Macrodebugging: Global Views of Distributed Program Execution

Tamim Sookoor, Timothy Hnat, Pieter Hooimeijer, Westley Weimer, and Kamin Whitehouse

Department of Computer Science, University of Virginia
Charlottesville, VA, USA
{sookoor,hnat,pieter,weimer,whitehouse}@cs.virginia.edu

Abstract

Creating and debugging programs for wireless embedded networks (WENs) is notoriously difficult. *Macroprogramming* is an emerging technology that aims to address this problem by providing high-level programming abstractions. We present *MDB*, the first system to support the debugging of macroprograms. MDB allows the user to set breakpoints and step through a macroprogram using a source-level debugging interface similar to GDB, a process we call *macrodebugging*. A key challenge of MDB is to step through a macroprogram in sequential order even though it executes on the network in a distributed, asynchronous manner. Besides allowing the user to view distributed state, MDB also provides the ability to search for bugs over the entire history of distributed states. Finally, MDB allows the user to make hypothetical changes to a macroprogram and to see the effect on distributed state without the need to redeploy, execute, and test the new code. We show that macrodebugging is both easy and efficient: MDB consumes few system resources and requires few user commands to find the cause of bugs. We also provide a lightweight version of MDB called *MDB Lite* that can be used during the deployment phase to reduce resource consumption while still eliminating the possibility of heisenbugs: changes in the manifestation of bugs caused by enabling or disabling the debugger.

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, Macroprogramming, Wireless Embedded Networks

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SenSys'09, November 4–6, 2009, Berkeley, CA, USA.
Copyright 2009 ACM 978-1-60558-748-6 ...$5.00

1 Introduction

Creating and debugging programs for wireless embedded networks (WENs) is notoriously difficult. *Macroprogramming* is an emerging technology that aims to address this problem by providing high-level programming abstractions: the user writes a single *macroprogram* that specifies high-level distributed operations (i.e., leader election or contour finding), and the system automatically converts these into *microprograms* that specify local actions for each node (i.e., sensing, message passing, and local processing). Macroprograms do not actually execute on any node; all nodes execute microprograms, and the operations specified in the macroprogram are thus executed in a distributed fashion. Macroprogramming provides the user with the illusion of programming a single machine by abstracting away the low-level details of message passing and distributed computation. This promising approach has recently attracted dozens of prototype implementations with a wide array of programming models, including relational databases [33], geographic regions [48], and logical rules [10]. None of these systems, however, provide support for debugging, which is a crucial stage in the development cycle as a macroprogram moves from the drawing board to real deployments.

We present *MDB*, the first system to support the debugging of macroprograms. MDB allows the user to set breakpoints and step through a macroprogram using traditional source-level debugging commands, much like GDB [47]. This provides the same abstraction as debugging a sequential program on a single machine, even though the macroprogram executes in a distributed, asynchronous manner on the network. We call this process *macrodebugging*. A key challenge is to allow the user to step through a macroprogram in a sequential order, even if the nodes are not all executing the same distributed operations at any given time. MDB addresses this challenge by providing two ways to view distributed state: (1) *logical views* depict the distributed state where each node is executing the same logical operation in the macroprogram, although possibly at different times, and (2) *temporal views* depict distributed state of the entire system at a fixed time, even though nodes may not all be executing the same distributed operation. Both of these interfaces support *time travel*, which means that the user can step both forward and backward through the code.

In addition to the ability to view distributed state, MDB provides two additional functions that are not supported by

most existing source-level debuggers. First, *historical search* allows the user to search for the manifestation of a bug over the entire historical sequence of distributed states, without manually stepping forward and backward through the code. Second, MDB allows the user to make *hypothetical changes* to a macroprogram at debugging time, and to observe the effect of these changes on distributed network state without the need to redeploy, execute, and test the new code.

We evaluate MDB on three macroprogams running on a 21 node wireless testbed, and find that MDB has modest memory, execution, and energy overhead: approximately 300 B of memory, 0.5% of the CPU, and 30% energy overhead. This energy overhead is substantial enough that the user would probably disable MDB during the deployment phase, introducing the possibility of *heisenbugs*: changes in the manifestation of bugs caused by enabling or disabling the debugger. Therefore, we introduce a lightweight implementation called *MDB Lite* that only has 0.9% energy overhead. MDB Lite does not provide debugging support, but it does preserve the timing and memory characteristics of MDB, allowing the user to reduce energy overhead while still eliminating the possibility of heisenbugs.

MDB is implemented for a macroprogramming system called *MacroLab* [23], and is a *post-mortem* debugger, which means that it collects data about the system at run time and allows the user to inspect program execution after the logs are retrieved. However, the underlying principles of MDB can be applied to other macroprogramming systems, and at least some of them can be applied to on-line debugging, as discussed in Section 7. This paper makes the following five contributions to the fields of macroprogramming and debugging:

1. The first debugger to support macroprogramming
2. The first debugger for WENs to support time travel
3. The first debugger for WENs to allow searching for bugs over the historical sequence of distributed states, without manually stepping forward and backward
4. The first debugger for WENs to show the effect on distributed state of hypothetical changes to the code, without the need to redeploy, execute, and test the new code
5. The first debugger for WENs to provide a lightweight implementation to reduce energy overhead while preserving timing and memory characteristics

2 Background and Problem Definition

MDB is the first debugger that allows the user to navigate and view the distributed state of a program in terms of high-level macroprogramming abstractions. In this section, we describe the fundamentals of macroprogramming, provide an overview of MacroLab, and illustrate an example macroprogram. We also identify several possible bugs that could appear in the macroprogram, which we use to motivate the MDB user interface in Section 3.

2.1 Macroprogramming

Node-level programming is the process in which a developer manually creates the program that will run on each node, specifying node-local actions such as sensing, message passing, and local processing. These programs then execute on the nodes and the interactions between them produce emergent, network-wide behaviors. This programming model is notoriously difficult to use because the emergent network behavior is never explicitly specified and is instead fragmented among the programs of multiple different nodes. Furthermore, the emergent network behavior is difficult to predict: the user must have a mental model of each node and be able to mentally simulate the interactions between the nodes. This is particularly challenging given the complex, dynamic, and non-deterministic nature of WENs: execution flows non-deterministically between nodes via unreliable broadcast messages and starts spontaneously on nodes due to timer and sensor interrupts. Despite these challenges, node-level programming is the most common way to program a WEN [4, 15, 20].

Macroprogramming systems address these problems to some extent by allowing the user to program an entire network as if it were a single machine, using abstract, distributed data structures such as database tables [33], logical facts [10], or data streams [49]. The user creates a *macroprogram* that uses the abstractions to specify network-wide operations, which the system automatically converts into *microprograms* that specify the appropriate local actions for each node. Thus, the macroprogram never executes on any device in the network; all nodes execute microprograms that cooperatively execute the global operations specified in the macroprogram. Macroprogramming enables the user to write a single program that explicitly specifies global, network-wide operations, and it eliminates the need to manually specify local node actions.

Dozens of macroprogramming prototypes have recently been proposed [2, 3, 22, 23, 27, 48], but unfortunately none of these systems provide support for debugging. Thus, macroprogramming systems make it easier to create programs, but do not make it easier to debug them: the user must still resort to the examination of low-level details such as message traces and the local state on each node. Arguably, high-level macroprogramming abstractions make debugging even more difficult because the user must analyze the execution microprograms that were automatically generated by the system. Debugging is an important phase in the development cycle and the lack of debugging support in existing macroprogramming systems decreases any ease-of-use advantages that they may have over node-level programming. The goal of MDB is to fill this important gap in the macroprogramming tool chain by allowing the user to debug a macroprogram using the same high-level abstractions that were used to create it.

2.2 MacroLab and a Motivating Example

MDB is implemented for a macroprogramming system called *MacroLab*, which provides a vector-based syntax similar to Matlab [1]. All data on nodes, including sensor values, internal state, and parameters for actuation, are abstracted for the user as vectors called *macrovectors*. The user can operate on macrovectors with Matlab's standard set of vector operations such as `max`, `min`, `sum`, or `find`[1], and the

[1] Matlab's `find` operator returns the indices of non-zero elements in a vector

```
1  motes = RTS.getMotes('type', 'tmote')
2  magSensors = SensorVector(motes, 'magnetometer')
3  magVals = Macrovector(motes)
4  neighborMag = neighborReflection(motes, magVals)
5  THRESH = 500
6  every(1000)
7     magVals = magSensors.sense()
8     active = find(sum(neighborMag>THRESH, 2) > 3)
9     maxNeighbor = max(neighborMag, 2)
10    leaders = find($\ldots$
11            maxNeighbor(active) == magVal(active))
12    focusCameras(leaders);
13 end
```

Figure 1. MacroLab code for a WEN that tracks an object. Every 1000 *ms*, nodes take a reading from their magnetometers and share the value with their neighbors. If more than three nodes in a neighborhood sense a magnetometer value above a threshold, a leader is elected from among them and a camera is focused on it.

