

Clairvoyant: A Comprehensive Source-Level Debugger for Wireless Sensor Networks

Jing Yang, Mary Lou Soffa, Leo Selavo and Kamin Whitehouse
Department of Computer Science
University of Virginia

{jy8y, soffa, selavo, whitehouse}@cs.virginia.edu

Abstract

Wireless sensor network (WSN) applications are notoriously difficult to develop and debug. This paper describes *Clairvoyant* which is a comprehensive *source-level debugger* for wireless, embedded networks. With *Clairvoyant*, a developer can wirelessly connect to a sensor network and execute standard debugging commands including *break*, *step*, *watch*, and *backtrace*, as well as new commands that are specially designed for debugging WSNs. *Clairvoyant* attempts to minimize its effect on the program being debugged in terms of network load, memory footprint, execution speed, clock consistency, and flash lifetime.

Categories and Subject Descriptors

D.2.5 [Software Engineering]: Testing and Debugging—*debugging aids, distributed debugging*

General Terms

Design, Experimentation, Performance

Keywords

Source-Level Debugging, Distributed Debugging, Embedded Debugging, Wireless Sensor Networks

1 Introduction

Source-level debuggers allow a developer to execute a program one statement or code block at a time and to watch the program as it is being executed. They are called source-level debuggers because they operate on symbols and statements defined in the program's source code. For example, the developer can set a breakpoint on a line of code, observe or modify a live variable, and invoke an arbitrary function using the symbol names defined in the source code. Also known as *symbolic debuggers*, or often simply *debuggers*, these tools are an essential part of most modern debugging environments. They make software development much easier, to the point that a good debugger can sway a program-

mer's decision to use a particular language [26] or hardware platform [29].

Remote debugging is the process in which a *host* machine is used to debug a process on a *target* machine. Remote debuggers have existed for many years and are commonly used to debug single embedded devices but, as we discuss in Section 2, cannot be easily applied to wireless sensor networks (WSNs). This is because hardware debuggers are expensive and sometimes impossible to deploy in large numbers or over large geographic areas, while software approaches do not accommodate the multi-hop, resource-constrained, and timing-dependent nature of WSNs. The most common approach to debug a WSN today is to use a simulator for large-scale networks and a hardware debugger for low-level device drivers. This is problematic because many bugs only manifest themselves when deployed on real hardware, in a real and noisy environment, and at large scale. The lack of a source-level debugger for real WSN deployments increases development time by making it difficult to diagnosis and fix these bugs. It also makes it difficult to verify correct operation, increasing the chance that buggy code will survive the testing process and make its way into production code.

In this paper, we present a system called *Clairvoyant* that provides comprehensive source-level debugging solution for WSNs. It can operate on the deployment hardware and in the deployment environment, without requiring any additional hardware or wires. Furthermore, no modifications need to be made to the application's source code. *Clairvoyant* provides a suite of standard debugging commands like *break*, *step*, *watch*, and *backtrace*, among others. It also provides new functionality that is specially designed for WSNs, including access to typical hardware such as external flash and interrupt vectors, global commands such as *gstop* and *gcontinue* that operate not on a single node but on all nodes in the network, and the ability to log variables to LEDs, RAM, external flash, or radio. These commands are described in more detail in Section 3. *Clairvoyant* can be used to debug race conditions, stack overflow, errors in global state, and errors in low-level hardware drivers, all of which are common yet difficult-to-debug errors in WSNs. It has been implemented on the Mica2 sensor node and is demonstrated in Section 6 through two case studies on real, documented bugs from the TinyOS repository: stack overflow and deadlock.

The process of debugging will always affect how a program executes, and *Clairvoyant* attempts to minimize this

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SenSys'07, November 6–9, 2007, Sydney, Australia.
Copyright 2007 ACM 1-59593-763-6/07/0011 ...\$5.00

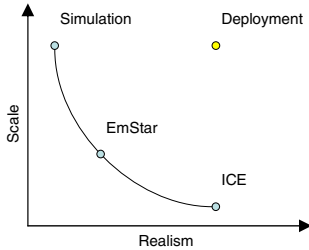


Figure 1. Existing source-level debugging tools for WSNs cannot deal with scale and reality simultaneously.

effect in order to reduce the number of so-called *Heisenbugs* – those bugs that change or disappear once a debugger is used. To this end, Clairvoyant implements debugging commands by modifying the target binary, an approach called *dynamic binary instrumentation*. This is the only approach that runs the target program at native speed directly on the MCU, or “on the metal”, without using extra hardware or making changes to the program’s source code. However, sensor nodes have limited space for debugging software and have flash-based program memory, which limits the number of modifications that can be made to the target binary. In Section 4, we describe the techniques we developed to make dynamic binary instrumentation work on resource-constrained sensor nodes. In Section 5, we explore and empirically evaluate the costs, trade-offs and limitations of these techniques in terms of network load, memory footprint, execution speed, clock consistency, and flash lifetime.

2 Background and Related Work

Debugging is a multi-step process that requires several tools to collaborate with each other, including source-level debuggers. For example, tools like Tarantula help with *fault localization* by identifying the lines of code most highly correlated with failed test cases [9]. Integrated development environments (IDEs) like Eclipse [18] and Visual Studio [13] help with *fix determination* by suggesting possible changes to syntactical errors in the code. Dynamic reprogramming systems like Deluge help with a process called *fix application* [8]. The main goal of source-level debuggers is called *cause identification*, in which the developer tests a hypothesis about the cause of a bug.

Source-level debuggers such as GDB [21] and DBX [20] are typically implemented in one of four ways. Most debuggers use the processor’s built-in hardware support: special registers store the locations of breakpoints or variable watches and, when the program counter or a memory access matches these registers, the processor halts execution and transfers control to the debugger. This approach cannot be used for WSNs because the 8-bit microcontroller units (MCUs) most commonly used on sensor nodes do not provide hardware support for software debuggers. A second approach is to control the run-time system through APIs like the Java Debugger API [19] and the ICorDebug API [14] for the .NET Common Language Runtime. However, most WSN applications do not run in this type of run-time system. Smaller virtual machines, software dynamic translators, or interpreters [7] could be used to implement a debugger

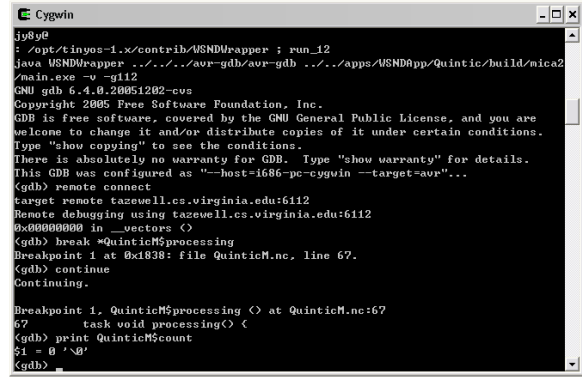


Figure 2. The Clairvoyant user interface consists of multiple GDB-like terminals.

for WSNs, but we did not employ these techniques because they can significantly change the temporal characteristics of a program and therefore increase the number of heisenbugs. A third approach is *hardware emulation* through an in-circuit emulator (ICE) such as the AVR JTAG ICE [2], which is an external piece of hardware that interfaces with GDB and controls firmware execution by physically attaching to the I/O pins on the MCU. ICEs are most commonly used for debugging embedded devices, but they require a physical wire and additional hardware for every node and thus do not scale with the number of nodes or the geographic size of a network. Furthermore, they cannot be used spontaneously because the hardware must be deployed in advance. A fourth approach is to implement debugging commands in software, through modification of the target executable. For example, a breakpoint can be implemented by inserting a trap into the binary executable; whenever the trap is reached, control will be transferred to the debugger. This approach may be used when the developer creates more breakpoints or watches a larger section of memory than the hardware supports, and is most similar to the approach used by Clairvoyant. To the best of our knowledge, Clairvoyant is the first implementation of this approach on a platform with flash-based instead of RAM-based program memory.

The most common way to employ the source-level debugging technology described above in a WSN application is through *simulation*, which allows the application to be executed and debugged on a machine with full OS and hardware debugging support [10, 16, 24]. Simulation makes it easy to watch program execution, and can be used to debug a large network with 1000s of nodes. However, all sensor readings and radio communication are based on models that can never capture the full complexity of the real world. This problem is ameliorated by a system called EmStar [6], which runs the application in a simulation-like environment, but uses real-world I/O by connecting the simulation to the sensors and radios of a real-deployed sensor network. EmStar requires a physical wire to be run to each node, which limits the maximum size and geographic area of any network to be debugged. Furthermore, EmStar affects the speed with which the application executes by running the application on a high-powered, multi-tasking processor and by introducing

Command Format	Description
target remote host:6.xxx detach	connect this terminal to the sensor node with ID xxx release the target program from debugging control and resume its execution
continue step stepi next finish	resume execution execute until another source line is reached execute until another machine instruction is reached execute the next source line, including function calls execute until the selected stack frame returns
break [file:]line func	set a breakpoint at line number line function func [in source file]
cond n [expr]	make breakpoint n unconditional [or conditional according to expression expr]
watch expr delete [n]	set a watchpoint for expression expr delete all breakpoints/watchpoints [or breakpoint/watchpoint number n]
print [f] [expr]	show last printed value [or value of expression expr] [according to format f]
set var=expr call func	evaluate expression expr and use it for altering program variable var call function func
backtrace [n] frame [n]	print trace of all frames [or of n frames] on the stack select the current frame [or frame number n or frame at address n]
info frame	describe the selected frame, including locations of arguments, local variables, and registers
list	show next ten lines of the source

Table 1. Basic commands provided by Clairvoyant.

