# Simulation of TinyOS Wireless Sensor Networks Using OPNET

Daniel Sumorok, David Starobinski, Ari Trachtenberg
*Boston University*
*Boston, MA 02115*
*E-mail: staro@bu.edu*

**Abstract**
*Many of the sensors finding their way into sensor networks run a lightweight operating system developed at U.C. Berkeley called TinyOS. This open-source operating system, designed specifically for highly-constrained wireless devices, enables building a variety of applications using highly modular code. Over one hundred groups worldwide, and several company products, use TinyOS.*

*Our main contribution in this paper is a simulation interface for compiling TinyOS applications to OPNET model. Our approach harnesses the wealth of tools made available by OPNET, such as wireless channel modeling, scenario management, and collected data management. Using OPNET it is thus possible to simultaneously simulate multiple instantiations of different TinyOS applications, as well as the interaction of TinyOS sensors with other hardware devices (e.g., Ethernet and IP nodes).*

*As part of our preliminary results, we have incorporated the timer, LED, and radio interfaces of TinyOS sensors into an OPNET model. Within our implementation, TinyOS radio packets are converted to OPNET packets and sent and received using the OPNET API (although currently all communications links are modeled as full-duplex serial links). We present two simulation results as proofs of our concept: an LED-based counter, and a location-detection system. These promising results demonstrate the viability of our approach to bridging the TinyOS world with the available infrastructure in the OPNET world.*

## I. Introduction

The TinyOS operating system has been specifically designed to support resource-constrained sensing devices [1]. Each sensing device (or simply sensor) typically contains an embedded processor, a digital radio, several A/D converters, and connectors for attaching sensor boards. Sensors support a wide range of applications, from environmental habitat monitoring to indoor location detection and homeland security. However, the TinyOS operating system software running on each individual sensor is heavily optimized for that sensor's particular application.

Developing and debugging distributed algorithms on TinyOS sensors presents many challenges. In particular, as sensor networks increase in size, scaling considerations cause the distributed algorithms running on these sensors to increase in complexity. Furthermore, deploying a large sensor network simply to verify the implementation of a distributed algorithm can be time-consuming and cumbersome. This is compounded by the fact that due to the limited capabilities of the sensors, debugging on the target hardware is difficult. Therefore, developing and implementing complex distributed algorithms for TinyOS sensors requires an effective simulation tool. An ideal simulation tool should support a "shared code" model, in which the same application code is shared between the target executable and the simulation model. Towards achieving this goal, we have developed a simulation interface to compile a TinyOS application into an OPNET model. This work is devoted to a description of this interface, its capabilities, and the more prominent technical challenges of its development.

Our work is closely related to, and based in part on, another shared-code simulator for TinyOS, called TOSSIM [2]. The TOSSIM simulator is a free tool, created from scratch, that enables multiple instantiations of the same TinyOS application to be simulated. It has been used successfully to debug the TinyOS radio stack [2]. While TOSSIM is a capable tool, it does not provide means for scenario management and statistics management, does not allow instantiations of different applications to be simulated together in the same memory space, and does not have the wealth of link models available in OPNET. In short, the maturity of the OPNET simulator will give it an edge over TOSSIM for large, sophisticated TinyOS simulation projects.

This paper is organized as follows. In section II we give an overview of the TinyOS operating system, and in section III we describe our OPNET implementation of TinyOS, including how the shared application code is compiled and linked to the TinyOS OPNET process model. In section IV, we show a simulation of two sample TinyOS programs. Concluding remarks are offered in section V.

## II. TinyOS Overview

The TinyOS operating system design emphasizes modularity and compactness. To enforce the modularity, TinyOS applications are written in a custom "C"-based programming language called NesC. The NesC programming language requires functionality to be encapsulated in components with well-defined interfaces. A TinyOS application consists of a collection of these NesC components connected, or wired together. Some components are specific to the target platform, other components are generic to the TinyOS operating system, and still other components are specific to the application being built. When a TinyOS application is built for a particular platform, the build system knows how to wire the application-specific and TinyOS generic components to the hardware-specific components for that platform.