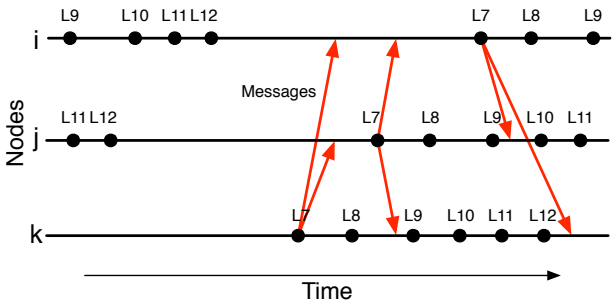


Figure 2. Nodes i, j, and k execute lines of the macroprogram in Figure 1 asynchronously. Dots indicate when a node executed a particular line of code, and arrows indicate the transmission of cache update messages.

system compiles these down into local actions for each node that cooperatively execute the vector operations [23]. Thus, macroprograms are written in a Matlab-like syntax and compiled down to microprograms in nesC [20] that run on mote-class devices.

Macrovectors are similar to traditional vectors except that each row is indexed by node ID instead of an ordinal set of integers. Macrovectors can be stored in the network in multiple different ways. For example, all elements of a *centralized* macrovector are stored on a single device, whereas each element of a *distributed* macrovector is stored on the node corresponding to that row. The MacroLab compiler automatically chooses the mode of storage for each macrovector. MacroLab also offers several types of *caching macrovectors*. For example, all elements of a *reflected* macrovector are stored on all nodes in the network and each time a node writes to one element, the value is broadcast to all other nodes and cached. A *neighborhood* macrovector is similar, but values are only reflected over a local neighborhood. These caching macrovectors allow the system to achieve various forms of distributed concensus. MacroLab only supports *best-effort* data synchronization, which means that it makes no coherence guarantees on cached values. However, users can add traditional synchronization techniques to a macroprogram, such as locks [12] or barriers [24].

Figure 1 shows an example of a MacroLab program that will be used throughout the rest of the paper. This program implements the Object Tracking Application (OTA), in which a network of sensors cooperate to locate and focus a camera on a moving object [50]. The overall algorithm is to find *active* neighborhoods with at least three nodes that detect an object (to prevent false positives), and to focus a camera on the node with the highest sensor value in that neighborhood, which is likely to be closest to the moving object. In lines 1–3, the code intializes the `motes` vector of node IDs, the `magSensors` vector of magnetometer sensors, the `magVals` macrovector of magnetometer readings. Line 4 initializes `neighborMag`, which is an $n \times n$ neighbor reflection vector, where each element i, j has a cached copy of the jth element of `magVals` if i is a neighbor of j, and an invalid value otherwise. Every 1000 *ms*, line 7 reads from all magnetometer values, and line 8 creates an `active` vector with the IDs of all nodes that have at least three neighbors with values higher than `THRESH`. Line 9 creates a `maxNeighbor` vector with the highest sensor value in each node's neighborhood, and lines 10–11 create a `leaders` vector, which contains the IDs of those nodes that have the highest sensor value in an active neighborhood. Finally, line 12 uses a proprietary function called `focusCameras` to focus all available cameras on the leader nodes, using a pre-defined mapping from nodeID to location. The user or camera manufacturer would need to write the `focusCameras` function, which is basically a hardware driver, while the standard Matlab functions such as `find`, `sum`, and `max` are provided by the MacroLab system. The hardware drivers for the sensors are also provided by the MacroLab system by using the TinyOS operating system. See the MacroLab paper for more details [23].

The MacroLab compiler could decompose the program in Figure 1 and execute it on the network in multiple different ways. For example, a completely centralized decomposition would store all vectors and perform all computation on a base station; the nodes would simply read from the sensors and forward the values to the base station. In a completely distributed decomposition, every node would store the elements of each macrovector corresponding to its own node ID and all operations would be performed in a distributed manner: each node would read from its own sensor and store the value in its `magVals` element, which would be *reflected* to all neighboring nodes and cached in their `neighborMag` vector. Each node would then calculate whether it was in an active neighborhood and whether it was the leader. If so, it would send its node ID to the base station, which would call the `focusCameras` function.

The program in Figure 1 does not include any data synchronization, and so all nodes will execute it *asynchronously*, reading from their sensors, broadcasting their values, and calculating their own leadership status independently of the other nodes. Figure 2 illustrates this asynchronous execution using a space-time diagram of three neighboring nodes i, j, and k. Time progresses from left to right and the dots

on the lines indicate the points in time when each node executes a particular line in the macroprogram. For instance, the leftmost point on each line indicates that the three nodes are executing lines 9, 11, and 7 of the macroprogram. When the nodes reach line 7 to read and store their magnetometer values, these values automatically get reflected to neighboring nodes via a radio broadcast message and populate the caches of the `neighborMag` vector. This is illustrated by the arrows in the diagram. In this example, no node is elected the leader, and so no messages are sent to the base station when the nodes reach line 12.

2.3 Three Types of Macroprogramming Bugs

Although the program in Figure 1 is very simple, several bugs are possible due to factors such as human error, message loss, and message races. In this section, we enumerate three bugs that are representative of the types of bugs that might appear in a typical MacroLab program. These bugs will be used in Section 3 to motivate the design of the MDB interface.

Bug 1 – Logical Error: A logical error occurs when the logic of the program is specified incorrectly, perhaps due to a typographical error or a poor understanding of the application by the human user. For example, on line 5 of the code in Figure 1, the user could accidentally set `THRESH` to be 5000 instead of 500, or the user could mistakenly write line 8 as:

```
>> active = find(sum(magVals > THRESH, 2) > 3)
```

Either one of these typos would result in the `active` variable always being an empty vector, in which case no node would ever be elected a leader in the network. Either of these bugs would produce the same manifestation: that moving objects are never detected by the application.

Bug 2 – Configuration Error: A configuration error occurs when the logic of the application is correct, but does not match the details of the deployment topology or physical stimuli. For example, line 6 in the example macroprogram reads from the sensors every 1000 *ms*, but the objects that are being tracked may move past the nodes much more quickly than that. Similarly, the example program requires at least three neighboring nodes to detect a mobile object, but the nodes may be spaced out so far from each other that only one or two nodes ever detect the mobile objects. Either of these bugs would produce the same manifestation: that moving objects are sporadically or inconsistently detected by the application.

Bug 3 – Synchronization Error: A synchronization error occurs when the logic and the configuration of the application are correct, but the desired result is not produced because of message loss, data races, or asynchronous execution. For example, if node 1 has the maximum sensor reading in its neighborhood and node 2 has the second highest reading, node 1 should be elected the leader and node 2 should not. If node 2 does not receive the cache update message with node 1's sensor reading, however, the local computations on node 2 would compute that it has the highest reading and it would also be elected the leader. This data synchronization bug would produce the manifestation of having two leader nodes in the same neighborhood. In a similar scenario, node 1 may read from its sensor before receiving the cache updates from any of its neighbors, perhaps due to network delays or because node 1 sensed much earlier than its neighbors. Its local computations would therefore observe that its neighborhood was not active, and it would not be elected the leader. Meanwhile, all of node 1's neighbors would conclude that node 1 was the neighborhood leader because it has the highest sensor reading. This bug would manifest in the application missing the detection of a mobile object.

3 The MDB User Interface

The goal of MDB is to provide a source-level debugging interface for macroprograms with which the user can place breakpoints, step through the code, and inspect the values of variables. MDB allows the user to debug a single macroprogram, navigating execution and viewing system state in terms of high-level operations and data structures. This alleviates the need to debug the node-level programs running on each node, including details such as radio messages and hardware interrupts. The key challenge for MDB is that nodes can execute *asynchronously*: each node may be executing a different part of the macroprogram at any given time, and nodes may execute the parts of any given distributed operation at different times. For this reason, MDB provides two different types of views: *logically synchronous views* depict distributed state where all nodes are executing the same line of code, and *temporally synchronous views* depict the distributed state at a given time.

MDB is a post-mortem debugger, which means that the user steps through a pre-recorded execution trace and does not view distributed state at execution time. As a result, MDB is able to provide *time travel* commands, allowing a user to step both forward and backward through the code. MDB also provides *historical search*, which allows the user to search over the historical sequence of distributed states for the manifestation of a bug without manually stepping forward and backward. For example, the user can find the time that a variable had a particular value, or can plot all sensor readings from a particular node.

