TOSSIM: Accurate and Scalable Simulation of Entire TinyOS Applications

Philip Levis[†‡], Nelson Lee[†], Matt Welsh[♯], and David Culler[†‡]
{pal,culler}@cs.berkeley.edu, mdw@eecs.harvard.edu

[†]Computer Science Division
University of California
Berkeley, California

[‡]Intel Research: Berkeley
Intel Corporation
Berkeley, California

[♯]Harvard University
Cambridge, Massachusetts

Abstract

Accurate and scalable simulation has historically been a key enabling factor for systems research. We present TOSSIM, a simulator for TinyOS wireless sensor networks. By exploiting the sensor network domain and TinyOS's design, TOSSIM can capture network behavior at a high fidelity while scaling to thousands of nodes. By using a probabilistic bit error model for the network, TOSSIM remains simple and efficient, but expressive enough to capture a wide range of network interactions. Using TOSSIM, we have discovered several bugs in TinyOS, ranging from network bit-level MAC interactions to queue overflows in an ad-hoc routing protocol. Through these and other evaluations, we show that detailed, scalable sensor network simulation is possible.

1. INTRODUCTION

Development of the right simulation tools has been a key step in systems research progress for several areas. In general, simulation can provide a way to study system design alternatives in a controlled environment, explore system configurations that are difficult to physically construct, and observe interactions that are difficult to capture in a live system.

A debate often emerges over how much fidelity is required to observe critical phenomena, since simulation cost typically increases quickly with the level of detail. Several times in the past, we have seen a tool that captured the essential elements of the area of study; these tools dramatically accelerated progress in emerging research areas. For example, SimOS [24] used binary rewriting techniques to provide enough detail and yet enough speed and flexibility to allow workload-driven evaluation of machine architectures and operating systems for multiprocessors by running whole programs. `ns-2` [2], using an object approach, provided a common toolbox for studying a wide range of network protocols and implementations against various traffic models. Proteus [5] provided broad feedback on the design of parallel programs.

We believe that developing the right simulation tools will be especially important for sensor network research. Not only does this area address issues of architecture, operating systems, network protocols, and distributed in-network processing, it does so in a highly application-specific manner. Additionally, sensor network systems are closely connected to the physical world; this adds noise, variation, and uncertainty to execution. We need simulation tools that will allow studying entire applications, including the operating system and the network stack. These tools must handle very large numbers of nodes with sufficient detail that the subtle interactions caused by unpredictable interference and noise can be observed.

Many researchers have concluded that it is intractable to address this collection of requirements simultaneously and have, instead, employed very abstract simulations for studying behavior at scale and detailed simulations for individual nodes. `ns-2` has been adapted to provide a rough analog of sensor network behavior to evaluate protocols against synthetic workloads [4, 23, 29].

Usually, the algorithm representations evaluated through simulation are quite different from those of a real implementation. While these simulation approaches are valuable, the lesson from SimOS, Proteus, and numerous similar tools, is that it is very important to have a vehicle that allows study of the actual implementation of algorithms at scale running real applications [5, 24]. This is especially true in new areas where there are not years of experience in what can be safely abstracted without sacrificing accuracy. Also, if simulation and deployment programming environments are very different, having an accurate simulator can inhibit deployments: one has to implement algorithms twice.

In this paper, we investigate how we might exploit the characteristics of the sensor network domain to obtain a scale, fidelity, and completeness that would be intractable in a general purpose context. Specifically, we consider the simulation of TinyOS sensor networks. TinyOS, developed at UC Berkeley, is a sensor network operating system that runs on custom 'mote' hardware [12]. Over one hundred groups worldwide use it, including several companies within their products. Many published systems [8, 14], protocols [11, 20, 26, 28, 31] and applications [15, 17, 18, 29] have been built on TinyOS. Providing a TinyOS simulation environment would help all of these efforts.

A TinyOS simulator has four key requirements:

- **Scalability:** The simulator must be able to handle large networks of thousands of nodes in a wide range of configurations. The largest TinyOS sensor network that has ever been deployed was approximately 850 nodes; the simulator must be able to handle this and the much larger systems of the future.
- **Completeness:** The simulator must cover as many system interactions as possible, accurately capturing behavior at a wide range of levels. Algorithm and network protocol simulations are helpful, but the reactive nature of sensor networks requires simulating complete applications.
- **Fidelity:** The simulator must capture the behavior of the network at a fine grain. Capturing subtle timing interactions on a mote and between motes is important both for evaluation and testing. The simulator must reveal *unanticipated* interactions, not just those a developer suspects.

- **Bridging:** The simulator must bridge the gap between algorithm and implementation, allowing developers to test and verify the code that will run on real hardware. Often, algorithms are sound but their implementations are not.

To investigate whether these four requirements could be met at once, we built TOSSIM, a discrete event simulator. Compiling unchanged TinyOS applications directly into its framework, TOSSIM can simulate thousands of motes running complete applications. By only replacing a few low-level TinyOS systems that touch hardware, it can capture mote behavior at a very fine grain, allowing a wide range of experimentation. Taking advantage of several characteristics of TinyOS sensor networks enables achieving this fidelity at scale. As individual motes have limited storage and CPU resources, TOSSIM can simulate many of them at once. TinyOS's event-driven execution maps well to a discrete event simulation, requiring a very simple simulation engine. Whole-system compilation can be used to integrate entire applications into the simulation infrastructure. By seamlessly supporting the PC-based TinyOS tool-chain, TOSSIM allows developers to easily transition between running an application on motes and in simulation. To improve its usefulness to TinyOS developers, TOSSIM has mechanisms that allow GUIs to provide detailed visualization and actuation of a running simulation.

This paper has three contributions. First, it addresses a critical need in sensor network development: scalable, controlled and accurate simulation of these non-deterministic systems. Second, by taking advantage of characteristics of the sensor network domain, TOSSIM provides insight on how to resolve the tensions between scalability, completeness, fidelity, and bridging. Finally, by describing and evaluating a concrete implementation, TOSSIM, this paper shows that such a simulator can be built.

Section 2 summarizes the aspects of TinyOS relevant to simulation. Section 3 presents TOSSIM's architecture, implementation, and capabilities. Section 4 evaluates TOSSIM's accuracy and performance. We address relevant prior work in Section 5, discuss our results and future work in Section 6, and conclude with Section 7.

2. TINYOS

The event-driven nature of sensor networks means that testing an individual mote is insufficient. Programs must be tested at scale and in complex and rich conditions to capture a wide range of interactions. Deploying hundreds of motes is a daunting task; the focus of work shifts from research to maintenance, which is time-consuming due to the failure rate of individual motes. A simulator can deal with these difficulties, by providing controlled, reproducible environments, by enabling access to tools such as debuggers, and by postponing deployment until code is well tested and algorithms are understood. In this section, we provide background on TinyOS, presenting points of its design that are of particular interest for simulation.

TinyOS is an operating system specifically designed for sensor networks. It has a component-based programming model, provided by the nesC language [9], a dialect of C. TinyOS is not an OS in the traditional sense. It is a programming framework for embedded systems and set of components that enable building an application-specific OS into each application.

A TinyOS program is a graph of components, each of which is an independent computational entity. Each TinyOS component has a frame, a structure of private variables that can only be referenced by that component. Components have three computational abstractions: *commands*, *events*, and *tasks*. Commands and events are

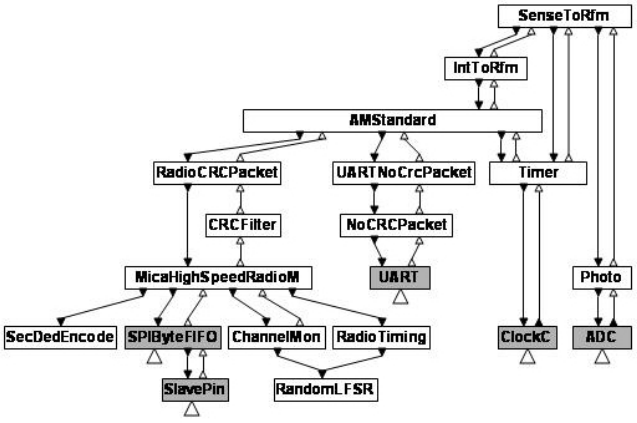


Figure 1: Simplified nesC Component Graph of the SenseToRfm TinyOS Application. ***The networking stack dominates the left side, while timers and sensors are on the right.***

mechanisms for inter-component communication, while tasks are used to express intra-component concurrency.

