The Design of MDB: a Macrodebugger for Wireless Embedded Networks

A Dissertation

Presented to

the faculty of the School of Engineering and Applied Science

University of Virginia

In Partial Fulfillment

of the requirements for the Degree

Master of Science

Computer Science

by

Tamim I. Sookoor

December 2009

Approvals

This dissertation is submitted in partial fulfillment of the requirements for the degree of

Master of Science

Computer Science

Tamim I. Sookoor

Approved:

Kamin Whitehouse (Advisor)

John A. Stankovic (Chair)

Westley R. Weimer

Accepted by the School of Engineering and Applied Science:

James H. Aylor (Dean)

December 2009

Abstract

Creating and debugging programs for 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.

Contents

List of Figures

List of Tables

Chapter 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 [38], geographic regions [60], and logical rules [11]. 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 [13]. 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.

MDB fills an important gap in the macroprogramming tool chain, thereby making it easier to debug and deploy macroprograms. Furthermore, macrodebugging is both easier and has lower overhead than node-level debugging: the ability to view global, distributed state using the high-level macroprogramming abstractions eliminates the need to trace through the execution details of multiple nodes, such as message passing and hardware interrupts. Thus, MDB produces a marriage between debugging and macroprogramming that is mutually beneficial to both fields.

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* [27], 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 8. 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

MDB is also a contribution in its own right: we find that debugging macroprograms is both easier and more efficient than debugging node-level programs. The goal of MDB is to make macroprogramming a feasible approach by providing a more complete tool chain. We find that macrodebugging is easier that low-level debugging for the same reason that macroprogramming is easier: high-level abstractions. We also find that macrodebugging is more efficient than low-level debugging: we need to collect less information about system execution to provide visibility into execution.

Chapter 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 [23,

4, 16].

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 [38], logical facts [11], or data streams [61]. 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 [1, 2, 26, 27, 31, 60], 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 [44]. 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`,

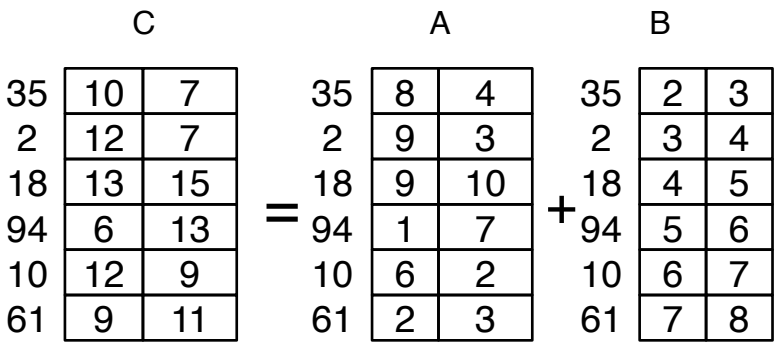


Figure 2.1: Two $n \times 2$ macrovectors A and B can be added and stored into a third macrovector C. The values on the left of each vector indicate which node ID the cells are associated with.

or `find`[1], and the system compiles these down into local actions for each node that cooperatively execute the vector operations [27]. Examples of operations on macrovectors include:

```
maxLight = max( light )
```

which returns the maximum light value in the network, or:

```
hotLight = light( find( temp > 100 ) )
```

which returns the light values on nodes where the `temp` value is higher than 100. If these vectors were tables in TinyDB, this would be similar to posing the SQL query

```
SELECT light WHERE temp > 100
```

Elements associated with the same node ID are paired together for binary operations that involve multiple macrovectors. For example, Figure 2.1 shows the operation `C = A + B` performed on three $n \times 2$ macrovectors. In this operation, the elements of A and B corresponding to node 35, for example, are added together and stored in the elements of C corresponding to node 35.

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.

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

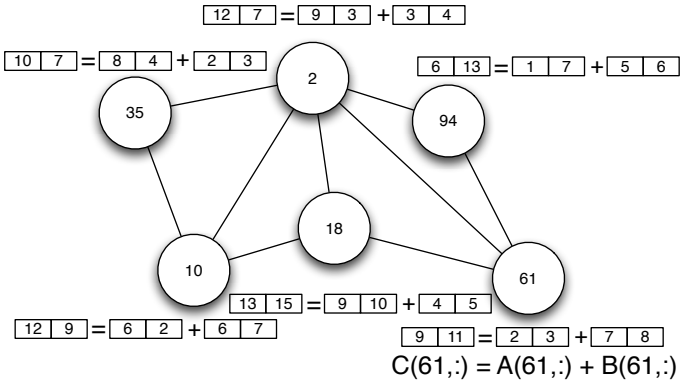


Figure 2.2: When a macrovector is distributed, nodes read and write their own elements of the vector.

Figure 2.2 shows how the elements of the macrovectors *A*, *B*, and *C* from Figure 2.1 can be stored on each node and how the `addition` operation is performed. 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. As Figure 2.3 illustrates, when node 35 writes to its own element in the vector the value is *reflected* to all other nodes currently caching it. 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 consensus. 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 [14] or barriers [28].

The macroprograms written in the Matlab-like syntax are compiled down to microprograms in nesC [23] that run on mote-class devices. MacroLab uses a Matlab-like syntax because Matlab is the most familiar programming environment for application domain experts [49]. This would make it easier for domain experts to implement applications that demand complex network behavior without having to rely on WEN researchers as is currently the norm.

Figure 2.4 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

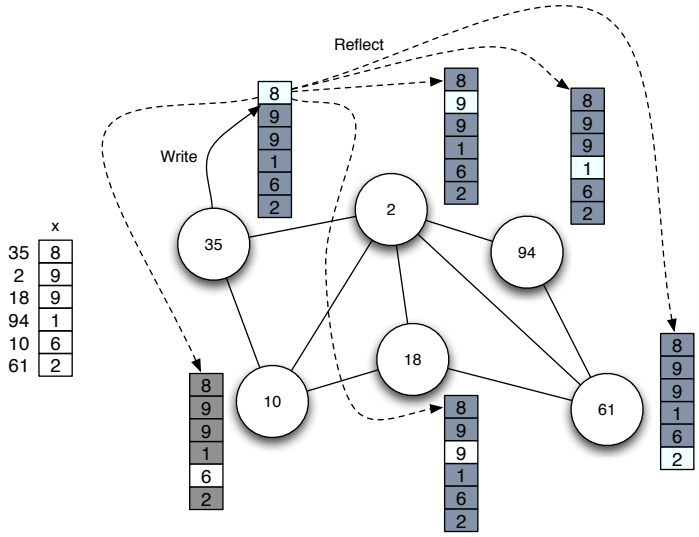


Figure 2.3: When a macrovector is reflected, nodes can read all values in the vector, but can only write to their own elements of the vector.

```
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(...
11           maxNeighbor(active) == magVal(active))
12    focusCameras(leaders);
13 end
```

Figure 2.4: 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.

sensors cooperate to locate and focus a camera on a moving object [62]. 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`. Since `sum` returns the sum over columns by default, the second parameter, 2, to `sum` indicates the sum across rows to be calculated since each row contains a particular node's neighborhood. Line 9 creates a `maxNeighbor` vector with the highest sensor value in each node's neighborhood. Here too the 2 as the second parameter indicates that the maximum value across nodes should be calculated instead of across columns. 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 [27].