The disadvantage of post-mortem debugging is that the user cannot modify system state during execution, as one might when using an on-line debugger such as GDB [47]. MDB overcomes this limitation to some extent by providing *hypothetical changes*, which allows the user to view how certain changes to the program would affect system state. Hypothetical changes allow a user to test whether a particular change to a program would fix a given bug, without the need to redeploy, execute, and test the new code.

Table 1 provides a summary of the main commands offered by MDB. In the following subsections, we illustrate how these commands can be used to address the bugs discussed in Section 2.3. The examples and system outputs are derived from real experiments in which the authors created the conditions for each bug, executed the example program and debugging commands, and saved the output for exposition purposes.

3.1 Logically Synchronous Views

Logically synchronous views allow the user to inspect and analyze the distributed state of the system when all nodes

Command	Description
tjump (t)	change the state of the system to time t μs
tstep [(t)]	change the state of the system to next logged time [or current time + t]
lbreak (l)	place a breakpoint at line l
lstep [(l)]	increment to the next line [or step l lines]
lcont	move forward to the next breakpoint
lstatus	list all breakpoints
lclear [(l)]	remove breakpoint [at line l]
isCoherent (x, y)	check if x is coherent with y
diff (x, y)	compare views x and y of a vector
alt (hc, tl)	produce a timeline by altering tl using hypothetical change hc
getTime	get current debugger time

Table 1. Basic commands provided by MDB. These commands allow the user to (1) navigate the trace temporally, (2) navigate the trace logically, (3) compare macrovectors, and (4) make hypothetical changes to the code.

are executing the same logical operation, (i.e., the same line of code.) MDB provides three commands for generating and navigating through logical views: `lbreak`, `lcont`, and `lstep`. These commands are used to debug logical errors such as Bug 1 from Section 2.3, where the system fails to detect moving objects because of a typo on line 8. The user wants to first inspect the sensor values stored in `magVals` and so places a breakpoint on line 8 using the `lbreak` command:

```
>> lbreak(8)
Breakpoint set at line 8
```

The user then executes the `lcont` command, which advances the state of the application on all nodes until just before they execute the operation on line 8 for the first time:

```
>> lcont
At Line 8
```

The programmer views the value of the `neighborMag` macrovector by simply typing the variable name:

```
>> neighborMag
neighborMag =
   (1,1)    540
   (1,4)    505
   (1,7)    523
   ...
```

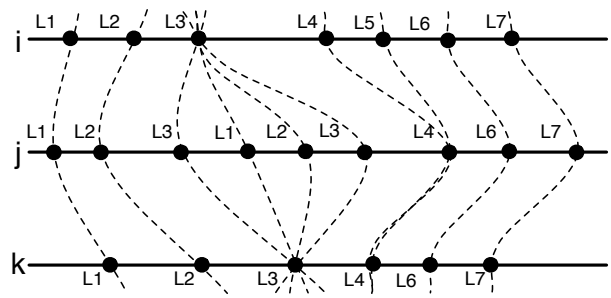
Since node 1 has at least three neighbors with sensor readings above `THRESH`, the programmer expects it to be in an *active* neighborhood. The programmer progresses past line 8 with the `lstep` command:

```
>> lstep
At Line 9
```

and views the value of the `active` variable:

```
>> active
```

This action reveals that `active` is an empty vector, even though node 1 has at least three active neighbors. At this point, the programmer inspects line 8 of the macroprogram and finds that the logic of the program was specified incorrectly: line 8 compares `THRESH` to the `magVals` variable instead of the `neighborMag` variable.




```
1   while( magVals < 10 )
2     magVals = magSensor.sense()
3   end
4   if magVals > 3000
5     magSensor.calibrate()
6   end
7   Display(magVals)
```

Figure 3. Logically synchronous views (shown by dashed lines) depict nodes i, j, and k progressing through the code in unison, even though node j executes the loop twice and only node i executes line 5.

The key challenge to providing a logically synchronous view of macroprogram execution is that the nodes may take different paths through the macroprogram, but the system must still allow the user to step sequentially through the macroprogram. MDB allows this by using an intuitive forward ordering of program statements: when the user executes the `lstep` command, MDB only advances those nodes for which the next instruction has the lowest line number of all nodes' next instructions. More formally, we define $\mathbf{l}$ to be the vector of line numbers l_i for each node i in the current logically synchronous view, and define $\mathbf{n}$ to be the vector of the next line numbers n_i that each node i executed and logged immediately after line l_i. When the user executes the `lstep` command, the next logically synchronous view will be the vector $\mathbf{l}'$ of line numbers l'_i, which are defined as

$$l'_i = \begin{cases} n_i & \text{if } n_i = min(\mathbf{n}) \\ l_i & \text{otherwise} \end{cases}$$

Figure 3 illustrates the effect of intuitive linearization, where the solid lines show the sequence of statements that each node executes in the example program, and the dashed lines illustrate the ten logically synchronous views that MDB would create as a user executes the `lstep` command. In the first three views, all nodes progress through lines 1, 2, and 3 simultaneously. Node j iterates through the loop twice, executing lines 1, 2, and 3 again. In the logically synchronous views, nodes i and k remain at line 3 until node j exits the loop, at which point all nodes progress to line 4 together, as illustrated by the dashed lines. Similarly, nodes j and k remain at line 4 in the logically synchronous views until node i finishes executing line 5, when the views show all three nodes progressing to line 6 together. Thus, logically

synchronous views depict all nodes progressing through the code in unison, even though they may take different paths through the code or execute the same lines of code at different times.

Logically synchronous views provide a convenient abstraction for navigating back and forth through the logic of a macroprogram, but they are limited in that the views of distributed state may include values that correspond to different points in time. In WENs, the user may also want to see distributed state at the specific time that a physical event took place in the network. For this reason, MDB also provides temporally synchronous views, described in the next section.

3.2 Temporally Synchronous Views

Temporally synchronous views allow the user to inspect and analyze the distributed state of the system at a specific time. MDB provides two commands for moving forward and backward in time: `tjump` jumps to a specific time, and `tstep` moves forward or backward by a given duration. These commands can be used to debug configuration errors such as Bug 2, described in Section 2.3, in which the mobile objects move by the nodes much more quickly than the rate at which they sample from the sensors. In our experiment, the programmer observes an object moving through the network approximately 0.4 seconds after execution started, but the object is not detected by the network. The programmer jumps to this time using the `tjump` command:

```
>> tjump(400000)
At 400000 microseconds
```

The programmer then inspects the `neighborMag` macrovector:

```
>> neighborMag
neighborMag =
   (1,1)    508
   (1,4)    125
   (1,7)    207
   ...
```

The user can see that only one node has a sensor value above `THRESH` at that time time. Then the user steps forward in time by entering the `tstep` command:

```
>> tstep
At 404129 microseconds
```

When issued with no parameter, `tstep` continually steps forward to the next log entry. The programmer notices that node 1's neighboring nodes do not read from their sensors until long after the mobile object has left the vicinity, indicating that the sampling frequency is not high enough. This example illustrates how temporally synchronous views allow the user to view distributed state at the actual time a bug manifestation was observed. This is particularly useful in WENs, where bug manifestations may correspond to physical events that do not correspond to the logic of the macrocode.

Temporally synchronous views allow the user to navigate through the execution trace of a WEN, but are limited by the fact that the user must be able to specify the exact times of interest. Identifying the time that an interesting event occurred to within milliseconds or microseconds can be a challenge, especially when the timestamps do not necessarily correspond to the values of any given wall clock. For this reason, MDB also provides capabilities for *historical search*, as described in the next section.

3.3 Historical Search

Historical search allows the user to inspect and analyze the historical sequence of distributed system states without manually stepping forward and backward through the code. Like most source-level debuggers, the temporally synchronous and logically synchronous views only show the user one state of the system at a time. This requires the user to step forward and backward through execution while trying to remember and correlate values from different states to find the cause of a fault. Historical search allows the user to operate over the state of the system at all times at once, eliminating the need to step forward and backward in time.