A *command* is typically a request to a component to perform some service, such as initiating a sensor reading, while an *event* signals the completion of that service. Events may also be signaled asynchronously, for example, due to hardware interrupts or message arrival. From a traditional OS perspective, commands are analogous to downcalls and events to upcalls. Commands and events cannot block: rather, a request for service is *split phase* in that the request for service (the command) and the completion signal (the corresponding event) are decoupled. The command returns immediately and the event signals completion at a later time.

Rather than performing a computation immediately, commands and event handlers may post a *task*, a function executed by the TinyOS scheduler at a later time. This allows commands and events to be responsive, returning immediately while deferring extensive computation to tasks. While tasks may perform significant computation, their basic execution model is run-to-completion, rather than to run indefinitely. This allows tasks to be much lighter-weight than threads. Tasks represent internal concurrency within a component and may only access that component's frame. The TinyOS task scheduler uses a non-preemptive, FIFO scheduling policy.

TinyOS abstracts all hardware resources as components. For example, calling the `getData()` command on a sensor component will cause it to later signal a `dataReady()` event when the hardware interrupt fires. While many components are entirely software-based, the combination of split-phase operations and tasks makes this distinction transparent to the programmer. For example, consider a component that encrypts a buffer of data. In a hardware implementation, the command would instruct the encryption hardware to perform the operation, while a software implementation would post a task to encrypt the data on the CPU. In both cases an event signals that the encryption operation is complete.

Figure 1 contains a simplified component graph of a simple application, SenseToRfm, that periodically samples the light sensor and broadcasts the value in a network packet. The nodes of the graph are components and the edges are interfaces. The network stack dominates the left side of the graph, while the sensing stack is on the right side. `ADC`, `ClockC`, `UART`, `SlavePin` and

```
task void packetReceived(){
  TOS_MsgPtr tmp;
  state = IDLE_STATE;
  tmp = signal Receive.receive((TOS_Msg*)rec_ptr);
  if(tmp != 0) rec_ptr = tmp;
  call ChannelMon.startSymbolSearch();
}
```

Figure 2: TinyOS MicaHighSpeedRadioM Receive Task Code

`SpiByteFifo` are example hardware abstraction components. Upward arrows represent event flow, downward arrows represent command flow, and large triangles represent interrupts.

Figure 2 shows sample nesC code for a packet reception task, taken from `MicaHighSpeedRadioM`. The task signals the `receive` event on the `Receive` interface, calls the `startSymbolSearch` command on the `ChannelMon` interface, and uses two frame variables, `rec_ptr` and `state`. It uses the return value of the receive event to perform a buffer swap for its next receive buffer.

TinyOS commands and events are very short, due to limited code space and a finite state machine style of decomposition. The rich event processing model means an event or command call path can traverse several components. Understanding all of the possible control flows can be difficult, especially when many executions are asynchronous. Our experience with TinyOS has shown that while failure bugs are usually quickly found, bugs that produce operational but aberrant behavior, such as low channel utilization, are far more difficult to discover; on the surface they are indistinguishable from transient network conditions. Additionally, while individual components are usually sound, they are written as separate entities; this ignores the possible interactions that can result from complex compositions.

The TinyOS component model allows us to easily change the target platform from mote hardware to simulation by only replacing a small number of low-level components. The event-driven execution model can be exploited for efficient event-driven simulation, and the whole program compilation process can be re-targeted for the simulator's storage model and native instruction set. As individual mote resources are very small, we can simulate many of them within the simulator's address space. The static component memory model of TinyOS simplifies state management for these large collections. Setting the right level of simulation abstraction can accurately capture the behavior and interactions of TinyOS applications; the challenge is to remain scalable and efficient. In the next section, we present TOSSIM, our solution to this challenge.

3. TOSSIM

TOSSIM captures the behavior and interactions of networks of thousands of TinyOS motes at network bit granularity. Figure 3 shows a graphical overview of TOSSIM. The TOSSIM architecture is composed of five parts: support for compiling TinyOS component graphs into the simulation infrastructure, a discrete event queue, a small number of re-implemented TinyOS hardware abstraction components, mechanisms for extensible radio and ADC models, and communication services for external programs to interact with a simulation.

TOSSIM takes advantage of TinyOS's structure and whole system compilation to generate discrete-event simulations directly from TinyOS component graphs. It runs the same code that runs on sensor network hardware. By replacing a few low-level components (e.g., those shaded in Figure 3), TOSSIM translates hardware interrupts into discrete simulator events; the simulator event queue delivers the interrupts that drive the execution of

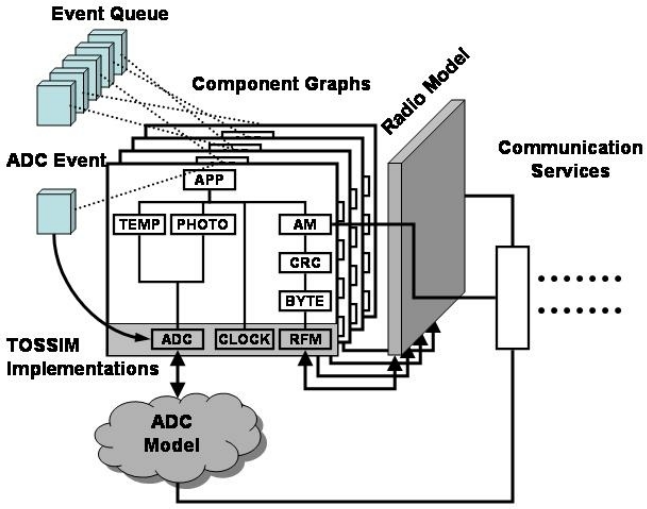


Figure 3: TOSSIM Architecture: Frames, Events, Models, Components, and Services

a TinyOS application. The remainder of TinyOS code runs unchanged.

TOSSIM uses a very simple but surprisingly powerful abstraction for its wireless network. The network is a directed graph, in which each vertex is a node, and each edge has a bit error probability. Each node has a private piece of state representing what it hears on the radio channel. This abstraction allows testing under perfect transmission conditions (bit error rate is zero), can capture the hidden terminal problem (for nodes a,b,c, there are edges (a, b) and (b, c) but no edge (a, c)), and can capture many of the different problems that can occur in packet transmission (start symbol detection failure, data corruption, etc.).

The simulator engine provides a set of communication services for interacting with external applications. These services allow programs to connect to TOSSIM over a TCP socket to monitor or actuate a running simulation. Details of the ADC and radio models, such as readings and loss rates, can be both queried and set. Programs can also receive higher level information, such as packet transmissions and receptions or application-level events.

TOSSIM supports the TinyOS tool-chain, making the transitions between simulated and real networks easy. Compiling to native code allows developers to use traditional tools such as debuggers in TOSSIM. As it is a discrete event simulation, users can set debugger breakpoints and step through what is normally real-time code (such as packet reception) without disrupting operation. It also provides mechanisms for other programs to interact and monitor a running simulation; by keeping monitoring and interaction external to TOSSIM, the core simulator engine remains very simple and efficient.

3.1 Compiler Support

Closing the gap between simulation and deployment, we modified the nesC compiler (`ncc`) to support compilation from TinyOS component graphs into the simulator framework. With the change of a compiler option, an application can be compiled for simulation instead of mote hardware, and vice versa. This compiler integration is also a key element to providing scalability and fidelity simultaneously. In the TinyOS memory model, component frames

nesC TinyOS Code	Mote C Code	TOSSIM C Code

`result_t StdControl.init(){`
`  state = 0;`
`  return SUCCESS;`
`}`
	`result_t Counter$StdControl$init(void){`
	`  Counter$state = 0;`
	`  return SUCCESS;`
	`}`
		`result_t Counter$StdControl$init(void){`
		`  Counter$state[tos_state.current_node] = 0;`
		`  return SUCCESS;`
		`}`

Figure 4: nesC Code from Counter Component with Resulting Mote and TOSSIM C Code

constitute the entire in-memory storage of a program. In TOSSIM, component variables are replaced with an array of them, one copy for each simulated mote. All frame variable references index into the variable's array, based on which mote is currently "running." Figure 4 shows a snippet of TinyOS code, the resulting C code for a mote, and the resulting code for simulation. This C code is compiled to the simulator instruction set rather than the mote instruction set.

