TOSSIM: A Simulator for TinyOS Networks

Philip Levis and Nelson Lee
pal@cs.berkeley.edu

September 17, 2003

Version 1.0
June 26, 2003

Contents

1 Introduction **2**

2 Compiling and Running a Simulation **4**

2.1 dbg 5

2.2 Network Monitoring and Packet Injection 6

3 Radio Models **7**

3.1 Using LossyBuilder 8

3.2 Bit Errors and Packet Errors 9

3.3 Lossy Model Actuation 9

4 ADC Models **9**

5 EEPROM **9**

6 TinyViz **10**

6.1 TinyViz Plugins 12

7 Using `gdb` **12**

8 Concurrency Model **13**

8.1 Sample Execution 14

9 Internals **14**

9.1 TOSSIM Architecture 15

9.2 TOSSIM Implementations 15

9.3 RFM 16

1 Introduction

TOSSIM is a discrete event simulator for TinyOS sensor networks. Instead of compiling a TinyOS application for a mote, users can compile it into the TOSSIM framework, which runs on a PC. This allows users to debug, test, and analyze algorithms in a controlled and repeatable environment. As TOSSIM runs on a PC, users can examine their TinyOS code using debuggers and other development tools. This document briefly describes the design philosophy of TOSSIM, its capabilities, its structure. It also provides a brief tutorial on how to use TOSSIM for testing or analysis.

TOSSIM's primary goal is to provide a high fidelity simulation of TinyOS applications. For this reason, 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, and should not be used for absolute evaluations.

TOSSIM is not always the right simulation solution; like any simulation, it makes several assumptions, focusing on making some behaviors accurate while simplying others. One of the most common questions about TOSSIM is whether it can "simulate X" or whether it "has an accurate X model." In hope of answering most of such questions, here is a brief summmary of its characteristics:

- **Fidelity:** By default, TOSSIM captures TinyOS' behavior at a very low level. It simulates the network at the bit level, simulates each individual ADC capture, and every interrupt in the system.

- **Time:** While TOSSIM precisely times interrupts (allowing things like bit-level radio simulation), it does not model execution time. From TOSSIM's perspective, a piece of code runs instantaneously. Time is kept at a 4MHz granularity (the CPU clock rate of the rene and mica platforms). This also means that spin locks or task spin locks will never exit: as the code runs instantaneously, the event that would allow the spin to stop will not occur until the code completes (never).
- **Models:** TOSSIM itself does not model the real world. Instead, it provides abstractions of certain real-world phenomena (such as bit error). With tools outside the simulation itself, users can then manipulate these abstractions to implement whatever models they want to use. By making complex models exterior to the simulation, TOSSIM remains flexible to the needs of many users without trying to establish what is "correct." Additionally, it keeps the simulation simple and efficient.
  - **Radio:** 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.
  - **Power/Energy:** 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 (e.g., turned on or off). After a simulation is run, a user can apply a energy or power model to these transitions, calculating overall energy consumption. Because TOSSIM does not model CPU execution time, it cannot easily provide accurate information for calculating CPU energy consumption.
- **Building:** TOSSIM builds directly from TinyOS code. To simulate a protocol or system, you must write a TinyOS implementation of it. On one hand, this is often more difficult than an abstract simulation; on the other, it means you can then take your implementation and run it on actual motes.
- **Imperfections:** Although TOSSIM captures TinyOS behavior at a very low level, it makes several simplifying assumptions. This means that it is very possible that code which runs in a simulation might not run on a real mote. For example, in TOSSIM interrupts are non-preemptive (a result of being a discrete event simulator). On a real mote, an interrupt can fire while other code is running. If pre-emption can put a mote in an unrecoverable state, then simulated motes will run without mishap while real-world motes may fail. Also, if interrupt handlers run too long, a real-world mote may crash; as code in TOSSIM runs instantaneously, no problems will appear in simulation.
- **Networking:** Currently, TOSSIM simulates the 40Kbit RFM mica networking stack, including the MAC, encoding, timing, and synchronous acknowledgements. It does not simulate the mica2 ChipCon CC1000 stack. We are waiting to have a better understanding of the behavior of CC1000 before writing a simulation implementation.
- **Authority:** Initial experience from real-world deployments has shown that TinyOS networks have very complex and highly variable behavior. While TOSSIM is useful to get a sense of how algorithms perform in comparison to one another, TOSSIM results shouldn't be considered authoritative. For example, TOSSIM can tell you that one algorithm behaves better than another under high loss, but the question remains as to whether the specified loss scenario has any basis in the real world. TOSSIM should not be considered an end-point of evaluation; instead, it is a system that allows the user to separate out environmental noise to better understand algortithms.