MDB allows the user to access the history of a macrovector by adding a new dimension to it which represents time. For example, in the object tracking application in Figure 1, `magVals` is a single-dimension macrovector that is indexed by node ID. The value of `magVals` at node 5 and time 1000 can be accessed with the command:

```
>> magVals(5, 1000)
ans =
   17316
```

Macrovectors conform to standard Matlab syntax even with the additional time dimension, and standard Matlab operators can be applied to them. This produces a powerful programmatic debugging interface that can be used to search for the manifestation of bugs. Historical search can be used to find timestamps of all instances of Bug 3 in which more than one leader was detected, using a single command:

```
>> find(numel(leaders(:, :))  > 1)
ans =
   17317316
   39824496
```

where `numel` is a standard Matlab operation that returns the number of elements in a vector. In this scenario, the developer finds that the IDs of more than one node were stored in the `leaders` macrovector at two times during program execution. To identify the cause of the bug, the programmer then jumps to one of these times using the command:

```
>> tjump(17317316)
At 17317316 microseconds
```

MDB's historical search functionality allows the user to exploit Matlab's rich plotting and analysis tools. For example, the command

```
>> plot(magVals(5,:))
```

produces the graph shown in Figure 4 that contains all sensors values read by node 5. Historical search provides a convenient mechanism for searching through and analyzing the historical sequence of distributed system state.

3.4 Hypothetical Changes

MDB allows the user to observe how *hypothetical changes* to a macroprogram would affect distributed state, without redeploying and executing the new code. This functionality is useful for testing whether a particular change will fix all occurrences of a bug that appeared in a given execution trace. As a proof-of-concept, MDB currently provides four hypothetical changes that highlight the effects of adding

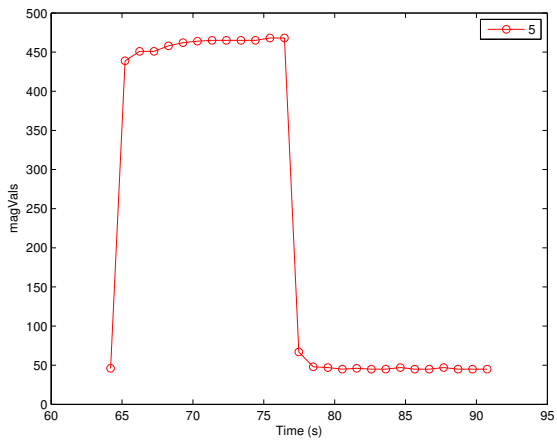


Figure 4. MDB's views of distributed state can be combined with Matlab's rich plotting and analysis tools. This figure is the result of a single debugging command: `plot(magVals(5, :))`

process and data synchronization primitives to a macroprogram: (i) a hypothetical barrier (ii) hypothetical cache coherence (iii) a hypothetical time delay, and (iv) hypothetical cache expiration.

The original execution trace that was collected during program execution is called the *base timeline* and is denoted *bt*. Hypothetical changes are applied to the base timeline to generate an *alternative timeline* denoted *at*. For example, the programmer could make a hypothetical change to a program by setting `magVals = 0` at line 7 by executing the `alt` command:

```
>> at = alt('magVals = 0', 7, bt)
```

This command would produce `at` by modifying `bt` such that `magVals` is always set to 0 at line 7. The value of a macrovector in this alternative timeline can be viewed by passing a handle to the timeline as an optional last parameter when indexing that macrovector. For example, a user can view node 7's value of `magVals` at time 10000 in the alternative timeline by executing the commands:

```
>> magVals(7, 10000, at)
```

3.4.1 Hypothetical Barrier

A *barrier* is a point in the source code that all nodes must reach before any node can proceed. The *hypothetical barrier* illustrates how the distributed state of the system would change if a barrier were placed at a particular line of macrocode. For example, the user can use hypothetical barriers to test whether a barrier at line 12 in the example program from Figure 1 would fix Bug 3 from Section 2.3. To do so, the programmer first creates an alternative timeline with a hypothetical barrier at line 12 using the command:

```
>> hb = alt('barrier()', 12, bt)
```

The user then checks if multiple leaders still arise with a barrier by apply the same command that was used in Section 3.3 to the new timeline:

```
>> find(numel(leaders(:, :, hb)) > 1)
```

Because this command no longer returns any timestamps, the user concludes that applying a barrier to line 12 would fix Bug 3. Thus, this hypothetical change allows the programmer to test whether this code modification, with respect to the given trace, fixes all occurrences of this bug, some occurrences, or no occurrences, without the need to redeploy, execute, and test the code.

We illustrate the semantics of a hypothetical barrier using the example execution timelines shown in Figure 5(a). MDB first creates a logically synchronous view at line 12, as illustrated by the dashed line. If the nodes had implemented a barrier, the state of the system would be identical to the state represented by this logically synchronous view, except that additional cache update messages would have been received. Specifically, any node *a* that waited at the barrier would have received all messages from another node *b* that were sent before *b* reached line 12 and that were received before all nodes reached line 12. Thus, MDB creates the hypothetical barrier by updating the state of the logically synchronous view at line 12 with all such cache update messages. This hypothetical change to the order in which cache update messages are received is illustrated by a dashed arrow in Figure 5(a) to illustrate the difference between the messages that were received in the real execution trace *bt*, and the messages that were received in the alternate timeline *at*. The distributed state resulting from this message re-ordering is the view of a hypothetical barrier applied to line 12.

3.4.2 Hypothetical Time Delay

Barriers are expensive operations and are not always desirable in WENs due to their high message cost. A cheap alternative is to have all nodes wait for any cache updates to arrive for only a fixed period of time before continuing execution. If execution on all nodes is already synchronized or is otherwise known to be unsynchronized by a bounded duration, this approach may provide sufficient data synchronization guarantees. The *hypothetical time delay* produces the distributed state that would have been produced if a time delay of Δt were inserted at a particular point in the macrocode. The programmer could create a hypothetical time delay using the `deltat` command:

```
>> dt = alt('deltat(10)',12, bt)
```

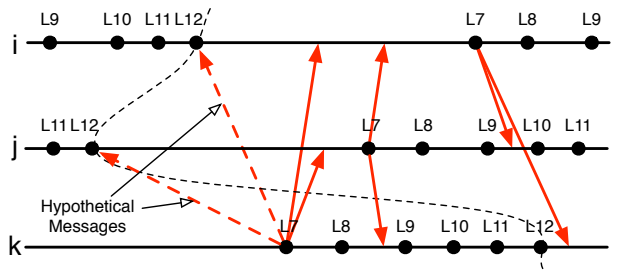
`deltat` takes a parameter to indicates the delay Δt in milliseconds. To create the hypothetical time delay, MDB first creates a logically synchronous view on line 12. Then, the distributed state is updated with any cache update messages received by a node withing the Δt of the time it reached line 12. Figure 5(b) illustrates that a hypothetical time delay applied at line 12 would cause the message sent from line 7 of node *k* to update the state of node *i* after executing line 12.

3.4.3 Hypothetical Cache Coherence

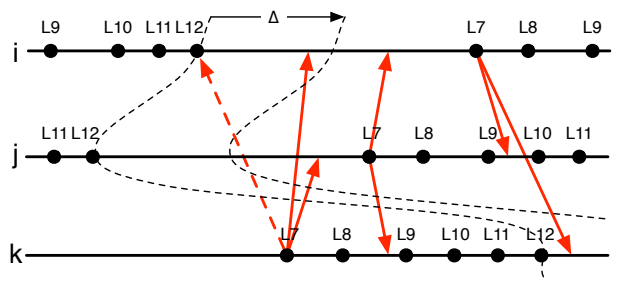
Hypothetical cache coherence shows the hypothetical distributed state if all caches were coherent at a given time. This view can be produced with the command:

```
>> cv = alt('coherent()', 12 bt)
```

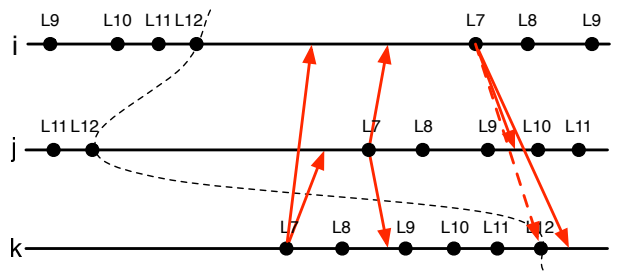
To create hypothetical cache coherence, MDB rearranges any cache updates that were sent before, but received after the line of code specified. Figure 5(c) illustrates how message reception is altered by the command above: the



(a) Hypothetical Barrier



(b) Hypothetical Time Delay



(c) Hypothetical Cache Coherence

Figure 5. MDB emulates the effect of hypothetical code changes by reordering messages appropriately. Solid arrows illustrate message reception in an actual execution trace. Dashed arrows illustrate how these messages are re-ordered to emulate the effect of hypothetical changes to the code.

message from node i is received at node k on line 12, because it was sent before k finished executing line 12. Perfect cache coherence is expensive to implement because it requires two-phase locks to be acquired on all cached versions of a variable before any new values can be written to it. If hypothetical cache coherence makes a bug manifestation disappear, the user may attempt to use end-to-end reliability for cache updates, low-latency communication protocols, or other mechanisms that will improve, but not guarantee, cache coherence.