3.2 Execution Model

A simulator event queue sits at the core of TOSSIM. Interrupts are modeled through simulator events, which are distinct from TinyOS events. A simulator event calls an interrupt handler in hardware abstraction components. The interrupt handler signals TinyOS events and calls TinyOS commands, just as it would on a mote. These TinyOS events and commands post tasks and cause further simulator events to be enqueued, driving execution forward.

TOSSIM keeps time at mote instruction clock cycle granularity: 4MHz. All simulator events are timestamped and processed in global time order. When the simulation starts, motes are given randomized and staggered times at which their boot sequence is called, to prevent artificial synchronization. Every TOSSIM event occurs at a specific virtual time and runs instantly in regards to the virtual clock. There is a notion of delta time[1] for event execution, similar to VHDL simulators [1]. This allows code within a handler to have a temporal ordering, as well as providing a total order on events occurring at a certain virtual instant.

Each simulator event is associated with a specific mote. After running a simulator event, the scheduler executes the tasks on that mote's TinyOS task queue, following the FIFO run-to-completion model of the normal TinyOS scheduler. When the core loop executes a simulator event, it sets global state to indicate which mote is currently running, which is used by frame variable references. Once control transfers outside of a hardware abstraction component, TOSSIM executes standard TinyOS code.

After each simulator event, TOSSIM runs the task queue of the mote associated with the event until the queue is empty. Following the TinyOS model, TOSSIM runs each task to completion, without preemption. This concurrency model is identical to the TinyOS one, with one exception: interrupt handlers, being discrete events, cannot preempt tasks. Section 6 discusses this in greater depth.

3.3 Hardware Emulation

TinyOS abstracts each hardware resource as a component. By replacing a small number of these components, TOSSIM emulates the behavior of the underlying raw hardware. These include the Analog-to-Digital Converter (ADC), the Clock, the transmit strength variable potentiometer, the EEPROM, the boot sequence component, and several of the components in the radio stack.

The low level components that abstract sensors or actuators also provide the connection point for the simulated environment. For example, the simulated ADC still provides the standard split phase interface to the application graph; calling the `getData()` command on the TOSSIM ADC component will cause it to signal a `dataReady()` event later, just as mote hardware does. Under simulation, however, an external function provides the ADC reading.

The radio transceiver is effectively a sensor/actuator pair, but it is handled specially. A TOSSIM network model is a shared resource that captures the inter-mote interactions. Each bit transmission engages the model, which changes the state of the channel observed by receive events on other nodes. By modeling at the bit level, TOSSIM allows users to develop, test, and evaluate low level network protocols in addition to high level applications, and everything in-between. We discuss in depth how TOSSIM models the network later, in Section 3.7. By controlling these models, developers can simulate a wide range of environments. Section 3.6 describes one tool for doing so.

3.4 Sample Execution

Figure 5 contains a sample execution that demonstrates the workings of TOSSIM. A simple program, `SenseToLeds`, is running on a single mote. This program has a 1Hz timer that samples the light sensor and displays the three high order bits of the reading on the mote LEDs. Since simulator time is kept in terms of a 4MHz clock, the timer events are four million ticks apart, and the 50μs ADC capture takes 200 ticks between request and interrupt. In Figure 5 TOSSIM-specific execution is shown in plain text, while unchanged TinyOS execution is shown in italics. This sample execution shows how a small number of TOSSIM events results in the series of TinyOS events and commands comprising an entire (albeit very simple) application.

3.5 Radio Models

TOSSIM provides mechanisms for TinyOS developers to choose the accuracy and complexity of the radio model as necessary for their simulations. The models are external to the simulator, which can remain simple and efficient.

TOSSIM support consists of a directed graph of bit error probabilities. Each edge (u, v) in the graph represents the error rate when mote u sends to v, and is distinct from the edge (v, u); this allows accurate simulation of asymmetric links. Bit errors are independent. Link probabilities can be specified by the user and changed at runtime. Transmission events propagate to the simulated input channel of each connected node. Each mote has its own local view of the network channel. The network does not model RF cancellation. If a bit error occurs, TOSSIM flips the transmitted bit.

As an example, consider mote T transmitting to mote R on an error-free channel. On each of its bit events, T transmits a 0 or 1. This transmission modifies internal state of R, representing what it hears over the air. On each of its bit events, R reads this state, and passes the heard bit up to a TinyOS component.

TOSSIM also has two built-in special case radio-models for what we have observed to be common needs. The first, `simple`, places all of the motes in a single radio cell, with error-free transmission. `simple` is very useful when testing protocols for correctness in the

[1] An infinite amount of delta time adds up to zero virtual time. That is, events do not occur simultaneously, being ordered in delta time, but also do not take any virtual time to execute [3].

Time (4MHz ticks)	Action
3987340	Simulator event is dequeued and handled at time 3987340.
	The clock interrupt handler is called, signaling the application Timer event.
	The application's Timer handler requests a reading from the ADC.
	The ADC component puts a simulator ADC event on the queue with timestamp 3987540.
	The interrupt handler completes; the clock event re-enqueues itself for the next tick.
3987540	Simulator ADC event is dequeued and handled at time 3987540.
	The ADC interrupt handler is called, signaling an ADC ready event with a sensor value.
	The application event handler takes the top three bits and calls LEDs commands.
	The ADC interrupt handler completes.
7987340	Simulator event is dequeued and handled at time 7987340.
	The clock interrupt handler is called, signaling the application Timer event.
	. . . execution continues as above

Figure 5: Sample Execution

absence of any multi-hop or hidden terminal behavior. The second, `static`, is a static undirected graph of error-free connections; this allows testing protocol correctness in the multihop case.

3.6 Communication Services

TOSSIM provides mechanisms that allow PC applications to drive, monitor, and actuate simulation by communicating with TOSSIM over TCP/IP. Drawing on the abstractions of TinyOS, the simulation-application protocol is a command/event interface. TOSSIM signals events to applications, providing data on a running simulation. Examples of events sent from TOSSIM are debug messages added by developers in TinyOS source code, radio and UART packets sent, and sensor readings. Applications call commands on TOSSIM to actuate a simulation and modify its internal state. Commands include operations to change radio link probabilities and sensor reading values, turn motes on and off, and inject radio and UART packets. The communication protocol is abstract so that developers can write their own systems that hook into TOSSIM in new ways, if needed. The monitoring/actuation hooks and statements are removed when compiling for a mote.

3.6.1 TinyViz

TinyViz, the TOSSIM visualization tool, illustrates the capabilities of TOSSIM's communication services. TinyViz is a Java-based graphical user interface for TOSSIM, allowing simulations to be visualized, controlled, and analyzed. TinyViz provides visual feedback on the simulation state and mechanisms for controlling the running simulation, e.g., modifying ADC readings and radio loss probabilities. TinyViz provides a plugin interface allowing developers to implement their own application-specific visualization and control code within the TinyViz engine. Figure 6 shows a screenshot.

The core TinyViz engine, by itself, does very little besides managing the event/command interface to TOSSIM. Users interact with a simulation by loading *plugins* that provide desired functionality. The TinyViz engine publishes TOSSIM events to loaded plugins. This allows, for example, a network plugin to visualize network traffic as motes receive messages. Plugins can also send commands to TOSSIM, actuating a simulation. For example, when a user turns off a mote in the visualization, the control plugin sends the corresponding power off command to TOSSIM.

TinyViz has a set of default plugins that provide basic debugging and analysis capabilities. Two of these, the network and control plugins, were described above. Further examples include a plugin that displays in list format all debug messages and another that graphically displays the data in radio and UART packets. A sensor plugin that displays mote sensor values in the GUI allows the user to set individual mote sensor values during simulation; this plugin can be extended to more complicated sensor models. A radio model plugin changes radio connectivity based on distances between motes in the GUI and graphically displays link probabilities, providing basic mechanisms for experimenting with how networks behave under change.