2 Compiling and Running a Simulation

TOSSIM is automatically built when you compile an application. Applications are compiled by entering an application directory (e.g. `/apps/Blink`) and typing `make`. Alternatively, when in an application directory, you 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; refer to the nesC manual for further information on the options available.

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:
  -h, --help        Display this message.
  -gui              pauses simulation waiting for GUI to connect
  -a=<model>        specifies ADC model (generic is default)
                    options: generic random
  -b=<sec>          motes boot over first <sec> seconds (default: 10)
  -ef=<file>        use <file> for eeprom; otherwise anonymous file is used
  -l=<scale>        run sim at <scale> times real time (fp constant)
  -r=<model>        specifies a radio model (simple is default)
                    options: simple static lossy
  -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. Section 4 describes these models.

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. Section 5 describes how the EEPROM in TOSSIM works.

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.). TOSSIM can only run so fast; specifying it to run faster than it can will cause it to run as quickly as possible. 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. Section 3 describes these models

The `-s` option tells TOSSIM to only boot a subset of the number of nodes specified. This is useful if you 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.

2.1 dbg

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. 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 the `-h` option; all available modes are printed. The current modes are:

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)
usr1:	User output mode 1
usr2:	User output mode 2
usr3:	User output mode 3
temp:	For temporary use

As an example, the statement

```
dbg(DBG_CRC, "crc check failed: \%x, \%x\n", crc, mcrc);
```

will print out the failed CRC check message if the user has enabled CRC debug messages, while

```
dbg(DBG_BOOT, "AM Module initialized\n");
```

will be printed to standard out if boot messages are enabled. DBG flags can also be combined, as in this example:

```
dbg(DBG_ROUTE|DBG_ERROR, "Received control message: lose our network name!.\n");
```

This will print out if either route or error messages are enabled.

There are a set of bindings from ASCII strings to debug modes. For example, setting DBG to "`boot`" will enable boot message (by setting the appropriate bit in the mask). Setting DBG to "`boot,route,am`"[1] will enable boot, routing and active message output. Generally, the name of the flag in TinyOS source is "`DBG_`", followed by the name of the DBG option in all caps. For example, the environment variable value "`am`" enables `DBG_AM` messages, and the value "`usr1`" enables `DBG_USR1` messages. Note that if you don't set DBG, then all debug output is enabled and the simulation will run very slowly.

Four modes are reserved for application components: usr1, usr2, usr3, and temp. Developers are urged to use these modes in their code. When TinyOS is compiled for mote hardware, all of debug statements are removed; we have verified that they add no instructions to the TinyOS code image.

We have found one of the easiest way to run experiments and gather data is to use dbg statements and save them to a file. Generally, the more data printed out, the better (to a point); Perl scripts or other tools can then read in the log of the run and provide information on packet loss rates, routing state, etc. A useful function, defined in `external_comm.c`, is `printTime(char* buf, int len)`. It takes the current time in simulation and converts it to the form HH:MM:SS, putting it in `buf`..

2.2 Network Monitoring and Packet Injection

To interact with a simulated network, you must use SerialForwarder, the standard TinyOS interface tool. To work with TOSSIM, SerialForwarder's input source must be set appropriately. TOSSIM provides two modes: communication through a serial port to mote 0 and network snooping.