3.4.4 Hypothetical Cache Expiration

Cache expiration is the process of invalidating a cached copy of a value once it becomes too old. This is cheaper to implement than perfect cache coherence, but it also provides few correctness guarantees because the original value may change before the cached copies of the old value expire. Furthermore, when nodes run asynchronously and aperiodically in a network, it can be difficult to decide when to expire cache entries. Expiring too quickly can result in insufficient data, but expiring too slowly can result in the use of stale values. *Hypothetical cache expiration* shows the hypothetical state that would result if cache expiration were used at a given line of code, with a particular expiration time. This view helps the user evaluate the effect of different expiration times and can be created using a command such as:

```
>> ev = alt('expire(100000)', 12, bt)
```

The values in the view are produced by finding the last value written to each element of the macrovector in the time interval $[t - t_e, t]$ where t is the time the node executed line 12 in the current instance of the logically synchronous view and t_e is the expiration time, in this case 100000 μs. If an element has had no value written to it within that time interval, the resulting view has a *NaN* value for that element.

4 The MDB Implementation

In this section, we describe the implementation of MDB, including the collection of execution traces (Section 4.1), how MDB Lite reduces energy cost (Section 4.2), how timekeeping is performed on data traces (Section 4.3), how MDB's global views and historical searches are implemented using these data traces (Section 4.4), and how hypothetical changes are emulated by creating modified copies of these data traces (Section 4.5).

4.1 Creating Execution Traces

MDB is a post-mortem debugger, which means that it must collect execution traces at run-time that will be analyzed after execution is complete. Most post-mortem debuggers create *event traces*: they log all non-deterministic events such as interrupts, I/O, and messages. These event traces are sufficient to recreate the state of the system at any point of the original execution using *execution replay*. In contrast, MDB creates *data traces*: it logs all changes to the system state, including writes to variables and changes to the program counter. Event traces generally produce shorter logs than data traces because events are relatively rare while state changes occur with every line of code executed, especially for traditional distributed applications that are compute intensive but have relatively little I/O or network communication. In contrast, each line of a macroprogram may correspond to many machine instructions, I/O operations, and network events. Therefore, data traces are more efficient than event traces for logging macroprogram execution.

MDB log entries consist of the (macro)program counter, the variable location and value being written, a 48 *bit* microsecond-accurate timestamp, and the sequence number. The sequence number helps correlate variable write events with remote cache events of that same value, since message transmission and reception are not logged. After execution is complete, the logs are retrieved from the nodes using the collection tree protocol from the TinyOS source tree, although any data collection protocol can be used.

To minimize interference with the application, MacroLab logs data in two phases: first in RAM and then to external

flash. While the MacroLab application is executing, trace entries are stored in a circular RAM buffer, requiring only 105 machine instructions per log. Then, when the node has no more instructions to process, and right before it enters sleep mode, the RAM buffer is dumped to external flash, which takes approximately 50 *ms*. This approach reduces MDB's interference with the application by reducing contention for the CPU. One disadvantage of this approach, however, is that any log entries in the RAM buffer may be lost if the node crashes. All logging code required for MDB is automatically added by the MacroLab compiler when the debugging option is enabled.

The external flash is 1 *MB* in size, and when it is full the earliest trace entries are overwritten. One advantage of MDB's use of data traces is that old trace entries can be overwritten when the external flash is full. In contrast, entries in an event trace cannot be overwritten because all events from the beginning of execution are required for execution replay. Therefore, systems that collect event traces must use checkpointing techniques, which can introduce additional overhead [53].

4.2 MDB Lite

MDB Lite is a light-weight version of MDB that conserves energy by not writing log entries to the external flash, although all other logging code is included in the microprograms and is identical to MDB. Furthermore, MDB Lite emulates the process of writing to flash by using a hardware timer to turn off the appropriate interrupts for the appropriate period of time. The logging commands and flash emulation ensure that the timing and memory characteristics of a program are the same when executing MDB and MDB Lite. Thus, MDB Lite cannot be used for debugging, but it can be enabled during the deployment phase to reduce energy overhead while also eliminating the possibility of heisenbugs: a change in the manifestation of bugs when the debugger is enabled or disabled. The user can toggle between MDB and MDB Lite to enable or disable debugging.

4.3 Distributed Timekeeping

After a program executes and the logs are collected, MDB must ensure that the timestamps in the logs are *causally consistent*: any event e that causes event e' must have a smaller timestamp than e'. Distributed time keeping is a challenging problem because nodes do not share a common clock, and is the main distinguishing feature between debugging on a distributed system and debugging concurrent threads on a single machine or a shared memory multiprocessor (SMMP). MDB adopts a well-known approach that is sometimes called the *Lamport* algorithm: all logs are timestamped with a value from the node's microsecond counter, and clock skew is accounted for by enforcing that a receiver's local time when a message is received is greater than the sender's local time when the message was transmitted. Since message passing is typically the only form of interaction between nodes, this approach guarantees that any events on the sender have an earlier timestamp than events they might cause on the receiver. Any distributed timekeeping scheme that satisfies this property is said to support *Lamport time* [28]. Other timekeeping schemes such as vector clocks have more precise causal semantics [17, 38]. MDB's use of the Lamport algorithm occurs off-line after the logs are collected, which eliminates the need for any run-time overhead due to on-line time synchronization [16, 36]. One disadvantage of this approach is that all nodes must receive periodic messages from neighboring nodes. In a partitioned network where some nodes do not have radio connectivity with other nodes, the logs cannot be made causally consistent. This is the case for all distributed systems, and not a limitation specific to MDB.

Distributed timekeeping in WENs is complicated by the fact that nodes can also interact through their sensors and actuators: two nodes can sense the same stimulus, or a node's sensor reading can be affected by another node's actuator. This causal relationship is not explicit in the program, cannot be verified at run time, and cannot be enforced by the Lamport timekeeping algorithm. Such node interactions through events external to the synchronization messages is called *anomalous behavior* by Lamport, who suggested physical clocks to eliminate this limitation. We performed empirical studies of the CPU clocks on the Tmote Sky nodes and found an average clock skew of 139 $\mu s/s$, similar to the observations mentioned in RBS [16]. We also measured the inherent uncertainty of sensor readings on the same nodes to be about 20 *ms*, which is the average time required for the ADC to digitize the analog readings from the sensors, plus software overhead. Thus, a minimum message rate of about one message every 2 minutes is required to prevent clock skew from growing larger than the inherent uncertainty on sensor reading timestamps. The minimum message rate may even be much larger when measuring physical stimuli such as temperature or object mobility that change with periods much lower than 20 *ms*.

4.4 Generating Global Views

Once the execution traces are retrieved and synchronized, MDB can use them to create views of any system variable at any given time. To view the value of variable x at time t, MDB indexes the trace of the node on which variable x was logged and retrieves the last value written before time t. For example, to produce the value of node k's element of the `magVals` vector at time 1000, the system may retrieve the value that was written to `magVals` by node k at time 998. This basic functionality helps implement MDB's logically synchronous views, temporally synchronous views, and historical search.

Logically synchronous views are implemented using an `lline` vector which stores all the line numbers at which a breakpoint has been placed and an `ltime` vector which stores a time for each node to indicate MDB's location in its trace. When the user enters `lbreak` with a line number, the line number is appended to `lline`. When the user enters `lstep`, MDB identifies which nodes executed the next line of the macroprogram next and sets the value of `ltime` for those nodes to be the time when the node finished executing that line. When the user executes `lcont`, MDB executes `lstep` repeatedly until the macrocode line matches one of the values in `lline`.

Temporally synchronous views are implemented using a variable called `ttime` which stores the parameter passed to `tjump`. `ttime` is updated when the user issues the `tstep`

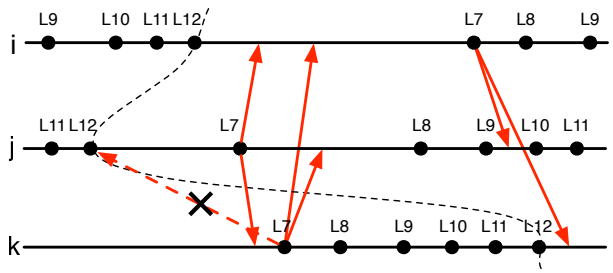


Figure 6. A hypothetical barrier cannot be applied in this scenario because the hypothetical advancement of the message from node k to node j would create a dependence cycle.

command. If the user views a macrovector after `ttime` has been set, the view of the macrovector is generated by searching through the trace of each node for the last state update to the macrovector at or before `ttime`. Historical search is implemented by searching through the execution traces for all writes to the macrovector specified by the historical search.

