MULE: Hybrid Simulator for Testing and Debugging Wireless Sensor Networks

David Watson and Mikhail Nesterenko[*]
Computer Science Department
Kent State University
Kent, OH, 44242
dgwatson@kent.edu, mikhail@cs.kent.edu

Abstract

Wireless sensor networks present a number of challenges to software development. Debugging and testing applications for such networks is especially difficult. We present MULE: a hybrid simulator that combines the ease of debugging multiple simulated motes on a host PC with high fidelity of message transmission and sensor data acquisition of physical motes. We describe MULE's architecture and functionality. We also present experimental results for several test applications run under MULE.

1 Introduction

Wireless sensors are a promising emergent computing platform with a lot of applications. The applications range from monitoring humidity in a redwood forest, to detecting the intrusion of enemy combatants into a protected area, to collecting status information about the machines on a factory floor. There is a number of unique features of sensor networks that distinguish them from other computer architectures: (a) each individual sensor node has limited computing, networking and power resources; (b) the applications for sensor networks necessitate the deployment of thousands, even hundreds of thousands of sensor nodes; (c) the sensor networks closely interact with the physical environment due to the sensing of environmental parameters that such networks accomplish and low-power radios that they use for communication.

The peculiarities of sensor networks present unique challenges to software development, in particular to debugging and testing of the software. Traditional instruction-level debuggers provide the programmer detailed access to the program state and a fine degree of control over the code execution. However, the limited resources of individual sensors, the distributed nature of sensor applications and tight interaction with the environment limit the usefulness of such debuggers in producing software for sensor networks. Field testing is the ultimate trial for a sensor application. Yet the programmers cannot rely on field testing alone: due to resource constraints and the distributed nature of the software, obtaining an adequate picture of the network behavior and controlling the experiment is difficult. In addition, the logistics of deploying a large number of sensors render field testing a supplement rather than the main means of testing. Hence, software designers need to rely on simulation as a primary testing technique for the sensor applications.

A simulator is a convenient debugging tool. It enables developers to run the code of multiple sensor nodes on a single host machine. The state of the program of the simulated sensors can usually be easily examined during the run and the execution flow can

[*]This research was supported in part by DARPA contract OSU-RF#F33615-01-C-1901 and by NSF CAREER Award 0347485

be conveniently controlled. A major challenge for a successful simulator is to faithfully model the interaction of the application with the physical environment. Usually, the simulator expects the experimenter to provide the test readings for the sensors and uses a deterministic or probabilistic mathematical model of radio signal propagation. Unfortunately, using test readings denies the designer the interaction with actual sensors and thus moves the debugging of this part of the application to field testing. Likewise, the signal propagation patterns of low-power radios are rather bizarre [4, 5]. The propagation of radio signals is influenced by a variety of diverse factors: walls, obstacles, concurrent transmission, time of the day, etc. Thus, the fidelity of any mathematical model is at best approximate.

In this paper we present MULE — a simulator that combines the ease of host-PC debugging with the fidelity of actual sensor readings and radio transmissions. As a testing platform we use Berkeley prototype sensors [6, 7] called *motes*. These sensors are quickly gaining in popularity due to their simplicity, ease of use and application readiness. The motes run TinyOS [7]. TinyOS is a lightweight event-based operating system that implements the networking stack, handles communication with the sensors and provides a programming environment for this platform.

Related Work. Atmel's AVR JTAG ICE module allows real-time instruction-level debugging of software on physical motes. JTAG attaches to a mote and enables developers to debug applications using a modified version of GDB. This approach has the distinct advantage that real code is used on actual hardware. JTAG, however, shares the GDB's lack of ability to debug distributed applications.

There is a number of simulators and related tools that can be used to simplify software development for sensor networks. There are a few high-level simulators such as ns-2 [3], GloMoSim [13], and Prowler [11] which can be used to prototype algorithms for sensor applications at an abstract level. These tools are useful for the initial phase of application development. Yet, they do not let the designers debug the code for the target platform.