The MacroLab compiler could decompose the program in Figure 2.4 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

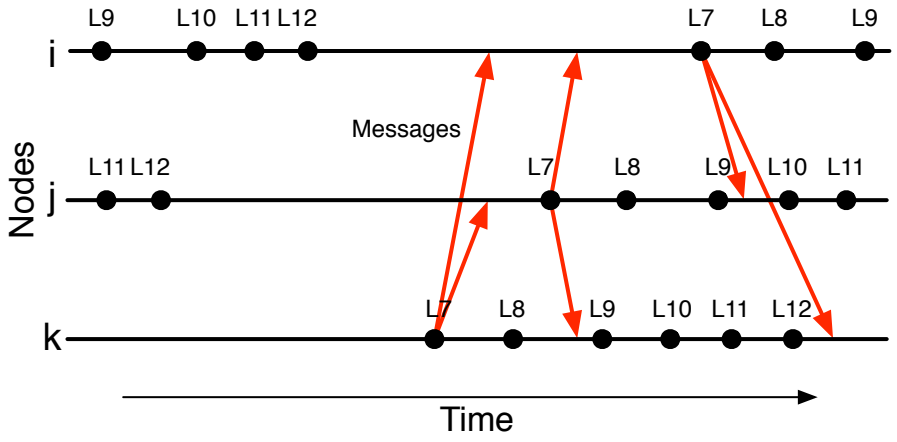


Figure 2.5: Nodes *i*, *j*, and *k* execute lines of the macroprogram in Figure 2.4 asynchronously. Dots indicate when a node executed a particular line of code, and arrows indicate the transmission of cache update messages.

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 2.4 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 2.5 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 2.4 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 2.4, 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 read-

ing, 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.

Chapter 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 [13]. MDB overcomes this

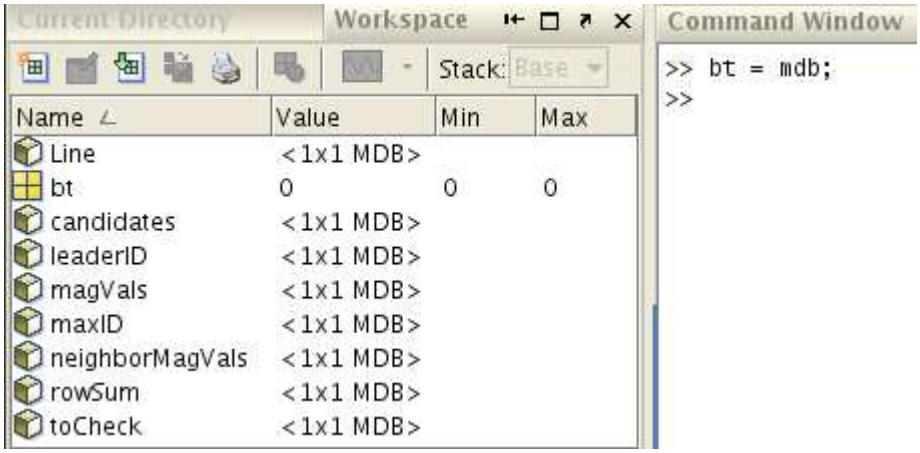


Figure 3.1: Starting the MDB debugger creates new objects in the Matlab workspace for every macrovector in the program being debugged.

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.

Our prototype implementation of MDB is designed for the MacroLab programming environment, although the principles can apply to many different macroprogramming abstractions. Because MacroLab uses a Matlab-like syntax, the current implementation of MDB is designed to be used through the Matlab command prompt. To debug the execution of a MacroLab program using MDB, the user executed the `mdb` command which starts up Matlab with the workspace initialized with the macrovectors in the program being debugged as shown in Figure 3.1.

Table 3.1 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.

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 3.1: 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.