The modularity of TinyOS also facilitates building very compact applications. This is important for sensor platforms. Sensor platforms have limited processing and memory resources, and usually perform well-defined functions. It is desirable that only the parts of the operating system necessary for the specific sensor application be present. The design of TinyOS allows unused TinyOS components, such as unused sensor components, to be omitted from the target application object code.

Building TinyOS applications requires the NesC compiler, which converts NesC code into standard *C* code suitable for target-specific compilation with the GNU Compiler Collection (gcc). From gcc's point of view, the NesC compiler is a preprocessor that processes header files and pre-processor directives. Its job is to output a single pre-processed "C" source file.

Functionally, TinyOS sensor applications can often be represented as event-driven state machines. While in an idle state, the sensor powers down resources and waits for an event. When the sensor application receives an event, such as a timer expiration or radio packet reception, the application does some processing immediately, and then asks TinyOS to schedule, or *post*, a Task. A Task is simply a function or procedure that is executed at a later time. TinyOS maintains a queue of tasks, and runs them to completion serially in the order they were posted. Although the tasks cannot interrupt each other, tasks may be interrupted by events. Each task can also post its own tasks. When the task queue is empty, TinyOS goes to sleep and waits for another event.

## III. OPNET TinyOS Implementation

### A. Build Strategy

Integrating TinyOS applications with OPNET presents several technical challenges. The TinyOS build system permits applications to be built for different platforms, including the TOSSIM simulator platform. In our case, we defined a TinyOS OPNET simulator platform consisting of OPNET node and process models. We created new OPNET models for each TinyOS application.

As part of our implementation, we also designed a tool to transform TinyOS NesC application code into OPNET node and process models. These synthesized OPNET models combined OPNET specific code that implemented TinyOS functionality, such as posting tasks, with application specific code. Our tool was designed to conform as much as possible to the existing TinyOS build system and also to maintain compatibility with existing TinyOS programs.

The synthesis process relied on a special generic set of OPNET node and process models for a TinyOS application, which were customized and linked to an OPNET external code module generated from the NesC application code. The customized OPNET models were then copied to the appropriate models directory, and compiled using OPNET command line tools during the build process.

### B. NesC Compilation

All application-specific and TinyOS generic code was compiled to a single C file (the OPNET external code module) by the NesC compiler. In the typical scenario, this step is followed by compilation using the gcc compiler. However, since we used the Windows version of OPNET, OPNET models had to be compiled with the Microsoft compiler. Unfortunately, some of the code generated by the NesC compiler was not compatible with the Microsoft compiler without some tweaking. This tweaking would probably not have been necessary had we used the Unix version of OPNET that uses gcc. However, we believe that code output by the modified NesC compiler is compatible both with gcc and with the Microsoft compiler.

Another challenge of our simulation was enabling multiple instantiations of the same TinyOS application. Each application instantiation needed to have its own data space. By default, when an application is compiled from NesC to C, its variables are represented as global variables. Storing data in global variables is a logical thing to do when compiling for the target sensor platform, but creates problems when compiling for a simulation platform where data from different instantiations could clash over the same variable names.

Our solution to the instantiation problem involved modifying the NesC compiler, and was largely based on the TOSSIM solution to the same problem. The NesC compiler was modified to collect all global variables into a unified structure that was referenced by a global pointer. Each instantiation of a TinyOS application maintained a different structure, and the global pointer would affect context switching by changing its dereferenced structure to correspond with the application being evaluated. In order to ensure that the global pointer variable was set correctly, we made use of *state variables* in the OPNET process models that are ultimately linked to the NesC compiler output. State variables allow different instantiations of a process to have their own data storage space. The OPNET process model code (and its state variables) interfaced with the NesC derived code (and its global pointer variable) using function calls. During application initialization, each instance of the OPNET process model code allocated and kept track of the global data area for the NesC derived code. Then, during the simulation, the OPNET process model code appropriately set the global pointer variable before calling the NesC derived code.

