TinyOS: An Operating System for Wireless Sensor Networks

Praveen Budhwar
Dept. of Computer Science and Engineering, BPSMV, Khanpur Sonepat, India

Abstract
TinyOS is an open-source, flexible and application-specific operating system for wireless sensor networks. Wireless sensor network consists of a large number of tiny and low-power nodes, each of which executes simultaneous and reactive programs that must work with strict memory and power constraints. The wireless sensor network's challenges of event-centric concurrent applications, limited resources and low-power operation impel the design of TinyOS. TinyOS meets these challenges and has become the platform of choice for sensor network research. It is very prevalent in sensor networks these days and supports a broad range of applications and research topics.

Keywords
NesC, TinyOS, TinyViz, TOSSIM, Wireless Sensor Networks

I. Introduction

TinyOS [1] an open-source embedded operating system designed directly for wireless embedded sensor networks [2]. Salient features of TinyOS are component-based architecture, a simple event-based concurrency model and split-phase operations that influence the development phases and techniques when writing application code. It has a component-based architecture which provides rapid innovation and implementation while reducing code size as required by the rigorous memory constraints inherent in wireless sensor networks. TinyOS's component library includes network protocols, distributed services, sensor drivers, and data acquisition tools – all of which can be used as it is or be further refined for a custom application. TinyOS's event-driven execution model enables fine grained power management, yet allows the scheduling flexibility made necessary by the unpredictable nature of wireless communication and physical world interfaces.

TinyOS is not an OS in the conventional sense instead it is a programming framework for embedded systems and set of components that enable building an application-specific OS into each application. A typical application is about 15K in size, of which the base OS is about 400 bytes; the largest application, a database-like query system, is about 64K bytes. TinyOS is strictly tied to the NesC [3]— Network embedded system C — programming language having native support for TinyOS's features (Gay et. al, 2003). NesC is a dialect of the C programming language optimized for the memory limitations of sensor networks. Its supplementary tools are mainly in the form of Java and shell script front-ends. Associated libraries and tools, such as the NesC compiler and Atmel AVR binutils toolchains are mostly written in C.

Section II presents the aspects of TinyOS, section III describes TOSSIM- TinyOS simulator, section IV explains TinyViz and finally section V concludes the paper.

II. TinyOS

TinyOS [4] provides a set of reusable system components. An application connects components using a wiring specification that is independent of component implementations; each application customizes the set of components it uses. All system processes are placed into separate components so that all of the data and functions inside each component are semantically related. Because of this principle, it is often said that components are modular and cohesive. With regard to system-wide co-ordination, components communicate with each other via interfaces. When a component offers services to the rest of the system, it adopts a provided interface which specifies the services that can be utilized by other components and how. This interface can be seen as a signature of the component - the client does not need to know about the inner workings of the component (implementation) in order to make use of it. This principle results in components referred to as encapsulated.

Unlike traditional computers, the motes' behavior with the environment is interactive: they are collecting sensor information and controlling the local environment rather than executing deterministic batch computations. As a consequence, motes need to react to changes in the environment. Since TinyOS is designed to be run on motes, it should provide support for various embedded events such as the microcontroller's internal timer is fired (the timer's current value is at zero), the ADC is about to provide a digitalized value measured by a sensor on the connected sensorboard, the radiochip raises an interrupt on the microcontroller to signal that a message reception has occurred, and so on and so forth. TinyOS [5] is completely non-blocking: it has a single stack. Therefore, all I/O operations that last longer than a few hundred microseconds are asynchronous and have a callback as known as Deferred Procedure Calls. To enable the native compiler to better optimize across call boundaries, TinyOS uses NesC's features to link these callbacks, called events, statically. Being non-blocking, TinyOS is enabled to maintain high concurrency with a single stack, it forces programmers to write complex logic by stitching together many small event handlers. To support larger computations, TinyOS [6] provides tasks, which are similar to a Deferred Procedure Call and interrupt handler bottom halves. A TinyOS component can post a task, which the OS will schedule to run later. Tasks are non-preemptive and run in FIFO order. This simple concurrency model is typically sufficient for I/O centric applications, but its difficulty with CPU-heavy applications has led to several proposals for incorporating threads into the OS. The following fig. 1 shows TinyOS architecture.