The basic TinyViz plugins are simple, but provide visualizations rich enough to debug and analyze a sensor network. The debug message plugin can examine internal mote state, and a breakpoint plugin can pause TOSSIM on user-specifiable events, so a user can drop into a debugger. In larger contexts, routing pattern visualizations can reveal algorithm behavior.

Using the simple models provided by TOSSIM, developers can write more powerful TinyViz plugins. For example, one can model in-network obstructions such as metal barriers by changing bit error rates. A plugin can model failures by turning motes off at scripted times. Plugins can also use simulation data to examine and analyze application behavior. Using the communication services of TOSSIM, TinyViz allows a user to take an omniscient view of a large network, examining internal mote in a running simulation.

3.7 Data Link Layer

The most complex system TinyOS provides is its networking stack. Composed of 12 components, the networking stack uses CSMA and single error correction/double error detection data encoding with a full-packet CRC. As perhaps the single most important shared resource of a sensor network, it deserves special attention in a simulator. In order for TOSSIM to be useful for studying data link and physical protocols and accurately capture interactions between applications and low-level protocols, it must simulate the behavior of the TinyOS networking stack at high fidelity.

Emulating the behavior of the hardware at the component level offers insight into the subtle interactions of the networking stack not captured in other simulations, bridging algorithmic simulation and implementation. In contrast, emulating the stack at the packet-level does not allow observations of the delicate interactions in mote communication. Furthermore, it is not clear that emulating the actual radio hardware provides additional insight into the behavior of the stack as our implementation does, while such an implementation would significantly reduce the simulator's scalability.

We provide a detailed description of the TinyOS protocol stack, followed by an explanation of how TOSSIM simulates it. The description illustrates the importance of bit-level simulation for capturing the complexity of the TinyOS networking stack. Some of our evaluations in Section 4 illustrate how high-level protocol failures can be due to interactions with very low-level data-link protocol details; these failures can only be discovered by capturing the network at a high fidelity. Instead of worrying about modeling each possible error in the networking stack (e.g., detection failure, encoding failure, synchronization failure, acknowledgment failure, etc.), a developer need only provide a single probability.

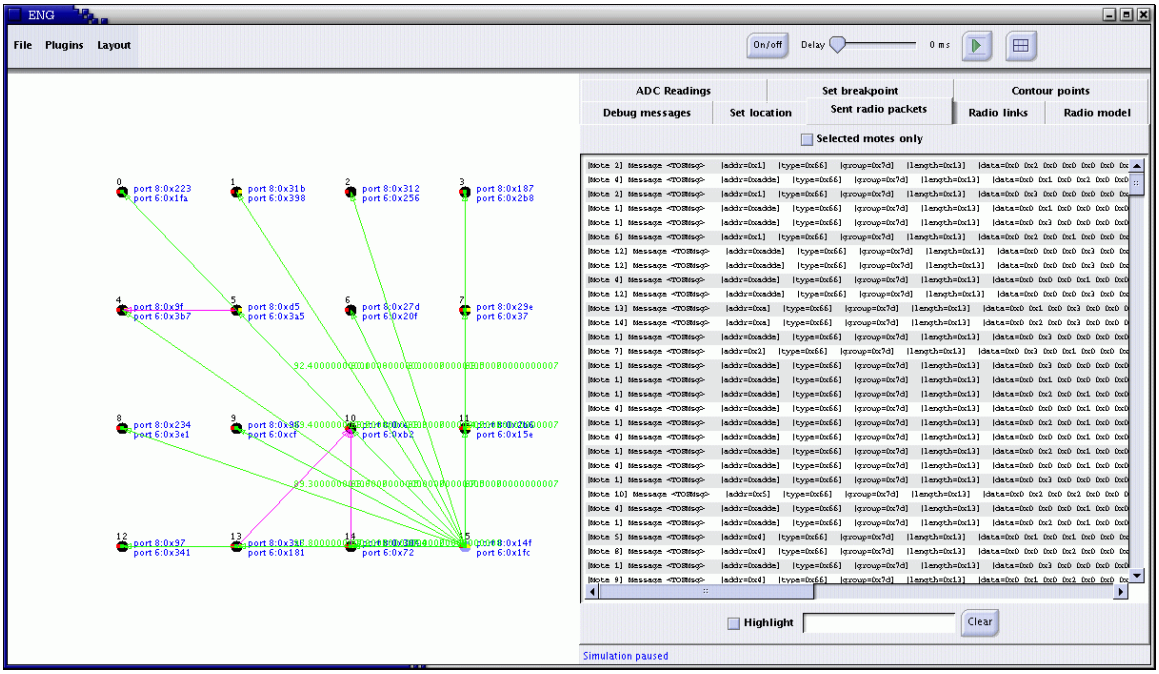


Figure 6: TinyViz connected to TOSSIM running an object tracking application. ***The right panel shows sent radio packets, the left panel exhibits radio connectivity for mote 15 and network traffic. The arrows represent link quality and packet transmissions.***

3.7.1 TinyOS Networking: AM and Below

The TinyOS packet abstraction is an Active Message [25]. AM packets are an unreliable data link protocol, and the TinyOS network stack handles media access control and single hop packet transmission. Active Messages provide precise timestamps as well as synchronous data-link acknowledgments. TinyOS provides a namespace of up to 256 AM message types, each of which can be associated with a separate software handler.

Figure 7 shows the different phases of packet transmission and reception. The sender first enters a CSMA delay period, listening for an idle channel. The sender then transmits a packet start symbol at 10Kbps, which a receiver samples for at 20Kbps. As the following data is transmitted at 40Kbps, after the start symbol the receiver must synchronize to the sender at a finer granularity. The sender pauses for a few bit times after the start symbol, then transmits a one bit. The receiver, after the start symbol, polls the channel to identify the falling edge denoting the sender pause. Once it has done so, it polls again, this time for the next rising edge, using an input capture register to take a precise timestamp. These two spin loops take roughly 400-600 μs. The receiver adjusts its radio clock so that its 40Kbps data sample rate is synchronized with the sender's signal. The sender starts transmitting encoded packet data, which the receiver decodes into a packet buffer.

Once the packet data has been read in, the sender transmits a pulse of ones for the receiver to use for measuring signal strength. After this strength pulse, the sender transitions into receive mode and the receiver transitions into send mode. The sender introduces a short pause in its timing as part of this phase shift; as the receiver was sampling after the sender actually transmitted a bit, the sender must shift its sampling point to be after the receiver. The receiver then transmits a short bit pattern indicating acknowledgment; if the sender hears it, it marks the sent packet as acknowledged before passing it back in a send done event. The receiver checks the packet CRC, discarding corrupted packets. If the packet is addressed for the receiver or the broadcast address, the networking stack signals a reception event of the proper AM type.

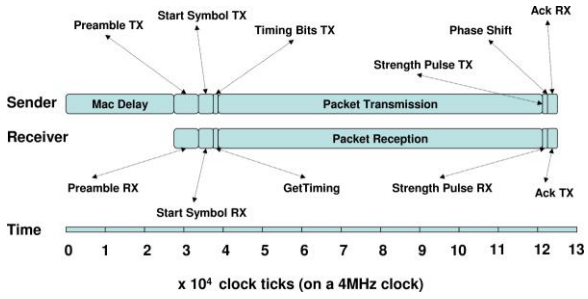
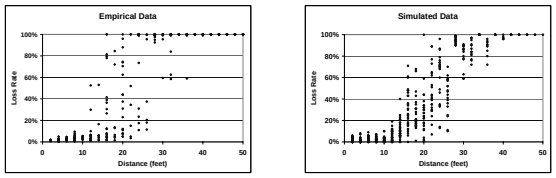


Figure 7: TinyOS Packet Sending/Reception

3.7.2 Network Simulation

The TinyOS stack uses three network sampling rates at different phases of packet reception and transmission: 40Kbps for data, 20Kbps for receiving a start symbol, and 10Kbps for sending a start symbol. In TOSSIM, adjustments to radio bit-rates are made by changing the period between radio clock events. The combination of bit sampling and bit-rate changes nearly captures the entire stack. There is one exception: the pair of spin loops to synchronize a sender signal, the one place where TinyOS breaks its event-driven methodology.