There are a number of simulators capable of running TinyOS code. TOSSIM [8] and TOSSF [10] and SENS [12] allow the designer to compile the TinyOS program for Intel architecture and run the program on a PC. ATEMU [1] emulates the instruction set of the mote's processor on a PC which provides an interesting debugging option. The four above tools are capable of simulating thousands of motes on a single PC. However, the mote hardware is simulated, hence simulation fidelity is an issue.

We are aware of two simulators that allow running of the code on physical sensor nodes. SensorSim [9] is a modification of ns-2 which allows simulated sensor nodes to communicate with real sensor nodes through gateway machines. This capability extends the usefulness of a high-level simulator as a debugging tool. Yet the limitations of high-level simulators outlined above apply to SensorSim as well. EmStar [9] has a range of tools that can be used for software development in sensor networks. EmStar uses a Linux PC as a host machine. EmStar maintains Unix device files and uses them to connect Unix processes. EmStar uses this connection to simulate radio communication. This simulation can be either purely software or hybrid. In case of hybrid simulation EmStar uses physical motes to transmit the messages. Each Unix process connected to the EmStar device file can run a simulated mote. Since each simulated mote runs as a separate process, the examination of mote's state and control of its execution flow has to be done individually. Hence, coordinated debugging and testing of a distributed application comprising many motes is difficult with EmStar.

Our Contribution. We developed a hybrid simulator — MULE. It simulates multiple motes on a host PC. This makes it possible to inspect the state of the simulated motes as well as suspend and restart the execution of the entire application. The radio transmission and sensor readings are carried out by the physical motes which improves the fidelity of the simulation. MULE handles the coordination between the simulated time of the motes running on the host PC and the real time of transmission and data sensing. The hardware configuration for MULE allows it to be run on a laptop in the field where communication

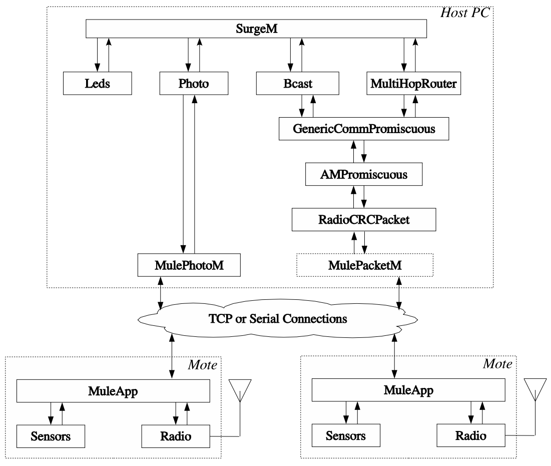

Figure 1: Components involved in hybrid simulation

and sensing characteristic resemble those of actual deployment. MULE is based on TOSSIM. MULE is freely available online [2].

Outline. The rest of the paper is organized as follows. In Section 2 we describe the architecture, functionality and operation of MULE. In Section 3 we present experiments showcasing MULE's capabilities. We cover extensions and future work for our simulator in Section 4.

2 Architecture, Functionality, and Operation

Overview. The configuration requirements for MULE include a PC and a set of physical motes. Each mote is attached to either a serial port of the PC or a Crossbow MIB 600 ethernet interface board.

We describe the architecture of MULE with an example of a standard TinyOS application – `Surge`. `Surge` takes readings from the photosensor and reports them over the radio to the base station. Figure 1 shows MULE running `Surge`.

A TinyOS program consists of a hierarchy of components with well-defined interfaces. A component encapsulates either a hardware module (e.g. photosensor or a clock) or a software module (e.g. a network layer protocol). MULE allows TOSSIM to run the application on the host PC but relays the communication and sensing requests to the physical motes connected to the PC. This is done by replacing the components that simulate communication and sensing in TOSSIM with components that handle the interaction with the motes. There are three MULE communication components: `MulePhotoM`, `MulePacketM` and `MuleApp`.