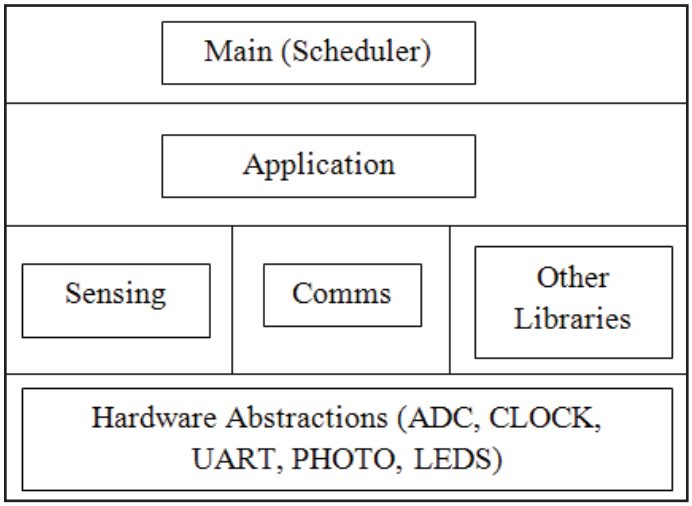

Fig. 1: TinyOS Architecture

Building applications and systems around an event-driven architecture allows these applications and systems to be constructed

in a manner that facilitates more responsiveness, because event-driven systems are, by design, more normalized to unpredictable and asynchronous environments. An event driven architecture is extremely loose coupled and well distributed. The great distribution of this architecture exists because an event can be almost anything and exist almost anywhere.

Event-drivenness means heavy concurrency. The previously mentioned tasks are synchronous while events and commands (interface member functions) with an async property are asynchronous. So in TinyOS, code runs either asynchronously in response to an interrupt (event), or in a synchronously scheduled task. Since dynamic memory allocation is not available, it is likely to have shared variables and states. Combining these two together, data races could occur due to concurrent updates to shared state.

Tasks are deferred computation mechanisms. Tasks do not preempt each other and can be posted by program components. The post operation does one thing: registers the task to be run in the scheduler, and immediately returns, deferring the computation until the scheduler executes the task later. Components can use tasks when timing requirements are not strict; this includes nearly all operations except low level communication. Events also run to completion, and may preempt the execution of a task or another event. Events are signaled by the environment (e.g. message reception or sensor reading), or by indicating completion of a split-phase operation. Furthermore, events can be sync and async.

The simple concurrency model of TinyOS [7] allows for high concurrency with low overhead, in contrast to a thread-based concurrency model in which thread stacks consume precious memory while blocking on a contended service. However, as in any concurrent system, concurrency and non-determinism can be the source of complex bugs, including deadlock, resource starvation and data race conditions. On the other hand, the presence of synchronous and asynchronous contexts makes the execution model complex, since various transitions can occur from one to other.

III. Introduction to TOSSIM

TOSSIM [8] captures the behavior and interactions of networks of thousands of TinyOS motes at network bit granularity (Levis et. al, 2003). Fig. 2 shows a graphical overview of TOSSIM. The TOSSIM architecture is made up of five parts: method to compile 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 exterior programs to interact with a simulation.

TOSSIM takes benefit of TinyOS's structure and whole system compilation to produce discrete-event simulations directly from TinyOS component graphs. It runs the identical code that runs on wireless sensor network hardware. Fig. 2 shows that by replacing a few low-level components, TOSSIM translates hardware interrupt into discrete simulator events; the simulator event queue delivers the interrupts that drive the execution of a TinyOS application. The remainder of TinyOS code runs unchanged. TOSSIM uses a very simple but astoundingly powerful abstraction for its wireless network. 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.