### C. OPNET Process and Node Models

Bridging the OPNET and TinyOS environments required a combination of OPNET platform specific NesC source files and modifications to the NesC compiler itself (See Figure 1). Our overall process flow begins with the compilation of TinyOS application code with a modified NesC compiler that correctly handles global variables to allow for multiple application instantiations. The compiled code is also designed to be compatible with Microsoft's compiler, and it is thereby compiled to an OPNET external code module. Finally, a generic OPNET node and process model is created, customized, and linked to the modified NesC output. In the remainder of this section, we describe some of the details behind the OPNET process and node model components of the simulation.

Recall that a TinyOS application is a collection of NesC components, some of which are platform specific. In order to simulate these components within OPNET, we had to define a platform that translated TinyOS functionality into OPNET functionality. We did this by creating OPNET-specific NesC components that made calls to functions that were defined in an OPNET process model.

The OPNET process models were designed by means of *generic* OPNET models (i.e., they do not correspond to any particular application) that simulate sensor platforms running TinyOS. The build scripts ultimately customize the generic models for each TinyOS application. Technically, the customization only involves changing the names of functions and other identifiers. However, this step is important, as it allows the automated generation of independent OPNET models for each TinyOS application.

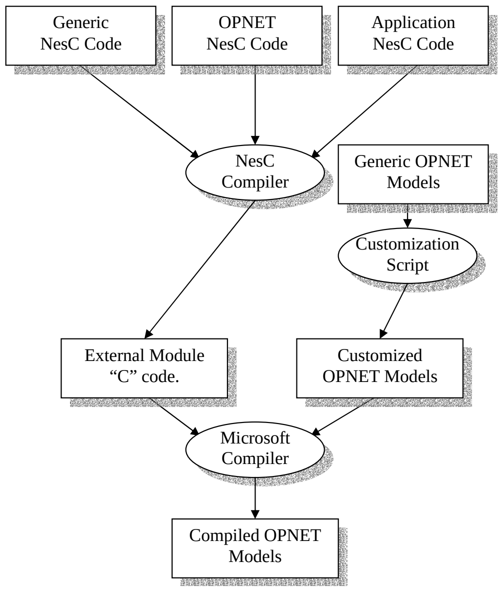
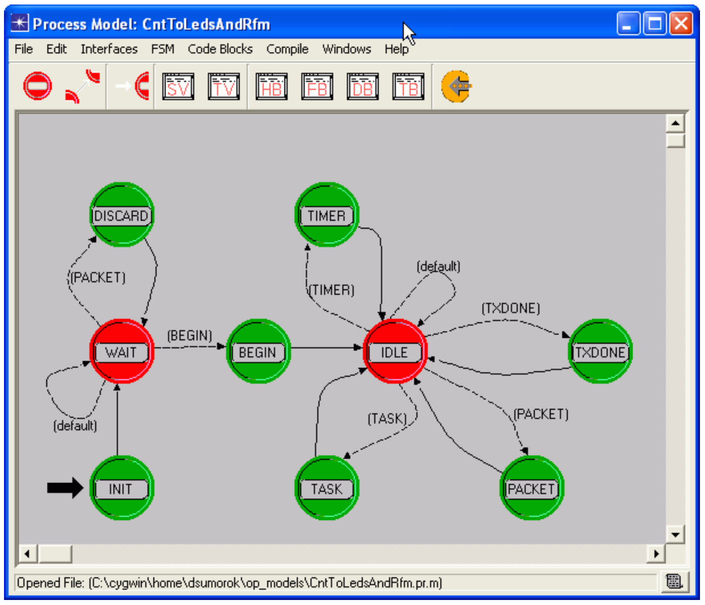


Figure 1

The OPNET process model is responsible for implementing the TinyOS operating system in the OPNET environment. From the point of view of the process model, the TinyOS application is an external code module with function call entry points. Events received by the OPNET process model are processed and sent to the TinyOS application code. The TinyOS application code, in turn, makes function calls back to the process model to cause simulation events to be scheduled.

We constructed the OPNET process model (See Figure 2) to handle TinyOS tasks, as well as the TinyOS Timer, Light Emitting Diode (LED), and Radio interfaces. Due to their small size, many sensors do not have display interfaces, and rely on LEDs to provide visual status information to operators. We simulated this LED interface using OPNET statistics: switching on a LED corresponds to toggling the statistic value from zero to one. This allows us to plot status of the LEDs as a function of time after the simulation (see Figure 4). We implemented tasks and timers by scheduling OPNET local interrupts. The Interface Control Information (ICI) associated with each interrupt kept track of the callback functions associated with the timers and tasks.