`Surge` uses the `Bcast` and `MultiHopRouter` components to handle broadcast and multi-hop radio communication. These components employ the standard TinyOS networking stack: `GenericCommPromiscuous`, `AMPromiscuous` and `RadioCRCPacket`. On physical motes, a bit-level radio driver is below `RadioCRCPacket`. In TOSSIM it is replaced by a bit-level radio simulator. In MULE, `MulePacketM` implements `RadioCRCPacket` interface and communicates with physical motes to transmit or receive a radio message. Similarly, `MulePhotoM` replaces a photosensor hardware component on real motes and a photosensor emulator in TOSSIM.

Communication with `MulePacketM` and `MulePhotoM` on the mote side is handled by the other major component of MULE – `MuleApp`. `MuleApp` receives requests for sensing and data transmission from the host PC, carries them out and replies with measurements and received messages.

Synchronization of real and simulated time. Reconciling the timing of real and simulated events is one of the trickiest tasks of a hybrid simulator. Even in a pure simulator, the problem of simulating the execution of multiple concurrently running motes is far from straightforward. A pure simulator has to overlay the execution of the code of multiple motes on the processor of the host PC. Thus, the duration of the simulated events is distorted. To preserve the fidelity, advanced simulators, such as TOSSIM, divorce the simulated execution from the real time. TOSSIM maintains *simulated time* to measure the duration of the events. TOSSIM advances the simulated time as the motes make progress in their execution. A hybrid simulator has to coordinate both real and simulated events. Thus, the problem of reconciling the real and simulated time arises.

When the simulation requires a real event, MULE executes the following sequence. First, MULE records the simulated timing parameters of this event. Then MULE freezes the simulation, translates the parameters into real time, and executes the event in real time gathering the timing parameters. After the real event ends, MULE translates the timing information back into simulated time and resumes the simulation.

Concurrent message transmission coordination. Message transmission is particularly sensitive to concurrent execution due to potential collision between messages transmitted concurrently. If a simulator freezes the simulation after the first message transmission request, then the message transmission is serialized and the messages do not interact. We illustrate MULE's handling of message transmission in Figure 2.

MULE manipulates the following TOSSIM events: `send`(m) — request for transmission of message m; `sendDone`(m) — successful message transmission notification; and `receive`(m) — message receipt notification. When a simulated mote M_1 generates `send`(m_1), MULE delays processing this request and continues the simulation for the approximate simulated time it takes to transmit a message. During this time, MULE records other requests for message transmission. In our case this is message m_2 from mote M_2. MULE notes the simulated time difference between transmission requests of m_1 and m_2. After this delay, the simulation is frozen and `MulePacketM` forwards the messages to `MuleApp` at the physical motes that correspond to the simulated ones. Message m_2 is delayed so that the real time difference between sending requests of m_1 and m_2 matches the simulated time difference.

At each of the physical motes `MuleApp` uses the radio stack of TinyOS to transmit the messages. Since the motes share the radio channel, only one mote at a time is able to proceed. `MuleApp` reports to `MulePacketM` successful transmission of the messages as well as messages received. In our example, `MuleApp` at M_1 reports that it finishes sending m_1, later it reports that M_1 receives m_2. Likewise,