Under simulation, we maintain the event-driven concept by ignoring the first spin loop (for the zero) and handling the second (for the one) with additional state. Whenever a mote transmits the synchronization bit, it checks if any of the motes that can hear it are in the synchronization listening state. If it finds such a mote, it enqueues a radio event for the receiver representing the occurrence of the input capture.



(a) Empirical (b) Simulated

Figure 8: Empirical and Corresponding Simulated Packet Loss Data

This implementation results in an almost perfect simulation of the TinyOS networking stack at a bit level; TOSSIM accurately simulates the hidden node problem and can simulate errors at all phases of packet reception. When two nodes have interfering transmissions, a third listening node will just see the union of the two sender's bits, leading to both signals being corrupted. Additionally, delay arises when motes repeatedly enter CSMA wait because they continue to hear a signal on the channel. A single bit error during the data phase can be handled with the data encoding, but a single bit error during start symbol detection will prevent reception and a single bit error during acknowledgment transmission will cause it to fail. This granularity changes the methodology with which one normally approaches network simulation. For example, instead of modeling latency, by modeling the network itself TOSSIM simulates contention and backoff, which are causes of latency.

4. EVALUATION

We evaluate how well TOSSIM meets the four core needs of a sensor network simulator: fidelity, completeness, bridging, and scalability. To evaluate its fidelity and completeness, we used Surge, a sample application that comes packaged with TinyOS. Surge is a simple send/report program; nodes periodically collect sensor readings and route them to a base station, which collects and graphically displays them for the user. We discuss the Surge protocol in greater depth in Section *4.4.1*.

4.1 Fidelity: Radio Noise

We have developed a tool that generates radio loss graphs from physical topologies, based on empirical loss data gathered from a real-world network. The empirical data is from a network of twenty-six motes, placed in a line, spaced at regular two foot intervals, with the TinyOS radio set at a medium power setting (50 out of 100). Each mote transmitted two hundred packets, with each other mote keeping track of the number of packets heard. Figure 8(a) shows a plot of this empirical data as loss rates over distance.

To generate lossy models, we took this empirical data and generated Gaussian packet loss probability distributions for each distance. Given a physical mote topology, the tool generates packet loss rates for each mote pair by sampling these distributions. The tool translates packet error rates into independent bit error rates. Figure 8(b) shows the results of the experiment used to gather loss data when run in TOSSIM.

This model is by no means perfect. Instead, it demonstrates that the simple bit-error mechanism provided by TOSSIM allows the implementation of complex models. For example, instead of sampling from these distributions, loss rates could be generated by functions of distance, set to be constant, or derived from other experimental data. TOSSIM remains uncommitted to a specific model, allowing developers to choose one that fits their needs. Protocols can be tested for correctness with error-free models, and also tested in the presence of packet loss.

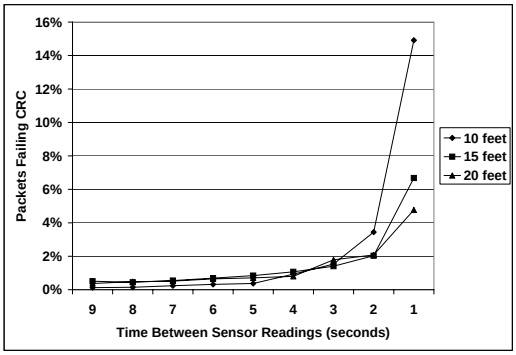


Figure 9: Packet CRC Failure Rates in Surge for Three Network Densities and Varying Sensing Frequencies. ***Each line shows a different network density, specified by the spacing between nodes in a 5x5 grid.***

4.2 Fidelity: Packet-level Interactions

Simulating network contention, the hidden terminal problem, and packet corruption are important requirements for TOSSIM. To evaluate whether the very simple networking model of TOSSIM can capture all of these complex interactions, we simulated Surge running at a range of sampling frequencies, from once per second to once every nine seconds. We simulated each frequency in a 5x5 grid network, at three network densities. The densest network had the 25 motes spaced 10 feet apart (a 40' by 40' area), while the sparsest had them spaced at 20 feet (an 80' by 80' area). For each simulation, we calculated the percentage of packets that were lost due to CRC failures. Figure 9 shows the results.

Two factors are at play in this study. The first is transmission rate. As the network send rate increases, so does the probability of collision. The second is network density. While the densest (single cell) networks do not suffer from the hidden terminal problem, neither do the sparsest (in which nodes have no neighbors). Real networks, of course, are more complex than this: cells are not simple and discrete. These two factors can be seen in the simulation results; the densest network (10 foot spacing) has the highest packet loss due to CRC failures, and that loss is most acute at the highest transmission rate.

If one thinks of a sensor network as a graph, then the hidden terminal problem occurs when there are three vertices a, b, and c, with edges (a, b) and (b, c) (but no (a, c)). Both a and c transmit at the same time, corrupting each other's signals at b. If, for a vertex v, one assumes each neighbor has an independent probability of transmitting, then as the degree of v increases, so does the probability that two neighbors will transmit concurrently. In other words, the more neighbors a node has (that cannot hear each other), the greater the chance they will interfere with one another. Of course, in the real world it is more complex: neighbors can often hear each other. Also, as a network becomes sparser, there will

be more CRC failures due to long, lossy links; additionally, more packets will be dropped due start symbol detection failure.

Higher network densities and sensing rates increase the possibility of the packet collisions. A network with 10-foot spacing and a 1Hz sampling rate caused the worst corruption rate. This complex behavior, caused by the interaction of a network of motes' packet encoding, the MAC, and hidden terminal problem all emerge from TOSSIM's simple network probability graph.

4.3 Fidelity: Subtle Race Conditions

The ability to simulate a large number of motes at a very fine granularity allowed us to uncover subtle bugs and race conditions in TinyOS code. Here we discuss one race condition we discovered in the TinyOS radio stack, involving a rare timing case. Until TOSSIM, the bug had been unnoticed, even though the stack had been in use by many research groups for over six months. We discovered the bug while testing applications in TOSSIM, verified its existence with its author, and fixed it.

This race condition occurred in the media access control component of the TinyOS radio stack, `ChannelMonM`, which is responsible for both sensing an idle channel (when sending packets) and for detecting the packet start symbol (when receiving packets). The idle channel and start symbol detection operations are concurrent and share a buffer of recently received bits. Before sending a packet, `ChannelMonM` attempts to detect whether the radio channel is idle, and performs a random backoff if not. Before performing backoff, `ChannelMonM` wrote bits into the shared buffer, possibly corrupting incoming bits being monitored by the start symbol detection code. The author of the offending component said the bug was a hold-over from an older, since-discarded MAC algorithm.

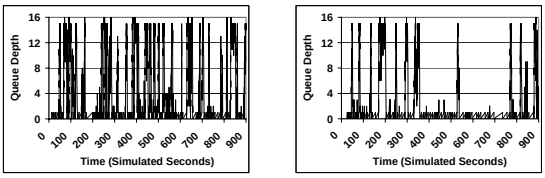
The bug fix was simple – removing the offending write. This bug was never detected in real deployments because it occurs very rarely, and causes packet loss that is indistinguishable from radio noise. Simulating the radio at the bit level allows these fine-grained timing issues to manifest in the simulation environment; they would be not be possible to detect or analyze with a higher-level, packet-level simulation.

4.4 Completeness: Surge

To evaluate whether TOSSIM can capture the wide range of interactions that occur in a TinyOS program, we present Surge as an example. We describe the Surge routing protocol in depth. Using TOSSIM, we discovered a significant flaw in the protocol, caused by an interaction of its routing cycle detection algorithm, retransmission policy, and the false negatives from the AM-level synchronous acknowledgments. Each of these policies are part of a separate TinyOS component: `MultiHopRouteM`, `QueuedSendM`, and `MicaHighSpeedRadioM`, respectively. TOSSIM was able to accurately capture this interaction, allowing us to analyze the complete Surge application and determine the causes of Surge's behavior in the real world.