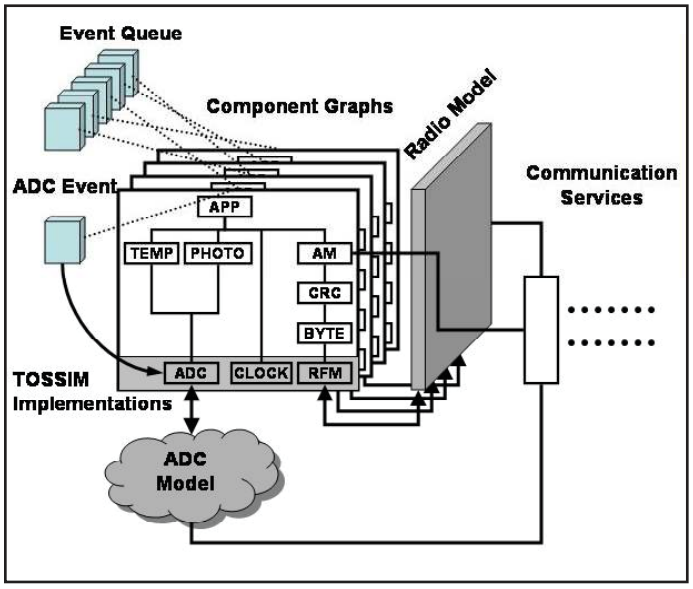

Fig. 2: TOSSIM Architecture

TOSSIM is a discrete event simulator for TinyOS wireless sensor networks. Instead of compiling a TinyOS application for a mote, users can compile it for the TOSSIM framework, which runs on a PC. This permits users to debug, test, and analyze algorithms in a restricted and repeatable environment. As TOSSIM runs on a PC, users can examine their TinyOS code using debuggers and other development tools. TOSSIM's primary goal is to provide a high fidelity simulation of TinyOS applications. Due to this, it focuses on simulating TinyOS and its execution, rather than simulating the real world.

While TOSSIM can be used to understand the causes of behavior observed in the real world, it does not capture all of them e.g. TOSSIM itself does not model the real world. Instead, it provides abstractions of certain real world phenomena. With tools outside the simulation itself, users can change these abstractions to implement whatever models they want to use. By making complex models outside the simulation, TOSSIM remains flexible to the needs of many users without trying to ascertain what is "accurate." TOSSIM does not model radio propagation; instead, it provides a radio abstraction of directed independent bit errors between two nodes. An external program can provide a desired radio model and map it to these bit errors. Having directed bit error rates means that asymmetric links can be easily modeled. Independent bit errors mean longer packets have a higher probability of corruption, and each packet's loss probability is independent. TOSSIM does not model power draw or energy consumption. However, it is very simple to add annotations to components that consume power to provide information on when their power states change.

TOSSIM [9] is automatically built when we compile an application. Applications are compiled by entering an application directory (e.g. /apps/Blink) and typing make. Alternatively, when in an application directory, we can type make pc, which will only compile a simulation of the application. There are several compilation options to ncc when compiling for TOSSIM, including the maximum number of motes that can be simulated. The default options in the TinyOS 1.1 makefile should fit almost any need. The TOSSIM executable is named main.exe, and resides in build/pc. It has the following usage:

Usage: ./build/pc/main.exe [options] num_nodes

[Options] are shown in Table 1.

Table 1 Options for running simulation on TOSSIM

Options	Usage
-h, --help	Display this message
-gui	Pauses simulation waiting for GUI to connect
-a=<model>	Specifies ADC model (<model>: generic, random) (generic is default)
-b=<sec>	All motes boot over first <sec> seconds (default: 10)
-ef=<file>	Use <file> for EEPROM; otherwise anonymous file is used
-l=<scale>	Run simulation at <scale> times real time (fp constant)
-r=<model>	Specifies a radio model (options: simple, lossy) (simple is default)
-rf=<file>	Specifies file input for lossy model (lossy.nss is default)
-s=<num>	Only boot <num> of nodes
-t=<sec>	Run simulation for <sec> virtual seconds
num_nodes	Number of nodes to simulate

The -h or --help options prints out the above usage message, and some additional information. The -a option specifies the ADC model to use. TOSSIM currently supports two models: generic and random. The -b option specifies the interval over which motes boot. Their boot times are uniformly distributed over this interval. The default value is ten seconds.

