruption, the most viable option is to preemptively assess how much stack could possibly be used by parameters of future procedure calls, and calculate how much additional "red zone" will ensure those parameters will not overflow the stack. To reiterate, a red zone is a buffer of stack space above the stack pointer that must be available to safely call procedures. For example, lets say a procedure A will call procedure B (int a, int b, int c) and procedure C (long a, long b, long c). Using a value of 2 bytes per int and 4 bytes per long, procedure B requires 6 bytes of stack for parameters, and procedure C requires 12 bytes. Procedure A's red zone must cover the largest quantity of parameters defined by either procedure B or procedure C, in this case 12 bytes for procedure C.

However, most of the expected red zone costs are mitigated within AVR-GCC. AVR-GCC uses volatile 8-bit registers 18-25 to pass parameters *without* pushing them on the stack. Through testing we were consistently able to pass 4 or fewer parameters totaling to 8 or fewer bytes without additional stack usage. Only excess parameters that are not able to be passed through these registers need to be covered by a red zone buffer. We anticipate that calculating the red zone required for excess parameters is most effectively achieved through static analysis and is ongoing research in this project.

Figure 2 shows a snapshot of the stack contents at the entry point for a given function 'a_procedure'. a_procedure has a total of 6 parameters and defines 4 local variables, all using 2 byte data structures for simplicity. At the call of a_procedure the first 4 parameters fill the available volatile registers. The extra parameters are pushed onto the stack, followed by the calling procedure's stack pointer and the return address. The predetermined size of local data (8 bytes) is then added to the stack pointer before the first lines of procedure code begin to execute.

In the current procedure context (between the old and new stack pointers), the stack can be divided into memory that has already been written, and memory that is still uninitialized and intact. Although stack space for the local data has been preallocated, no local definitions have yet been reached and no data has been written to those stack locations. The significant risk for stack overflow



```
void a_procedure (int a, int b, int c, int d, int e, int f) {
    char *local1, local2;
    int local3, local4;
    …
}
```

Figure 2: Stack snapshot at the point of entry to a_procedure, AVR-GCC assembly.

concerns the space used for parameters not passed through registers and the function call overhead. This chunk of memory is our red zone, the stack space we have to assume will be overwritten before our detection algorithm is able to run. Validation of the red zone defined in Figure 2 was done at the original stack overflow check in the *caller*. If the chunk of memory defined by the red zone was not available when the caller's procedure began, a stack overflow would have been detected. Consequently, since no stack overflow was detected in the caller, the red zone buffer at the call to a_procedure can be overwritten without risking a stack overflow. In effect, the historical sequence up to the snapshot is such: (1) the caller ensures the red zone is available, (2) the caller calls a_procedure, (3) a_procedure checks for stack overflow due to local data, (4) a_procedure ensures the new red zone is available (stack not shown).

On the AVR (Mica2/Z) platform the AVR-GCC compiler uses several tweaks and optimizations for parameter passing, but for all intents and purposes the visible behavior is the same. It is important to note that this design does utilize several features specific to the platform and compiler. Future adaptations will require compiler-specific algorithms, but an open research issue is whether an efficient generic approach can be found.

Finally, when this case is detected, we wish to avoid any further corruption of another thread's stack. Any continued execution risks further memory corruption. Thus, it's critical after detection 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. Deadlock occurs for example when two threads are blocked in a way that each one is waiting on some work of the other to complete, and thus neither proceeds.

Common cases of deadlock arise from a collective circular dependence on semaphores, mutexes, timer interrupts, and data conditions (the thread will unblock when the temperature has exceeded X degrees). Classic solutions to deadlock such as constructing resource allocation graphs and analyzing for circular dependencies appear to be quite heavyweight and complex for sensor nodes. Our detector implements a simpler solution suitable for resource-constrained WSNs.

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 could loop continuously in a section of its code, unable to ever meet the condition for exiting that section of the code, either because of programmer error or unexpected factors. The result would be that desired forward progress is stopped in the rest of the thread.

Many of the conditions that cause deadlock can also result in a livelock by polling on a condition rather than blocking. In addition to 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. In this case, the software detector would be unable to run and catch the livelock. Consequently, a hardware watchdog timer would be needed to reset the system. In the following, we propose a hybrid approach that combines software detection of livelock and deadlock with a hardware watchdog timer to handle the severest cases of interrupt-disabled livelock/deadlock.