The serial port mode ("tossim-serial" in the "Mote Communications" field or "-comm tossim-serial" at the command line) interacts with mote 0 over its serial port. Programs connecting to SerialForwarder can read messages mote 0 sends to its serial port, and send messages to mote 0 over its serial port.

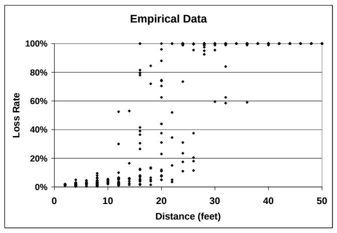
The snooping mode ("tossim-radio" in the "Mote Communications" field or "-comm tossim-radio" at the command line) sits on top of the TOSSIM network model. Programs connecting to SerialForwarder hear every radio message sent in the network, and can inject radio messages to arrive (without error) at any mote. Because this mode outputs every message *sent*, it does not consider loss; programs connecting will hear packets that might not arrive successfully at any mote.

It's often helpful to use the `-l` flag when using the serial port mode; external applications can then interact with a network in something approximating real time. Otherwise, in simulations of small numbers of motes, things like network timeouts can go off before a PC application has a chance to respond.

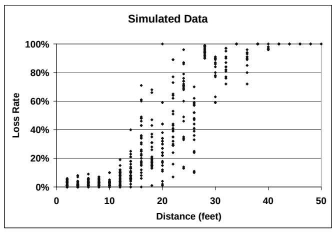
Message Interface Generator (MIG) is a tool that generates Java classes for TinyOS packets. The MIG tool parses C structures for TinyOS packets and builds a Java class with accessors for each of the packet fields. The message classes can also unpack from and pack to byte arrays.

Using MIG with TOSSIM has one hitch; you must specify the `-target=pc` flag when compiling your message classes. MIG classes generated for motes may not work with TOSSIM; the two platforms do not necessarily pack messages identically. Having different word boundaries (one byte on motes, four bytes on PCs) means that structures will have different padding. Problems will definitely appear if message structures use platform-dependent types such as `int`.

[1]E.g. "`export DBG=boot,route,am`" in bash, "`setenv DBG boot,route,am`" in tcsh, etc.



(a) Empirical



(b) Simulated

Figure 1: Empirical and Corresponding Simulated Packet Loss Data

3 Radio Models

TOSSIM simulates the TinyOS network at the bit level, using TinyOS component implementations almost identical to the mica 40Kbit RFM-based stack. TOSSIM provides two radio models: simple and lossy. The mica2 CC1000-based stack does not currently have a simulation implementation.

In TOSSIM, a network signal is either a one or zero. All signals are of equal strength, and collision is modeled as a logical or; there is no cancellation. This means that distance does not effect signal strength; if mote B is very close to mote A, it cannot cut through the signal from far-away mote C. This makes interference in TOSSIM generally worse than expected real world behavior.

The "simple" radio model places all nodes in a single cell. Every bit transmitted is received without error. Although no bits are corrupted due to error, two motes can transmit at the same time; every mote in the cell will hear the overlap of the signals, which will almost certainly be a corrupted packet. However, because of perfect bit transmission in a single cell, the probability of two motes transmitting at the same time is very, very low, due to the TinyOS CSMA protocol.

The simple model is useful for testing single-hop algorithms and TinyOS components for correctness. Deterministic packet reception allows deterministic results.

The "lossy" radio model places the nodes in a directed graph. Each edge (a, b) in the graph means a's signal can be heard by b. Every edge has a value in the range $(0, 1)$, representing the probability a bit sent by a will be corrupted (flipped) when b hears it. For example, a value of 0.01 means each bit transmitted has a 1% chance of being flipped, while 1.0 means every bit will be flipped, and 0.0 means bits will be transmitted without error. Each bit is considered independently.