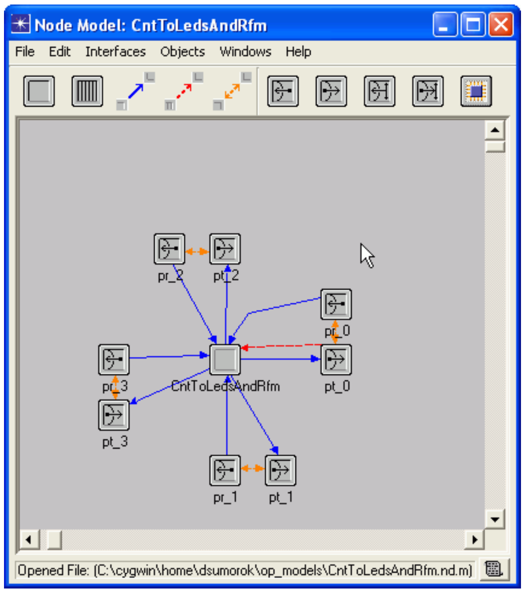


Opened File: (C:\cygwin\home\dsumorok\op_models\CntToLedsAndRfm.pr.m)

Figure 2

The OPNET process model converts TinyOS radio packets into OPNET packets, and then sends them using OPNET streams. For the purposes of this work, we used an OPNET serial line link model to simulate the OPNET radio channel. Such a serial link correctly models propagation and transmission delay of a radio link, but it does not model the interference due to simultaneous transmissions. This link also concretely demonstrates the ability to convert TinyOS packets to and from OPNET packets for the purpose of simulating inter-sensor communication. It is a simple (if potentially time-consuming) exercise to make a more accurate communication model by using the OPNET wireless toolkit to model wireless propagation and the TinyOS Medium Access Protocol (MAC), and we leave this for future work.

As usual, the OPNET process model needs to be part of an OPNET node model. Therefore, in addition to the process model, we constructed a generic node model for TinyOS applications (See Figure 3). The node model contains the OPNET TinyOS process model and some point-to-point transmitters and receivers. It also contains user-configurable attributes for the TinyOS node address, the length of the boot period, and the debug flags. During the simulation, the boot time of the sensor was set to a random variable uniformly distributed between zero and the value of the boot period attribute. This feature mirrors a similar feature in the TOSSIM simulator. The debug flags attributes enable TOSSIM-compatible debugging output.


Node Model: CntToLedsAndRfm
File Edit Interfaces Objects Windows Help
pr_2
pt_2
pr_0
pt_0
pr_3
pt_3
CntToLedsAndRfm
pr_1
pt_1
Opened File: (C:\cygwin\home\dsumorok\op_models\CntToLedsAndRfm.nd.m)
Figure 3

## IV. Simulation Examples

In this section we describe the simulation of two sample TinyOS applications: an LED-based counter and a location-detection service.

### A. Counter to LEDs and Radio

Our first sample application serves to validate our OPNET simulation technique and involves two related TinyOS programs: CntToLedsAndRFM and RfmToLeds. These applications are ideal for testing because they utilize the Timer, LEDs (which come in three colors: red, yellow, and green), and Radio interfaces (38 kbps device) of our sensors, despite being functionally very simple. Furthermore, these applications come as part of the standard TinyOS installation and were not themselves modified in any way to work with OPNET.

The CntToLedsAndRFM program sets a 4 Hz periodic timer, and increments a counter every time the timer expires. Also, when the timer expires, a radio packet containing the counter value is broadcasted, and the three sensor LEDs are set to the three least significant bits of the binary encoding of the counter value. For examples, if the timer value is 5, the yellow, green, and red LEDs are respectively On, Off, and On. If the timer Value is 17, the yellow, green, and red LEDs are respectively Off, Off, and On. Figure 4 shows the LED outputs for this program as a function of time.