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, i.e. rather than maintaining complex state in resource allocation graphs to analyze for circular dependencies, our approach is instead to identify their *symptoms* and draw a diagnostic conclusion based on those symptoms. A variety of cases arise:

- Some threads deadlocked (partial deadlock)
- All threads deadlocked
- At least one thread livelocked
- One thread livelocked in an 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 periodically repeated segments of code, within a while() block for example. When either a deadlock or livelock occurs in that thread, certain statements within that loop will fail to 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 a failure to recur. If the watchdog is reset at every iteration of a while statement, the system recovers itself when that reset does not occur. However, applying a watchdog to a multithreaded system presents a challenge: how can a single timer ensure that several threads are all executing properly? Yet another extensibility issue exists due to the logistics of watchdogs. On our target AVR (Mica2/Z) platform, the 8-bit hardware timers restrict the maximum watchdog length to 2 seconds.

Instead, the solution proposed by NodeMD takes a primarily software-based approach. We begin with an assumption that in a multithreaded sensor OS, each application thread can provide an estimate of its "period" of execution, i.e. the time it takes for its while() loop to iterate. We believe that this is quite reasonable for a wide cross section of today's WSN applications, e.g. sensing and networking applications, which are often driven by specific duty cycles with well known wake/sleep periods. Combining the repetitive nature of these 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, e.g. a timeout may be twice the thread's period. The application programmer effectively states some constraints about the program, i.e. that a thread ought to execute every Y seconds, and NodeMD's detection schemes determine if those application constraints have been violated, namely the thread has not executed for $2 * Y$ seconds so there must be a problem. While this simple approach requires estimation and some manual indication to the system by the application, e.g. by insertion of a small amount of code, it is a best-effort compromise. Our approach takes advantage of typical WSN application behavior while avoiding the system making assumptions about each application's timing.

Our implementation introduces the notion of a thread *checkpoint* to emulate the behavior of a hardware watchdog. As shown in Figure 3, each thread declares a checkpoint (1), and then *registers* that checkpoint, specifying the expected period of this checkpoint (2). Next, a checkpoint_reached(&mycheckpoint) call is added to a critical point in the thread that ought to be repeated during correct periodic execution of the thread (3). Whenever a checkpoint is reached during execution, a parameter in the checkpoint is reset to the current system real time, effectively time stamping the most recent iteration by the thread through this critical point in the code. At a periodic interval, the kernel inspects all registered checkpoints and compares them to the current real time (CRT). If the difference between the CRT and the thread's last time stamp exceeds the thread's timeout value, our algorithm assumes the thread has livelocked or deadlocked and enters error recovery code. As seen by the (#) indicators in Figure 3, this approach requires only 3 additional lines per checkpoint.

```
#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)
   {
      checkpoint_reached(&mycheckpoint);    (3)
      ...
      thread_sleep(sleep_time_a);
   }
}
```

Figure 3: Example checkpoint code.

The thread's period should be carefully estimated. As long as the thread period is not *under*estimated, the detection algorithm will correctly detect a failure due to livelock or deadlock, albeit with a greater delay. For example, suppose the true thread period is Y seconds per loop iteration, the estimated period Z is greater than or equal to Y, the timeout is twice the estimate, i.e. $2 * Z$, and NodeMD's detector is invoked with a delay D after the timeout. If a livelock/deadlock occurs for this thread immediately after the checkpoint has been reached, then the worst-case delay in detecting the failure is $2 * Z + D$. In the case that $Z = Y$, i.e. the thread period is accurately estimated, then the worst-case delay is $2*Y+D$. If in addition the timeout is set at the period, then the delay is $Y + D$. In all these cases, the algorithm will correctly detect the failure, albeit with some lag. This delay should not be a big limitation in typical deployments, as we believe the nature of deadlock/livelock does not typically require immediate detection. For example, if the detector is invoked every $D = 2$ seconds, sensor data is reported by this sensing thread every $Z = Y = 5$ seconds, and the timeout is set at $2 * Z = 10$ seconds, then NodeMD should catch the livelock at worst about 12 seconds after it occurs, which we believe to be acceptable for most deployments. However, if the thread period is significantly underestimated, e.g. $Z = Y/4$, then triggering of the detector can result in a false positive. In such a case, the timeout expires too soon, and the detector checks for expiration so quickly thereafter (D is small) that it finds the checkpoint to have expired and falsely identifies lack of progress, even though the thread is still legitimately making progress. As a result, it is important to err on the side of overestimating the period for this technique to be effective.