The graph of the lossy model can be specified at TOSSIM boot with a file. TOSSIM looks for the file "lossy.nss" by default, but an alternate file can be specified with the `-rf` flag. The file has the following format:

```
<mote ID>:<mote ID>:bit error rate
<mote ID>:<mote ID>:bit error rate
<mote ID>:<mote ID>:bit error rate ...
```

For example,

```
0:1:0.012333
1:0:0.009112
1:2:0.013196
```

specifies that mote 1 hears mote 0 with a bit error rate of 1.2%, mote 0 hears mote 1 with a bit error rate of 0.9%, and mote 2 hears mote 1 with a bit error rate of 1.3%. By making the graph directed, TOSSIM can model asymmetric links, which initial empirical studies have suggest are a common occurance in TinyOS RFM-stack networks.

By specifying error at the bit level, TOSSIM can capture many causes of packet loss and noise in a TinyOS network, including missed start symbols, data corruption, and acknowledgement errors.

TOSSIM includes a Java tool, `net.tinyos.sim.LossyBuilder` for generating loss rates from physical topologies. The tool models loss rates observed empirically in an experiment performed by Woo et al. on a TinyOS network [1]. 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 1(a) shows a plot of this empirical data.

To generate lossy models, the tool has Gaussian packet loss probability distributions for each distance, fit to match the empirical data. 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 1(b) shows the results of the experiment used to gather loss data when run in TOSSIM.

The lossy model models interference and corruption, but it does not model noise; if no mote transmits, every mote will hear a perfectly clear channel.

3.1 Using LossyBuilder

LossyBuilder assumes each mote has a transmission radius of 50 feet. Combined with the bit error rate, this means each mote transmits its signal in a disc of radius 50 feet, with the bit error rate increasing with distance from the center.

LossyBuilder can read in or generate physical topologies ((x,y) coordinates), and generate loss topologies from physical topologies by sampling from the model of the empirical distribution in Figure 1(a). Its usage is:

```
usage: java net.tinyos.sim.LossyBuilder [options]
options:
  -t grid:       Topology (grid only and default)
  -d <m> <n>:    Grid size (m by n) (default: 10 x 10)
  -s <scale>:    Spacing factor (default: 5.0)
  -o <file>:     Output file
  -i <file>:     Input file of positions
  -p:            Generate positions, not error rates
```

Running LossyBuilder will, by default, generate a loss topology for a 10 by 10 grid with a grid spacing of five feet (45' by 45'), and print it to standard out. It prints the loss topology in the form that TOSSIM expects. The `-i` option allows you to specify an input file of (x,y) positions, which has the format

```
x y
x y
x y ...
```

where the coordinate system is in feet. The first (x,y) pair is the position of mote 0, the second is the position of mote 1, etc.

For example,

```
java net.tinyos.sim.LossyBuilder -d 20 20 -s 10 -o 20x20-10.nss
```

will output a loss topology of a 20 by 20 grid, with ten foot spacing, and write it to the file "20x20-10.nss".

3.2 Bit Errors and Packet Errors

The forumula for calculating packet error rates (E_p) from bit error rates (E_b) for the mica RFM 40Kb stack with SecDed encoding is:

$$
\begin{aligned}
&E_p = 1 - (S_s \cdot (S_e)^d) \\
&S_s = (1 - E_b)^9 \\
&S_e = (1 - E_b)^8 + (8 \cdot E_b \cdot (1 - E_b)^{12}) \\
&E_p = 1 - ((1 - E_b)^9 \cdot ((1 - E_b)^8 + (8 \cdot E_b \cdot (1 - E_b)^{12}))^d)
\end{aligned}
$$

where S_s is the start symbol success probability, S_e the probability a packet byte is uncorrupted (zero or one bit errors), and d is the number of bytes in the packet.

The simple model can be represented in the lossy model: it is a fully connected graph, in which every edge has a loss probability of zero. However, doing so requires specifying the entire graph to TOSSIM; the simple model is an easy shortcut (and has an internal representation that's more efficient).