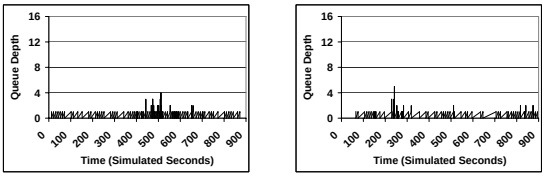
4.4.1 Surge

The Surge protocol forms a dynamic spanning tree, rooted at a single node. Nodes route packets to the root. Each mote maintains the network address of its parent in the tree as well as its depth. Nodes select their parent by eavesdropping on received messages; they attempt to use the parent with the lowest depth in the tree and best estimated link quality. Nodes select a new parent when the link quality falls below a certain threshold. Surge suppresses cycles in the routing by dropping packets that revisit their origin. A cycle's lifetime is limited by the sampling rate. When a node in a cycle takes a reading and sends it, the node will promptly discover and break the routing cycle.



(a) 14 feet from base (b) 28 feet from base

Figure 10: Send queue length over time for two motes in the Surge network: *send queues often overflow.*



(a) 14 feet from base (b) 28 feet from base

Figure 11: Send queue length for two motes in the Surge network over time, with cycle and duplicate suppression fixes: ***send queues do not overflow.***

Surge uses AM acknowledgments to provide a form of reliable transport. Nodes attempt to retransmit unacknowledged packets several times, and estimate the link quality to their parent as the fraction of messages that are acknowledged. Surge has been used in several applications and demonstrations. However, as far as we know, it had not been methodically analyzed in terms of performance or behavior. While running Surge in a test deployment, we observed a wide range of anomalous behaviors and troubling end-to-end reliability. We tried instrumenting the network in different ways, but were unable to get to the bottom of the problem; among other things, we did not know what to look for. Finally, after several days of fruitless effort, we turned to TOSSIM to gain some insight on the cause of Surge's behavior.

4.4.2 Simulation Environment

We used 5x5 mote grids with the lossy network model shown in Figure 8. Motes use the Surge protocol to route periodic sensor readings to the base station, located in one corner of the grid. We simulated grid spacings of 10, 15 and 20 feet to evaluate a range of network densities. We limited radio range to 50 feet. In the 10 foot case, most nodes can hear each other (although opposite corners cannot, for example), and in the 20 foot case, most packets will require several hops through the network. To separate the issue of network contention from the protocol, we configured Surge to sense at a low rate: once every ten seconds. In each case we simulated 15 virtual minutes, and longer simulations (2.5 virtual hours) validated that this represents steady-state behavior and not initial network startup.

4.4.3 Surge Analysis

Initial evaluation of the Surge protocol in TOSSIM uncovered a high end-to-end loss rate, even in the presence of many good network links. Given that channel utilization was low (each node generates only 1 packet every ten seconds), this seemed unlikely to be due to channel contention. After looking at a few packet traces, we suspected a problem with the packet send queue on each mote, which holds up to 16 packets awaiting transmission. When this queue overflows, packets are dropped, so a burst of activity in the network could cause increased loss. By instrumenting the length of the send queue in TOSSIM, we determined that this was in fact a problem. Figure 10 shows plots of the send queue length for two motes in the network at increasing distances from the base station. Surge concentrates traffic near the root of the spanning tree, and the mote closer to the base station frequently exhibits queue overflow. Given that the network generates packets at a very low rate, it seemed unlikely that the network should have so many packets in a queue at once due to bursts of communication.

Looking at transmission data, we determined the cause of the large number of packets to be an interaction between the retransmission policy and AM-level acknowledgments. Sometimes, a mote would successfully receive a packet, but the subsequent acknowledgement would be corrupted; the sender would see a false negative in the acknowledgment field of the packet buffer and retransmit. Bit errors in the radio channel were the cause of these false negatives. The Surge protocol has no build-in duplicate suppression; when a receiver receives a retransmission, it enqueues the packet in its forwarding queue along with the first copy. Essentially, every packet can duplicate at each hop in the network. This causes a possibly exponential increase in the copies of a packet over the number of hops.

While problematic, this duplication was not sufficient to disrupt a network by itself. For example, over a 4 hop network, a average replication factor of 1.5 would only result in 4 duplications over the complete traversal of the network. This is troublesome, but given the rate at which Surge was generating packets, should have been easy to handle.

An additional flaw in the protocol caused this duplication to overwhelm the network and overflow send queues. Surge suppresses cycles by dropping packets that revisit their origin. However, a packet originating at a node outside of a cycle will traverse it many times, until the routing protocol breaks the cycle. In the presence of even a short-lived cycle, a single packet can traverse ten or fifteen hops back and forth between two nodes. The possibly exponential growth from per-hop duplication occurs, send queues overflow, and the network saturates.

4.4.4 Fixing Surge

We instrumented the send queue to keep track of its maximum depth and redeployed Surge in our real-world network to validate that these queue overflows occur. They did. We implemented two changes to Surge to prevent this behavior. The first was duplicate suppression in the send queue; the second was to incorporate a more aggressive cycle detection algorithm that drops packets which do not traverse up the tree. Incorporating both of these fixes, we reran our simulations. Figure 11 shows the send queue length with the fixed protocol, which exhibits no cases of queue overflow.

The ability to instrument sensor network applications at different levels demonstrates TOSSIM's value for analyzing complex systems. This degenerate behavior in Surge was the result of low-level interactions between the link layer acknowledgment scheme, network-layer retransmission policy, and routing protocol. This problem would have been difficult to resolve using an abstract simulation of the networking stack; as with the `ChannelMonM` bug, low-level simulation of the actual TinyOS code was needed.

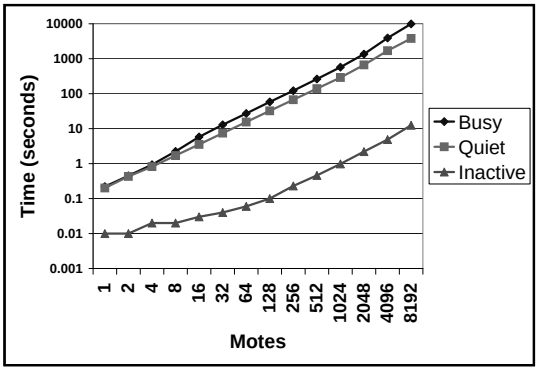


Figure 12: Scalability of TOSSIM on three applications as a function of the number of motes. ***Each simulation ran for ten virtual seconds.***

4.5 Completeness: TinyDB

Other research groups have used TOSSIM to test and evaluate their work. One example is TinyDB [17], a sensor query engine that runs on TinyOS developed by the Telegraph group at UC Berkeley. TinyDB provides an SQL-based interface to the sensor network, allowing users to pose declarative queries.

TinyDB uses a wide range of algorithms to route data tuples to query roots, aggregate sensor data to reduce traffic, and compute statistics over subregions of a network. Much of this processing occurs within the sensor network itself, reducing the amount of radio traffic by an order of magnitude compared with full collection and out-of-network processing.

The Telegraph group has used TOSSIM extensively in the design and evaluation of TinyDB; they report that it provides a useful environment for prototyping new features. Using TOSSIM and the lossy radio model, they have examined TinyDB's behavior and performance in networks of hundreds of nodes, a daunting task in the real world.

4.6 Bridging

TOSSIM compiles directly from TinyOS code; building a simulation instead of a mote image merely requires changing the single platform parameter to `make` (i.e., `make sim` instead of `make mica`). Besides the small number of components that TOSSIM replaces to emulate interrupts and other hardware abstractions, all of the simulation code (especially application code) is identical to that which runs on a mote. By supporting the TinyOS PC toolchain, users can transition between a simulation and real-world network by changing a single option in SerialForwarder, the TinyOS network access proxy server program. Users can step through simulation code with traditional debuggers without disrupting mote execution. All of these things combine to provide developers the ability to carefully test and analyze the implementations that will run on mote hardware.