The RfmToLeds program acts as the reception counterpart to CntToLedsAndRFM. It listens for radio packets containing broadcasted counter values and updates the LEDs of the receiving sensor to reflect the three least significant bits of these values.

Our simulation network topology is depicted in Figure 5. The topology consists of four nodes, with radio links between nodes 0 and 2, and between nodes 0 and 3. In our simulation, only node 0 ran the CntToLedsAndRFM program. The other three nodes ran the RfmToLeds application. For our simulation topology, the LEDs of nodes 2 and 3 are expected to mirror the LEDs of node 0, while the LEDs of node 1 are not. Figure 6 shows that the LED outputs of nodes 0, 1, 2, and 3 behave as expected.

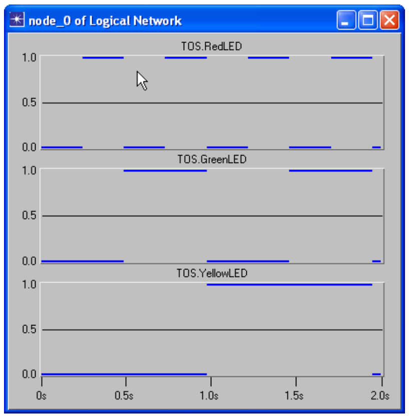
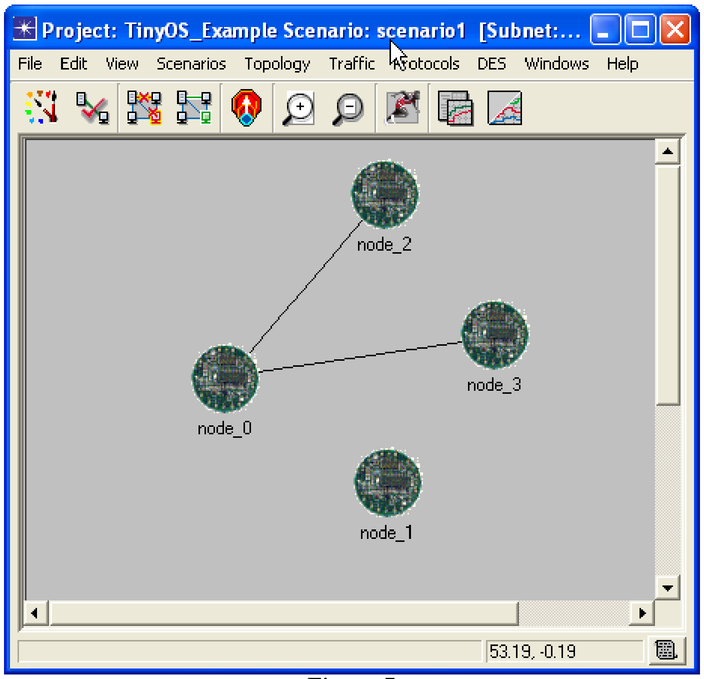


Figure 4


Project: TinyOS_Example Scenario: scenario1 [Subnet:...
File Edit View Scenarios Topology Traffic Protocols DES Windows Help
node_2
node_3
node_0
node_1
53.19, -0.19
Figure 5

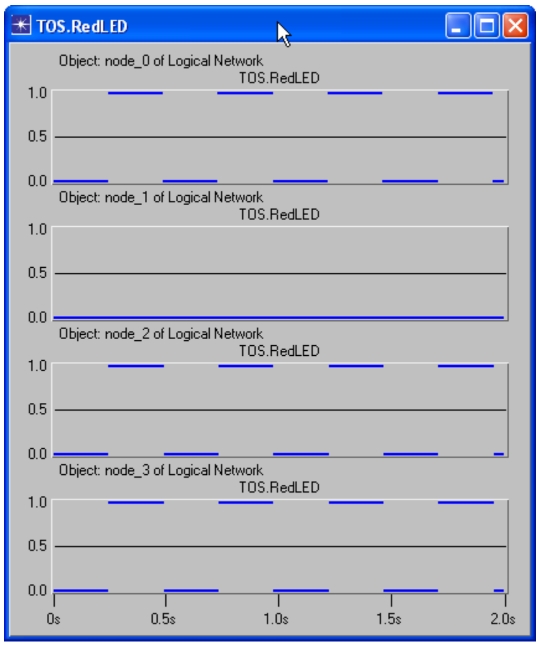


Figure 6

## B. Location Detection

Our second simulation example demonstrates a TinyOS application used for location detection. This application can be used in a system that enables users to determine their location in areas where GPS is not available (such as indoor settings) [3]. To set up the system, small sensors are scattered in fixed locations throughout an area. The sensors periodically broadcast beacons that identify themselves. By design, each sensor will receive beacons from unique set of other sensors.