The -e option is for named EEPROM files. If -e isn't specified, the logger component stores and reads data, but this data is not persistent across simulator invocations: it uses an anonymous file. The -l option is for making TOSSIM run at a rate representative of real time. The scale argument specifies what relative rate should be used. For example, -l=2.0 means twice as fast as real time (two virtual seconds run in one real second), while -l=0.1 means one tenth of real time (one virtual seconds runs in ten real seconds.). Using this option imposes a significant performance overhead; it shouldn't be used when trying to run simulations quickly. The -r option specifies the radio model to use. TOSSIM currently supports two models: simple and lossy. Earlier versions also supported a "static" model, but this has been subsumed by the lossy model. The -s option tells TOSSIM to only boot a subset of the number of nodes specified. This is useful if we want some to boot later, in response to user input. If the -s option is specified, TOSSIM boots mote IDs 0-(num - 1). The -t option tells TOSSIM to run for a specified number of virtual seconds. After sec seconds have passed, TOSSIM exits cleanly. The num nodes option specifies how many nodes should be simulated. A compile-time upper bound is specified in /apps/Makerules. The standard TinyOS distribution sets this value to be 1000. By default, all num nodes boot in the first ten seconds of simulation, with bootup times uniformly distributed. TOSSIM catches SIGINT (control-C) to exit cleanly. This is useful when profiling code.

TOSSIM provides configuration of debugging output at run-time. Much of the TinyOS source contains debugging statements. Each debugging statement is accompanied by one or more modal flags. When the simulator starts, it reads in the DBG environment variable to determine which modes should be enabled.

~$ export DBG= usr2, am

Modes are stored and processed as entries in a bit-mask, so a single output can be enabled for multiple modes, and a user can specify multiple modes to be displayed. The set of DBG modes recognized by TOSSIM can be identified by using -h option; all available modes are printed. The current modes are shown in Table 2.

Table 2: Modes Available for DBG Statement

Modes	Usage
all	Enable all available messages
boot	Simulation boot and StdControl
clock	The hardware clock
task	Task enqueueing/dequeueing/running
sched	The TinyOS scheduler
sensor	Sensor readings
led	Mote leds
crypto	Cryptographic operations (e.g., TinySec)
route	Routing systems
am	Active messages transmission/reception
crc	CRC checks on active messages
packet	Packet-level transmission/reception
encode	Packet encoding/decoding
radio	Low-level radio operations bits and bytes
logger	Non-volatile storage
adc	The ADC
i2c	The I2C bus
uart	The UART (serial port)
prog	Network reprogramming
sounder	The sounder on the mica sensor board
time	Timers
sim	TOSSIM internals
queue	TOSSIM event queue
simradio	TOSSIM radio models
hardware	TOSSIM hardware abstractions
simmem	TOSSIM memory allocation/deallocation (for finding leaks)
usrx	User output mode x (x=1,2,3) (e.g. usr1)
temp	For temporary use

IV. Introduction to TinyViz

TinyViz is a Java visualization and actuation environment for TOSSIM. The main TinyViz class is a jar file, tools/java/net/tinyos/sim/tinyviz.jar. TinyViz can be attached to a running simulation. Also, TOSSIM can be made to wait for TinyViz to connect before it starts up, with the -gui flag. This allows users to be sure that TinyViz captures all of the events in a given simulation. TinyViz is not actually a visualizer; instead, it is a framework in which plugins can provide desired functionality. By itself, TinyViz does little besides draw motes and their LEDs. However, it comes with a few example plugins, such as one that visualizes network traffic. Fig. 3 shows a screenshot of the TinyViz tool. The left window contains the simulation visualization. The right window is the plugin window; each plugin is a tab pane, with configuration controls and data. The second element on the top bar is the Plugin menu, for activating or de-activating individual plugins. Inactive plugins have their tab panes greyed out.