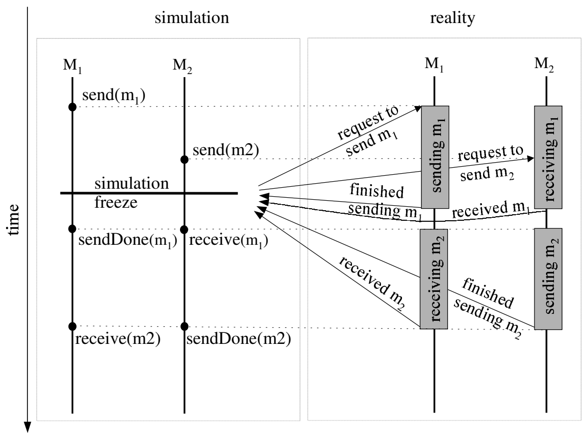

Figure 2: Interaction of radio messages sent by two motes

`MuleApp` at M_2 reports the end of transmission of m_2 and the receipt of m_1. `MulePacketM` notes the timing of arrival of these events, translates the real into simulated time, restarts the simulation and generates corresponding `sendDone()` and `receive()` events at appropriate times in the simulation.

Note that the picture in Figure 2 is simplified. Observe that due to message contention in the radio channel, transmission of m_2 was delayed. Consequently, transmission of m_2 can potentially interfere with with another message m_3 that is requested to be sent after the simulation freeze. MULE notes such requests and executes the subsequent simulation freezes similarly to the simulation freeze described above. Observe also that if the transmission of m_2 overlaps with later messages, it will be resubmitted for transmission by physical motes at the next simulation freeze so that message interference is properly modeled.

In our example M_1 and M_2 can communicate with each other. If they are placed outside the transmission range, then they can send messages concurrently. MULE accommodates this possibility. Hence, a complicated message propagation patterns are properly modeled.

Sensor operation. MULE handles sensor simulation similarly to radio transmission simulation. Each of the sensor types has a module that communicates with `MuleApp`. Sensor setting (such as adjusting the input gain or selecting a particular channel) are forwarded to the instance of `MuleApp` running on the real mote. `MuleApp` issues the appropriate command to the sensor hardware. Some of the sensors can use an asynchronous analog-to-digital converter (ADC) embedded in the motes. If the sensor reading needs to be digitized, `MuleApp` is so notified. In this case `MuleApp` uses the ADC to process the input and sends only the resultant digital reading back to the host PC.

3 Experiments

We describe a few experiments to showcase the capabilities of MULE. MULE's user interface is the same as TOSSIM. Hence, the experiments are demonstrating the hybrid capabilities of MULE. In particular, MULE's ability to operate with physical motes.

Experimental Setup. The equipment used for the experiments is as follows. The simulation host was a 1.70 GHz Pentium 4 with 512 MB RAM running Red Hat Linux 9.0 and using TinyOS v. 1.1.4. We use up to five Mica 2 motes with Mica sensor boards. Two motes were connected to the host PC using serial ports; the others — Crossbow MIB 600 ethernet interface board. In all experiments involving radio transmissions, the radio transmission power was set to the TinyOS default value. The motes did not have external antennas.

Radio transmission with changing mote positions. This experiment used two motes to illustrate how hybrid simulation with MULE supports changing radio channel conditions. The first mote transmitted at a rate of 20 messages per second (5 messages per 250ms) while the second one received the incoming messages. The results of the experiment are summarized in Figure 3. Initially, the sender was placed outside the receiver's radio range. During the first second of the experiment, the sender was gradually moved into the reception range of the receiver. The sender remained in range for two seconds and then moved back out of range in the last two seconds of the test. The results in Figure 3 indicate gradual improvement of message reception followed by fading.

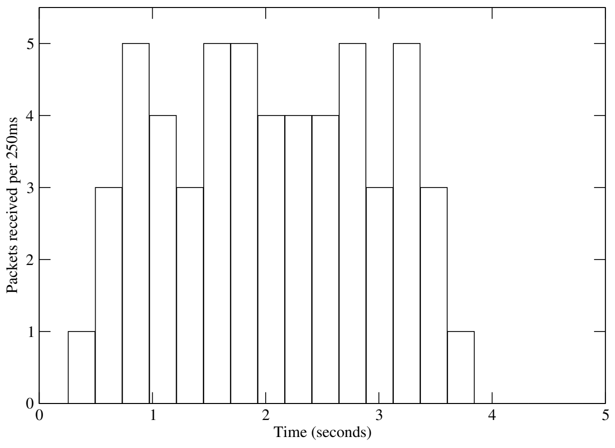

Figure 3: Radio messages received with changing sender position.