I/O latency. This can make it difficult to debug low-level hardware drivers or identify timing-sensitive bugs such as race conditions.

Figure 1 illustrates the trade-off between simulators, EmStar, and ICES. Simulators can be used to debug sensor networks at large scale, but cannot capture real-world dynamics. EmStar can be used to debug mid-scale networks with realistic sensor values and communication channels, but lacks realistic timing characteristics. ICES run the program on the actual deployment platform and therefore provide realistic I/O and timing characteristics, but because they require additional hardware to connect to each node, only a small number of nodes can be debugged at a time. Thus, WSN applications currently cannot be debugged in a realistic environment when the network either has a large number of nodes or covers a large geographic area. This is problematic because many bugs only manifest themselves when deployed on real hardware, in a real and noisy environment, or at large scale.

One other approach to source-level debugging in WSNs is called *record and replay*, in which all I/O is logged during execution and later replayed for debugging purposes [15, 28]. This approach is common in distributed systems and has been implemented for WSNs by the EnviroLog system [12], which stores and replays the I/O on the sensor nodes. If the I/O were instead replayed through an emulator, this technique could provide a limited form of source-level debugging. However, because it provides *post-mortem* debugging, record and replay debugging can only provide visibility into and not control over execution; once the execution path is modified, the logged data will no longer apply.

Because of the lack of adequate source-level debugging tools for WSNs, the community has predominantly used what could be called *compile-time* debugging, in which the programmer modifies the application’s source code to report state back to the developer as it executes. For example, the DiagMsg utility for TinyOS allows the devel-

Command Format	Description
led [-c] [file:]line func var col [cond]	display variable var to LEDs of color col at line number line function func [in source file] [when condition cond is satisfied]
ram [-c] [file:]line func var loc [cond]	log variable var to RAM of address loc at line number line function func [in source file] [when condition cond is satisfied]
flash [-c] [file:]line func var loc [cond]	log variable var to external flash of address loc at line number line function func [in source file] [when condition cond is satisfied]
radio [-c] [file:]line func var [cond]	send variable var to the base station at line number line function func [in source file] [when condition cond is satisfied]

Table 2. Logging commands provided by Clairvoyant.

oper to dump variable values to the display when particular points in the code are reached [23]. The Sympathy debugger uses periodic reports from the application to automatically classify routing-related bugs [17]. SNMS allows the developer to provide a messaging interface to read and write certain variables on a sensor node [25]. Marionette builds upon SNMS by allowing the developer to also invoke functions [27]. However, none of these tools provide source-level debugging: the developer does not interact with the program through a source-code based interface. Instead, the debugging interface must be pre-programmed into the executable and the only way to change this interface is to modify, recompile, and reprogram the executable.

3 Clairvoyant Overview

Clairvoyant is a comprehensive source-level debugger that allows a developer to connect to and debug an embedded device in a multi-hop wireless network. Clairvoyant does not require any special effort from the developer before debugging a program: the source code does not need to be modified and no libraries need to be linked into the executable. Clairvoyant simply needs to be programmed into the boot sector of each sensor node once, possibly at fabrication time. Once Clairvoyant has booted, it can load other applications from external flash into program memory and run them, just like the Deluge bootloader [8]. Clairvoyant also does not require any additional hardware and does not affect program execution unless the developer issues a debugging command. Therefore, Clairvoyant can be left on the sensor node all the time, even during deployment, to be used *spontaneously* and *only when needed*.

An important limitation of Clairvoyant is that it requires about 32 KB of program memory and 1 KB of data memory, and therefore large programs that require all the memory of a sensor node cannot be debugged. For comparison, the Mica2 sensor node has 128 KB of program memory and 4 KB of data memory. If large programs must be run but not debugged, Clairvoyant can be stored in the sensor node’s external flash and loaded into program memory by another, perhaps smaller bootloader. Another important limitation of Clairvoyant is that it must write to the sensor node’s program memory during the debugging process. Because most MCUs use flash-based program memory with a limited number of write/erase cycles, debugging with Clairvoyant will shorten the sensor node’s lifetime. The effect of each command on the sensor node’s lifetime will be discussed in Section 4.

Clairvoyant provides both standard debugging commands

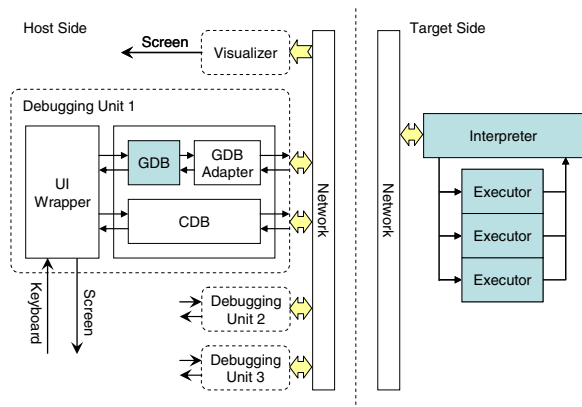


Figure 3. The host/target architecture of Clairvoyant.

and new commands that are designed to meet the specific needs of WSNs. The new commands are described in more detail later in this section, and provide access to typical hardware, the ability to command all nodes in the network at once, and the ability to efficiently log variables, among other things. All of these commands are source-level commands because they operate on symbols defined in the source code of the program.

All Clairvoyant commands are entered through a GDB-like terminal, similar to that shown in Figure 2. Each terminal connects to a single node in the network, so the developer creates a different terminal for each sensor node that needs to be debugged. In many cases, it should be sufficient to debug only a small subset of key nodes in the network, such as cluster heads, localization anchors, or routing gateways. Just like GDB, Clairvoyant’s text-based interface could be converted into a graphical user interface through an IDE such as Eclipse.

3.1 Basic Commands

Clairvoyant provides the most commonly used commands that are provided by GDB. These commands are listed in Table 1, and allow the developer to (i) remotely connect or disconnect to the target program, (ii) step through its execution, (iii) insert or delete breakpoints and watchpoints, (iv) inspect or modify data values and call functions, (v) change or examine stack frames, and (vi) list lines of source code. Two basic commands that Clairvoyant provides are not available in GDB. The *stop* command triggers the sensor node to enter debugging mode, even while the target program is executing. This produces an effect similar to hitting *Ctrl-C* in the GDB terminal on a PC. The *reset* command is similar to *detach* except that it restarts the target program from the very beginning instead of resuming it from the trapped location.

3.2 Hardware Access Commands

Clairvoyant provides commands specially designed to access the hardware on a typical sensor node. The *interrupts* command can be used to describe each enabled interrupt, including both its functionality and the corresponding handler routine. This command can be helpful in identifying race conditions since it lists the interrupts that are enabled at a particular execution point. It could also be useful in recog-

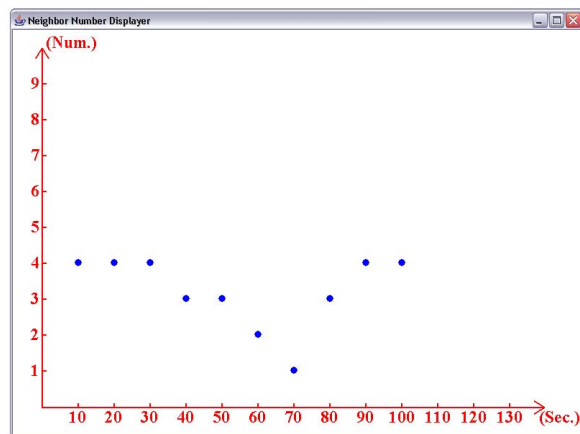


Figure 4. An example visualizer that displays the neighbor number of a sensor node every ten seconds.

nizing logical errors, such as why a particular interrupt was not triggered as expected. Finally, the name of the handler routine allows the developer to simulate an interrupt by using the *call* command to invoke that routine, which can help when debugging event-driven code.

Clairvoyant also provides the ability to read arbitrary values from RAM or external flash through the *ramread* and *flashread* commands. The latter command is important for many WSN applications that use external flash instead of RAM to hold important parameters or data logs. Both commands can be useful when reading variables logged by the logging commands described in Section 3.4.

3.3 Network-Wide Commands

Clairvoyant provides a number of global commands that operate on all nodes in the network, instead of only on a single node. These commands include *gstop*, *gcontinue*, *gdetach*, *greset*, and *gbreak*. All commands perform the same functionality as their corresponding single-node versions, except that they address all nodes in the network instead of only a single node. The only exception is the *gbreak* command, which is just like *break* except that a *gstop* command is issued when the breakpoint is reached. Thus, this command does not set a breakpoint on all nodes in the network; it sets a breakpoint at one node but all nodes stop execution when the breakpoint is reached.

Clairvoyant’s network-wide commands are designed to exploit the fact that sensor network communication happens through a wireless broadcast medium, unlike the wired environments where parallel and distributed debuggers have previously been used. Thus it is more efficient to send a single global command to all nodes via a network flood than to send the same command to each node individually. Besides increased efficiency, this approach is also important for allowing sensor nodes to start and resume execution in a fairly synchronized manner.

3.4 Logging Commands

Clairvoyant provides a set of commands for logging a variable every time a particular point in the code, called a *logpoint*, is reached. The commands to insert logpoints are summarized in Table 2, and allow variable values to be sent

Action Format	Description
c [addr]	resume execution
D	detach GDB from the target program
m addr leng	read bytes of memory
M addr leng vals	write bytes of memory
p n	read an register
P n val	write an register
s [addr]	execute a single machine instruction
z type addr leng	remove a breakpoint/watchpoint
Z type addr leng	insert a breakpoint/watchpoint

Table 3. Primitive actions defined by the GDB remote protocol.

to the LEDs, RAM, external flash, or the radio. LEDs can display continuous, real-time value changes, but must be physically observed by the developer. RAM access is fast but Clairvoyant only reserves 10 bytes for logged values due to data memory limitations. External flash is more spacious but also requires longer access time. Finally, logging to the radio can transmit an arbitrary amount of data to the developer but introduces large and non-deterministic delays during program execution.