3.1 Logically Synchronous Views

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

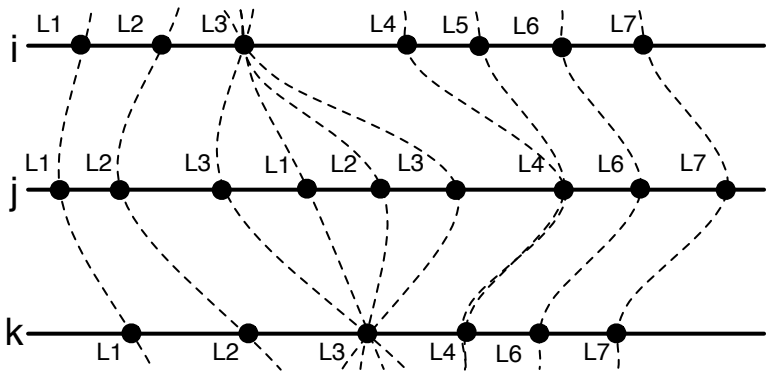
and views the value of the `active` variable:

```
>> 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.

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

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




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

Figure 3.2: 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.

Figure 3.2 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. This could lead to *causally inconsistent* states being presented to the user. For example, by the time node i reaches line L, node j may already have passed line L and sent i a cache update. Thus, a logically synchronous view at line L would show nodes i and j in a causally inconsistent state. Although all elements of the global state shown in such a view are not causally consistent, the view itself is still useful for debugging. 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.

The technique described above for representing branches and loops in logically synchronous assumes that all nodes execute the same code and that no node fails or blocks indefinitely on a variable. The first assumption is inherent in our technique for stepping through non-sequential code sequentially and, therefore limits the usability of the logically synchronous views in MDB while the second assumption can easily be eliminated by monitoring for a node failure or indefinite block when generating the next logically synchronous view in response to an `lstep`. In the event that no node's state can be updated based on our technique above, yet all the nodes are not at the same macrocode line number, any node that is not at the common macrocode line can be assumed to have failed and be ignored in generating future logically synchronous views.

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 2.4, `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 program-

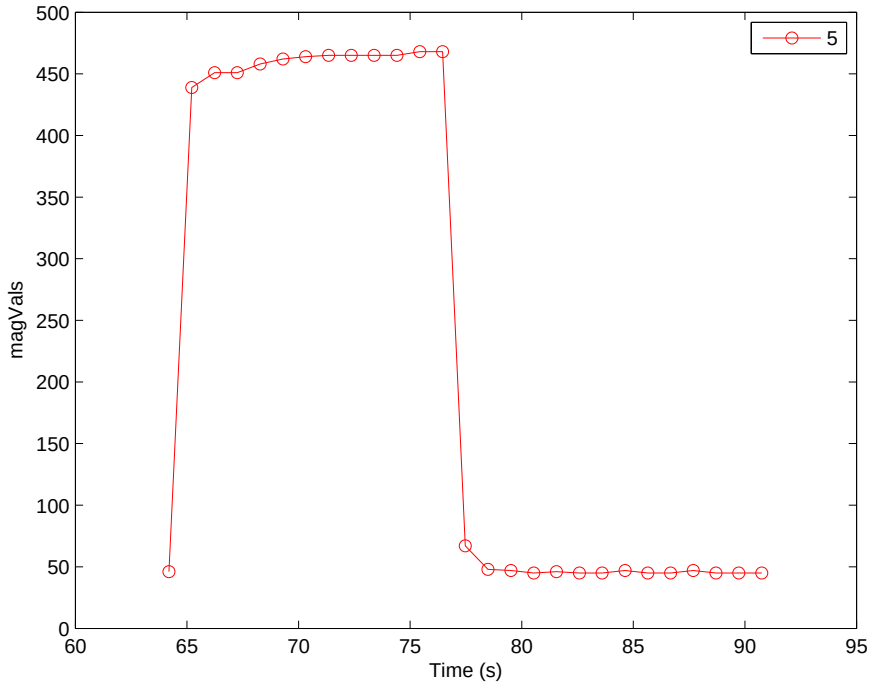


Figure 3.3: MDB's views of distributed state can be combined with Matlab's rich plotting and analysis tools. This figure is the result of a single debugging command: `plot(magVals(5, :))`

mer 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 3.3 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 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 2.4 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 3.4(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 3.4(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 3.4(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 3.4(c) illustrates how message reception is altered by the command above: the 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.

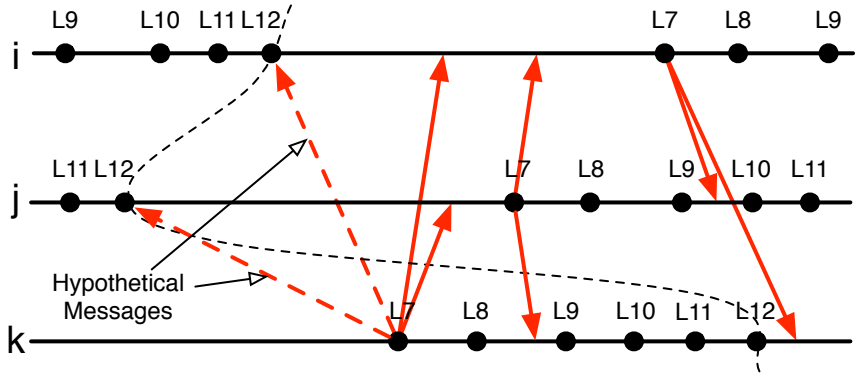
3.4.4 Hypothetical Cache Expiration

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

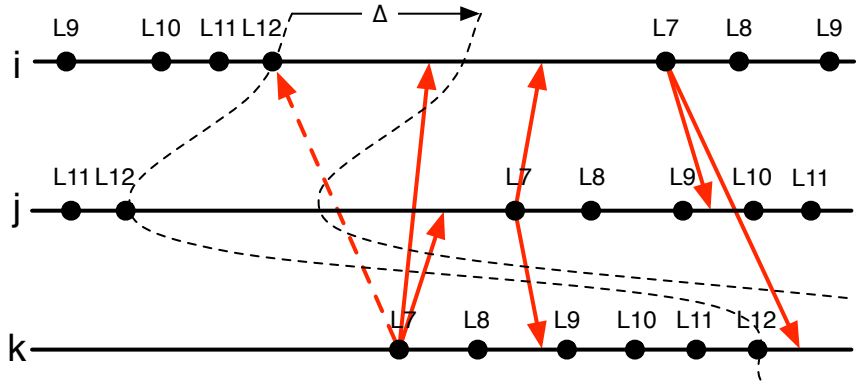
expiring too slowly can result in the use of stale values. *Hypothetical cache expiration* shows the hypothetical state that would result if cache expiration were used at a given line of code, with a particular expiration time. This view helps the user evaluate the effect of different expiration times and can be created using a command such as:

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

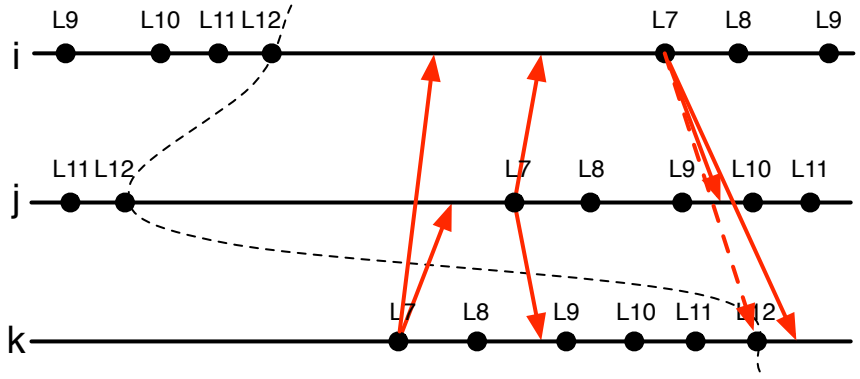
The values in the view are produced by finding the last value written to each element of the macrovector in the time interval $[t - t_e, t]$ where t is the time the node executed line 12 in the current instance of the logically synchronous view and t_e is the expiration time, in this case 100000 μs.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



(a) Hypothetical Barrier



(b) Hypothetical Time Delay



(c) Hypothetical Cache Coherence

Figure 3.4: 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.

Chapter 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 [65].

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* [32]. Other timekeeping schemes such as vector clocks have more precise causal semantics [45, 20]. 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 [18, 42]. 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 [18]. 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

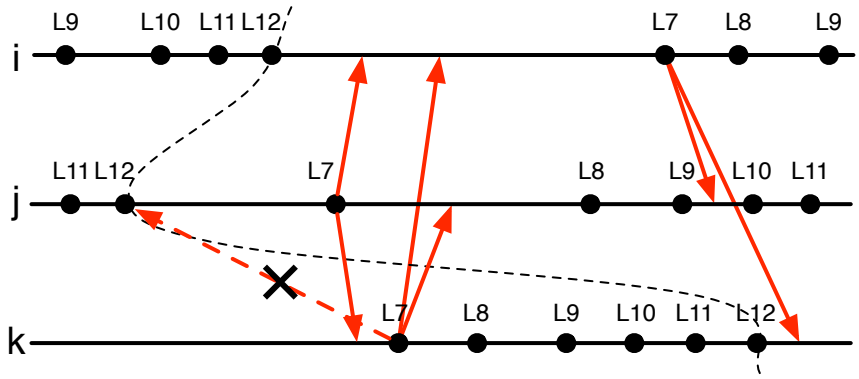


Figure 4.1: A hypothetical barrier cannot be applied in this scenario because the hypothetical advancement of the message from node k to node j would create a dependence cycle.

parameter passed to `tjump`. `ttime` is updated when the user issues the `tstep` 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 3.4(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.

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 4.1, 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.

Chapter 5

Macroprogramming Properties for MDB

While MDB is currently implemented for MacroLab, its principles apply to other macroprogramming systems. In this section we define the language and system properties necessary to support various aspects of MDB and identify other macroprogramming systems that share these properties.

High-level abstraction. The high-level programming abstraction of MacroLab concisely describes complex system states in a small number of lines of code and variables. This allows for efficient logging on resource-constrained devices. All macroprogramming abstractions, by definition, allow for such a succinct descriptions [39, 51, 60, 31, 25].

No message passing. MacroLab forbids user-written message passing, instead managing all communication in the run-time system. MDB can thus log only cache update messages. Systems such as Hood [62], Pleiades [31], COSMOS [1], and Semantic Streams [61] also abstract away message passing from the programming model.

Clear mapping from macro-code to micro-code. This is essential to log the ends of macrocode lines in microcode. Most macroprogramming systems, such as [31] translate the macrocode statements into appropriate blocks of microcode and therefore possess this property.

In-order message delivery. This enables cache updates from a given node to be received in the order they occurred in locally. Most macroprogramming systems are built on this assumption [5]. If not, sequence numbers can be used to to prevent out-of-order cache updates.

Sequential imperative language. MacroLab allows logically synchronous views to be generated where the user can place breakpoints and step through code. This property is supported by all sequential imperative macroprogramming systems, such as Marionette [63].

Data caching. Data caching, in the form of neighbor reflections, yields Lamport time stamps without additional logging. This property also makes hypothetical views both possible and useful since they show how data caching would be affected by application changes. All macroprogramming abstractions that support data caching, such as Hood [62], provide this property.

Single machine abstraction. MDB displays the state of the system as the state of the macrovectors if they were centralized. This requires the system to provide a single machine abstraction to the network. Systems such as RuleCaster [6] also possess this property.

Data-centric. The data-centric nature of MacroLab enables data traces, rather than event traces, to be logged. Semantic Streams [61], COSMOS [1], and TinyDB [38] are similar data-centric languages.

Chapter 6

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.

6.1 Experimental Setup

The evaluations were carried out on a testbed with 21 TMote Sky nodes with photoresistor sensors (Figure 6.1). 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 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 [52] which makes use of the MSPSim [19] 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 2.4), 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 6.2), which is a simple data collection

Figure 6.1: We evaluated MDB on three macroprograms by running them on a 21 node testbed. Light sensors and a projector created the sensor stimuli.

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

Figure 6.2: Surge reads sensor values and displays them at the base station.

application that periodically samples from a sensor and displays the readings at a base station. The third is an acoustic monitoring application (Figure 6.3), 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.

6.2 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