Radio transmission among five motes. This experiment demonstrates the operation of MULE with a modest scale. It uses five motes. These motes were arranged in a line, with five inch spacing between adjacent motes. Thus, motes 1 and 5 were approximately twenty inches apart, whereas 2 and 5 were fifteen inches apart. All motes run the same program. Each mote transmitted packets at 50 ms intervals, and listened for incoming packets from the others. The number of packets was recorded, and any lost packets were noted. The experiment was run for 6 seconds, and each mote transmitted between 105 and 115 packets. Figure 4 shows the results of the experiment.

Photosensor Operation. This experiment shows sensor operation under MULE. For this we used an array of three motes. These motes were arranged in a line with approximately three inches between adjacent motes. Initially, the lab was dark. Then a flashlight shined back and forth across the motes three

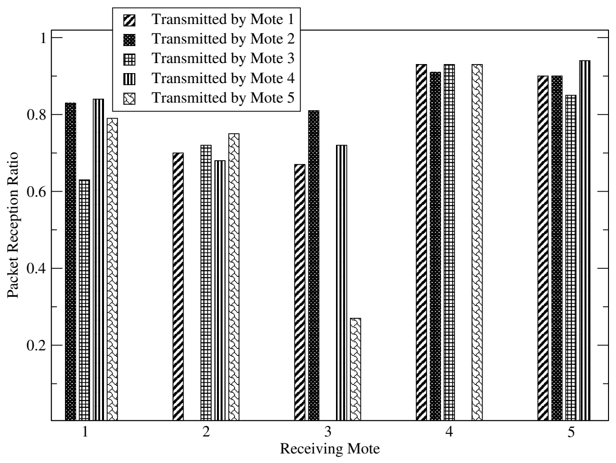

Figure 4: Radio transmissions between five motes

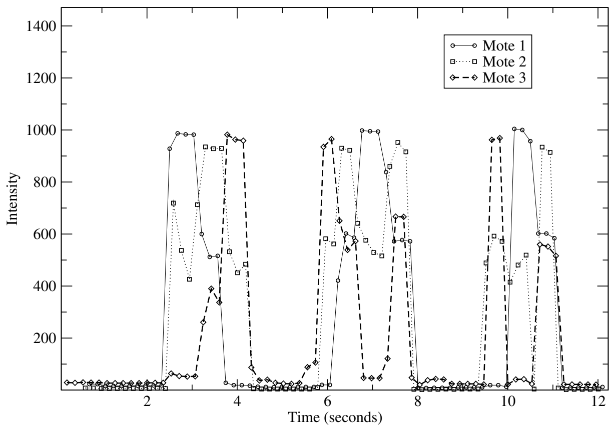

Figure 5: Light sensing in a 3-mote array

times. The motes reported the values given by their photo sensors. The results are shown in Figure 5. No scaling or calibrating of the sensor readings is done. The values shown are the output of the motes' ADCs. The three peaks correspond to the three intervals in which the photosensors were illuminated.

4 Future Work

In this paper we presented MULE: a tool for hybrid debugging and testing of applications for distributed sensor networks. MULE simplifies the arduous process of developing such applications. This paper presents the first version of the tool. In the future, we plan to work on extending the capabilities of MULE in order to increase its effectiveness. Below are the some of features we consider adding to MULE.