4.7 Scalability

We evaluate TOSSIM's scalability using three different applications with varying degrees of complexity. Because of the bit-

level resolution of the radio stack simulation, performance of a simulation is based largely on the amount of communication, rather than on the complexity of the application logic itself. Hence, while these applications are simple, their simulation performance accurately represents that of larger applications.

The first application, `CntToLeds`, displays the bottom three bits of a 4 Hz counter on the mote's three LEDs. This application does not use the networking stack, and therefore represents the best case of simulation performance. The second application, `RfmToLeds`, displays the value contained in a received radio packet on the LEDs. This application performs only listening, and does not send any radio packets; its performance demonstrates the overhead of the receive side of the radio stack. The third application, `CntToRfm`, transmits the value of the 4 Hz timer as a radio message, and displays the value of received timer messages on the LEDs. This application represents a busy network scenario involving both sending and receiving of messages.

Figure 12 shows the performance of these three applications as a function of the number of simulated nodes. In each case, 10 virtual seconds of execution time were simulated on a 1.8 GHz Pentium 4 machine with 1GB of memory running Linux 2.4.18. The memory footprint of motes is very small, so memory consumption is not a limiting factor in our simulations. For a network of 10000 motes running the `CntToRfm` application, approximately 4 MB of memory is required. Dense network configurations can lead to greater memory requirements.

As the figure shows, simulation time is mostly linear in the number of nodes, simulating the radio stack at the bit level introduces significant complexity. The network-inactive application scales extremely well, and 8192 motes can be simulated in 12.5 sec. The network-intensive application, on the other hand, requires about 2.75 hours to simulate 8192 motes. Much of this overhead is due to the MAC layer used in our radio stack; a TDMA scheme would allow motes to sample the channel infrequently, yielding a significant performance improvement.

5. PRIOR WORK

Simultaneously meeting all of the requirements posed to a sensor network simulator is difficult. Several other sensor network simulators have been proposed, but none meet all of the requirements as TOSSIM does.

`ns-2` [2] is the predominant discrete event simulator used in network systems. It simulates networks at the packet level, and allows a wide range of heterogeneous network configurations. Complex models to determine packet loss rates from physical topologies are written in Tcl or C, separate from a protocol implementation. `ns-2` is, as its name suggests, a network simulator, while TOSSIM is a TinyOS simulator that provides a network model.

Numerous sensor network research efforts have evaluated algorithms with simulations using `ns-2`. For example, research on geographic hash tables used a 50-200 node simulation with an 802.11 MAC[23]. Simulations for the PSFQ routing protocol used 25 nodes with 2Mbps links[26]; there are other examples of an 802.11 MAC being used in `ns-2` to simulate sensor networks[30], although initial evidence indicates it is inappropriate[13].

Additionally, `ns-2` does not model application behavior. For trace driven simulations of layered protocols, this is appropriate. In sensor networks, however, protocols and applications interact; for example, protocol layers are often crossed or ignored to provide time-dependent aggregation of sensor readings, following a model of integrated layer processing instead of strict layering [6]. `ns-2` is inappropriate to model this sort of behavior; while `ns-2` is a much more general network simulator, TOSSIM simulates applications, the network, and their interaction, thereby providing a degree of fidelity, bridging, and completeness that `ns-2` cannot.

SensorSim[19] is built on top of an `ns-2` 802.11 network model using DSR. SensorSim is intended for the heavy-weight WINS [22] platform, and requires applications writers to use SensorWare Tcl scripts. It uses similar techniques to achieve different goals than TOSSIM; for example, it closely tracks energy usage based on data taken from real-world WINS nodes.

EmStar [7] provides a flexible environment for transitioning between simulation and deployment for iPAQ-class sensor nodes running Linux. Users have three options: running many virtual nodes on a single host with a simulated network, running many virtual nodes on a single host with each virtual node bridged to a real-world one for networking, and running a single real node on a host with a network interface.

TOSSF[21], a simulator developed at Dartmouth, compiles TinyOS code into the SWAN[16] framework for wireless simulation. TOSSF borrows many techniques used in TOSSIM. However, TOSSF translates to a very different target simulation framework. TOSSF uses dynamic function binding and frame allocation to allow different mote applications to run at once. It includes idealized radio propagation models; in contrast, TOSSIM contains models derived from empirically gathered data. TOSSF seeks to scale to tens or hundreds of thousands of nodes, but there currently are not yet any published evaluations of its scalability, fidelity, or completeness. In contrast, TOSSIM has been in use by several groups (such as the Telegraph group at UC Berkeley) for over six months.

TOSSIM simulates not only the network, but also the execution of each node; the distributed nature of sensor networks makes this aspect of TOSSIM similar to work in processor simulation. Many of the same issues arise, such as the trade-off between performance and inter-CPU timing accuracy, which in the case of a sensor network is inter-mote timing and performance. Systems such as Embra [27](part of SimOS [24]) emulate binary code; TOSSIM compiles directly to native code instead, as the relative simplicity of mote hardware allows accurate simulation at the component level.

Proteus allowed the user to configure the relative fidelity and resulting performance of the separate components in a multiprocessor, such as interrupts, instruction quanta, and networking[5]. Work with Tango Lite demonstrated that trace-based simulations were only worthwhile if execution was timing independent[10], something which is obviously not the case with an event-driven sensor network.

6. DISCUSSION AND FUTURE WORK

By exploiting several characteristics of TinyOS and the sensor network domain, TOSSIM can provide a scalable, high-fidelity simulation. As individual motes are resource-constrained, TOSSIM can simulate large numbers of them. Event-driven execution efficiently maps to event-driven simulation. The TinyOS component model provides an easy way to capture the execution of entire applications by re-implementing a few low-level abstractions. The component model also allows TOSSIM to use the same code that will run on motes, so developers to test their algorithm implementations. By representing a network as a bit error graph, TOSSIM can capture many of the subtle interactions seen in reality while keeping configuration and simulation simple.

Because TOSSIM can replace arbitrary TinyOS components, it has a flexible level of abstraction. A desire for extremely high fidelity motivated our choice for where the simulation/TinyOS boundary lies, but this boundary and the models beneath it can be easily changed. For example, while we designed TOSSIM to be

a closed TinyOS mote simulator, its architecture turns out to be much more flexible than we anticipated, allowing a user to trade off between fidelity and performance.

For example, the performance measurements in Section 4.7 are from simulating the network at the bit level. As a simple experiment, we wrote a simulator version of the TinyOS packet-level radio component, `RadioCRCPacket`. This simulator implementation models packet loss due to bit errors (using the lossy model), but does not model packet collision or MAC delays. Using this higher-level abstraction, simulating 8192 motes running `RfmToLeds` for 10 virtual seconds takes approximately 25 real-world seconds; in comparison, as shown in Figure 12, simulating this at the bit level approximately takes an hour. By allowing users to replace any part of the application component graph, TOSSIM provides flexibility in the spectrum of fidelity and performance.

Researchers at UCLA, drawing on their experiences with EmStar [7], are exploring connecting real-world motes to the TOSSIM networking model by replacing TinyOS networking components with a bridge to real motes. In this system, when a TOSSIM mote sends a packet, the TOSSIM engine tells a real-world counterpart mote to send a packet, and when a real-world mote receives a packet, it signals its TOSSIM counterpart with the received packet. This would allow debugging and analysis of high-level application implementations on a PC with a real-world network. Other groups have modified TOSSIM so that multiple running copies can communicate through a network proxy, allowing developers to program palmtop PCs in TinyOS, including even an AM interface over 802.11. This is useful when experimenting with heterogeneous networks that have a few powerful motes, without having to design and fabricate new hardware platforms.

The comparatively low mote bandwidth allows TOSSIM to simulate protocols at bit granularity. As Figure 12 showed, network activity is the principal simulation cost. Real-world motes are also limited by network activity, but for a different reason: energy. In an interesting turn of different but parallel constraints, optimizing an application for energy consumption will also speed its simulation.

An important issue is whether TOSSIM's techniques are limited to a specific OS running on certain hardware, or can be generalized to a broader class of sensor network platforms. There is also the question of how TOSSIM can be improved. We consider three points: modeling CPU time, modeling energy, and supporting thread-based execution models.