A hypothetical user of the system would carry a small sensing device that listens to the various transmitted beacons. If the user is close to a fixed-position sensor, the user will receive beacons from the same sources as the fixed position sensor. Therefore, assuming that the user has information on the fixed position of each sensor, she could then identify her location based on the unique set of beacons that were heard.

We implemented this location detection system with a TinyOS application and simulated it using our framework on OPNET. Figure 7 depicts the simulated network topology. Four fixed-position sensors (nodes 0 though 3) are arranged in a line topology, and the user (node 4) is very close to the sensor in fixed position 3. Since the user receives beacons from sensors 1, 2, and 3, she should identify her location as the location of node 2.

Unlike the LED-counter example, this example uses the TOSSIM-compatible debug output that sends debug output to the console. The flexible debug system allows debug output to be enabled or disabled for different parts of a TinyOS application. For example, the AM debug attributes allows debugging of the radio component. The USR1, USR2, and USR3 are reserved for higher-level TinyOS application components. Figure 8 shows the interface being used to enable the USR1 debug output for node 4.

The debug output of a 20 second simulation is shown in Figure 9. The Figure shows that the user identified her location as that of node 2. It also shows that there were zero differences between what the user received and what she would expect to receive if she were at the location of node 2. The location detection algorithm from [3] picks the location that minimizes this "differences" number. Figures 10 and 11 respectively show the user at location of node 1, and the results of the simulation. As expected, the user identified her location correctly. It is possible to use some of OPNETs more sophisticated analysis tools to experiment with the system, deriving, for example, error probabilities in various topologies and transmitter configurations. We leave the results of such analysis to future work.

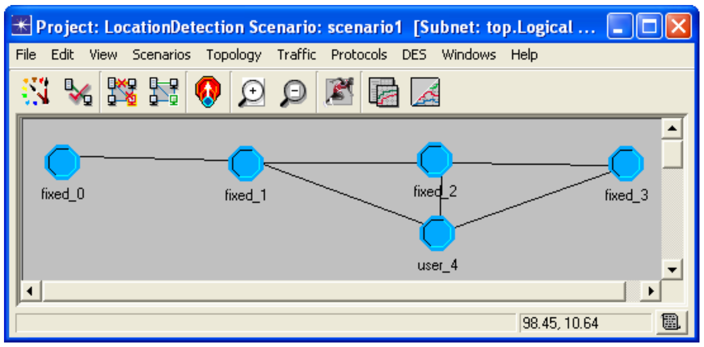
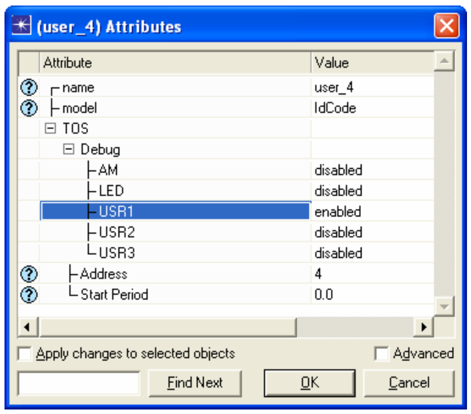
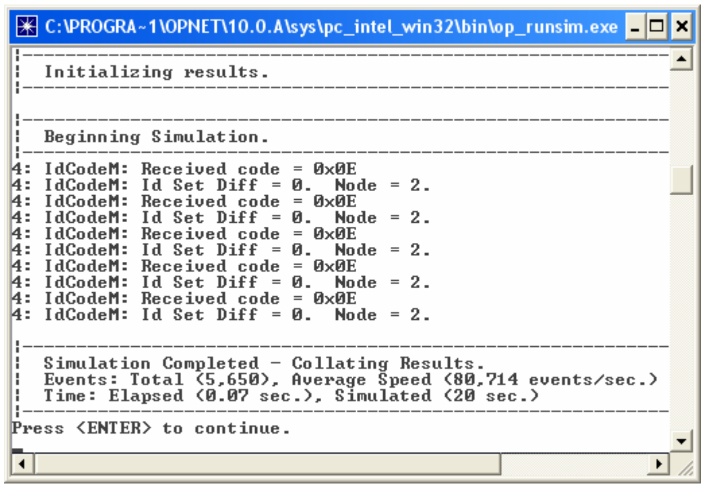