MDB uses a special *NaN* value to represent any value in a logically synchronous or temporally synchronous view that does not exist in the logs. This situation occurs in a temporally synchronous view, for example, when a node fails or stops executing and the log from that node does not contain values beyond a certain point in time. It may also occur in logically synchronous views, for example, if a node blocks on a variable indefinitely or dies and its log has fewer instances of a particular line number than the logs of other nodes.

4.5 Generating Hypothetical Changes

Hypothetical changes are made by modifying an initial timeline denoted `it` to generate an alternate timeline denoted `at`. This process involves two phases: (1) applying hypothetical changes to `it`, and (2) propagating the hypothetical changes into the future. MDB carries out the first stage of alternative timeline generation by copying all the values from `it` into a new timeline `at`. Then, it generates logically synchronous views at each instance of line l specified by the user and MDB modifies the cache update times based on the hypothetical change specified, as described in Section 3.4. For example, to implement the hypothetical barrier depicted in Figure 5(a), MDB would reorder the trace entries in the new timeline `at`, such that the message from line 7 of node k is received by nodes i and j when they reach the barrier. After the message reordering is complete, MDB propagates this change into the future by re-executing the instructions corresponding to every subsequent log entry in *at*, in the order of their timestamps. Thus, MDB re-executes the instructions logged in the original execution trace to propagate the state from the hypothetical change into all future states. After re-executing each line of code, MDB re-creates a new trace entry and adds it to the alternative timeline `at`. Thus, alternative timeline `at` should have exactly the same timestamps on all log entries as the original timeline `it`, but might have slightly different values.

```
1 every(uint16(1000))
2  lightVals = lightSensor.sense();
3  basedisplay(lightVals);
4 end
```

Figure 7. The macroprogram for Surge reads sensor values and displays them at the base station.

```
1 every(uint16(1000))
2  trigger = microphone.sense();
3  meanTrigger = mean(neighborTrigger, 2);
4  candidates = find(meanTrigger > THRESHOLD);
5  soundLog(candidates) = microphoneHF(candidates).
       sense();
6  baselog(soundLog);
7 end
```

Figure 8. The macroprogram for acoustic monitoring reads values from a microphone and reads from a higher-fidelity microphone on nodes whose neighborhoods detect high average noise levels.

When a node reads from a sensor, MDB retrieves the sensor reading that was collected in the original execution trace. MDB does not attempt to generate simulated sensor values based on a model of the stimulus. Therefore, creation of the alternative timeline cannot proceed once the control flow of execution diverges from the control flow observed in the original execution trace. Thus, hypothetical changes can only be propagated into the future insofar as they do not change control flow; any request for a view of the alternative timeline after control flow changes produces the *NaN* value, to indicate that the view cannot be generated. This limitation could be overcome by incorporating simulation and sensor models into MDB, but this is beyond the scope of this paper.

In addition to control flow changes, alternative timeline generation also fails in certain instances when the user specifies a hypothetical change that results in a *causality cycle*: an event e that is caused by event e' reordered to occur before e'. For example, in Figure 6, the hypothetical barrier applied to line 12 would cause message m from line 7 of node k to be received before message m' is sent at line 7 on node j. However, this produces a causality cycle because message m' is also received by node k before it sends message m. We call this scenario the *grandfather paradox*. MDB checks for the grandfather paradox during the creation of alternative timelines, and all subsequent values that are causally related to m or m' are assigned the *NaN* value.

5 Evaluation

Our system evaluation is composed of two parts. First, we show that data traces are more effecient for macrodebugging than the traditional approach of creating event traces. Then, we evaluate the RAM, flash, energy, and CPU cycle overhead of MDB. These evaluations were carried out on a testbed with 21 TMote Sky nodes with photoresistor sensors. Since all the nodes in the testbed are within one hop of each other, we artificially limit their communication ranges by modifying the packet reception module of the CC2420 radio. We

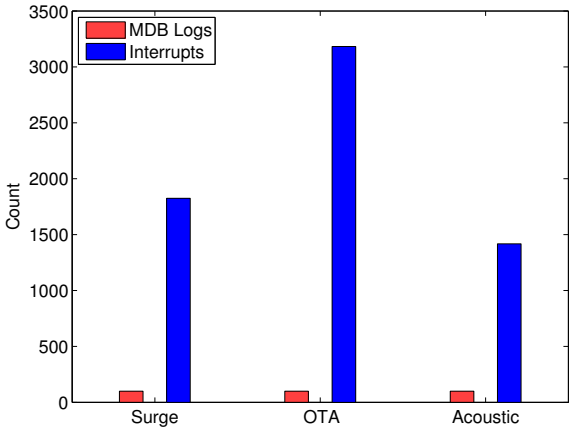


Figure 9. Number of interrupts generated per 100 data states written to flash. WEN applications produce 14–32 times more interrupts and message events than macroprogram state updates.

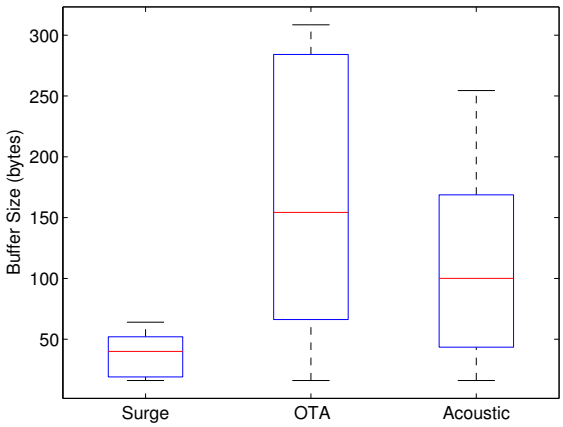


Figure 10. Log data is stored temporarily in RAM before being written to flash. In our experiments, no node needed more than 304 *B* of RAM. The box plot shows the minimum, lower quartile, mean, upper quartile, and maximum values.

restrict nodes to communicate with neighbors next to them vertically, horizontally, and on the two diagonals. To collect additional data that could not be obtained from the testbed, we used the node-level Cooja network simulator [42] which makes use of the MSPSim [14] TMote Sky simulator. The microcode executing on the nodes in the Cooja simulator is exactly the same microcode that executes on the real Tmote Sky hardware.

We evaluated MDB with three macroprograms. The first application is OTA, described earlier (Figure 1), for which we emulated the movement of objects using the light sensors by moving a white circle on a black background that was projected from a laptop onto the testbed. The intensity of the circle decreases radially outward to emulate the effect of varying sensor readings by nodes that detect the object. The second application is Surge (Figure 7), which is a simple data collection application that periodically samples from a sensor and displays the readings at a base station. The third is an acoustic monitoring application (Figure 8), which first senses from a microphone sensor, and depending on the number of neighbors that also heard a sufficiently loud sound, samples from a second high-fidelity microphone, stores the values, and reports them to the base station. The sensor values for both of these applications are generated using the photoresistors on the nodes.

5.1 Data vs. Event Logging

We evaluate the cost of creating data traces by counting the number of logging statements required to record all variable writes and program counter changes for a single run of each of the three macroprograms. We compare this count to the number of interrupts that would need to be logged to create an event trace of the same program execution. Since we could not count the interrupts generated on real hardware without changing the timing characteristics of the program, we obtained these values by analyzing the programs as they executed on the Cooja network simulator. Figure 9 shows that, for these applications, the number of hardware interrupts is 14–32 times larger than the number of updates to macroprogram state. These results clearly show that MDB's approach of collecting data traces is much more efficient for macroprograms than the traditional approach of collecting event traces.

Application	Flash (*Bps*)	Wraparound (*hr*)
Surge	31	9
Accoustic	187	2
OTA	288	1

Table 2. Flash memory consumption is low enough to debug hours of execution. The buffer is circular, so it can always store data to debug the last few hours of execution.

5.2 RAM and Flash Overhead

Memory is not typically a concern for traditional debuggers that execute on PCs, but WENs are composed of highly resource-constrained devices and efficient memory usage is essential to making MDB practical in this domain. We measured the amount of memory required by MDB by instrumenting the RAM buffer portion of the logger and tracking the maximum difference between the head and tail pointers of the circular buffer. Figure 10 contain box plots that show the minimum, maximum, mean, and the lower and upper quartiles of RAM consumption for each of the three test applications executing on our 21 node testbed. This data reveals that MDB has modest RAM requirements, and needs to store a maximum of 304 *B* of data, while the Tmote Sky nodes have about 10 *KB* of RAM available.