```
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 6.3: The acoustic monitoring application reads values from a microphone and reads from a higher-fidelity microphone on nodes whose neighborhoods detect high average noise levels.

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

Table 6.1: 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.

these values by analyzing the programs as they executed on the Cooja network simulator. Figure 6.4 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.

6.3 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 MDB requires 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 6.5 depicts 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 *Bytes* of data, while the Tmote Sky nodes have about 10 KB of RAM available.

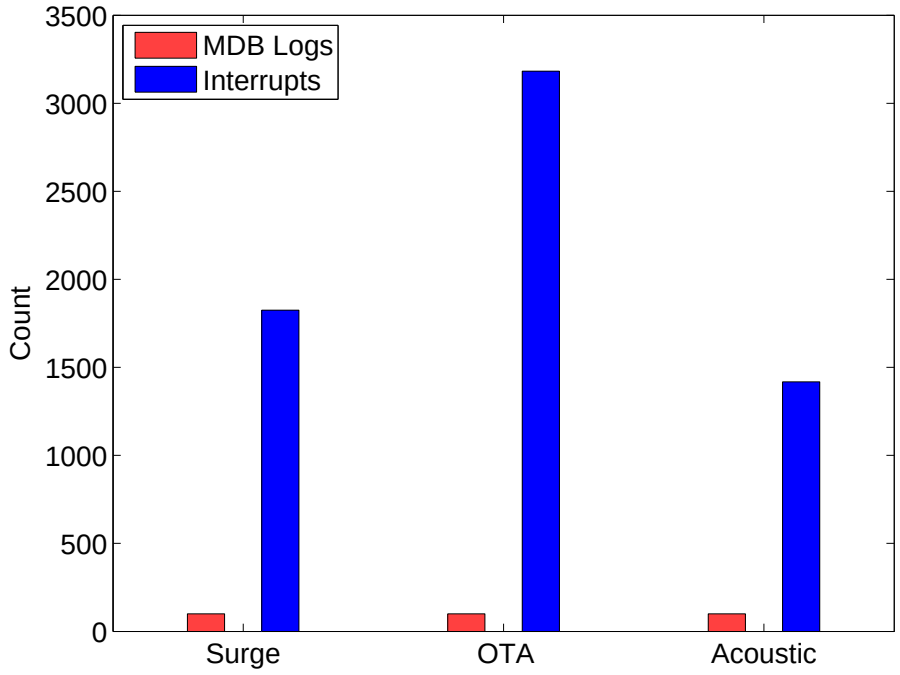


Figure 6.4: Number of interrupts generated per 100 data states written to flash. WEN applications produce 14–32 times more interrupts and message events than macroprogram state updates.

We also measured the amount of Flash memory required by MDB to store the complete data traces for each of the three applications. Table 6.1 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 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.

6.4 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,

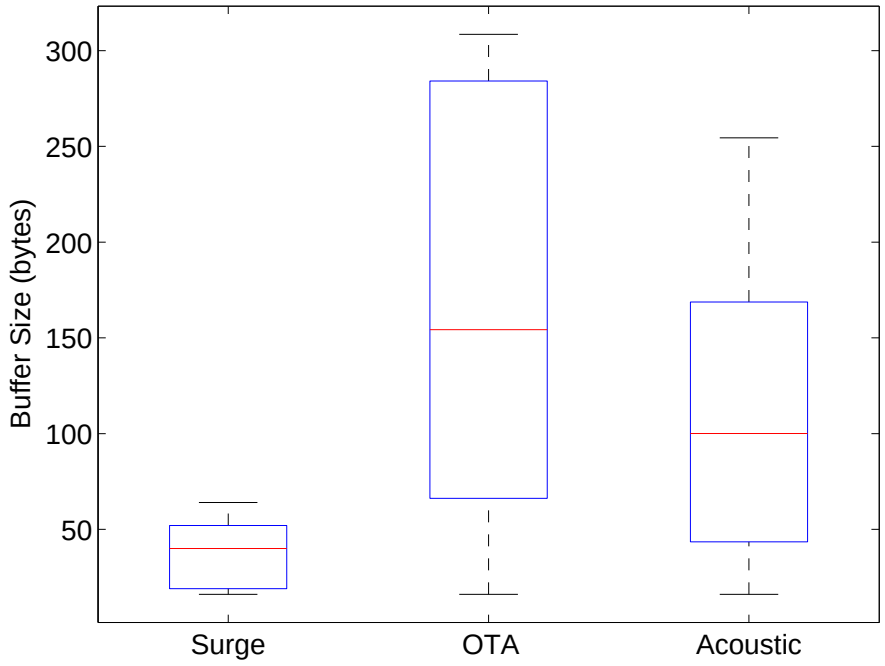


Figure 6.5: Log data is stored temporarily in RAM before being written to flash. In our experiments, no node needed more than 304 *Bytes* of RAM. The box plot shows the minimum, lower quartile, mean, upper quartile, and maximum values.