Figure 7



| Attribute | Value |
|---|---|
| name | user_4 |
| model | IdCode |
| TOS | |
| Debug | |
| AM | disabled |
| LED | disabled |
| USR1 | enabled |
| USR2 | disabled |
| USR3 | disabled |
| Address | 4 |
| Start Period | 0.0 |

Figure 8


C:\PROGRA~1\OPNET\10.0.A\sys\pc_intel_win32\bin\op_runsim.exe

```
 Initializing results.

 Beginning Simulation.
4: IdCodeM: Received code = 0x0E
4: IdCodeM: Id Set Diff = 0.  Node = 2.
4: IdCodeM: Received code = 0x0E
4: IdCodeM: Id Set Diff = 0.  Node = 2.
4: IdCodeM: Received code = 0x0E
4: IdCodeM: Id Set Diff = 0.  Node = 2.
4: IdCodeM: Received code = 0x0E
4: IdCodeM: Id Set Diff = 0.  Node = 2.
4: IdCodeM: Received code = 0x0E
4: IdCodeM: Id Set Diff = 0.  Node = 2.

 Simulation Completed - Collating Results.
 Events: Total (5,650), Average Speed (80,714 events/sec.)
 Time: Elapsed (0.07 sec.), Simulated (20 sec.)
Press <ENTER> to continue.
```

Figure 9

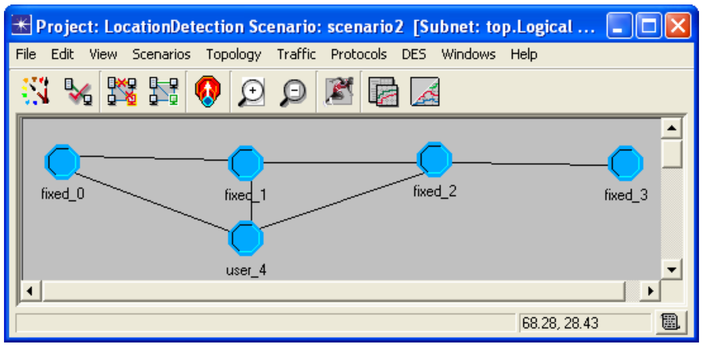
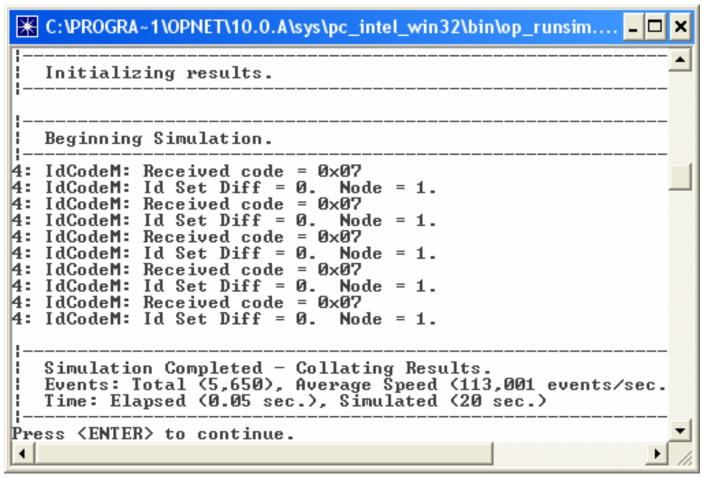