Any logging operation can be predicated on a condition that can be expressed using one of four basic formats: (i) a comparison of two variables, (ii) a comparison of a variable and a constant, (iii) a comparison of the current stack pointer with a constant, and (iv) a combination of two above formats using a logical *and* or *or*. Format (iii) is especially useful for identifying stack overflow, which is a relatively common occurrence in WSN applications due to the tight RAM constraints and event-driven operation. It is also a difficult bug to diagnose because the sensor node typically reboots after a stack overflow, leaving no way to identify the cause of the error. All the logging commands provide the *-c* option to allow the developer to choose whether or not to save clock registers before logging the value. This functionality is discussed further in Section 4.2.3.

4 Clairvoyant Implementation

Clairvoyant consists of two parts: the host side runs on a PC and interacts with the developer, and the target side runs on a sensor node and interacts with the firmware. The overall architecture is illustrated in Figure 3.

The host side of Clairvoyant is composed of one or more *debugging units* and a *visualizer*. Each debugging unit runs GDB as a child process in order to reuse its sophisticated debugging functionality as much as possible. It also contains a separate CDB (Clairvoyant DeBugger) process to deal with all specialized commands that are not recognized by GDB. Finally, a *UI wrapper* provides a text-based interface that dispatches commands from the developer to either the GDB or CDB process and prints any output from them to a display. The visualizer allows the developer to extend beyond this textual GDB-like interface and create graphical representations of the debugging information being retrieved from the sensor network. Figure 4 shows an example visualizer that displays the neighbor number of a sensor node every ten seconds. The data used to plot the figure is from a *radio* command which logs the corresponding variable.

The target side of Clairvoyant mainly consists of an *interpreter* and multiple *executors*, each of which implements

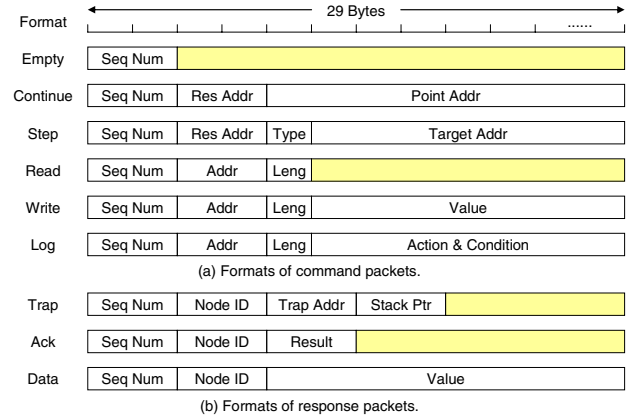


Figure 5. Formats of command packets and response packets used by the proprietary packet protocol.

a different command. It is compiled into an embedded program that is typically loaded into the boot sector of the Mica2 sensor node. Once Clairvoyant has booted, it automatically loads the target program to be debugged into program memory, as illustrated in Figure 7 (a). After that, it notifies the developer who can then use commands like *break* to set breakpoints in advance, or *continue* to start execution. All commands are implemented completely in software. For example, when the *break* command is issued, Clairvoyant modifies the target binary to transfer control back to it when the breakpoint is reached. If the *continue* command is issued, Clairvoyant transfers control to the appropriate point in the binary to resume the application. Thus, when the target program is running, it does not run in any sort of virtual debugging environment; it runs at native speed directly on the MCU, or “on the metal”.

Our current implementation is targeted towards debugging firmware written in the nesC language [5], using the TinyOS libraries [4], and running on the Mica2 sensor node platform [22]. However, Clairvoyant is not limited to this hardware/software platform. Clairvoyant uses a hardware abstraction layer to separate debugging functionality from low-level device drivers so that they can easily be replaced to support new hardware platforms. Clairvoyant also uses GDB to the extent possible, so it already has supported a wide range of languages and MCUs as GDB does. The only change required would be to update Clairvoyant’s modifications to the target program’s radio stack, which are described in the next section.

4.1 Host-Target Communication

Although Clairvoyant uses GDB as a child process on the host, it does not use the GDB remote protocol to communicate between the host and the target. The GDB remote protocol was designed to allow a GDB process to debug a target running on a different machine, such as an OS kernel or an embedded program. When GDB receives commands from the developer, it converts them into the set of *primitive actions* listed in Table 3 and sends them to the target. To interpret these messages, the target program must either (i) link in a *GDB stub* library that receives and interprets the

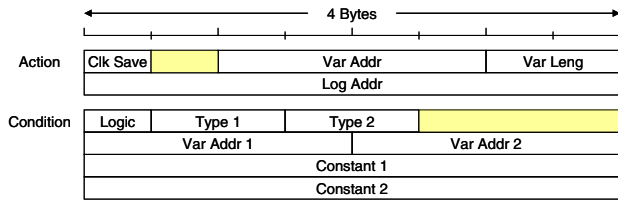


Figure 6. Detailed formats of the *action* & *condition* field in the *Log* command packet.

messages, or (ii) run a separate *gdbserver* process, which is a lightweight debugger that is smaller than GDB but still requires the same degree of OS support.

Clairvoyant does not use the GDB remote protocol for several reasons. It was designed for wired serial or TCP networks, where network bandwidth is not a limiting design factor. Therefore, it issues primitive actions one by one, causing excess network traffic. Furthermore, on a sensor node many of these actions require writes to flash which could be eliminated if several actions could be combined into a single operation. Finally, a GDB stub needs to be linked into each executable while a *gdbserver* requires complete OS support, neither of which are desirable for deployment-time debugging of WSNs.

To address these issues, Clairvoyant uses a *GDB adapter* on the host to extract information from GDB. When receiving a sequence of primitive actions, the GDB adapter converts them into *atomic commands* that can be packed into a single message and executed at once by Clairvoyant’s target side. There are six types of command packets and three types of response packets, listed in Figure 5 and Figure 6.

Most of these are self explanatory, and here we explain only a few packet formats. When a breakpoint is reached, the target side of Clairvoyant sends back a *Trap* response packet, which includes the trap address and the current stack pointer. The *Continue* command packet contains the address at which to resume the program and the locations of all active insertion points that need to be reinstated. When the number of insertion points is too large to fit, further packets of the same format are needed. The *Step* command packet is very similar. The *Log* command packet specifies the location to insert the logpoint, the target variable together with the its logging address, as well as the condition if applicable. The *Empty* packet is used to invoke atomic commands with no parameters; the command is identified from the message type alone. We do not discuss in detail how each GDB command is converted into the sequence of primitive actions and how those are in turn converted into atomic commands. However, in Sections 5.1 and 5.2, we show how this implementation outperforms the GDB remote protocol in terms of radio messages and flash writes.

Clairvoyant commands and responses follow three general traffic patterns: (i) most commands are sent from the base station to an arbitrary sensor node, (ii) network-wide commands are sent from the base station to all nodes in the network, and (iii) responses are sent from an arbitrary sensor node to the base station. These three traffic patterns are very typical in sensor networks, and are currently

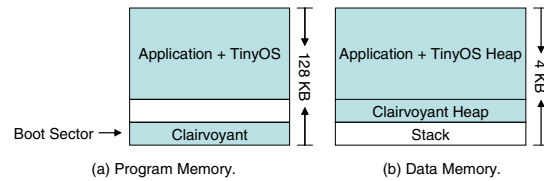


Figure 7. (a) Clairvoyant is programmed into the boot sector of the Mica2 sensor node, and shares program memory with the target program. (b) It also shares data memory, using the same stack and an adjacent heap.

supported by a wide range of routing protocols. The efficient dissemination of commands and the collection of responses is not the main contribution of Clairvoyant, and the choice of routing protocols depends on several factors including the network topology, application requirements, and personal preferences. Therefore, Clairvoyant currently sends all commands and responses using a reliable flooding protocol called Trickle [11], and defines the choice of particular routing protocols to be outside the scope of this paper.

4.2 Target Execution

One key design decision was to implement the debugging operations using an approach called *dynamic binary instrumentation* [30]. For example, when the developer specifies a breakpoint, Clairvoyant modifies the target binary so that control is transferred to it when the breakpoint is reached. The target state is then frozen while Clairvoyant processes any additional commands such as memory reads or writes, function calls, and stack traces, among others. When the developer issues the *step* or *continue* command, Clairvoyant modifies the binary accordingly and transfers control back to the target program. In this way, Clairvoyant and the target program alternately take full control of the sensor node. This approach allows Clairvoyant to debug an unmodified binary executable while executing it at native speed directly on the MCU, or “on the metal”, minimizing any effects on execution timing. In this section, we describe the techniques used to switch between application and debugging context.

4.2.1 Switch to Debugging Context

When a breakpoint is reached, the sensor node must be changed from application context to debugging context. A common technique for doing this is to replace an instruction block in the target program with a *jmp* instruction targeted at what is called the *trampoline* [30]. When that code point is reached, the application jumps to the trampoline, which (i) saves the application context, (ii) executes any instrumentation code, (iii) restores the application context, (iv) executes the instruction block, and (v) jumps back to the application. One limitation of this approach is that each instrumentation instance requires its own instance of a trampoline that contains both the instrumentation code and the original instruction block. This approach would quickly exhaust valuable program memory on the sensor node.