TOSSIM's run-instantly execution model does not capture CPU time. Since interrupts are discrete events, TOSSIM does not model preemption and the resulting possible TinyOS data races. Achieving this degree of fidelity would require either interpretation of mote binaries or instrumenting them to correlate PC ISA operations with mote ones and maintaining cycle counts. While using such techniques would be feasible for a small number of motes, it would limit scalability. We are exploring possible ways to provide similar degrees of fidelity while avoiding most of the costs. Additionally, current versions of nesC perform compile-time data race checks, greatly reducing this class of bugs, and therefore limiting the utility of such functionality.

Another thing that TOSSIM currently does not capture is energy consumption. Initial work deploying sensor networks has shown there to often be a large disparity between expected and real power draw [18]. However, simulation can provide comparative results for analyzing algorithms. Providing energy data requires adding hooks to the simulator implementations of hardware abstraction components, to keep track of changes in their energy states. The importance of energy in sensor network systems makes it one of the next intended additions for TOSSIM.

7. CONCLUSION

TinyOS's event-based execution maps easily to discrete event simulation. Unfortunately, it is unlikely that TOSSIM would be effective for simulating thread-based systems; the cost of the large number context switches (even if in user-land) would be prohibitive. Additionally, maintaining accurate time across motes and interrupt modeling in the presence of possible spin loops would require the same techniques as capturing preemption – a tremendous performance cost.

We tackled the problem of testing and analyzing sensor network applications, and have demonstrated that it is possible to build scalable, high fidelity simulations of full sensor network applications. The number of bugs and flaws we found in core TinyOS services suggests that simulation should be an important phase in application development, and the ease with which one can transition between deployments and simulation means doing so is not prohibitive. Our hope is that TOSSIM will be of great use to the sensor network community, enabling research in this new and wide domain.

8. REFERENCES

[1] 3170 vhdl simulator. `www.midwestcad.com/pdf/dig/vhdl_sim.pdf`.

[2] The network simulator. `http://www.isi.edu/nsnam/ns/`.

[3] *1076-1993 VHDL Language Reference Manual*. (ANSI/IEEE), 1993.

[4] D. Braginsky and D. Estrin. Rumor Routing Algorithm for Sensor Networks. In *Proceedings of the First ACM International Workshop on Wireless Sensor Networks and Applications*, 2002.

[5] E. A. Brewer, C. Dellarocas, A. Colbrook, and W. E. Weihl. PROTEUS: A High-Performance Parallel-Architecture Simulator. *Measurement and Modeling of Computer System*, pages 247–8, 1992.

[6] D. D. Clark and D. L. Tennenhouse. Architectural Considerations for a New Generation of Protocols. In *Proceedings of SIGCOMM*, september 1990.

[7] J. Elson, S. Bien, N. Busek, V. Bychkovskiy, A. Cerpa, D. Ganesan, L. Girod, B. Greenstein, T. Schoellhammer, T. Stathopoulos, and D. Estrin. Emstar: An environment for developing wireless embedded systems software. Technical Report Technical Report 0009, CENS, Mar. 2003.

[8] J. Elson, L. Girod, and D. Estrin. Fine-Grained Network Time Synchronization using Reference Broadcasts. In *Fifth Symposium on Operating Systems Design and Implementation (OSDI 2002)*, Boston, MA, USA., dec 2002.

[9] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC Language: A Holistic Approach to Networked Embedded Systems. In *Proceedings of Programming Language Design and Implementation (PLDI)*, June 2003.

[10] S. R. Goldschmidt and J. L. Hennessy. The Accuracy of Trace-driven Simulations of Multiprocessors. In *Proceedings of the 1993 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, 1993.

[11] J. S. Heidemann, F. Silva, C. Intanagonwiwat, R. Govindan, D. Estrin, and D. Ganesan. Building Efficient Wireless Sensor Networks with Low-Level Naming. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, Banff, Canada, October 2001.

[12] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. E. Culler, and K. S. J. Pister. System architecture directions for networked sensors. In *Architectural Support for Programming Languages and Operating Systems*, pages 93–104, Boston, MA, USA, Nov. 2000.

[13] C. Intanagonwiwat, R. Govindan, and D. Estrin. Directed Diffusion: A Scalable And Robust Communication Paradigm For Sensor Networks. In *Proceedings of the International Conference on Mobile Computing and Networking*, Aug. 2000.

[14] P. Levis and D. Culler. Maté: A Tiny Virtual Machine for Sensor Networks. In *International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, CA, USA*, Oct. 2002.

[15] J. Liu, P. Cheung, L. Guibas, and F. Zhao. A Dual-Space Approach to Tracking and Sensor Management in Wireless Sensor Networks. In *Proceedings of First ACM International Workshop on Wireless Sensor Networks and Applications*, September 2002.

[16] J. Liu, Y. Yuan, M. Liljenstam, and L. F. Perrone. SWAN: Simulator for Wireless Ad-Hoc Networks. `http://www.cs.dartmouth.edu/research/SWAN`.

[17] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. TAG: A Tiny AGgregation Service for Ad-Hoc Sensor Networks. In *OSDI*, 2002.

[18] A. Mainwaring, J. Polastre, R. Szewczyk, D. Culler, and J. Anderson. Wireless Sensor Networks for Habitat Monitoring. In *ACM International Workshop on Wireless Sensor Networks and Applications (WSNA'02)*, Atlanta, GA, USA, Sept. 2002.

[19] S. Park, A. Savvides, and M. B. Srivastava. SensorSim: A Simulation Framework for Sensor Networks. In *Proceedings of the 3rd ACM International Workshop on Modeling, Analysis and Simulation of Wireless and Mobile Systems*, 2000.

[20] A. Perrig, R. Szewczyk, V. Wen, D. Culler, and J. D. Tygar. SPINS: Security Protocols for Sensor Networks. In *International Conference on Mobile Computing and Networking (MobiCom 2001)*, Rome, Italy, July 2001.

[21] L. F. Perrone and D. Nicol. A Simulator for TinyOS Applications. In *Proceedings of the 2002 Winter Simulation Conference*, 2002.

[22] G. J. Pottie and W. J. Kaiser. Wireless Integrated Network Sensors. *Communications of the ACM*, 43(5):51–58, 2000.

[23] S. Ratnasamy, B. Karp, L. Yin, F. Yu, D. Estrin, R. Govindan, and S. Shenker. GHT: A Geographic Hash Table for Data-Centric Storage. In *Proceedings of the First ACM International Workshop on Wireless Sensor Networks and Applications*, 2002.

[24] J. Redstone, S. J. Eggers, and H. M. Levy. An Analysis of Operating System Behavior on a Simultaneous Multithreaded Architecture. In *Architectural Support for Programming Languages and Operating Systems*, pages 245–256, 2000.

[25] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: a Mechanism for Integrating Communication and Computation. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 256–266, May 1992.

[26] C.-Y. Wan, A. T. Campbell, and L. Krishnamurthy. PSFQ: A Reliable Transport Protocol for Wireless Sensor Networks. In *Proceedings of the First ACM International Workshop on Wireless Sensor Networks and Applications*, pages 1–11. ACM Press, 2002.

[27] E. Witchel and M. Rosenblum. Embra: Fast and Flexible Machine Simulation. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, 1996.

[28] A. Woo and D. Culler. A Transmission Control Scheme for Media Access in Sensor Networks. In *International Conference on Mobile Computing and Networking (MobiCom 2001)*, Rome, Italy, July 2001.

[29] Y. Yao and J. Gehrke. The Cougar Approach to In-Network Query Processing in Sensor Networks. *ACM SIGMOD Record*, 31(3):9–18, 2002.

[30] F. Ye, H. Luo, J. Cheng, S. Lu, and L. Zhang. A Two-Tier Data Dissemination Model for Large-Scale Wireless Sensor Networks. In *Proceedings of the First ACM International Workshop on Wireless Sensor Networks and Applications*, 2002.

[31] W. Ye, J. Heidemann, and D. Estrin. An Energy-Efficient MAC Protocol for Wireless Sensor Networks. In *Proceedings of IEEE Infocom 2002*, New York, NY, USA., June 2002.