Placement of the checkpoint in a loop is critical for correct detection of livelock/deadlock, i.e. lack of thread progress. For example, consider the thread in Figure 4 that has both an outer loop OL and multiple inner loops $I1$ and $I2$. If the checkpoint is placed in the inner loop $I1$ at (1), then a livelock may occur within the inner loop $I1$, causing the checkpoint to be continuously updated even though the thread is not making progress through the outer loop. In this case, incorrect placement of the checkpoint fails to detect the livelock. However, if the checkpoint is placed in the outer loop OL at (2), then a livelock within any of the inner loops will be detected on the next invocation of the livelock detector, as the checkpoint will not be reached on the outer loop.

Verifying the timeout of each checkpoint is done at the kernel level. In a multi-threaded system, control is typically returned to the kernel's scheduler via the hardware time slice timer, so that threads can be periodically context switched. Even if multiple application threads deadlock and/or livelock in a "normal" manner, i.e. contending for shared system resources in a non-interrupt-disabled context, then control still returns to the scheduler, and application deadlock/livelock do not paralyze such a system. In this way, the OS can periodically invoke NodeMD's deadlock/livelock detector to check to see whether any checkpoints have expired. Also, this approach enables NodeMD, immediately after detecting a livelock/deadlock, to gracefully drop the system into a stable debug mode that is still responsive to interaction from the remote human debugger. If additional hardware timers are available, then this approach can be augmented with a dedicated hardware timer devoted just to deadlock/livelock detection.

```
void thread_a()
{
   /* Outer Loop OL */
   while(1)
   {

      /* Inner Loop I1 */
      while(...)
      {

      /* put checkpoint_reached() here?  (1) */
      ...
      }

      ...

      /* Inner Loop I2 */
      while(...)
      {

      ...
      }

      /* put checkpoint_reached() here?  (2) */
      ...
      thread_sleep(sleep_time_a);
   }
}
```

Figure 4: Checkpoint placement is critical to correctly capturing lack of progress in a thread due to deadlock/livelock.

Another interesting parameter in the checkpointing scheme is the choice of timeout value. Currently NodeMD enforces a default timeout equal to twice the period, but the multiplier can be set differently at compile time. Our reasoning was that doubling the estimated period would err on the side of overestimation, and thus reduce the occurrence of false positives. We leave this as an open issue for further research.

A thread may also declare more than one checkpoint to track different periods of recurrence in different regions of its code. This would be useful in cases where an application thread may have multiple legitimate duty cycles whose progress needs to be checked. For example, a sensing thread may have a 10% energy-based duty cycle that wakes/sleeps every 100 seconds, i.e. 90 seconds asleep and 10 seconds awake, and may also have a second data requirement that samples be collected every second during the wake time. In this case, the thread is making progress only if it is awake every 100 seconds *and* collecting data every second while awake. In this case, separate checkpoints would be needed, one for the outer loop of energy-based duty cycling, and a second for the inner loop of data sampling.

The above checkpointing 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, and 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 is reset by the watchdog. 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.

4.3 Application-specific faults

Many data integrity rules for 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.

NodeMD supports an API that the application programmer can call when custom code detects that domain-specific constraints are violated. Our system introduces the ASSERT(condition) macro to allow the system to validate that certain application constraints are not untrue. This is similar to the approach introduced by Design by Contract [11], but would not kill a program. Instead, if an assertion fails, then NodeMD directly drops the system into a stable debug mode, suspending all application threads.

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, which uses assertions to verify 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).

This approach gives the application programmer considerable flexibility to invoke the debug mode and suspend a node through any number of (failed) assertion conditions. We believe that this capability to perform 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 a difficult challenge to any WSN debugging system. With a wired interface, JTAG debugger units provide a multitude of information to any connected node, but would triple the cost of each node [22], and don't apply to remote wireless deployments. Conventional string logging [16] of events is an approach more commonly used for wired devices. Storing events via strings is a fairly inefficient approach in WSNs, and also incurs a higher energy cost in transmitting more data. Logging via 'printf' statements is also a heavyweight operation on embedded devices like WSNs, often interfering with the timing of lightweight programs.

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

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.