We also measured the amount of Flash memory required by MDB to store the complete data traces for each of the three applications. Table 2 shows that the applications store less than 300 *Bps* to the flash. At these rates, simple applications such as Surge can collect logs for several hours before

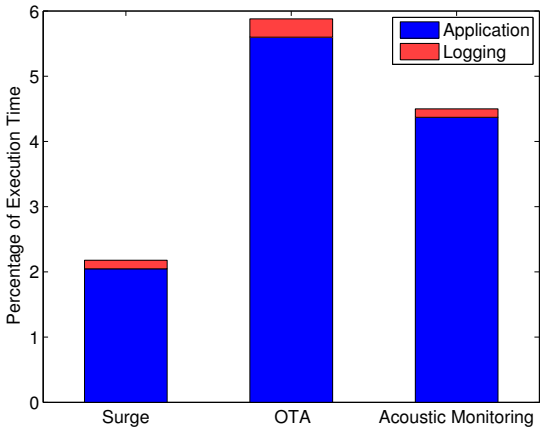


Figure 11. MacroLab context constitutes less than 6% of all code running on the nodes. Logging code constitutes less than 0.5%.

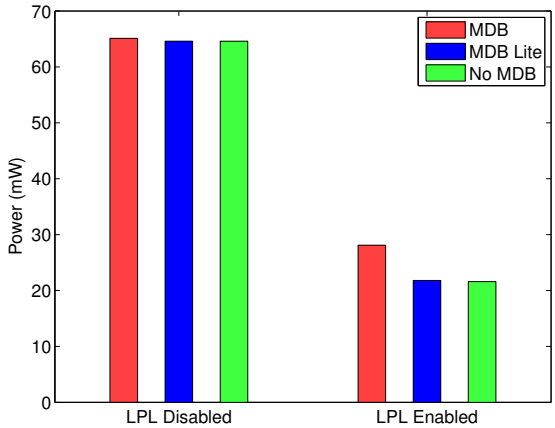


Figure 12. Without low-power listening enabled, applications consume 30% more energy with MDB enabled, but only 0.9% more energy with MDB Lite.

exhausting the 1 *MB* of external flash available on the Tmote Sky. More complicated applications such as OTA and acoustic monitoring may exhaust the flash after about one and a half hours. Thus, bugs in these applications must be detected within one and a half hours in order to debug them before the log entries are overwritten.

5.3 CPU Overhead

We evaluate the effect of MDB on execution speed by counting the logging instructions executed during a particular run of the three applications. Since it was difficult to collect this information from the actual nodes without substantially modifying the timing characteristics of the application, this data was collected by running the application on the Cooja simulator. Since Surge has no interaction between nodes, we simulated it using 1 node. Acoustic monitoring used 5 nodes and OTA used 10 nodes. Figure 11 shows the number of MacroLab application instructions and MDB logging instructions that execute, as a percentage of the total execution time. The values in Figure 11 are averages over all the nodes simulated in Cooja. As Figure 11 shows, the MacroLab application code executes for between 2% and 6% of the total execution time and the logging code executes for less than 0.5% of the total execution time.

5.4 Energy Consumption

We evaluate MDB's energy consumption by executing the OTA application, with the senor being sampled every 10 seconds, on the 21 node testbed and measuring its average current draw over a period of 100 seconds. We executed this experiment with MDB, MDB-lite, and without MDB. We repeated the experiment both with and without low-power listening (LPL) [43]. Figure 12 shows that an application consumes 30% more energy with MDB, when LPL is enabled with a sleep interval of 1 second. This energy overhead is due to the process of saving logs to the external flash chip. With MDB Lite, this overhead is reduced to only 0.2 *mW* or 0.9%. This is because MDB Lite does not write to flash, and the timer-based implementation of MDB Lite allows the node to sleep when possible. The overhead of MDB and MDB Lite when LPL is disabled is only 0.5 *mW* because the node does not go into sleep mode in any of these implementations.

6 Related Work

A plethora of debuggers and debugging abstractions have been created for distributed computing. For example, source-level debuggers allow the user to halt and step through execution on each individual machine [5,7,31]. Distributed breakpoints [18, 19, 40] and snapshots [8] allow the user to stop all nodes in a particular consistent state across the network. Some debuggers can be programmed so that they interact with and analyze the program as it executes without user interaction [21, 39, 52]. Others provide logical or SQL-like query languages for searching, inspecting, and analyzing state or execution flow [13, 30, 44]. Each of these systems provides a high-level abstraction for debugging, but are not designed to work with systems that provide a high-level abstraction for programming; these debugging systems allow the user to inspect node-local actions and messaging protocols which are abstracted away by macroprogramming systems.

Many debugging systems have been implemented in ways that accommodate the strict resource requirements of WENs. For example, Marionette [51] and Clairvoyant [54] provide access to source-level symbols, while Declarative Tracepoints [6] and Wringer [46] provide a programmable interface for describing debugging operations that can be downloaded and executed on the nodes at run-time. DustMiner [25] and LiveNet [9] eavesdrop on messages in the network for visibility into network operations without consuming node resources. EnviroLog [32] logs some nondeterministic events to produce efficient in-network execution replay. Sympathy collects a small amount of data to identify the cause of network failures [45]. However, all of these systems require the inspection of node-local actions or message traces. The primary contribution of MDB is to

avoid this by debugging a WEN using high-level distributed abstractions provided by macroprogramming systems.

MDB is a *static* debugger; it collects execution traces and the debugging process is off-line or *post-mortem*. This approach has long been used for distributed systems because on-line debugging can change the timing characteristics of execution, making it difficult to analyze message races. Most off-line distributed debuggers use *execution replay* to recreate the system state at a specific point in the original execution, in which the debugging runtime records only high-level events and re-executes (or simulates) the intermittent code where necessary to recreate the full system state [29, 53]. Execution replay incurs a debug-time cost of recreating the state, and some systems must use checkpointing to limit the amount of replay required [53].

In contrast, MDB avoids most of the cost of replay by logging all macrovector writes, creating a data trace rather than an event trace. This approach has been known to be possible in traditional distributed systems, but not practical. Mall notes that data traces are impractically expensive but shows that the cost can be reduced to some extent with a technique called *inverse statement analysis* [34]. Maruyama et al. state that *data replay* is still uncommon due to its high cost but shows that it is becoming possible with the increasing capacity and decreasing cost of storage [37]. MDB is unique in that it uses data traces because they are more efficient than event traces for its application domain, as shown in Section 5.1.

Several debugging abstractions have been created for use during execution replay. For example, TraceView [35] allows the user to visualize event traces. Garg et al. detect weak unstable predicates using traces [19]. Kilgore, et al. replay and change the order of messages to detect *race messages*, where the order in which messages are received change the results of the distributed computation [26]. Finally, D3 [11] allows the user to write a query in a high-level declarative language called NDLog to create a data model of the logs. D3 then applies the query to the incoming data that conforms to the model. Many of these abstractions are similar to MDB's ability to visualize data, find logical conditions in the network, reorder messages, or analyze the history of variables. One key difference is that MDB's abstractions were specifically designed for use with data traces, whereas the abstractions above were designed for use with event traces. Another difference is that MDB provides these abstractions on high-level, abstract distributed data structures that are specified in a macroprogram, while the existing systems are applied using symbols from the programs of each individual node.

7 Conclusions

Macroprogramming systems make programming a WEN easier by providing high-level distributed abstractions such as database tables, logical facts, or data streams so that the user does not need to build a mental model of the individual nodes and their interactions. However, to the best of our knowledge, no macroprogramming systems have debugging support, which is a crucial link in the development chain as a macroprogram progresses from the drawing board to real deployment. We present MDB, the first system that allows the user to inspect and analyze the execution of a WEN using the high-level abstractions provided by a macroprogramming system. This process that we call *macrodebugging* simplifies the debugging process and eliminates the need to analyze traces of low-level events and message passing algorithms. We show that macrodebugging is not only easier, but also more efficient than the debugging of node-level programs.

The MDB macrodebugger is built for the MacroLab macroprogramming system [23], but the underlying principles can be applied to other macroprogramming systems. The collection of data traces can be applied to any system that has a high-level abstraction for which the overhead of logging a single high-level operation is small compared to the number of low-level instructions required to execute that operation. This property holds for most macroprogramming systems. The source-level debugging interface can be applied to any system that uses a sequential imperative language and has a clear mapping between macrocode and microcode, such as Kairos [22], Plaeiades [27], and Marionette [51]. Other systems that use functional [41] or declarative abstractions [33] must present information from data traces using a different interface. The four hypothetical changes presented illustrate the affects of adding data synchronization to a macroprogram, and are only useful to systems like MacroLab and Hood [50] that make heavy use of data caching. However, the general concept of creating hypothetical changes to illustrate how theoretical changes to a program or execution state *would* affect global state could be applied to aspects of other systems besides data synchronization.