To address this issue, we created a *common trampoline*, which is a single piece of code used for all breakpoint instances. Instead of replacing the instruction block with a *jmp* instruction, the target side of Clairvoyant replaces it with a *call* instruction, which places the return address onto the

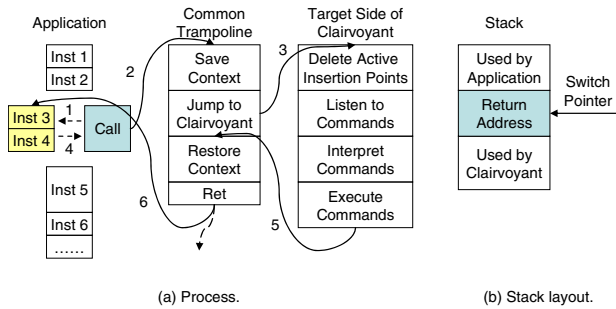


Figure 8. (a) A context switch using the common trampoline requires six steps. All insertion points can share the same trampoline. (b) The `call` instruction stores the address of the reached insertion point in the switch pointer on the stack.

stack, in a location called the *switch pointer*. When that `call` instruction is reached, the trampoline saves the application context and jumps to Clairvoyant, which removes all active breakpoints, watchpoints, and logpoints (uniformly, *insertion points*) to return the executable to its original, unmodified form. Clairvoyant then reports a trap to the host and waits for further commands. When the program is to resume, Clairvoyant jumps back to the trampoline which restores the application context and returns to the location from which the trampoline was first called. This process is depicted in Figure 8.

The common trampoline is efficient for sensor nodes because the context-switch code is only stored once in program memory. This is made possible in part by the use of the `call` instruction, whose return address indicates the location of the insertion point from which the trampoline was called. The common trampoline also relies on the fact that the target side of Clairvoyant restores the instruction block to its original condition before returning. This approach obviates the need for the trampoline to execute the original instruction block; it will be executed once control is returned to the target program. In Section 4.2.2 we discuss how control is transferred to the correct location, and how breakpoints are reinstated if necessary.

The common trampoline saves two types of application context. First, the 256 general-purpose and I/O registers are stored in a reserved location in data memory called the *register pool* because they might be overwritten when the target side of Clairvoyant executes. Later, whenever a program register has to be read or written, it is performed by accessing the register pool. Second, the common trampoline stores the target program's interrupt vector values and writes the locations of Clairvoyant's handler routines instead. This ensures that the application can never preempt Clairvoyant.

Clairvoyant leaves the target program resident in memory to avoid saving its stack and heap to external flash. Clairvoyant and the target program have separate heaps adjacent to each other, but they share the same stack, as illustrated in Figure 7 (b). This reduces the amount of memory available to both Clairvoyant and the target program but dramatically simplifies context switches, both increasing the execution speed and reducing the number of flash writes required.

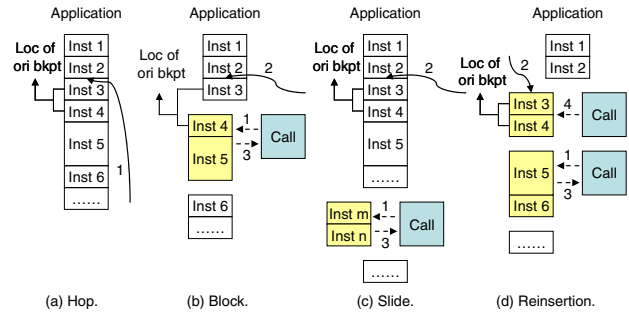


Figure 9. (a) The hop operation returns control to the target program. (b) The block operation returns control for only one instruction. (c) The slide operation returns control for multiple instructions. (d) Reinsertion of the reached point requires a temporary system breakpoint.

It also makes the execution of many debugging commands more efficient because the up-to-date values of program variables can always be retrieved directly from their allocated memory addresses instead of a stored location in external flash.

The context switch saves the complete state of the target program but does not stop or save the state of external hardware such as radio, flash, or external clocks. If a context switch occurs in the middle of an I/O operation, such as the receipt of a radio packet, unpredictable behavior may occur when the context is restored. This is not unusual, and other debugging techniques including AVR JTAG ICE also do not have the ability to stop external hardware. In some cases, however, where the external hardware only interacts with the program through I/O registers, such as an external clock, saving the I/O registers does stop the external hardware from the application's point of view.

Figure 8 illustrates that the `call` instruction can occupy the space of two instructions. Thus, it may appear that a branch could jump to the second instruction, which is now the second half of the `call` instruction to the trampoline. However, in Clairvoyant, breakpoints can only be placed at the granularity of a statement, not at arbitrary instructions. Therefore, this problem can only occur if an entire statement can be compiled into a single instruction such that a breakpoint on that statement would overwrite both itself and the first instruction of the next statement. To the best of our knowledge, this could never happen because all statements in the nesC language are translated into at least two instructions.

4.2.2 Switch to Application Context

If the common trampoline were to simply invoke the `return` instruction to resume the application, it would not return to the head of the instruction block replaced by the reached breakpoint, but to the instruction immediately following that block (Inst 5 in Figure 8). Therefore, the target side of Clairvoyant must add 2 to the return address stored at the switch pointer before invoking the `return` instruction so that it points to the correct location. This operation is called a *hop* and is depicted in Figure 9 (a). If all active insertion points had been deleted and the developer issued the `continue` command, a hop operation is all that would be required.

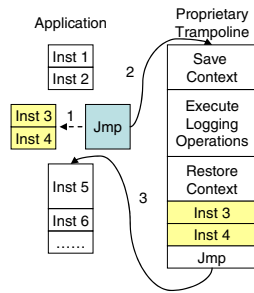


Figure 10. The context switch can be faster if a specialized trampoline is used for each insertion point, but this requires more program memory.

The return address can also be overwritten by a different value to resume the application from an arbitrary address. This is another advantage of using the *call* instruction to enter the trampoline instead of the typical *jmp* instruction: changing a *jmp* instruction would require a write to flash while changing the return address is a write to RAM.

If the developer issues the *stepi* command, the target side of Clairvoyant must execute one and only one instruction. To implement this, Clairvoyant first performs a hop operation and then adds a *system breakpoint* to the target of the first instruction to be executed. The system breakpoint is used internally and is not visible to the developer. If the target address can be calculated statically, it will be provided to Clairvoyant when the developer issues the *stepi* command. Otherwise, for instructions like indirect jumps or calls, the target address must be calculated by Clairvoyant dynamically given the state of the program. This operation is called a *block* and is depicted in Figure 9 (b). When the trampoline invokes the *return* instruction after a block operation, a single instruction is executed before the system breakpoint is encountered, control is transferred back to Clairvoyant, and the system breakpoint is removed.

If the developer issues a *step* or *next* command, the target side of Clairvoyant must execute an undefined number of instructions until the end of a statement is reached. GDB would execute this as if there were a number of *stepi* commands. However, each block operation requires two writes to program memory and one context switch, which are both slow and consume flash lifetime. Therefore, Clairvoyant uses GDB to calculate the longest sequence of deterministic instructions as possible, and places a system breakpoint at the last one of them. This optimization is called a *slide* and is depicted in Figure 9 (c). Like in the block operation, only the target address of indirect jumps or calls can not be statically determined, which makes the slide optimization very effective. When the trampoline invokes the *return* instruction after a slide, several instructions are executed before the system breakpoint is encountered, control is transferred back to Clairvoyant, and the system breakpoint is removed. Thus, Clairvoyant can sometimes execute a *step* or *next* command with only two writes to program memory and one context switch, which is the same cost as a *stepi* command even though multiple instructions have to be executed.

When a *continue* command is issued by the developer and

```
call Timer.start(TIMER_REPEAT, 1000);

event result_t Timer.fired() {
    post process();
}

task void process() {
    TRIGGER(PW0); // Start timing
    count = count ^ 5; // Quintic of count
    count++;
    TRIGGER(PW0); // End timing
    call SendMsg.send(count);
}
```

Figure 11. The program used to time the execution speed of *cond*, *watch* and all the logging commands.

the original breakpoint must be reinstated in its original location, a different process has to be used. The original breakpoint must be placed at the head of the restored instruction block, but this is also the location to which the target side of Clairvoyant must transfer control. Thus, Clairvoyant has to single step past the instruction block before reinstating the original breakpoint. In some cases, this is equivalent to two block operations, reinstating the breakpoint, and one hop operation. If the instruction block only contains one long instruction, however, only one single block operation needs to be performed. Similarly, if the target of the instruction block can be calculated statically, Clairvoyant can also implement the slide optimization. Therefore, in the best case, a *continue* command and the reinstatement of the original breakpoint can be executed with three writes to program memory and one context switch, as depicted in Figure 9 (d). Otherwise, five writes to program memory and two context switches are required to perform two separate block operations.

4.2.3 Partial Context Switch

The last two sections described the process of switching context between the application and Clairvoyant using the common trampoline. Although this approach saves program memory, it requires additional execution time because all commands require the full application context to be saved and restored. In most cases, this additional time is not significant compared to the time of reporting a trap and waiting for further commands. However, the logging commands do not interact with the developer, so use of the common trampoline could slow them down significantly.