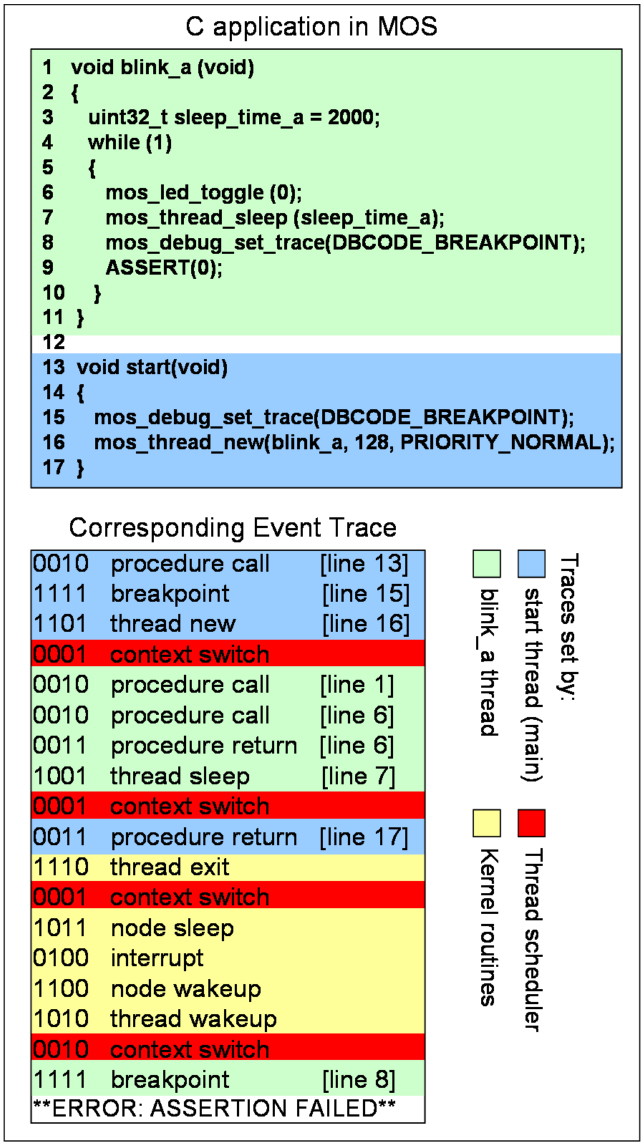
The solution NodeMD implements is to keep an execution trace of recent system events within a circular bitmap, similar to work found in ARTS [17] and the Wind River System Viewer [18]. Each defined event can be described as a unique order of bits, and is compressed to a length dependent on the number of combinations needed to express all recorded events. Events are encoded as bit patterns, and multiple events could even be stored in a single byte, depending on the need for compression. Events are then entered into a circular buffer in main memory (RAM). When memory allocated to the buffer is exhausted we begin overwriting the oldest events first. 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 major system events that we have found to paint a fairly extensive picture of execution history. These include procedure entry/exit, thread behavior (context switches, blocking, sleeping), timer behavior, and interrupts. In particular, NodeMD tracks the following system 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

Most system events are logged at various levels in the operating system. However, our parser discussed in Section 4.1 also adds debugging code to the application when necessary.

In addition to the system defined events, NodeMD gives the application programmer the flexibility to specify custom application events that should be logged to aid in the 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.



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 5: Example application and corresponding trace data.

An example of how NodeMD generates an event trace for an application is shown in Figure 5. First, we see the C code for two simple MOS applications, start and blink_a, which are shaded blue (medium gray on black-and-white printers) and green (light gray on b&w) respectively. These applications are each executed in a separate thread. The trace at the bottom of the figure has also been coded with the same colors/shades to correlate system event behavior with the corresponding code, and line numbers have been added to traces to help identify the correlation. We can clearly identify different running thread contexts, context switches, and kernel routines for thread scheduling and power management. For example, the breakpoint code 1111 followed by the thread new event 1101 clearly identifies that the initial context belongs to the start thread. A context switch then creates a pattern of events that corresponds with blink_a's code. The second context switch returns to the start thread, which quickly completes. The third context switch ushers in a sequence of events that correspond to energy-based system management. The final context switch returns control to blink_a, which then calls the ASSERT(0) statement. This statement will fail, resulting in controlled detection of an application-specific software fault that causes the system to halt, i.e. NodeMD will detect the application-specific fault and gracefully drop the system into a stable debug mode. Note that in this example that we have taken advantage of NodeMD's ability to insert custom application fault detectors via the ASSERT() statement to aid in debugging, i.e. injecting an artificial fault.

Note that this example also leverages NodeMD's ability to create custom application events to be logged. At line 8 the application code sets a custom application BREAKPOINT trace code 1111 in order to help identify key locations in the application code. "Breakpoints" 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 5 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 surrounded with a different sequence of events) we can conclude where the error occurred. Note that in our terminology a "breakpoint" is merely a special application-defined identifier in the event trace, and does not actually halt execution as in a debugger's breakpoint.

This example illustrates the memory trade off that NodeMD makes 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. NodeMD's current implementation pushes the envelope towards one extreme, namely very spare event detail, while favoring longer traces. Given the 15 system events, we chose to initially devote only 4 bits to each event, thus admitting only one extra application-specific breakpoint event. Such a compact event code also removes the ability to provide any qualifying information to a given event, i.e. to what thread is the context switched, what procedure is being called, etc.