The third element is the layout menu, which allows us to arrange motes in specific topologies, as well as save or restore topologies. TinyViz can use physical topologies to generate network topologies by sending messages to TOSSIM that configure network connectivity and the loss rate of individual links. The right side

of the top bar has three buttons and a slider. TinyViz can slow a simulation by introducing delays when it handles events from TOSSIM. The slider configures how long delays are. The On/Off button turns selected motes on and off; this can be used to reboot a network, or dynamically change its members. The button to the right of the slider starts and stops a simulation; unlike the delays, which are for short, fixed periods, this button can be used to pause a simulation for arbitrary periods. The final button, on the far right, enables and disables a grid in the visualization area. The small text bar on the bottom of the right panel displays whether the simulation is running or paused. The TinyViz engine uses an event-driven model, which allows easy mapping between TinyOS' event based execution and event-driven GUIs. By itself, the application does very little; drop-in plugins provide user functionality. TinyViz has an event bus, which reads events from a simulation and publishes them to all active plugins.

Users can write new plugins, which TinyViz can dynamically load. A simple event bus sits in the center of TinyViz; simulator messages sent to TinyViz appear as events, which any plugin can respond to. For example, when a mote transmits a packet in TOSSIM, the simulator sends a packet send message to TinyViz, which generates a packet send event and broadcasts it on the event bus. A networking plugin can listen for packet send events and update TinyViz node state and draw an animation of the communication. Plugins can be dynamically registered and deregistered, which correspondingly connect and disconnect the plugin from the event bus. A plugin hears all events sent to the event bus, but individually decides whether to do anything in response to a specific event; this keeps the event bus simple, instead of having a content-specific subscription mechanism.

A plugin must be a subclass of net.tinyos.sim.Plugin. Plugin has the following signature:

```
public abstract class Plugin {
public Plugin() {}
public void initialize(TinyViz viz, JPanel pluginPanel) {...}
public void register() {...}
public void reset() { /* do nothing */}
public abstract void deregister();
public abstract void draw(Graphics graphics);
public abstract void handleEvent(SimEvent event);}
```

Plugins register themselves with the TinyViz event bus, which then notifies them of all events coming in from TOSSIM; it is up to an individual plugin whether to do something. The draw method is used to draw visualizations in the left pane of the TinyViz window.