To address this issue, we revert to something similar to the traditional use of proprietary trampolines for the logging commands, as depicted in Figure 10. A *jmp* instruction is placed in the target binary, and a unique trampoline is used for each and every instance of logpoints. This trampoline performs a *partial context switch*; only the registers that will be used by the logging operations are preserved. The developer can even choose not to save and restore the clock registers in order to get the maximum execution speed. The trampoline then executes the logging operations directly, instead of transferring control to Clairvoyant. Finally, the trampoline restores the application context, executes the instruction block that was replaced by the *jmp* instruction, and returns control to the application. This approach requires fewer writes to program memory than the normal context switch because it executes the instruction block in-place without

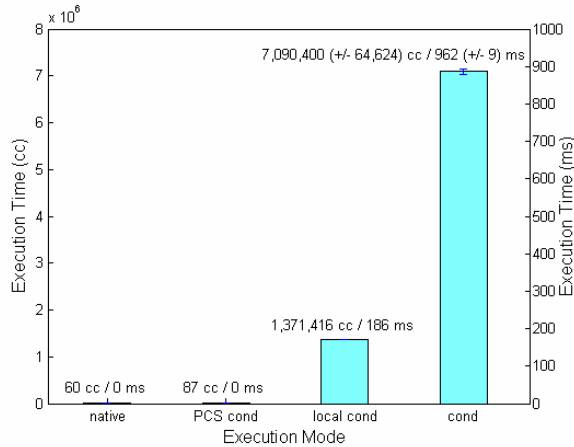


Figure 12. Execution speed of *cond* w/ no optimizations (*cond*), w/ local condition checking (*local cond*), w/ a partial context switch (*PCS cond*), and w/ *cond* disabled (*native*).

restoring it. One limitation of this approach is that only a limited number of logpoints can be instantiated simultaneously, because each one requires a unique trampoline.

4.3 Sharing the Radio

The approach described above causes the target program and the target side of Clairvoyant to alternately take full control of the hardware. One problem with this approach is that Clairvoyant must relinquish control of the radio while the target program is running, but the target program has no way of understanding debugging messages or forwarding them to Clairvoyant. Therefore, Clairvoyant cannot receive messages while the node is in application context.

This is problematic when debugging multi-hop sensor systems, where debugging messages must be routed through *mixed-context* networks where some nodes are in application context while others are in debugging context. In order to solve this problem, Clairvoyant modifies the target program’s radio stack the first time it takes control of the sensor node and links it with its own binary so that debugging messages are forwarded to Clairvoyant. Thus, Clairvoyant can receive debugging messages even when the sensor node is running in application context. If the message needs to be forwarded, Clairvoyant would switch the sensor node into debugging context, forward the message to its neighbors, and switch back into application context immediately. This hook into the target program’s radio stack is also used by the *stop* and *gstop* commands which, when received by the target, stop normal execution of the program and switch the sensor node into debugging context.

One limitation of this approach is that all messages received by a TinyOS application, including those addressed to Clairvoyant, are dispatched in *task context*. Thus, the *stop* and *gstop* commands can not stop program execution at arbitrary points, for example, when an asynchronous event handler is being executed. Another limitation of this approach is that Clairvoyant is subject to the application’s control of the radio while the sensor node is in application context; if

	Num	Time (cc) / Num	Summed Time (cc)
Total			1,371,416
Native Code Execution	1	60	60
Flash Writes	4	101,336	405,344
Saving of 256 Registers	2	2,540	5,080
Saving of Other Contexts	2	120	240
Restoring of 256 Registers	2	2,086	4,172
Restoring of Other Contexts	2	25	50
Clairvoyant Initialization	2	478,235	956,470

Table 4. Execution time breakdown of code w/ *local cond*.

the application puts the sensor node to sleep or turns the radio off, Clairvoyant can neither route debugging messages nor receive the *stop* and *gstop* commands. A third limitation of this approach is that the application must use a low-level radio stack similar to Clairvoyant’s; if Clairvoyant uses the standard TinyOS, 36-byte, CSMA-based radio stack, but the application uses TDMA or a 15-byte packet size, then multi-hop communication could not take place in mixed-context networks. A final limitation of this approach is that a breakpoint in the application’s radio stack would prevent Clairvoyant from receiving any debugging messages while in application context. This is not a serious limitation because the sensor node would switch into debugging context once the breakpoint were reached, and Clairvoyant would receive all subsequent messages through its own radio stack.

5 Impact on the Application

Clairvoyant tries to minimize its effect on the program being debugged. All debuggers affect program execution, but minimizing this effect reduces the number of so-called Heisenbugs – those bugs that change or disappear once the debugger is used. In this section, we evaluate the effect of Clairvoyant on network load, memory footprint, execution speed, clock consistency, and flash lifetime.

5.1 Execution Speed

Most WSN applications are very I/O intensive, and so small changes in execution speed can break a hardware driver or change the semantics of the sensing data that is collected. Furthermore, many networking protocols rely on implicit assumptions about the relative timing of neighboring nodes. As explained in Section 4, Clairvoyant runs the target program at native speed directly on the MCU, so execution speed is not affected by Clairvoyant unless a debugging command is executed. This is a key feature of Clairvoyant.

When a debugging command does execute, it does affect the temporal characteristics of the target. However, the time required for most debugging commands to execute is inconsequential because the largest delays will always be the developer’s idle time. For example, the developer could never step through execution of the low-level radio stack no matter how quickly Clairvoyant operates, because the physical hardware operates much more quickly than a human.

Clairvoyant does offer three types of commands for which execution speed is critical: *watch*, *cond*, and the logging commands. These commands do not require developer interaction, so the execution speed with which they can be implemented is a critical metric of the Clairvoyant’s temporal fidelity.

We evaluate the execution speed of these commands by

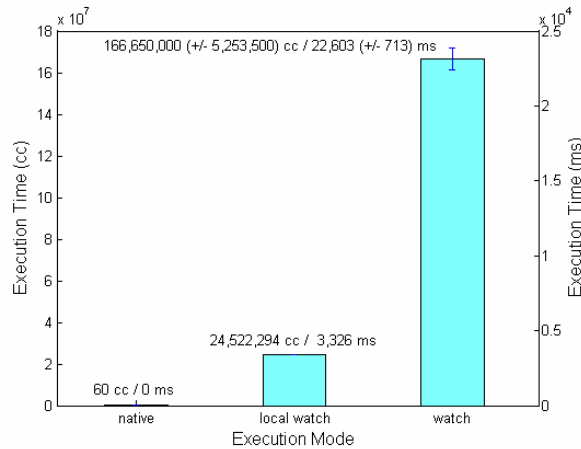


Figure 13. Execution speed of *watch* w/ no optimizations (*watch*), w/ local condition checking (*local watch*), and w/ *watch* disabled (*native*).

adding them to the program in Figure 11, and measuring total execution time on a Mica2 sensor node, which uses the 7.3728 MHz Atmel ATmega128 MCU [1]. This program simply calculates the fifth power of an integer and passes it on to the radio stack. It takes 60 clock cycles to execute when no debugging commands are being executed.

We timed execution speed with an external device that could measure execution time down to a single clock cycle by capturing the triggers of the PW0 output pin added into the program. For most experiments there was no measurement error since all execution took place on the sensor node itself and was totally deterministic. However, for those experiments that involved network communication, random delays were introduced by the MAC layer. In these cases, we repeated the experiment thirty times and reported the observed mean and standard deviation.

5.1.1 Cond

The *cond* command is used to convert a breakpoint into a *conditional breakpoint*, so that the application is only trapped if a logical condition is satisfied. This can be a very important command in WSNs because manually stepping through execution of a timing-sensitive operation is not possible. The developer can instead place a conditional breakpoint which freezes the state of the program when and only when the bug is likely to have occurred.

The simplest way to implement *cond* is to have the host side of Clairvoyant check the condition each time the breakpoint is reached, and resume the application automatically if the condition is found to be false. This requires several normal context switches and round-trip messages, but maximizes reuse of GDB functionality, and is also how the GDB remote protocol implements *cond*. Using this implementation, the conditional breakpoint `break process, cond 1 count>8` increased execution time from 60 clock cycles to 7 million clock cycles (0.962 seconds), with a relative standard deviation of 1% due to non-deterministic network communication.

We improved on this implementation by checking the

Command	Execution Time (cc)			
	Condition Satisfied		Condition Not Satisfied	
	Save Clk Regs	Not Save Clk Regs	Save Clk Regs	Not Save Clk Regs
led	89	35	N/A	N/A
led <i>cond</i>	100	46	83	29
ram	75	21	N/A	N/A
ram <i>cond</i>	86	32	83	29
flash	93,003	92,949	N/A	N/A
flash <i>cond</i>	93,010	92,956	83	29
radio	551,971 (±10,754)	551,917 (±10,754)	N/A	N/A
radio <i>cond</i>	551,978 (±10,754)	551,924 (±10,754)	83	29

Table 5. Execution overhead of logging commands.

condition locally on the sensor node. This requires certain GDB functionality to be reimplemented on the target side of Clairvoyant, increasing its program memory footprint but eliminates the need for round-trip messages. This implementation increased the execution time from 60 clock cycles to 1.4 million clock cycles (0.186 seconds), whose detailed breakdown is listed in Table 4. Most of this time is consumed by the two context switches required when a breakpoint is reached and then reinserted, as described in Section 4.2.2. This optimization trades program memory size for execution speed and reduced network load.

We further improved the *cond* implementation by using the partial context switch technique described in Section 4.2.3. Instead of using the common trampoline, a unique trampoline was created for each conditional breakpoint to save only enough registers to check the breakpoint's condition. Control is transferred back to the target program if the condition is false and a full context switch into debugger context is performed if the condition is true. Using this implementation, condition checking only increased execution time from 60 clock cycles to 141 clock cycles. In other words, the context switch and the condition checking only require 81 clock cycles in total. Because this optimization requires a new trampoline for each conditional breakpoint, it trades execution speed for program memory size.