Our reasoning was that sensor systems are relatively simple, with typically only a handful of threads, so qualifying information such as which thread context was under execution at a given time could be inferred by careful analysis of the event trace, and need not be included in event detail. Sensor systems don't have the luxury of including extraneous information in such extremely limited RAM, so our philosophy was to see how how far we could go in streamlining events and still provide useful debugging information. As illustrated by this example, each application exhibits a largely identifiable behavior or signature, as revealed by the sequence of logged events, that helps to uniquely identify which thread is currently executing and where in that thread's code that execution is taking place. In addition, the ability to inject application-specific breakpoints/events at strategic locations in the code further distinguishes application behavior and helps to triangulate to the execution point in the code. In this example, we were able to infer substantial contextual information about execution behavior even with extremely spare event detail. Though this is admittedly a simple example, we have also tested this approach on other applications and found the event trace to be useful. In fact, we describe in the evaluation section how this sparse approach towards event tracing was nonetheless quite helpful in pinpointing a bug in MOS despite having only 4 bits/event for debugging.

While our current implementation explores an important test case, NodeMD is not restricted to any specific number of bits/event. A developer that wishes to log more system and/or application-specific events can increase the number of bits/event, though the system code will have to be instrumented at the appropriate locations to log each additional event, and new codes will have to be defined. Similarly, a developer that wishes to include more detail for each event will need to devote more bits to provide qualifying information for each such event.

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 system faults addressed in this paper, we believe we have solutions to the previously identified faults that ensure that the notification is properly sent.

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 bidirectional communication. A faulty node uses the wireless mesh network to inform the remote administrator that the system is in a faulty state and uploads 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. At this point, any memory location (including complete memory dump useful for debugging on simulators) could be sent on user request. 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.

While NodeMD has a limited implementation of this debugging mode, a variety of open research issues remain. How can we ensure that the debug mode itself is not buggy? While it can't be guaranteed that there are no bugs in debug mode, the error recovery code used in debug mode is only a small portion of the rest of the system and so can be scrutinized more carefully to remove bugs. Moreover, this code will be reused over multiple deployments, with the likelihood that in the long run that bugs in the debug mode will be more readily discovered and ironed out, thus appearing more rarely than in the rest of the system. Other open 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 that still needs to be done regarding reliable network communication between the programmer and the debugging mode.

6. FAULT DIAGNOSIS - CLOSING THE LOOP

The final piece of NodeMD's architecture is closing the loop to enable interaction between the human user and faulty nodes in the system. Following delivery of an initial error report that includes the event trace, our system drops into an interactive debug mode that leaves many decisions to the user about how best to proceed with further debugging. The philosophy is that the human has the expertise and domain knowledge to determine how best to debug the problem, so it is incumbent on our system to support the user in this endeavor to the extent practicable for WSNs. This means that NodeMD supports two key capabilities in this mode: the ability for the human to interrogate the node by sending on-demand requests querying for more detailed information; and the ability to upload a remedy to the node in the form of new code updates.

6.1 Remote Debugging

Our controls allow the human to obtain all available fault information on a node. This can range from obtaining system parameters that were not included in the initial brief error report, to a full dump of memory. 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.

Queries are sent on-demand, and it is incumbent on the human user to balance how much usable information should be retrieved versus the energy costs of retrieval, i.e. how much strain on system resources would be incurred by transferring a full memory dump. For example, transferring a full 4 KB of RAM using packet sizes of 50 bytes over a lossy multi-hop wireless network can necessitate many transmissions and retransmissions, not to mention the considerable inefficiency of header overhead in such small packets.

Although we do not write in flash memory due to the performance impact during deployment, NodeMD could be modified to allow for writing the event queue in flash memory for nodes that have been deployed specifically for debugging, such as in a testbed. This would allow for much larger buffer sizes at the expense of execution times, which may be appropriate for debug-only testing. In this case, the larger event trace could be retrieved from flash when the node drops into this interactive debug mode.

6.2 Code Updates

In terms of remote code updates, our intent is to choose a reasonable combination of reliable code propagation and degrees of operating system modularity to enable dynamic reprogramming. The prior work in this area [8, 9, 10] offers many options for closing the loop in fault management systems for WSNs.