Project: LocationDetection Scenario: scenario2 [Subnet: top.Logical ...
File Edit View Scenarios Topology Traffic Protocols DES Windows Help
fixed_0
fixed_1
fixed_2
fixed_3
user_4
68.28, 28.43

Figure 10


C:\PROGRA~1\OPNET\10.0.A\sys\pc_intel_win32\bin\op_runsim....

```
|------------------------------------------------------------------
|  Initializing results.
|------------------------------------------------------------------

|------------------------------------------------------------------
|  Beginning Simulation.
|------------------------------------------------------------------
4: IdCodeM: Received code = 0x07
4: IdCodeM: Id Set Diff = 0.  Node = 1.
4: IdCodeM: Received code = 0x07
4: IdCodeM: Id Set Diff = 0.  Node = 1.
4: IdCodeM: Received code = 0x07
4: IdCodeM: Id Set Diff = 0.  Node = 1.
4: IdCodeM: Received code = 0x07
4: IdCodeM: Id Set Diff = 0.  Node = 1.
4: IdCodeM: Received code = 0x07
4: IdCodeM: Id Set Diff = 0.  Node = 1.

|------------------------------------------------------------------
|  Simulation Completed - Collating Results.
|  Events: Total (5,650), Average Speed (113,001 events/sec.
|  Time: Elapsed (0.05 sec.), Simulated (20 sec.)
|------------------------------------------------------------------
Press <ENTER> to continue.
```

Figure 11

## V. Conclusion

We have demonstrated the viability of using OPNET to simulate TinyOS applications. Due to the simplicity of the target sensor platforms, it is very difficult to debug a TinyOS platform without a simulation tool. With the exception of TOSSIM, there are currently no other simulation options for TinyOS. While TOSSIM is adequate for some simulation work (and is a free tool), OPNET's mature simulation library and data management and scenario management tools make OPNET a preferable tool for large, complicated, or heterogeneous simulations.

Our specific implementation provides OPNET code supporting the timer, radio, and LED TinyOS components, and models wireless communication as a full-duplex serial link. As such, it is only a proof of concept; a complete simulation would require implementation of all the TinyOS sensor components together with a more realistic communication model using OPNET's wireless toolkit to model RF interference and TinyOS's MAC protocol. We believe that such extensions should be fairly straightforward and would not require any further changes to the NesC compiler.

An important feature of our simulation technique is that it uses the shared code approached and shares the application code between the target sensor and simulation environments. This allows target code to be simulated exactly at it would be used on the target sensors, at the cost of requiring some hacking of the build process. Our simulation tool can thus be used to develop and verify distributed algorithms on sensor networks.

Finally, we have demonstrated our tool on three TinyOS applications (two of which were simulated together). Due to their shared-code nature, these applications were also built for and ran on the Crossbow Mica2dot target sensor platform. These simple examples show that OPNET has the potential to be an extremely useful, and effective, tool for developing and implementing TinyOS applications for sensor networks.

The source codes of this project are available at `http://nislab.bu.edu/opnet_tinyos.html`

**Acknowledgments**

This work was supported in part by the US National Science Foundation under grants CAREER ANI-0132802, CAREER CCR-0133521, and ANI-0240333.

## References

[1] Philip Levis, Sam Madden, David Gay, Joe Polastre, Robert Szewczyk, Alec Woo, Eric Brewer and David Culler, "The Emergence of Networking Abstractions and Techniques in TinyOS," *Proceedings of the First USENIX/ACM Symposium on Networked Systems Design and Implementation, 2004.*

[2] Philip Levis, Nelson Lee, Matt Welsh, David Culler, "TOSSIM: Accurate and Scalable Simulation of Entire TinyOS Applications," *Proceedings of the First International Conference on Embedded Networked Sensor Systems, 2003.*

[3] Saikat Ray, David Starobinski, Ari Trachtenberg and Rachanee Ungrangsi, "Robust Location Detection with Sensor Networks," *IEEE Journal on Selected Areas in Communication (Special Issue on Fundamental Performance Limits of Wireless Sensor Networks), in press.*