3.3 Lossy Model Actuation

Specifying a loss topology with a file defines a static topology over an entire simulation. There are simulation situations, however, in which changing topologies are needed. TOSSIM therefore allows users to modify the loss topology at run-time. Applications can connect to TOSSIM over a TCP socket and send control messages to add, delete, or modify network links. Currently, the only application that does so is TinyViz, discussed in Section 6. The TinyViz network plugins use the same empirical model of LossyBuilder to generate link loss rates; moving motes causes TinyViz to send the appropriate commands to TOSSIM.

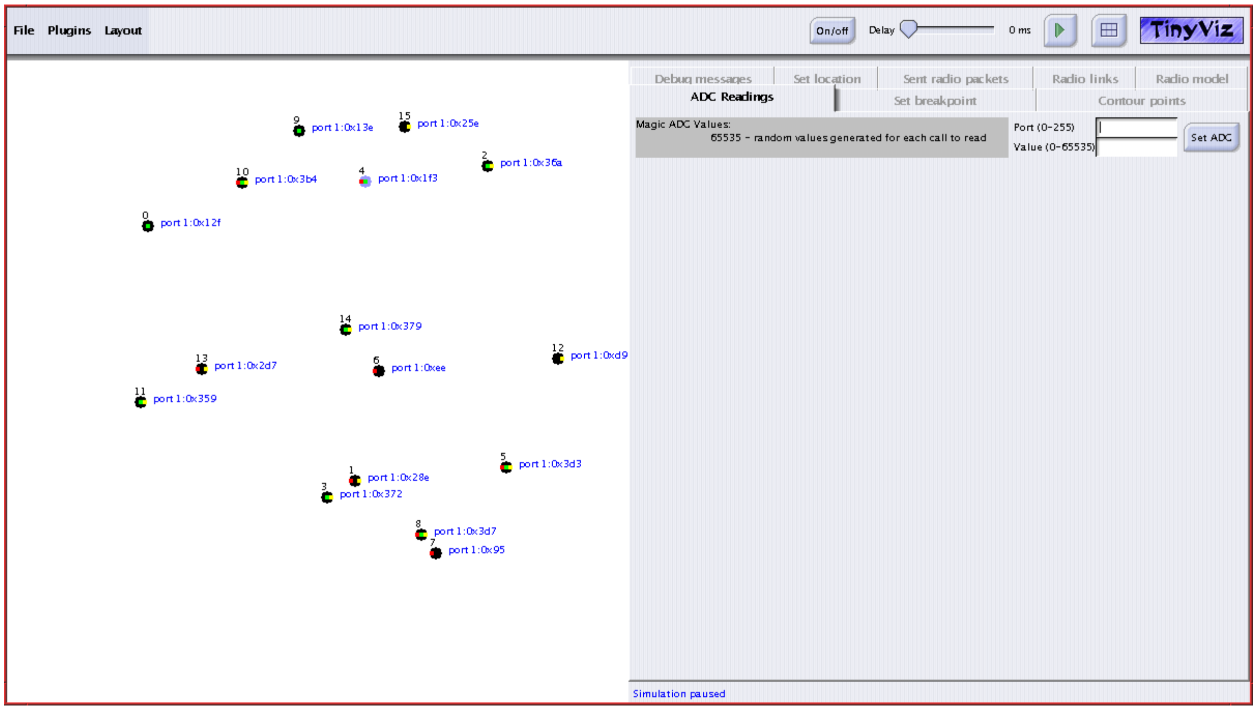
4 ADC Models

TOSSIM provides two ADC models: random and generic. The model chosen specifies how readings taken from the ADC are generated. Whenever any channel in the ADC is sampled in the random model, it returns a 10-bit random value (the rene and mica ADCs are 10 bits).

The general model also provides random values by default, but has added functionality. Just as external applications can actuate the lossy network model, they can also actuate the generic ADC model using the TOSSIM control channel, setting the value for any ADC port on any mote. Currently, only TinyViz, (discussed in Section 6) supports this, through the ADC plugin. Left-clicking on a mote selects it; using the ADC panel, you can set the 10-bit value read from any of the mote's ADC ports. Figure 2 shows a screenshot of this TinyViz functionality.