Execution time was reduced even further by removing the code to preserve the clock registers. The Atmel Atmega128 MCU has four clocks that are saved and restored to preserve temporal consistency for the application – an operation that requires 54 clock cycles. By not preserving the clock registers, the program executed in 87 clock cycles, in other words, only 27 clock cycles of overhead were introduced by the condition checking. This optimization trades execution speed for clock consistency, and the effect of not preserving the clock registers on clock consistency is evaluated in Section 5.4. A comparison of execution speed for the three different *cond* implementations is illustrated in Figure 12.

5.1.2 Watch

The *watch* command traps the program when the value of a designated expression changes. The simplest way to implement this command is to have the host side of Clairvoyant single step through the program one instruction at a time and check the expression value in between. This maximizes reuse of GDB functionality and is also how the GDB remote protocol implements *watch*. However, this implementation

Variable	Meaning
M	number of statically unpredictable control-transfer instructions
B	number of active insertion points
I	number of machine instructions executed
F	number of previous frames on the stack
N	number of sensor nodes that the command applies to
R	number of registers modified during the call

Table 6. Meanings of the variables in Table 7 and Table 8.

of `watch` count increased execution time of the program in Figure 11 from 60 clock cycles to 167 million clock cycles (22.6 seconds), with a relative standard deviation of 3% due to non-deterministic network communication. Most of this execution time is consumed by round-trip messages between the host and target machines, and by performing a full context switch after each and every instruction is executed.

Similar to our implementation of `cond`, we improved on the native implementation of `watch` by checking the expression value locally on the sensor node. Again, this requires a partial reimplement of GDB functionality on the target side of Clairvoyant, but obviates the need for round-trip messages. This implementation increased the execution time from 60 clock cycles to 25 million clock cycles (3.3 seconds).

A comparison of execution speed for the two different `watch` implementations is illustrated in Figure 13. Using the partial context switch could dramatically increase the execution speed of `watch`, just like it did for `cond`. However, this implementation is not feasible because a new trampoline would be required for each instruction to be executed. Furthermore, even a partial context switch requires 27 clock cycles, so `watch` would never be able to run the application less than 27 times native speed and will never be able to debug timing-sensitive code, anyway.

5.1.3 Logging Commands

The logging commands are specifically designed for extracting state from timing-sensitive operations. To evaluate their execution speed, we issued commands to log the value of `count` to the LEDs, RAM, external flash, and radio, respectively. We then repeated this experiment while making the logging commands conditional on the value of `count`.

The results of this experiment are shown in Table 5. Each command has four values, depending on whether the condition is satisfied and whether the clock registers are preserved during the partial context switch. When the condition is not satisfied, all applicable commands introduce overhead of less than 30 clock cycles if not preserving clock registers. When the condition is satisfied, logging to external flash or radio takes a relatively long time due to I/O latency, but logging to LEDs or RAM introduces less than 40 clock cycles. If the clock registers are preserved, every command takes 54 more clock cycles to execute whether or not the condition is satisfied. For the `radio` command, a relative standard deviation of 2% was observed due to non-deterministic network communication. Based on these results, logging to LEDs or RAM could be useful for extracting state from timing-sensitive code such as hardware drivers, while logging to flash or radio could be useful for extracting state from application logic.

Command	Num of FWs	Num of RMs Total (R, S)	Comments
stop	0	2 (1, 1)	
reset	0	2 (1, 1)	
led/ram/flash/radio	1	2 (1, 1)	when defined
	0	0 (0, 0)	when reached
interrupts	0	2 (1, 1)	
ramread	0	2 (1, 1)	read length <= packet length
flashread	0	2 (1, 1)	read length <= packet length
gstop	0	N+1 (1, N)	
gcontinue	0	1 (1, 0)	
gdetach	0	N+1 (1, N)	
greset	0	N+1 (1, N)	
gbreak	0	0 (0, 0)	when defined
	B	N+1(0, N+1)	when reached

Table 8. The number of flash writes and the theoretical number of radio messages required by each CDB command. The variable meanings are described in Table 6.

5.2 Flash Writes and Radio Messages

Clairvoyant will affect the application by sending and receiving debugging messages that could interfere with the normal network traffic. We compare three possible implementations of Clairvoyant: (i) if Clairvoyant were implemented by strictly obeying the GDB remote protocol, (ii) the current implementation which reuses GDB functionality but communicates with a proprietary packet protocol, and (iii) if Clairvoyant had reimplemented all debugging functionality from scratch on the sensor node.

The GDB remote protocol issues a separate command for each primitive action listed in Table 3. For example, when a breakpoint is reached GDB removes all active insertion points one after another, and when a `step` or `next` command is issued GDB executes through the statement by single stepping each instruction. Thus, the fifth column of Table 7 shows that the number of radio messages required for the purely GDB-based `break` is a function of the number of active insertion points B . Similarly, the number of radio messages required for the purely GDB-based `step` or `next` depends on the number of executed instructions I . The numbers in this table indicate the number of logical radio messages, and do not account for retransmission due to multi-hop routing or packet losses.

As explained in Section 4.1, Clairvoyant reuses GDB functionality but communicates with a proprietary packet protocol to reduce the number of radio messages required for each command. This optimization reduces the number of radio messages required for most commands down to a constant number, as shown in the sixth column of Table 7. Three exceptions are `step`, `next`, and `backtrace`, for which the core functionality is implemented in GDB, but the execution depends on the dynamic state which can only be retrieved from sensor nodes.

The seventh column of Table 7 shows the hypothesized number of radio messages that would be required if all GDB functionality could be reimplemented on the sensor node. A minimum number of radio messages are required due to the sensor node receiving commands from the developer and notifying the developer of corresponding events. Therefore, the current implementation of Clairvoyant approaches the hypothetical minimum number of required radio messages for

Command	Number of Flash Writes (FWs)			Number of Radio Messages (RMs) Total (Receiving (R), Sending (S))			Comments
	GDB RP	PPP	LI	GDB RP	PPP	LI	
target remote	0	0	0	2 (1, 1)	2 (1, 1)	2 (1, 1)	
detach	0	0	0	2 (1, 1)	2 (1, 1)	2 (1, 1)	
continue	B	B	B	2B+1 (B+1, B)	1 (1, 0)	1 (1, 0)	command issued first time
	0	0	0	1 (1, 0)	1 (1, 0)	1 (1, 0)	not first time, w/o active insertion points
	B+4*	B+4*	B+4*	2B+5* (B+3*, B+2*)	3 (2, 1)	1 (1, 0)	not first time, w/ active insertion points
step	2I	2M	2M	2I+2 (I+1, I+1)	2M+2 (M+1, M+1)	2 (1, 1)	
stepi	2	2	2	4 (2, 2)	4 (2, 2)	2 (1, 1)	
next	2I	2M	2M	2I+2 (I+1, I+1)	2M+2 (M+1, M+1)	2 (1, 1)	statement not a call
	2I	2M	2M	2I+10 (I+5, I+5)	2M+6 (M+3, M+3)	2 (1, 1)	statement a call
finish	2B+4*	2B+4*	2B+4*	4B+8* (2B+4*, 2B+4*)	2 (1, 1)	2 (1, 1)	
break	0	0	0	0 (0, 0)	0 (0, 0)	0 (0, 0)	when defined
	B	B	B	2B+1 (B, B+1)	1 (0, 1)	1 (0, 1)	when reached
cond	0	0	0	0 (0, 0)	0 (0, 0)	0 (0, 0)	when defined
	2B+4*	0	0	4B+8* (2B+4*, 2B+4*)	0 (0, 0)	0 (0, 0)	when reached but not satisfied
	B	B	B	2B+3 (B+1, B+2)	1 (0, 1)	1 (0, 1)	when reached and satisfied
watch	2I	2I	2I	4I+2 (2I+1, 2I+1)	2 (1, 1)	2 (1, 1)	
delete	0	0	0	0 (0, 0)	0 (0, 0)	0 (0, 0)	
print	0	0	0	2 (1, 1)	2 (1, 1)	2 (1, 1)	read length \leq packet length
set	0	0	0	2 (1, 1)	2 (1, 1)	2 (1, 1)	write length \leq packet length
call	2B+4*	2B+4*	2B+4*	2R+4B+150* (R+2B+75*, R+2B+75*)	2 (1, 1)	2 (1, 1)	
backtrace	0	0	0	2F (F, F)	2F (F, F)	2 (1, 1)	
frame	0	0	0	0 (0, 0)	0 (0, 0)	0 (0, 0)	
info frame	0	0	0	0 (0, 0)	0 (0, 0)	0 (0, 0)	
list	0	0	0	0 (0, 0)	0 (0, 0)	0 (0, 0)	

Table 7. The number of flash writes and the theoretical number of radio messages required by each GDB command if Clairvoyant (i) strictly obeys the GDB Remote Protocol (GDB RP), (ii) uses the proprietary packet protocol (PPP), or (iii) uses a purely local implementation on the sensor node (LI). Each value marked with a star is the maximum. The variable meanings are described in Table 6.

most commands.

Besides radio messages, Clairvoyant will also affect the lifetime of a sensor node because it must rewrite the application’s binary stored in the flash-based program memory which, on the Mica2 sensor node platform, only has 10,000 write/erase cycles. The current implementation of Clairvoyant reduces the number of flash writes over the purely GDB-based implementation, as shown in the left-hand side of Table 7, although not as much as the number of radio messages. The GDB remote protocol would require two writes to flash for each instruction to be executed by the *step* or *next* command. Clairvoyant reduces this by the number of statically unpredictable control-transfer instructions M , primarily due to the slide optimization described in Section 4.2.2. Furthermore, when a conditional breakpoint is reached, a partial context switch is performed and the variable is evaluated, requiring no flash writes at all.

