Demo Abstract: Macrodebugging with MDB

Timothy W. Hnat, Tamim I. Sookoor, and Kamin Whitehouse
Department of Computer Science, University of Virginia
Charlottesville, VA, USA
{hnat,sookoor,whitehouse}@cs.virginia.edu

Abstract

Macroprogramming abstractions provide abstract distributed data structures to simplify the programming of wireless embedded networks. However, none of the current macroprogramming systems provide debugging support for application development. We have developed MDB, a GDB-like post-mortem debugger for the MacroLab macroprogramming abstraction. In this demonstration, we show how MDB enables application development and debugging at a single level of abstraction. MDB eliminates the need for a programmer to reason about low-level event traces and message passing protocols, instead allowing debugging in terms of abstract data types. We expect MDB to fill a crucial link in the development cycle as a macroprogram progresses from the drawing board to real deployment.

Categories and Subject Descriptors

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

General Terms

Design, Experimentation, Performance

Keywords

Source-Level Debugging, Macrodebugging, Wireless Embedded Networks

1 Introduction

Wireless embedded networks (WENs) are complex systems that are notoriously difficult to develop and debug. Each node is programmed with a set of local actions [3] and together the nodes produce emergent behaviors that are hard to predict or explain. When a fault occurs, the root cause is often difficult to identify. The user must trace execution [1,5] as it flows between nodes via unreliable broadcast messages or when it starts spontaneously on other nodes due to timer and sensor interrupts. We present *MDB*, a *macrodebugging* system that extends the benefits of macroprogramming to debugging. MDB provides a GDB-like interface that allows the user to inspect and analyze the execution of a WEN. This simplifies the debugging process, provides a common well understood interface, and eliminates the need to analyze traces of low-level events and messages.

SenSys'09, November 4–6, 2009, Berkeley, CA, USA.
ACM 978-1-60558-748-6

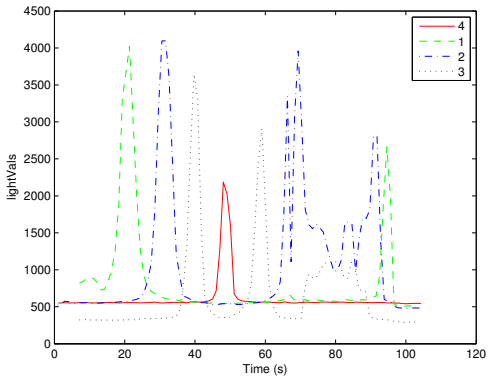

Figure 1. Historical plot for four light sensing nodes.

2 MDB

MDB is a debugging system for WENs that exploits macroprogramming to provide four *abstract views* of a system state: (1) the *temporally synchronous view*, (2) the *logically synchronous view*, (3) the *historical view*, and (4) the *hypothetical view*.

A *temporally synchronous view* shows the values of all macrovectors at a given, globally-consistent time while the *logically synchronous view* shows the state of the system when all nodes are at the same logical point of execution or line of code. *Historical views* go beyond time travel by allowing the user to inspect and analyze all of time in one single view and operate over the state of the system at all times at once, eliminating the need to step forward and backward in time. *Hypothetical views* show system state that would result if process synchronization were enforced at a particular point in execution. Many bugs in WENs are due to message races and often difficult to find. We present four different hypothetical views that help solve this problem.

- *Barrier views* show the values of a macrovector that would have been produced if a barrier was placed at a particular temporal point in execution on each node.
- *Time delay views* show what would happen if the nodes waited the specified amount of time.
- *Cache coherent views* show the hypothetical state of a macrovector if all cache updates sent before the refer-

```
every(uint16(1000))
  lightVals = light.sense();
  candidates = find(nlightVals > 200);
  rowSum = numel(candidates);
  toCheck = find(rowSum > 2);
  maxID = max(nLightVals(toCheck,:));
  leaderID = find(maxID-magVals(toCheck) == 0);
  baseDisplay(leaderID)
end
```

Figure 2. If more than two nodes detect a light value larger than 200, the ID of the higher valued node is displayed at the base station.

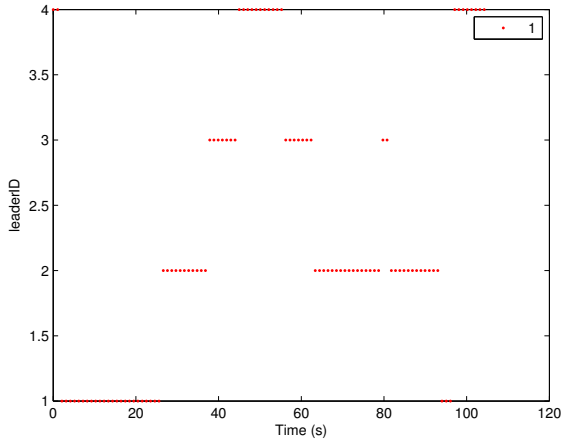

Figure 3. Historical plot for node 1 which shows the state of the variable *leaderID*.

ence time for each node were received by that node at the reference time.

- *Cache expiration views* show the cache after invalidating old values.

3 Demonstration

In this demo, we demonstrate the MDB debugger on a MacroLab [2] macroprogram (Figure 2) that tracks a flashlight. The user will modify the stimulus and the result will be shown in real-time on the base station by displaying the ID of the node with the largest light reading. For demonstration purposes, we are downloading the logs from the serial ports, but using a multi-hop wireless data collection protocol is possible. After the logs have been downloaded, MDB will process them to correct the timing information and present the user with its GDB-like interface.

The first feature of MDB is its ability to display historical views. This view allows the user to plot the logged data from each node. The user will plot *lightVals* with the following command:

```
>> plot(lightVals(:,:))
```

The result is shown in Figure 1.

We can plot more that just data traces. Figure 3 shows node one's view of who the leader of the group is. By using the following command:

```
>> plot(leaderID(1,:),'.')
```

the figure shows the value of *leaderID* every time it is written to.

The problem with the tracking application in Figure 2 is that there is no synchronization between the nodes. For example:

```
>> nLightVals(3,:)
ans =
    (3,1)    150
    (3,3)    500
    (3,2)    160
    (3,4)    100
```

will show the *nLightVals* state on node 3. Since there is only a single value above the 200 unit threshold, this is not sufficient to cause a leader message to be sent. When in reality, a value was not received from node 4. This could be due to a lost message or channel contention. Hypothetical views help diagnose this fault by allowing the user to test the outcome of the alternative application without having to run any new code. The user creates a hypothetical barrier with the following command:

```
>> barrierTL = alt('barrier')
```

The user will then run:

```
>> nLightVals(3,:,getTime,barrierTL)
ans =
    (3,1)    150
    (3,3)    500
    (3,2)    160
    (3,4)    700
```

to obtain the value of node 3's *nLightVals* vector. Node 4 is clearly going to be the elected leader where before there were not enough nodes in the neighborhood to reach a decision. This is just one example of how hypothetical views can be helpful. We will also have available data traces from a real deployment that the user can debug using commands similar to those described. For a more complete description of the MDB framework, please see [4].

Acknowledgment

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

4 References

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

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

[3] Levis, P., Madden, S., Polastre, J., Szewczyk, R., Whitehouse, K., Woo, A., Gay, D., Hill, J., Welsh, M., Brewer, E., et al. Tinyos: An operating system for sensor networks. *Ambient Intelligence '05*.

[4] T. Sookoor, T. Hnat, P. Hooimeijer, W. Weimer, and K. Whitehouse. Macrodebugging: Providing abstract views of system state. In *SenSys*, 2009.

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