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 6.6 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 6.6 are averages over all the nodes simulated in Cooja. As Figure 6.6 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.

It takes exactly 105 machine instructions to log data to the RAM buffer. Execution is minimal because on the MSP430 an instruction takes approximately 125 nanoseconds. This means the logging overhead is about 13 microseconds. This is extremely small when compared to the normal duty cycle of applications written in MacroLab.

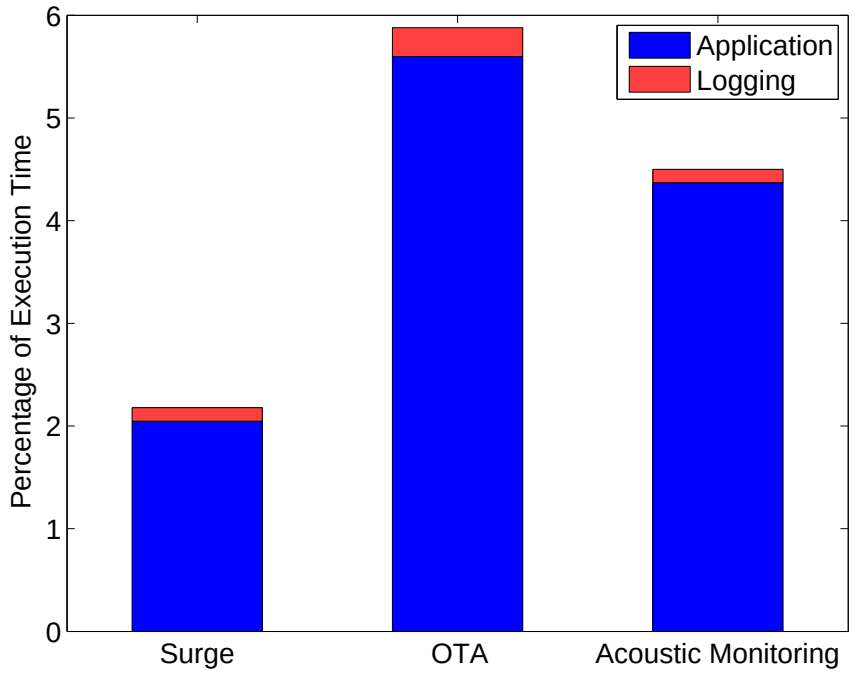


Figure 6.6: MacroLab context constitutes less than 6% of all code running on the nodes. Logging code constitutes less than 0.5%.

6.5 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) [53]. Figure 6.7 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.

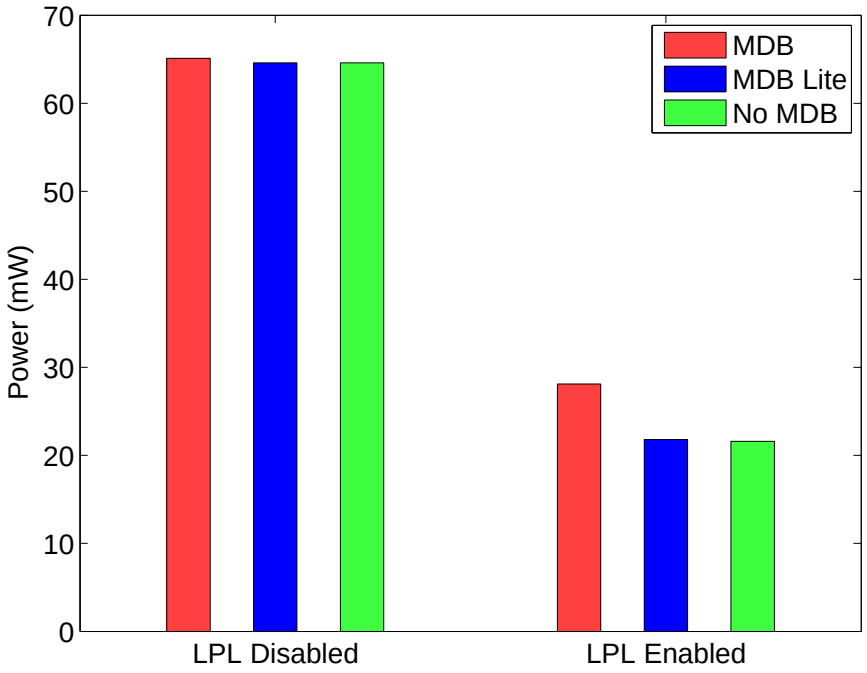


Figure 6.7: Without low-power listening enabled, applications consume 30% more energy with MDB enabled, but only 0.9% more energy with MDB Lite.

6.6 Discussion

The effectiveness of MDB as a WEN debugger depends on two factors: (i) the usefulness of the debugging abstraction and (ii) the efficiency of trace logging. We attempt to address the first factor by demonstrating that the features of MDB can be used to debug three classes of bugs in macroprograms (Chapter 3). MDB, by its very nature, does not enable the debugging of bugs in layers below the application layer. For example, a user cannot identify a bug in the MAC layer or routing layer because MDB hides message passing between nodes. This abstraction of low-level details conforms to the abstraction provided by the macroprogramming system since one of the goals of MDB is to allow the user to program and debug at the same level of abstraction. Also, MDB relies on the guarantees of macroprogramming systems that the low-level libraries have been rigorously tested and can be used in macroprograms to build more complicated applications. If a user of a macroprogramming system does not have such a guarantee about the function libraries, macropro-

gramming would become more of a burden on the user than writing the functions from scratch as the user would have to test the interoperability of various functions, that have been written by others, before they can be used to write an application. Thus, a debugger that allows the user to debug at the application level of a macroprogramming language is a useful debugging abstraction.

In order for MDB to be useful, execution traces should be collectible for a sufficient duration. MDB attempts to maximize the duration of execution for which traces can be collected by minimizing the number of trace entries that have to be logged per unit of time. This is done by exploiting the fact that the user is debugging at the macroprogram abstraction and, therefore, only changes that are reflected in the macroprogram need to be logged. Thus, the many events, such as interrupts, that take place in the TinyOS operating system for instance can be ignored since they are not visible at the macroprogramming abstraction. What MDB does log are the writes to variables declared in the macroprogram a macroprogram counter than indicates which part of the macroprogram each node is executing. This enables MDB to log 14–32 times fewer trace entries per unit of time than if MDB were logging all events within a node so as to recreate each node's execution.