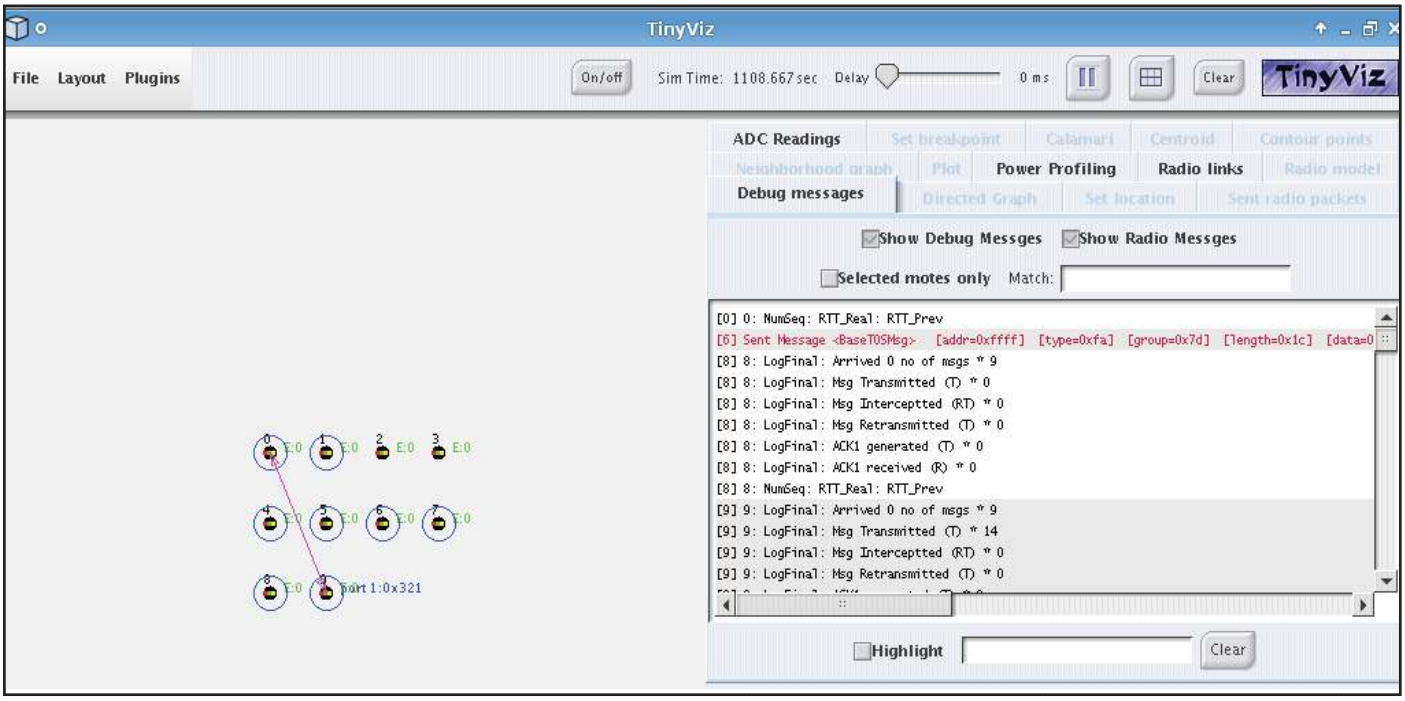

Fig. 3: TinyViz

V. Conclusion

In this article, we have studied TinyOS, an operating sytem for wireless sensor networks. It is an open source development environment with component based model. In this paper, we have studied NesC; TOSSIM, TinyOS simulator and TinyViz, TinyOS visualization tool. This paper explains all the important aspects related to TinyOS. In future, we can extend our study to include other operating systems for wireless sensor networks and can also present a comparison of various operating systems for wireless sensor networks.

References

[1] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, D. Culler,"TinyOS: An operating system for wireless sensor networks", in Ambient Intelligence. New York, NY: Springer-Verlag, New York, 2004.

[2] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, E. Cayirci, "Wireless Sensor Networks: A survey", Computer Networks 38(4), pp. 393–422, 2002.

[3] D. Gay, P. Levis, R. Behren, M. Welsh, E. Brewer, D. Culler,"The NesC Language: A Holistic Approach to Networked Embedded Systems", In Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language

Design and Implementation (PLDI).
[4] P. Levis, S. Madden, D. Gay, J. Polastre, R. Szewczyk, A. Woo, E. Brewer, D. Culler,"The emergence of networking abstractions and techniques in TinyOS", Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation, pp. 1-1, March 29-31, 2004, San Francisco, California.
[5] P. Levis,"TOSSIM System Description", [Online] Available: http://www.tinyos.net/nest/doc/tossim.pdf, 2002.
[6] P. Levis, D. Gay,"Tinyos design patterns", [Online] Available: http://www.cs.berkeley.edu/pal/ tinyos- patterns, 2004.
[7] C.L. Fok,"TinyOs tutorial", CS521, [Online] Available: www.princeton.edu/~wolf/EECS579/tutorial.pdf.
[8] P. Levis, N. Lee, M. Welsh, D. Culler,"TOSSIM: accurate and scalable simulation of entire TinyOS applications", Proceedings of the 1st international conference on Embedded networked sensor systems, November 05-07, Los Angeles, California, USA, 2003.
[9] P. Levis,"TinyOS Programming", 2006.

Praveen Budhwar received her M.E. degree in Computer Science & Engineering from PEC University of Technology, Chandigarh, India in 2014. She received her B. Tech. degree in Computer Science & Engineering from Vaish College of Engineering, Rohtak, Haryana, India in 2012. Her area of specialization is wireless sensor networks. She has published research papers in various international journals. Currently, she is working as an assistant professor in BPSMV, Khanpur, Sonepat, Haryana, India.