The Mantis research group is currently working on an implementation that modifies Mantis OS, our target platform, 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 [24]. 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 Effectiveness of Fault Detection

We begin by evaluating how effectively NodeMD detects stack overflow. We constructed numerous experiments that forced stack overflow, including recursive functions, highly nested function calls, and calls to functions with large local memory allocations, e.g. many local variables. For all of these cases of stack overflow, our experiments indicated that NodeMD was able to immediately detect the stack overflow using the function-based stack checking approach described previously. In addition, NodeMD provided an accurate event history leading up to the stack overflow event. Based on how it was designed, we expected that our stack overflow detector would not have any false positives with these cases. However, in our most recent testing we found a fourth case that exposed a weakness in our original algorithm. Passing enough parameters to a procedure, such that the compiler can no longer use only volatile registers to pass them, resulted in unexpected stack growth. As a result, we have incorporated a "red zone" to buffer the top of the stack from other areas of memory. However, determining the appropriate size of this red zone is difficult. If the red zone is too small, we risk a stack overflow due to parameter pushing. If the red zone is too large, false positive detections become a possibility. A goal of our ongoing research is to incorporate static analysis tools to determine the exact red zone value needed for the stack overflow check in each procedure. An exact value will ideally prevent any cases of stack overflow due to parameter pushing, and at the same time prevent most plausible false positives.

The stack overflow detector demonstrated its practical utility in helping to debug the implementation of NodeMD itself. 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 detector correctly identified the problem.

We evaluated how effectively NodeMD detected deadlock and livelock. 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 the checkpoint-based algorithm of NodeMD on a binary scale: either the deadlock/livelock occurrence was caught, or it was not.

The testing regime started a set of threads programmed to either run correctly, or encounter one of the problems above. Up to four threads would run simultaneously, and different combinations of livelock, total deadlock, partial deadlock, and interrupt-disabled livelock were induced in the threads. To emulate livelock, we inserted a while(1) loop in a thread. To emulate total deadlock, threads would develop circular dependencies on sets of semaphores. For partial deadlock, at least one thread was allowed to execute in a normal fashion, while others deadlocked.

The applications that we used to test the checkpoint approach were typical WSN applications that exhibited periodic behavior, e.g. sense-and-forward and a WSN base station. The sense-and-forward application typically loops through code that periodically gets sensor data, sends that data over the radio, and then sleeps. The base station application typically will make periodic calls to receive data from the radio. Similar to a select() call in UNIX, the receive call in MOS has a timeout parameter associated with it that permits the call to return after a maximum wait time if no data was received.

For all of these applications, the checkpoint technique required the application developer to estimate a thread period. We sought to emulate how a high-level WSN developer might estimate this period. Our approach was to inspect the code from a high level and determine an approximate upper bound on delay on each section of the code, sum the delays, and add some extra margin to arrive at a ballpark estimate of the thread period. We did not go in-depth to measure the exact execution time of each individual instruction in the loop, as we would not expect a typical WSN developer to know instruction-level timing details. For example, for sense-and-forward, the application code tells us that its timers are set up so that data must be sampled every T_{sample} seconds while awake for a total of N samples, that the send command takes relatively little time T_{send}, and that we sleep for T_{sleep} seconds per loop iteration. Thus, we set the thread period equal to $N * (T_{sample} + T_{send}) + T_{sleep} + \delta$, where δ is a rough estimate of all other instructions in the period. The timeout is then set to be twice the estimated thread period. Similarly, for the base station application, we know that in each loop iteration the thread waits a maximum of $T_{timeout}$ seconds to receive a packet before timing out, and sleeps T_{sleep}, so we set the thread period equal to $T_{timeout} + T_{sleep} + \epsilon$.

Given the experimental setup and estimation approach above, our checkpoint-based detector was able to accurately detect the presence of all combinations of complete deadlock, partial deadlock, livelock, and interrupt-disabled livelock. In all cases, the NodeMD detector was able to execute despite application threads being paralyzed, and correctly dropped into debug mode while providing its event trace in RAM. No false negatives were encountered in testing, although plausible scenarios have been suggested. However, we are confident that circumstances that lead to false negatives can likely be solved using a combination of deadlock/livelock detection, application-specific fault detection and Design by Contract practices. False positives were not encountered either, as our approach for selecting the thread period correctly overestimated the actual thread period.

In cases where more than one thread was livelocked and/or deadlocked, NodeMD only detected that one of the threads had violated their checkpoint timer, not each such thread. This was due to the nature of NodeMD, which halts the system on the first fault violation that it detects. Detecting concurrent faults is beyond the capacity of NodeMD, and is probably not needed, as NodeMD could, for example, detect and remove individual livelocks one by one.