MDB utilizes a temporary buffer in RAM to which macroprogram state changes are logged while each iteration of the macroprogram's every loop is executing. At the end of each iteration, this buffer is written to flash. The amount of RAM required for MDB depends on the number of variables in the macroprogram, the number of lines of code in the macroprogram, and the number of neighbors with which each node shares macrovectors. Our evaluations indicate that for OTA, which has 13 lines of macrocode with five vectors, on the 21 node testbed where each node has eight neighbors the maximum amount of RAM required is 304 *Bytes*. This is just 3% of the RAM available on Tmote Sky nodes. While larger, more complicated applications could require a greater amount of RAM for the buffer, the fact that OTA, which is one of the largest and most complicated WEN applications to data [62], requires such a small fraction of the available RAM indicates that MDB would not pose much of an overhead on the available RAM for most applications.

For the duration of the data collection phase the traces are stored on the node's external flash. For the object tracking application traces for one hour of execution can be collected before the flash buffer gets overwritten. This duration can be increased for debugging purposes by reducing

the sampling frequency or network density which would decrease the number of state changes that have to be logged. For most applications which would fall in the complexity spectrum between Surge, which is one of the simplest WEN applications, and OTA, around five hours of execution traces could be collected without overwriting the flash. For most WEN applications, five hours of execution should be sufficient during the debugging phase to analyze their performance and identify bugs. If the user needs data from a longer execution of the application, the traces can be collected periodically, just before the flash buffer gets overwritten, and the application could be allowed to continue executing.

Due to the controlled conditions under which the testing phase of application development usually takes place, the limitations placed by the flash capacity can be overcome as described above. Yet, if the user wishes to deploy the application with logging enabled so that any bugs that arise during deployment can be analyzed, the flash capacity is of greater concern. For example, the presence of a bug in a data collection application may only be noticed when the data is analyzed at the end of a deployment. MDB may not be usable for identifying this bug if the bug occurred before the last n hours of execution if the flash buffer fills up every n hours. Yet, if the bug manifested itself within the last n hours, the available data traces could still be used with MDB to identify the bug. This would not be possible if MDB logged execution traces without resorting to checkpointing or other techniques for storing the state of the system periodically.

MDB impacts the execution speed of applications by consuming CPU cycles to carry out the logging operation. As Figure 6.6 shows logging to RAM imposes only a 0.5%, or 13 μs, overhead on the application execution time. Since the inherent uncertainty of even a sensor readings is an order of magnitude greater at 20 ms (Section 4.3), the 13 μs overhead of logging can be ignored for almost all applications.

MDB also consumes energy while logging which could decrease the lifetime of the application. As Section 6.5 describes, an application with logging enabled consumes 30% more energy than the application without logging enabled. This is a considerable amount of energy since it could decrease the lifetime of a system by 30%, for instance causing an application that could have executed for a year to die in eight months. A user could mitigate this issue in two ways. The first is to increase

the power available to each node by at least 30%. This can be done by providing each node larger or additional batteries or augmenting each node with energy scavenging capabilities. This would enable the full logging capabilities of MDB to exist while the application is executing in the real world so that if a problem were to arise the user could obtain the traces and attempt to identify its cause.

An alternative is to deploy applications in two phases. The first is a debugging phase where the application, with MDB logging enabled, is deployed for a short period of time so that traces can be collected and bugs identified. Then, after the identified bugs have been fixed, the application is deployed for its full duration without the MDB logging capabilities enabled so that their is no energy overhead. This is ideal if the application does not need logging during execution or if it providing each node with additional power is not feasible and the application cannot tolerate the energy overhead of MDB. Yet, an application without MDB logging enabled could behave differently from an application with MDB logging enabled due to the difference in timing between the two. This is mainly the time MDB take to write the RAM buffer to flash which could be about 10 ms per log entry. This timing difference could cause a bug that was hidden when MDB was enabled to manifest itself during the actual deployment. Such bugs are called Heisenbugs. MDB allows a user to deploy an application without flash writing enabled while still maintaining the timing characteristics of MDB by enabling MDB Lite instead of MDB when compiling the application. MDB Lite functions just like MDB except when writing the RAM buffer to flash. During this phase MDB Lite disables the interrupts which MDB disables when writing to flash and puts the node into a sleep state for the same duration of time MDB takes to write the RAM buffer to flash. This operation ensures that MDB Lite behaves in an identical manner to MDB in terms of timing, yet by eliminating the flash writes decreases the energy overhead from 30% to 0.9% which should be tolerable for most applications. Thus, as other projects such as [58] have claimed, MDB eliminates the possibility of Heisenbugs by enabling the tracing to be left on at all times.

Chapter 7

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 [36, 9, 7]. Distributed breakpoints [48, 21, 22] and snapshots [10] 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 [24, 46, 64]. Others provide logical or SQL-like query languages for searching, inspecting, and analyzing state or execution flow [15, 54, 34]. 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 [63] and Clairvoyant [66] provide access to source-level symbols, while Declarative Tracepoints [8] and Wringer [59] provide a programmable interface for describing debugging operations that can be downloaded and executed on the nodes at run-time. DustMiner [29] and LiveNet [57] eavesdrop on messages in the network for visibility into network operations without consuming node resources. EnviroLog [37] logs some non-deterministic events to produce efficient in-network execution replay. Sympathy collects a small amount of data to identify the cause of network failures [55]. 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 [65, 33]. Execution replay incurs a debug-time cost of recreating the state, and some systems must use checkpointing to limit the amount of replay required [65].

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* [40]. 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 [43]. 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 6.2.

Several debugging abstractions have been created for use during execution replay. For example, TraceView [41] allows the user to visualize event traces. Garg et al. detect weak unstable predicates using traces [22]. 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 [30]. Finally, D3 [12] 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.

Debugging on traces allows MDB to provide time travel capabilities for users. This allows the user to go to step back through states, or time, in order to identify the root cause of a bug. A number of time travel debuggers have been proposed for traditional computer programs. Bhansali et al. allows Windows applications to be debugged by running them backwards in order to figure out the root cause of the problem [3]. The Omniscient Debugger (ODB) records every change in every accessible object or local variable in each thread of a Java program to allow a user to navigate backwards in time and look at objects, variables, and method calls [35]. ReVirt executes an application in a virtual machine which logs below the virtual machine so that sufficient information can be logged to replay and analyze bugs [17]. These techniques are not feasible in the domain of WENs due to its distributed resource constrained nature.

There have been a number of advances in debugging sequential programs. One such advance is delta debugging [67] which automatically minimizes test cases by comparing code differences between two versions of a program and tries to locate changes which cause errors by replacing code from a failed version with code from a succeeding version. Chianti [56] explores many version of a program through a set of tests. Model-based debugging [47], which attempts to automatically locate defects in a program by modeling the program from its source code is another area that has seen considerable research recently. Most of these techniques attempt to automate the task of failure localization. While such an approach is possible for certain classes of failures for WENs, such as communication failures in data collection applications which Sympathy attempts to localize, the distributed, event-driven nature of WEN applications make them too complicated for such a technique to be utilized to identify failures in general.

Chapter 8

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 [27], 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 [26], Plaeiades [31], and Marionette [63]. Other systems that use functional [50] or declarative abstractions [38] must present information from data traces using a different interface. The four hypothetical changes pre-

sented illustrate the affects of adding data synchronization to a macroprogram, and are only useful to systems like MacroLab and Hood [62] 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.