Component Migration. MULE makes it possible to divide the component stack into two parts. The lower part runs on the physical mote, the upper part runs under TOSSIM. The two parts are joined by `MulePacketM/MuleApp` pair (see Section 2 for details). Currently, the size and the architecture of the lower part is fixed. We are planning to extend MULE so it is possible to insert a `MulePacketM/MuleApp` pair in an arbitrary place in the component stack. This would give the programmer greater flexibility as to what portion of the TinyOS application they would like to test. The programmer would be able to develop his application incrementally "bottom-up". The physical motes would execute already tested components.

Tiling. Currently, there is a one-to-one mapping between simulated and physical motes in MULE. Thus, the number of motes MULE can simulate is limited by the number of physical motes attached to the host PC. The scale of the simulation can be significantly increased if MULE allowed to map multiple simulated motes on one physical mote. This amounts to tiling the physical motes segment into an arbitrary virtual topology.

Time Synchronization. Even though MULE improves the fidelity of simulation over pure simulators, the fidelity can be further enhanced. Since actual message transmission and sensing is separated from the simulated code execution, their interaction may not be properly represented. For example, a mote running a cryptographic signature computation and receiving a message at the same time will be able to accomplish both tasks simultaneously. Yet in reality, the two tasks will be competing for scant processor resources. Similarly, the time overhead of transmitting messages between the simulation and the physical motes is not perfectly accounted for. The proper solution for these problems is to implement tighter time synchronization between simulated and physical motes across the application. We contemplate implementing global clock ticks that advance the simulation in lock-step.

References

[1] ATEMU - sensor network emulator/simulator/ debugger. `http://www.isr.umd.edu/CSHCN/research/atemu`.

[2] MULE. `http://deneb.cs.kent.edu/mule`.

[3] Josh Broch, David A. Maltz, David B. Johnson, Yih-Chun Hu, and Jorjeta Jetcheva. A performance comparison of multi-hop wireless ad hoc network routing protocols. In *Mobile Computing and Networking*, pages 85–97, 1998.

[4] D. Ganesan, B. Krishnamachari, A. Woo, D. Culler, D. Estrin, and S. Wicker. Complex behavior at scale: An experimental study of low-power wireless sensor networks. Technical Report CSD-TR 02-0013, UCLA, 2002.

[5] D. Ganesan, B. Krishnamachari, A. Woo, D. Culler, D. Estrin, and S. Wicker. An empirical study of epidemic algorithms in large scale multihop wireless networks. Technical Report IRB-TR-02-003, Intel Research, 2002.

[6] J. Hill, R. Szewczyk, A. Woo, D. Culler, S. Hollar, and K. Pister. System architecture directions for networked sensors. *ACM SIGPLAN Notices*, 35(11):93–104, November 2000.

[7] J.L. Hill and D.E. Culler. Mica: A wireless platform for deeply embedded networks. *IEEE Micro*, 22(6):12–24, November/December 2002.

[8] Philip Levis, Nelson Lee, Matt Welsh, and David Culler. TOSSIM: accurate and scalable simulation of entire tinyos applications. In *Proceedings of the first international conference on Embedded networked sensor systems*, pages 126–137. ACM Press, 2003.

[9] Sung Park, Andreas Savvides, and Mani B. Srivastava. SensorSim: A simulation framework for sensor networks. In *Proceedings of MSWiM*, 2000.

[10] Luiz Felipe Perrone and David M. Nicol. A scalable simulator for TinyOS applications. In *WSC*, 2002.

[11] Gyula Simon, Péter Völgyesi, Miklós Maróti, and Ákos Lédeczi. Simulation-based optimization of communication protocols for large-scale wireless sensor networks. In *IEEE Aerospace Conference*, 2003.

[12] Sameer Sundresh, WooYoung Kim, and Gul Agha. SENS: A sensor, environment, and network simulator. In *The 37th Annual Simulation Symposium (ANSS37)*, 2004.

[13] Xiang Zeng, Rajive Bagrodia, and Mario Gerla. GloMoSim: A library for parallel simulation of large-scale wireless networks. In *Workshop on Parallel and Distributed Simulation*, pages 154–161, 1998.