Our current implementation was able to detect when an interrupt-disabled livelock case occurred only through the hardware watchdog reset. This is probably the least common deadlock state, as systems are spending the majority of their time with interrupts enabled. At the moment, our implementation is limited to entering the debug mode for this case so notification that something went wrong would be received, but the state of the memory would be lost.

Correctly diagnosing when an individual case of deadlock/livelock has actually occurred has also proven to be dependent on the event trace. Since NodeMD's checkpoint-based detection algorithm is a solution based on providing information to the human, it expects a programmer to correctly interpret the data in order to diagnose the problem. Experience from the hard-real time community using similar tools [18], [27] indicates that similar systems provide significant help in understanding system behavior, and determining what additional event categories would be useful.

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 [18] attests to their usefulness in practice. In addition, NodeMD provides more information than the programmer previously had available.

The effectiveness of application-defined ASSERT()'s was validated first by inspecting the code that checks the ASSERT() condition to see that it was error free. The ASSERT() statement was then tested in a variety of application scenarios, such as the one shown in Figure 5, and always halted the system appropriately.

7.2 Event Logging - A Case Study

During our implementation, the event tracing capability of NodeMD helped us to pinpoint the location of an actual legacy bug in MOS, previously only detected by unpredictable behavior and code analysis. Several MOS programmers had identified a bug where certain thread behavior would unknowingly 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 that a visible *1 second* delay would occur between the entry and return from an interrupt handler. We identified the occurrence of this phenomenon while testing this system. The before-and-after traces from the buggy code and then the corrected code are shown in Figure 6.

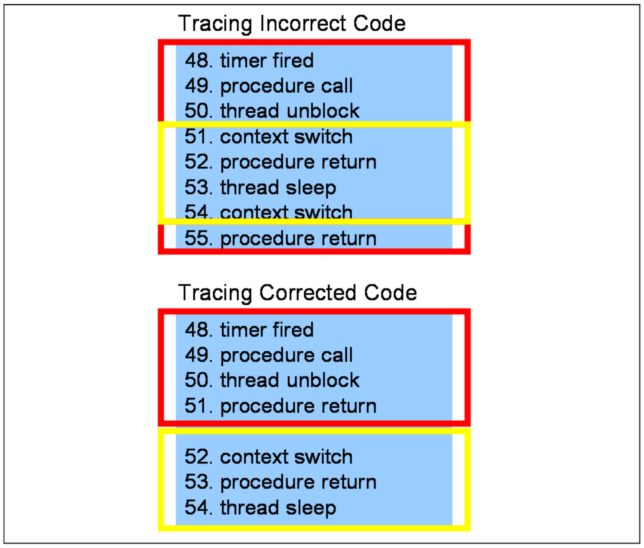


Figure 6: Before-and-after traces from a bug in MOS, where an application could unknowingly context switch out of an interrupt handler.

Notice the highlighted traces in the first trace section. Areas outlined in red (dark gray on b&w prints) are execution within the interrupt handler, while areas outlined in yellow (light gray in b&w prints) are outside of the handler. When a timer fires (48), its 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 yet returned, which results in several unpredicted conditions, one of which is the new running thread remaining in 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 outlined in red (dark gray on b&w) contains the entire interrupt handler routine without a context switch, we have verified that the bug has been fixed.

In this example, the event logging system of NodeMD provided a sufficiently accurate picture of the fault despite only encoding events at 4 bits/event. We could discriminate between system code being executed within and outside of the interrupt handler, which was all the precision we needed to locate the bug. The detail provided by the system's event trace was also sufficient to confirm that the corrected code had fixed the bug.

7.3 Event Logging Overhead

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 all 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. A simple call to blink the LED only logs 2 traces, whereas a more complex procedure such as com_recv_timed, which calls the radio, can conditionally generate up to 32 traces. Complex applications will call a mixture of these functions, resulting in varying impact of logging on the applications.

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.

Let's examine the impact of logging on a relatively advanced real world sensor networking application that we used in the FireWxNet deployment [3]. This is a very complex application encompassing nearly all of the features in MOS. Within the application, two threads are spawned. In the first, data is read from 4 different sensors into a packet buffer and sent over the radio every 1 second. The other 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, update the neighbor table, and handle state transitions for power management. After 1 minute of awake time, the node changes to a low power sleep state for the following 14 minutes.

In our experiments, we measured that processing a single iteration of FireWxNet's sending and receiving threads, plus all concurrent timers, required approximately 250 traces. Even one iteration of just the sending thread, without timers, required 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 would require around 6000 traces, or 3000 bytes using our approach. Given our memory limitations of 4 KB, which must also allow for execution space in RAM for the application, we would have to accept less than a full accounting of the entire awake period.