Bibliography

[1] A. Awan, S. Jagannathan, and A. Grama. Macroprogramming heterogeneous sensor networks using cosmos. *SIGOPS Oper. Syst. Rev.*, 41(3):159–172, June 2007.

[2] J. Bachrach and J. Beal. Programming a Sensor Network as an Amorphous Medium. Technical report, MIT, 2006.

[3] Sanjay Bhansali, Wen-Ke Chen, Stuart de Jong, Andrew Edwards, Ron Murray, Milenko Drinić, Darek Mihočka, and Joe Chau. Framework for instruction-level tracing and analysis of program executions. In *VEE '06: Proceedings of the 2nd international conference on Virtual execution environments*, pages 154–163, New York, NY, USA, June 2006. ACM.

[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. *Mob. Netw. Appl.*, 10(4):563–579, August 2005.

[5] Ken Birman. A response to cheriton and skeen's criticism of causal and totally ordered communication. *SIGOPS Oper. Syst. Rev.*, 28(1):11–21, Jan. 1994.

[6] U. Bischoff and G. Kortuem. RuleCaster: A Macroprogramming System for Sensor Networks. *OOPSLA '06: Proceedings OOPSLA Workshop on Building Software for Sensor Networks*, 2006.

[7] Shirley Browne, Jack Dongarra, and Anne Trefethen. Numerical libraries and tools for scalable parallel cluster computing. *Int. J. High Perform. Comput. Appl.*, 15(2):175–180, 2001.

[8] Qing Cao, Tarek Abdelzaher, John Stankovic, Kamin Whitehouse, and Liqian Luo. Declarative tracepoints: a programmable and application independent debugging system for wireless sensor networks. In *SenSys '08: Proceedings of the 6th ACM conference on Embedded network sensor systems*, pages 85–98, New York, NY, USA, November 2008. ACM.

[9] O.S. Center. Xmpi-a run/debug gui for mpi. Technical report, Ohio Superconputer Center, 1997.

[10] K. Mani Chandy and Leslie Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, February 1985.

[11] David Chu, Lucian Popa, Arsalan Tavakoli, Joseph M.Hellerstein, Philip Levis, Scott Shenker, and Ion Stoica. The design and implementation of a declarative sensor network system. In *SenSys '07: Proceedings of the 5th international conference on Embedded networked sensor systems*, pages 175–188, New York, NY, USA, 2007. ACM.

[12] Byung-Gon Chun, Kuang Chen, Gunho Lee, Randy H. Katz, and Scott Shenker. D3: declarative distributed debugging. Technical report, UC, Berkeley, 2008.

[13] The GDB developers. Gdb: the gnu project debugger. `http://sourceware.org/gdb`.

[14] Edsger Wybe Dijkstra. Cooperating sequential processes. Technical Report EWD-123, Technological University, Eindhoven, The Netherlands, September 1965.

[15] Mireille Ducassé. Coca: an automated debugger for c. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 504–513, New York, NY, USA, May 1999. ACM.

[16] A. Dunkels, B. Gronvall, and T. Voigt. Contiki-a lightweight and flexible operating system for tiny networked sensors. In *LCN '04: Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks*, pages 455–462, Washington, DC, USA, November 2004. IEEE Computer Society.

[17] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. Revirt: enabling intrusion analysis through virtual-machine logging and replay. *SIGOPS Oper. Syst. Rev.*, 36(SI):211–224, Dec. 2002.

[18] Jeremy Elson, Lewis Girod, and Deborah Estrin. Fine-grained network time synchronization using reference broadcasts. In *OSDI '02: Proceedings of the 5th symposium on Operating systems design and implementation*, pages 147–163, New York, NY, USA, December 2002. ACM.

[19] Joakim Eriksson, Adam Dunkels, Niclas Finne, Fredrik sterlind, and Thiemo Voigt. Mspsim – an extensible simulator for msp430-equipped sensor boards. In *Proceedings of the European Conference on Wireless Sensor Networks (EWSN), Poster/Demo session*, Delft, The Netherlands, JAN 2007.

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

[21] Jerry Fowler and Willy Zwaenepoel. Causal distributed breakpoints. In *In Proceedings of the 10th International Conference on Distributed Computing Systems*, pages 134–141, 1990.

[22] Vijay K. Garg and Brian Waldecker. Detection of weak unstable predicates in distributed programs. *IEEE Transactions on Parallel and Distributed Systems*, 5:299–307, 1994.

[23] David Gay, Philip Levis, Robert von Behren, Matt Welsh, Eric Brewer, and David Culler. The nesc language: A holistic approach to networked embedded systems. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 1–11, New York, NY, USA, 2003. ACM.

[24] M. Golan and D.R. Hanson. Duel-a very high-level debugging language. In *Proceedings of the Winter USENIX Technical Conference*, pages 107–117, San Diego, January 1993.

[25] Ben Greenstein, Eddie Kohler, and Deborah Estrin. A sensor network application construction kit (snack). In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 69–80, New York, NY, USA, November 2004. ACM.

[26] Ramakrishna Gummadi, Omprakash Gnawali, and Ramesh Govindan. Macro-programming wireless sensor networks using Kairos. In *International Conference on Distributed Computing in Sensor Systems (DCOSS)*, pages 126–140, 2005.

[27] Timothy W. Hnat, Tamim I. Sookoor, Pieter Hooimeijer, Westley Weimer, and Kamin Whitehouse. Macrolab: a vector-based macroprogramming framework for cyber-physical systems. In *SenSys '08: Proceedings of the 6th ACM conference on Embedded network sensor systems*, pages 225–238, New York, NY, USA, November 2008. ACM.

[28] H. F. Jordan. A special purpose architecture for finite element analysis. In *Proc. 1978 Intl Conf. Parallel Processing*, pages 263–266, August 1978.

[29] Mohammad Maifi Hasan Khan, Hieu Khac Le, Hossein Ahmadi, Tarek F. Abdelzaher, and Jiawei Han. Dustminer: troubleshooting interactive complexity bugs in sensor networks. In *SenSys '08: Proceedings of the 6th ACM conference on Embedded network sensor systems*, pages 99–112, New York, NY, USA, November 2008. ACM.

[30] Richard Kilgore and Craig Chase. Re-execution of distributed programs to detect bugs hidden by racing messages. In *In Proceedings of the International Conference on System Sciences*, page 423, 1997.

[31] Nupur Kothari, Ramakrishna Gummadi, Todd D. Millstein, and Ramesh Govindan. Reliable and efficient programming abstractions for wireless sensor networks. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 200–210, New York, NY, USA, 2007. ACM.

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

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

[34] Raimondas Lencevicius, Urs Hölzle, and Ambuj K. Singh. Dynamic query-based debugging of object-oriented programs. *Automated Software Engg.*, 10(1):39–74, January 2003.

[35] Bill Lewis. Debugging backwards in time. In *5th Workshop on Automated and Algorithmic Debugging (AADEBUG)*, Ghent, Belgium, September 2003.