The current implementation of MDB provides *post-mortem* debugging, which means that it collects data about the system at run time and allows the user to inspect program execution after the logs are retrieved. At least some of the underlying concepts, however, could be applied to on-line debugging. For example, the process of logging data instead of events to recreate system state would reduce the amount of data that needs to be collected during on-line debugging, in the same way that it reduces data collection requirements for off-line debugging. Similarly, the logically-synchronous and temporally-synchronous source-level debugging interface could also be used for on-line debugging, and could even be combined with off-line data trace analysis to provide both forward and backward stepping. Hypothetical changes would not be as useful for on-line debugging as they are for off-line debugging, because the user could test changes to the system by simply changing the current state of the system before allowing execution to proceed.

Acknowledgment

This material is based upon work supported by the National Science Foundation under Grant No. 0845761.

8 References

[1] Matlab - the language of technical computing. `http://www.mathworks.com/products/matlab/`.

[2] A. Awan, S. Jagannathan, and A. Grama. Macroprogramming heterogeneous sensor networks using cosmos. *SIGOPS*, 2007.

[3] J. Bachrach and J. Beal. Programming a sensor network as an amorphous medium. *DCOSS*, 2006.

[4] S. Bhatti, J. Carlson, H. Dai, J. Deng, J. Rose, A. Sheth, B. Shucker, C. Gruenwald, A. Torgerson, and R. Han. Mantis os: An embedded multithreaded operating system for wireless micro sensor platforms. *MONET*, 2005.

[5] S. Browne, J. Dongarra, and A. Trefethen. Numerical libraries and tools for scalable parallel cluster computing. *Int. J. High Perform. Comput. Appl.*, 15:175–180, 2001.

[6] Q. Cao, T. Abdelzaher, J. Stankovic, K. Whitehouse, and L. Luo. Declarative tracepoints: a programmable and application independent debugging system for wireless sensor networks. In *SenSys*, 2008.

[7] O. Center. XMPI-A Run/Debug GUI for MPI. Technical report, Ohio Supercomputer Center, 1997.

[8] K. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3:63–75, 1985.

[9] B.-R. Chen, G. Peterson, G. Mainland, and M. Welsh. Livenet: Using passive monitoring to reconstruct sensor network dynamics. In *DCOSS*, 2008.

[10] D. Chu, L. Popa, A. Tavakoli, J. M. Hellerstein, P. Levis, S. Shenker, and I. Stoica. The design and implementation of a declarative sensor network system. In *SenSys*, 2007.

[11] B.-G. Chun, K. Chen, G. Lee, R. H. Katz, and S. Shenker. D3: Declarative distributed debugging. Technical report, UC, Berkeley, 2008.

[12] E. W. Dijkstra. Cooperating sequential processes, technical report ewd-123. Technical report, 1965.

[13] M. Ducassé. Coca: An automated debugger for c. In *ICSE*, 1999.

[14] A. Dunkels, J. Eriksson, N. Finne, F. Osterlind, and T. Voigt. Mspsim - an extensible simulator for msp430-equipped sensor boards. In *EWSN*, 2007.

[15] A. Dunkels, B. Gronvall, and T. Voigt. Contiki-a lightweight and flexible operating system for tiny networked sensors. In *LCN*, 2004.

[16] J. Elson, L. Girod, and D. Estrin. Fine-grained network time synchronization using reference broadcasts. In *OSDI*, 2002.

[17] C. Fidge. Logical time in distributed computing systems. *Computer*, 24:28–33, 1991.

[18] J. Fowler and W. Zwaenepoel. Causal distributed breakpoints. In *ICDCS*, 1990.

[19] V. Garg and B. Waldecker. Detection of weak unstable predicates in distributed programs. *IEEE TPDS*, 5:299–307, 1994.

[20] 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 *PLDI*, 2003.

[21] M. Golan and D. Hanson. Duel-a very high-level debugging language. In *USENIXW*, 1993.

[22] R. Gummadi, O. Gnawali, and R. Govindan. Macro-programming wireless sensor networks using Kairos. In *DCOSS*, 2005.

[23] T. Hnat, T. Sookoor, P. Hooimeijer, W. Weimer, and K. Whitehouse. Macrolab: a vector-based macroprogramming framework for cyber-physical systems. In *SenSys*, 2008.

[24] H. F. Jordan. A special purpose architecture for finite element analysis. In *ICPP*, 1978.

[25] M. Khan, H. Le, H. Ahmadi, T. Abdelzaher, and J. Han. Dustminer: troubleshooting interactive complexity bugs in sensor networks. In *SenSys*, 2008.

[26] R. Kilgore and C. Chase. Re-execution of distributed programs to detect bugs hidden by racing messages. In *HICSS-30*, 1997.

[27] N. Kothari, R. Gummadi, T. Millstein, and R. Govindan. Reliable and efficient programming abstractions for wireless sensor networks. In *PLDI*, 2007.

[28] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21:558–565, 1978.

[29] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Trans. Comput.*, 36:471–482, 1987.

[30] R. Lencevicius, U. Hölzle, and A. Singh. Dynamic query-based debugging of object-oriented programs. *ASE*, 10:39–74, 2003.

[31] M. Linton. The evolution of dbx. In *USENIX*, 1990.

[32] L. Luo, T. He, G. Zhou, L. Gu, T. F. Abdelzaher, and J. A. Stankovic. Achieving repeatability of asynchronous events in wireless sensor networks with envirolog. In *Infocom*, 2006.

[33] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. Tinydb: an acquisitional query processing system for sensor networks. *ACM Trans. Database Syst.*, 30:122–173, 2005.

[34] R. Mall. A novel "bi-directional execution" approach to debugging distributed programs. In *HiPC*, 1999.

[35] A. Malony, D. Hammerslag, and D. Jablonowski. Traceview: A trace visualization tool. In *Proceedings of the First International ACPC Conference on Parallel Computation*, 1991.

[36] M. Maroti, B. Kusy, G. Simon, and A. Ledeczi. The flooding time synchronization protocol. In *SenSys*, 2004.

[37] M. Maruyama, M. Maruyama, T. Tsumara, and H. Nakashima. Parallel program debugging based on data-replay. *IPSJ*, 46:214–224, 2005.

[38] F. Mattern. Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms*, 1989.

[39] P. Maybee. Ned: The network extensible debugger. In *USENIX*, 1992.

[40] B. Miller and J. Choi. Breakpoints and halting in distributed programs. In *ICDCS*, 1988.

[41] R. Newton, G. Morrisett, and M. Welsh. The regiment macroprogramming system. In *IPSN*. ACM Press New York, NY, USA, 2007.

[42] F. Osterlind, A. Dunkels, J. Eriksson, N. Finne, and T. Voigt. Cross-level sensor network simulation with cooja. In *LCN*, 2006.

[43] J. Polastre, J. Hill, and D. Culler. Versatile low power media access for wireless sensor networks. In *SenSys*, 2004.

[44] M. Powell and M. Linton. A database model of debugging. *SIGSOFT Softw. Eng. Notes*, 8:67–70, 1983.

[45] N. Ramanathan, K. Chang, R. Kapur, L. Girod, E. Kohler, and D. Estrin. Sympathy for the sensor network debugger. In *SenSys*, 2005.

[46] A. Tavakoli, D. Culler, P. Levis, and S. Shenker. The case for predicate-oriented debugging of sensornets. In *HotEmNets*, 2008.

[47] The GDB developers. GDB: the GNU project debugger. `http://sourceware.org/gdb`.

[48] M. Welsh and G. Mainland. Programming sensor networks using abstract regions. In *NSDI*, 2004.

[49] K. Whitehouse, J. Liu, and F. Zhao. Semantic streams: a framework for composable inference over sensor data. In *EWSN*, 2006.

[50] K. Whitehouse, C. Sharp, E. Brewer, and D. Culler. Hood: a neighborhood abstraction for sensor networks. In *MobiSys*, 2004.

[51] 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 *IPSN*, 2006.

[52] P. Winterbottom. Acid: a debugger built from a language. In *USENIXW*, 1994.

[53] L. Wittie. The bugnet distributed debugging system. In *EW 2*, 1986.

[54] J. Yang, M. L. Soffa, L. Selavo, and K. Whitehouse. Clairvoyant: a comprehensive source-level debugger for wireless sensor networks. In *SenSys*, 2007.