Besides the GDB commands, Table 8 lists each CDB command along with the number of flash writes and the theoretical number of radio messages required to execute the command. Most CDB commands do not require any flash writes, and only require a single round-trip radio message between both sides of Clairvoyant. The logging commands require a single flash write when defined in order to place the logpoint, but do not require any flash writes or radio messages when they are reached (the *flash* and the *radio* logging commands, of course, do use the flash and the radio for logging purposes). The main traffic load is incurred by Clairvoyant’s network-wide commands such as *gstop*, which send a message from each stopped node in the network back to the host side of Clairvoyant.

Although Clairvoyant approaches the potential minimum

for most commands, the number of flash writes and radio messages can quickly add up and, as with any debugging system, it is the responsibility of the developer to know how each command will affect the application. For example, the *gstop* command in a 10-hop network of 100 nodes would produce less than 1,000 radio messages. Using today’s 802.15.4 radios and Clairvoyant’s 36-byte packet format, each radio message would require less than 2 milliseconds to transmit, including MAC layer delays. Therefore, with an efficient routing protocol, the *gstop* command could theoretically be executed in less than 2 seconds. The additional network traffic introduced by Clairvoyant could cause congestion if load exceeds network capacity, especially if the currently used flooding protocols are not replaced with proper routing algorithms. In our experiments, we were able to run network-wide debugging commands on a multi-hop network of 10 nodes without any noticeable network effect, where half of the nodes were being debugged and the other half sent out an application message every 1 second. Debugging commands such as *watch* and *step [n]* perform context switches after every instruction and statement, respectively. These commands could quickly exceed the 10,000 write/erase cycles on the Mica2 sensor node’s flash-based program memory: at two flash writes per context switch, only 5,000 instructions could be executed if they were in the same writable block. Thus, the program memory would be exhausted after about 10 minutes of executing *watch*, since each instruction takes about 0.1 second to execute. Theoretically, the damage of flash writes may be mitigated by using a load distribution algorithm to distribute the writes evenly across the different writable blocks of flash, but this still needs further investigation. Because of the way Clair-

	Data Memory	Program Memory
Mica2	4,096 Bytes	131,072 Bytes
Basic Clairvoyant	875 Bytes	31,360 Bytes
break	8 Bytes	
gbreak	8 Bytes	
Local watch	6 Bytes	
Local cond	26 Bytes	
PCS cond	21 Bytes	256 Bytes
Logging Command	21 Bytes	256 Bytes

Table 9. Data memory and program memory required by the Clairvoyant’s target-side operations.

voyant is implemented, we recommend that developers use the *cond* command instead of *watch* and *step [n]* whenever possible.

5.3 Data Memory and Program Memory

The consumption of data memory and program memory is not a problem for traditional debuggers on PCs, but it becomes critical in the WSN domain. Sensor nodes are resource-constrained and lack OS support for virtual memory. Similarly, most sensor nodes do not have support for paging large programs into limited memory and thus require the entire application to be loaded at one time. Therefore, if the debugger consumes too much data or program memory, it will prohibit certain large applications from running on the sensor node.

Table 9 lists the data and program memory consumed by the target side of Clairvoyant. The minimal implementation only supports one breakpoint and requires 875 bytes of data memory and 31,360 bytes of program memory. A breakdown of these resource requirements is shown in Table 10. Most program memory is used for the executors, which implement the debugging commands, and the hardware drivers for accessing radio, LEDs, program memory, and external flash. These drivers are not shared with the application so that hardware operations issued by Clairvoyant would not change the state of the application.

Each additional *break* or *gbreak* requires 8 more bytes of data memory to store the breakpoint address and the original block of instructions at that address. The local implementations of *watch* and *cond* require additional data memory to store the addresses and the current values of the locally-monitored variables. In contrast, a conditional breakpoint that uses a partial context switch requires slightly less data memory because some data can be hard-coded in its proprietary trampoline, but this requires 256 extra bytes of program memory for that trampoline, which is the page size of the flash-based program memory. Thus, Clairvoyant offers a trade-off between execution speed of conditional breakpoints and the amount of required program memory. An extra trampoline is also required for each supported instance of logging commands. The default Clairvoyant configuration supports 8 *break*/*gbreak*, 2 *watch*, and 4 *cond* or logging commands, requiring a total of 1,027 bytes of data memory and 32,384 bytes of program memory.

5.4 Clock and Timer Consistency

When running most source-level debuggers such as GDB, the clock on the target machine continues to run even when the application has been stopped. On sensor nodes, however, this can cause incorrect behavior of the software timers and

	Data Memory	Program Memory
Total	1,027 Bytes	32,384 Bytes
Communicator	139 Bytes	624 Bytes
Interpreter	0 Bytes	638 Bytes
Executors	201 Bytes	14,846 Bytes
Hardware Drivers	421 Bytes	14,996 Bytes
Register Pool	256 Bytes	
RAM Logging Pool	10 Bytes	
Trampolines		1,280 Bytes

Table 10. Data memory and program memory breakdown of the Clairvoyant’s target-side executable.

any other application code that uses the clock register values. For example, if the application sets a timer to fire after 100 clock ticks but the sensor node happens to be in debugging context at that time, the interrupt would be suppressed. Once the application is resumed, the timer may not fire at all, or may fire only after the clock register overflows, returns to 0, and then counts back up to 100. In another example, an application that compares subsequent readings of a clock register may observe negative time changes after switching into and out of debugging context.

To solve these problems, the target side of Clairvoyant saves and restores the clock registers with each context switch to preserve time *with respect to application context*. Atmel Atmega128 MCUs have four different clocks, which are named T/C₀ through T/C₃. T/C₀ and T/C₂ each have an 8-bit clock register while T/C₁ and T/C₃ each have a 16-bit clock register. The trampoline saves the clock registers at the very beginning of a switch out of application context. It then subtracts a value from each clock register and restores it right before switching back to application context. The value subtracted accounts for the time that passes before the trampoline can save the clock register and after the trampoline can restore it, thus, this value differs for each clock register.

Preserving the clock registers is necessary for most debugging commands because they take a long time. Some logging commands and our partial-context-switch implementation of *cond*, however, can execute very quickly, as shown in Section 5.1. If the partial context switch for the *led* and the *ram* logging commands does not preserve the clock registers, it only causes a loss of 35 and 21 clock cycles, respectively. Condition checking for the conditional breakpoints would only cause a loss of 27 clock cycles. Thus, these commands offer a trade-off: they can be executed 54 clock cycles more quickly, by causing a loss of clock consistency to the extent of less than 40 clock cycles. This trade-off may make sense depending on how timing-sensitive the debugged code is and how much the code correctness depends on the clock register values. This trade-off usually does not make sense for the *flash* and the *radio* logging commands because these relatively long-running commands cause a loss of clock consistency to 92,949 clock cycles and 551,917 clock cycles, respectively.

6 Case Studies

In this section, we use two case studies to demonstrate some of the commands that are provided by Clairvoyant. As with all source-level debuggers, Clairvoyant can help test a hypothesis about a bug, but the process of developing this hypothesis highly depends on the developer’s understanding

```

1 TOSH_INTERRUPT(SIG_INTERRUPT7) { --> [delete]
2     --> [insert] TOSH_SIGNAL(SIG_INTERRUPT7) {
3     call MicInterrupt.disable();
4     __nesc_enable_interrupt();
5     signal MicInterrupt.toneDetected();
6     }
7
8 event result_t Timer.fired() {
9     call MicInterrupt.enable();
10    return SUCCESS;
11 }
12
13 async event result_t MicInterrupt.toneDetected() {
14    call Leds.greenToggle();
15    return SUCCESS;
16 }

```

Figure 14. The code used to cause a stack overflow in the interrupt handler routine of microphone.

of the application and related debugging experiences. Thus, each of these case studies assumes a pre-developed hypothesis and illustrates the commands used to test the hypothesis, the outcome of these commands, and how they helped confirm or deny the hypothesis. We focus on the logging commands and our partial-context-switch implementation of *cond* because they are specially designed to debug timing-sensitive modules of WSN applications, such as low-level hardware drivers. Both of these case studies are derived from real, documented bugs that can be found in the TinyOS-1.x source tree.

6.1 Stack Overflow

This bug occurred in the interrupt handler routine of microphone (*/tinyos-1.x/tos/platform/mica2/HPLMicC.nc*) and was corrected in revision 1.2 of the TinyOS-1.x source tree. As Figure 14 illustrates, the problem is that interrupts are not disabled immediately after calling `TOSH_INTERRUPT` at line 1. Thus, a long whistle at the right frequency will cause the tone detector interrupt on the Mica Sensor Board to continually fire, preemptively invoking the handler routine multiple times in a row and causing a stack overflow. To fix this bug, the `TOSH_INTERRUPT` specifier should be replaced by `TOSH_SIGNAL` (line 2), for which all interrupts are automatically turned off to prevent preemption.

In this case study, the bug can be observed when the green LED does not toggle as expected and the sensor node reboots instead. To test the hypothesis of a stack overflow, we issued the `break _vector_8, cond 1 StackCheck 0x800` commands to check whether the application’s stack pointer at line 3 ever falls below the address `0x800`. Our partial-context-switch implementation of *cond* only introduces an overhead of less than 30 clock cycles, so it is unlikely to change the re-entrant characteristics of the suspended function.

Once this conditional breakpoint was in place, we ran the program and reached the breakpoint, which was strong evidence of stack overflow. The hypothesis was further confirmed with a `backtrace` command, which showed that the handler routine was on the stack multiple times. We corrected the program and repeated the above process to find that the sensor node worked correctly.