[36] M.A. Linton. The evolution of dbx. In *Proceedings of the Summer USENIX Conference*, 1990.

[37] 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.

[38] Samuel R. Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. Tinydb: an acquisitional query processing system for sensor networks. *ACM Trans. Database Syst.*, 30(1):122–173, 2005.

[39] Geoffrey Mainland, Greg Morrisett, and Matt Welsh. Flask: staged functional programming for sensor networks. In *ICFP '08: Proceeding of the 13th ACM SIGPLAN international conference on Functional programming*, pages 335–346, New York, NY, USA, September 2008. ACM.

[40] Rajib Mall. A novel bi-directional execution approach to debugging distributed programs. In *HiPC '99: Proceedings of the 6th International Conference on High Performance Computing*, pages 95–102, London, UK, December 1999. Springer-Verlag.

[41] AD Malony, DH Hammerslag, and DJ Jablonowski. Traceview: a trace visualization tool. In *Proceedings of the First International ACPC Conference on Parallel Computation*, 1991.

[42] M. Maroti, B. Kusy, G. Simon, and A. Ledeczi. The flooding time synchronization protocol. In *Proc. 2nd International Conference on Embedded Neteworked Sensor Systems*, Baltimore, MD, November 2004.

[43] M. Maruyama, M. Maruyama, T. Tsumara, and H. Nakashima. Parallel program debugging based on data-replay. *Transactions of Information Processing Society of Japan*, 46(SIG12(ACS11)):214–224, 2005.

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

[45] Friedemann Mattern. Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms*, pages 215–226. North-Holland, 1989.

[46] P. Maybee. Ned: The network extensible debugger. In *Proceedings of the Winter USENIX Technical Conference*, 1992.

[47] W. Mayer and M. Stumptner. Evaluating models for model-based debugging. In *ASE '08: Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 128–137, Washington, DC, USA, September 2008. IEEE Computer Society.

[48] B. P. Miller and J. D. Choi. Breakpoints and halting in distributed programs. In *Proceedings of the 8th International Conference on Distributed Computing Systems (ICDCS)*, pages 316–325, Washington, DC, 1988. IEEE Computer Society.

[49] J. Scott Miller, Peter A. Dinda, and Robert P. Dick. Evaluating a basic approach to sensor network node programming. In *SenSys '09: Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*, pages 155–168, New York, NY, USA, November 2009. ACM.

[50] Ryan Newton, Greg Morrisett, and Matt Welsh. The regiment macroprogramming system. In *IPSN '07: Proceedings of the 6th international conference on Information processing in sensor networks*, pages 489–498, New York, NY, USA, 2007. ACM.

[51] Ryan Newton and Matt Welsh. Region streams: functional macroprogramming for sensor networks. In *DMSN '04: Proceeedings of the 1st international workshop on Data management for sensor networks*, pages 78–87, New York, NY, USA, August 2004. ACM Press.

[52] F. Osterlind, A. Dunkels, J. Eriksson, N. Finne, and T. Voigt. Cross-level sensor network simulation with cooja. *Annual IEEE Conference on Local Computer Networks*, 0:641–648, 2006.

[53] Joseph Polastre, Jason Hill, and David Culler. Versatile low power media access for wireless sensor networks. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 95–107, New York, NY, USA, 2004. ACM.

[54] M.L. Powell and M.A. Linton. A database model of debugging. In *SIGSOFT '83: Proceedings of the symposium on High-level debugging*, pages 67–70, New York, NY, USA, 1983. ACM.

[55] Nithya Ramanathan, Kevin Chang, Rahul Kapur, Lewis Girod, Eddie Kohler, and Deborah Estrin. Sympathy for the sensor network debugger. In *SenSys '05: Proceedings of the 3rd international conference on Embedded networked sensor systems*, pages 255–267, New York, NY, USA, November 2005. ACM.

[56] Xiaoxia Ren, Fenil Shah, Frank Tip, Barbara G. Ryder, and Ophelia Chesley. Chianti: a tool for change impact analysis of java programs. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 432–448, New York, NY, USA, October 2004. ACM.

[57] Bor rong Chen, Geoffrey Peterson, Geoffrey Mainland, and Matt Welsh. Livenet: Using passive monitoring to reconstruct sensor network dynamics. In *Proceedings of the 4th IEEE/ACM International Conference on Distributed Computing in Sensor Systems (DCOSS 2008)*, pages 79–98, Santorini Island, Greece, June 2008.

[58] Michiel Ronsse, Mark Christiaens, and Koenraad De Bosschere. Cyclic debugging using execution replay. In *ICCS '01: Proceedings of the International Conference on Computational Science-Part II*, pages 851–860, London, UK, May 2001. Springer-Verlag.

[59] Arsalan Tavakoli, David Culler, Philip Levis, and Scott Shenke. The case for predicate-oriented debugging of sensornets. In *Proceedings of the 5th Workshop on Hot Topics in Embedded Networked Sensors (HotEmNets)*, Charlottesville, VA, June 2008.

[60] Matt Welsh and Geoff Mainland. Programming sensor networks using abstract regions. In *NSDI '04: Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation*, pages 3–3, Berkeley, CA, USA, 2004. USENIX Association.

[61] Kamin Whitehouse, Jie Liu, and Feng Zhao. Semantic streams: a framework for composable inference over sensor data. In *Proc. Third European. Workshop on Wireless Sensor Networks (EWSN)*, Zurich, Switzerland, February 2006.

[62] Kamin Whitehouse, Cory Sharp, Eric Brewer, and David Culler. Hood: a neighborhood abstraction for sensor networks. In *MobiSys '04: Proceedings of the 2nd international conference on Mobile systems, applications, and services*, pages 99–110, New York, NY, USA, 2004. ACM.

[63] Kamin Whitehouse, Gilman Tolle, Jay Taneja, Cory Sharp, Sukun Kim, Jaein Jeong, Jonathan Hui, Prabal Dutta, and David Culler. Marionette: using rpc for interactive development and debugging of wireless embedded networks. In *IPSN '06: Proceedings of the 5th international conference on Information processing in sensor networks*, pages 416–423, New York, NY, USA, 2006. ACM.

[64] P. Winterbottom. Acid: a debugger built from a language. In *Proceedings of the Winter USENIX Technical Conference*, pages 211–222, San Francisco, CA, January 1994.

[65] Larry Wittie. The bugnet distributed debugging system. In *EW 2: Proceedings of the 2nd workshop on Making distributed systems work*, pages 1–3, New York, NY, USA, September 1986. ACM.

[66] Jing Yang, Mary Lou Soffa, Leo Selavo, and Kamin Whitehouse. Clairvoyant: a comprehensive source-level debugger for wireless sensor networks. In *SenSys '07: Proceedings of the 5th international conference on Embedded networked sensor systems*, pages 189–203, New York, NY, USA, November 2007. ACM.

[67] Andreas Zeller. Yesterday, my program worked. today, it does not. why? *SIGSOFT Softw. Eng. Notes*, 24(6):253–267, Nov. 1999.