This example illustrates the extreme challenges that we face in designing a system to efficiently log events on resource-constrained nodes. Even at only 4 bits/event, the tracing mechanism reaches the capacity of our event trace buffer during one awake period and begins overwriting older events in the circular buffer. In our experiments, we have found that the event trace of only the most recent events nonetheless retained enough useful information to assist in locating the bug. However, more detailed studies need to be performed to determine how to most efficiently log the most useful events. We leave this as an open topic for future research.

7.4 General Overhead

How do our algorithms actually impact system performance? One of NodeMD's primary objectives is to remain lightweight and as 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 2 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 parser-added debugging code. The added amount is dependent on the complexity of the application. Given the features added by NodeMD, and the flexibility to tailor several trace buffer details to minimize overhead, we argue that 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 a 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 is enough to change timing of the program, this is a fairly small number. By comparison, calling the mos_led_on() function costs 35 cycles. Writing a trace is equivalent to turning on between one and two LEDs, which is essentially invisible to the application.

The logging operation 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 [23]. The FireWxNet application spends the majority of its awake cycle idling; waiting for software timers, data available on the radio, and data available on the ADC. Therefore the small number of cycles introduced by NodeMD should be easily absorbed without affecting application timing.

NodeMD's impact on battery life can be divided into two modes. During normal program and system execution, NodeMD does not use the radio, flash memory, or even LEDs to log events and/or execute its detectors, so all of its impact on battery life is related to the memory and processor's battery use. Since NodeMD introduces only a modest number of additional instructions for logging and/or detection, we do not expect any substantial impact on battery life due to NodeMD's execution during normal operation. When NodeMD drops into debug mode, the impact on battery life is likely to be more substantial. Spending time keeping the node and radio awake to support interactive debugging by the remote programmer will significantly drain energy resources and limit overall node lifetime. The amount of time spent diagnosing a problem would depend on the pattern of diagnosis by the programmer. However, the urgency to debug the detected software fault may override immediate energy concerns on the failed node.

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. Our plan is to instrument a WSN field application in the near future 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.

Having recently modified the approach NodeMD takes for stack overflow detection to include red zoning, more extensive testing is needed to determine what red zone sizes are most applicable to WSN applications. Including a static analyzer/preprocessor to identify and add at compile time the red zone requirements for application and OS procedures is a direction for future development on the NodeMD project.

NodeMD's approach for stack overflow detection is based on source based instrumentation. Although it is a reasonable assumption today that we have access to all source code in the system, that assumption may no longer hold in the future as commercial entities increase their presence in the area. Due to that, an interesting direction for future work is binary instrumentation.

We are interested in extending NodeMD so that it is equipped to detect illegal memory writes, which are another class of software faults that can also paralyze a node. Recent work has offered the approach of software-based memory protection [1], which we plan to investigate.

A potentially useful function for NodeMD 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.

Securing NodeMD involves several issues that were beyond the scope of this initial study. First, the wireless communication would have to be secured, i.e. authenticated and/or encrypted reporting of the event trace and sending of queries. It is still an open research question how best to secure routing in wireless sensor networks. Second, any remote code updates would need to be signed by the base station and efficiently authenticated by the node to avoid viral propagation. Early WSN security research focused on symmetric key approaches, but more recent work has explored the viability of public key techniques.

9. CONCLUSIONS

This paper has described NodeMD, a comprehensive system that implements detection, notification, and diagnosis of software failures in remote wireless sensor nodes. NodeMD is motivated by the need to minimize the high cost of 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 several specific detection algorithms: stack overflow detection; and application-defined thread *checkpoints* that act as custom software 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 in unit testing and in the real world application FireWxNet. We described how NodeMD has proven useful in practice in finding and diagnosing two real world bugs in the code of Mantis OS.

10. REFERENCES

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

[2] 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.

[3] 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.

[4] 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.

[5] Crossbow Technologies: "Mica2 Series (MPR4x0)", available at http://www.xbow.com.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

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

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

[12] 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.

[13] 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.

[14] 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).

[15] R. Wilson, "Shedding light on the Mars rover malfunction", EE Times, 02/20/04.

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

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

[18] 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

[19] 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

[20] 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.

[21] 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

[22] JTAG distributor. www.digikey.com

[23] 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.

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

[25] 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.

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

[27] 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.

[28] Glenn Reeves, "What really happened on Mars ?", available at http://research.microsoft.com/ mbj/Mars_Pathfinder/Authoritative_Account.html.