6.2 Deadlock

This bug occurred in the CC1000 radio stack (*/tinyos-1.x/tos/platform/mica2/CC1000RadioIntM.nc*) and was cor-

```

1 task void shortDelay() {
2     TOSH_uwait(10);
3 }
4
5 void postDelay() {
6     for (i = 0; i < 8 i++) {
7         post shortDelay();
8     }
9 }
10
11 .....
12 case TXSTATE_DONE:
13 default:
14     if (bTxPending == TRUE) {
15         postDelay();
16     }
17     call SpiByteFifo.rxMode(); --> [delete]
18     call CC1000Control.RxMode(); --> [delete]
19     bTxPending = FALSE;
20     if (post PacketSent()) {
21         --> [insert] call SpiByteFifo.rxMode();
22         --> [insert] call CC1000Control.RxMode();
23         RadioState = IDLE_STATE;
24         RSSIInitState = RadioState;
25         call RSSIADC.getData();
26         .....
27
28 event result_t Timer.fired() {
29     post processing();
30     return SUCCESS;
31 }
32
33 task void processing() {
34     call Send.send(TOS_BCAST_ADDR, sizeof(IntMsg), &readyMsg);
35 }

```

Figure 15. The code used to cause a deadlock in the CC1000 radio stack.

rected in revision 1.30 of the TinyOS-1.x source tree. As Figure 15 illustrates, the issue is that `SpiByteFifo.rxMode()` and `CC1000Control.RxMode()` are both called in the `TXSTATE_DONE` case, whether or not `post packetSent()` succeeds (lines 17 to 18). These calls invoke `TOSH_uwait()`, which makes the execution path fairly long (400 microseconds). When the task queue overflows, causing the post to fail, the same execution path will be taken the next time an SPI interrupt is handled. The problem is that the SPI interrupt rate is high enough that these calls will essentially starve the task queue from executing, which hangs the sensor node forever. The fix this bug, the function calls should be moved inside the block that executes only when the post succeeds (lines 21 to 22).

To repeat the bug on a sensor node, we hacked the CC1000 radio stack to overflow the task queue at a specific point (lines 14 to 16). This modification was fairly simple and our experiments showed that it did not affect the normal functionality at all.

In this case study, the bug can be observed when no messages from the sensor node are received. The hypothesis of a full task queue could be tested by issuing the `led CC1000RadioIntM.nc:733 TOSH_sched_free YBR` command to display the value of `TOSH_sched_free` on the three LEDs at line 20. The head of the task queue `TOSH_sched_free` always changes when a new task is successfully posted, so we would expect to see the LEDs change frequently. We chose *led* but not other logging commands for two reasons: (i) it requires less than 40 clock cycles to execute, which should not impair the timing-sensitive operations of the CC1000 radio stack, and (ii) we needed continuous, real-time status reporting.

After the logpoint was inserted, the LEDs displayed a value but did not change at all. This supported the hypothesis that the task queue was always full when `post packetSent()` was about to execute. After we corrected the program and repeated the above process, messages could be successfully sent out and the LEDs changed display frequently, further confirming that the bug was a deadlock caused by task queue overflow.

7 Future Work

In future work, we plan to explore the possibilities of implementing Clairvoyant on an MCU like TI MSP430 [3]. This MCU has several important differences from Atmel Atmega128, with much less program memory and the ability to run instructions from RAM. Clairvoyant would consume more than half of the MSP430's 48 KB program memory, but it may be possible to efficiently page in the debugged program and run its instructions from RAM. Other possibilities may be to reuse the hardware drivers from the application through dynamic re-linking and state preservation. We will also explore networking algorithms to efficiently transport debugging messages through large-scale, bandwidth-limited, multi-hop networks, as well as the possibility of running Clairvoyant over wired testbeds.

8 Conclusions

This paper proposes *Clairvoyant*, the first comprehensive *source-level debugger* for WSNs. It simultaneously satisfies the demands of several domains that are notoriously difficult to debug, including (i) embedded systems, (ii) distributed systems, (iii) event-driven, I/O-oriented computing, and (iv) multi-hop, wireless networking.

We demonstrated that a suite of specialized commands besides the standard ones provided by GDB are both useful and necessary for debugging WSNs because, among other reasons, we need to debug timing-sensitive operations, access physical hardware, and exploit the wireless broadcast channel. We designed new techniques for context switches on a firmware-based device without OS support. We evaluated our implementation and found that it is feasible to provide source-level debugging within the resource constraints of a sensor node, and without additional hardware. Through the implementation process, we uncovered fundamental challenges to debugging multi-hop, wireless, distributed, embedded systems, including the issues of routing through mixed-context networks and of sharing the same radio and communication channel with the target program. We also uncovered several fundamental trade-offs, including: (i) data and program memory for execution speed, (ii) clock consistency for execution speed, (iii) data and program memory for number of radio messages, (iv) data and program memory for number of flash writes, and (v) data memory for program memory. We hope that this work will form a foundation for further advances in source-level debugging of WSNs.

Acknowledgment

Special thanks to Na Zhang, Nguyet Nguyen, John Heidemann, and many anonymous reviewers for their insightful and constructive comments.

9 References

- [1] Atmel Corporation. Atmega128L Specifications. http://www.atmel.com/dyn/products/product_card.asp?part_id=2018.
- [2] Atmel Corporation. Mature AVR JTAG ICE. http://www.atmel.com/dyn/products/tools_card.asp?tool_id=2737.
- [3] J. Beutel. Metrics for sensor network platforms. In *Proc. REALWSN'06*, 2006.
- [4] D. Culler, J. Hill, P. Buonadonna, R. Szewczyk, and A. Woo. A network-centric approach to embedded software for tiny devices. In *Proc. EMSOFT'01*, 2001.
- [5] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC language: a holistic approach to networked embedded systems. In *Proc. PLDI '03*, 2003.
- [6] L. Girod, J. Elson, A. Cerpa, T. Stathopoulos, N. Ramanathan, and D. Estrin. EmStar: a software environment for developing and deploying wireless sensor networks. In *Proc. USENIX'04*, 2004.
- [7] L. Gu and J. Stankovic. *t*-kernel: providing reliable OS support to wireless sensor networks. In *Proc. SenSys'06*, 2006.
- [8] J. Hui and D. Culler. The dynamic behavior of a data dissemination protocol for network programming at scale. In *Proc. SenSys'04*, 2004.
- [9] J. Jones and M. J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proc. ASE'05*, 2005.
- [10] P. Levis, N. Lee, M. Welsh, and D. Culler. TOSSIM: accurate and scalable simulation of entire tinyOS applications. In *Proc. SenSys'03*, 2003.
- [11] P. Levis, N. Patel, D. Culler, and S. Shenker. Trickle: a self-regulating algorithm for code propagation and maintenance in wireless sensor networks. In *Proc. NSDI'04*, 2004.
- [12] L. Luo, T. He, G. Zhou, L. Gu, T. Abdelzaher, and J. Stankovic. Achieving repeatability of asynchronous events in wireless sensor networks with EnviroLog. In *Proc. INFOCOM'06*, 2006.
- [13] Microsoft Corporation. *Visual Studio User Reference*. Microsoft Developer Network, 2006.
- [14] Microsoft Corporation. *MSDN Library Technical Reference*. Microsoft Developer Network, 2007.
- [15] R. Netzer. Optimal tracing and replay for debugging shared-memory parallel programs. In *Proc. PADD'93*, 1993.
- [16] J. Polley, D. Blazakis, J. McGee, D. Rusk, and J. Baras. ATEMU: a fine-grained sensor network simulator. In *Proc. SECON'04*, 2004.
- [17] N. Ramanathan, K. Chang, R. Kapur, L. Girod, E. Kohler, and D. Estrin. Sympathy for the sensor network debugger. In *Proc. SenSys'05*, 2005.
- [18] S. Shavor, J. D'Anjou, S. Fairbrother, D. Kehn, J. Kellerman, and P. McCarthy. *The Java Developer's Guide to Eclipse*. Addison-Wesley Professional, 2003.
- [19] Sun Microsystems. *Java Development Kit Documentation*. Sun Developer Network, 2007.
- [20] SunSoft Inc. *Solaris Application Developer's Guide*. Prentice Hall, 1997.
- [21] The GDB developers. GDB: the GNU project debugger. <http://sourceware.org/gdb>.
- [22] The TinyOS developers. Hardware designs. <http://www.tinyos.net/scoop/special/hardware>.
- [23] The TinyOS developers. The TinyOS Message Center tool. <http://www.tinyos.net/tinyos-1.x/doc/mcenter.html>.
- [24] B. Titzer, D. Lee, and J. Palsberg. Avrora: scalable sensor network simulation with precise timing. In *Proc. IPSN'05*, 2005.
- [25] G. Tolle and D. Culler. Design of an application-cooperative management system for wireless sensor networks. In *Proc. EWSN'05*, 2005.
- [26] P. Wadler. Why no one uses functional languages. *SIGPLAN Notices*, 33(8), 1998.
- [27] K. Whitehouse, G. Tolle, J. Taneja, C. Sharp, S. Kim, J. Jeong, J. Hui, P. Dutta, and D. Culler. Marionette: using RPC for interactive development and debugging of wireless embedded networks. In *Proc. IPSN/SPOTS'06*, 2006.
- [28] L. Wittie. Debugging distributed C programs by real time replay. In *Proc. PADD'88*, 1988.
- [29] X. Wu, Q. Chen, and X.-H. Sun. A java-based distributed debugger supporting MPI and PVM. *Parallel Numerical Linear Algebra*, 2001.
- [30] J. Yang, S. Zhou, and M. L. Soffa. Dimension: an instrumentation tool for virtual execution environments. In *Proc. VEE'06*, 2006.