5 EEPROM

TOSSIM models the EEPROM at the line (16-byte block) level. TOSSIM models it with a large, memory-mapped file. By default, this file is anonymous, and disappears when a simulation ends. However, with the



Magic ADC Values:
65535 - random values generated for each call to read

Simulation paused

Figure 2: Setting ADC values in TinyViz

`-ef` option, a user can specify a named file to use; this allows the EEPROM data to persist across multiple simulation invocations.

6 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.

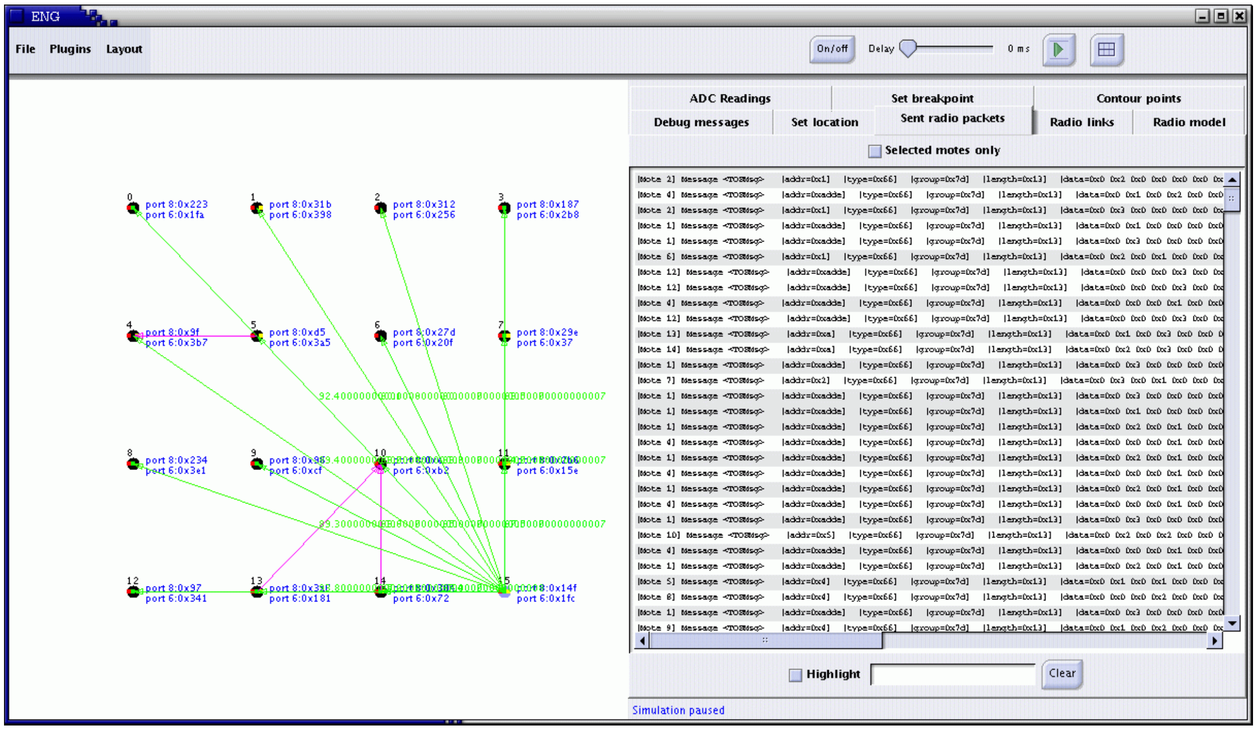
TinyViz is not actually a visualizer; instead, it is a framework in which plugins can provide desired functionality. By itself, TinyViz does little besides draw motes and their LEDs. However, it comes with a few example plugins, such as one that visualizes network traffic.

Figure 3 shows a screenshot of the TinyViz tool. The left window contains the simulation visualization, showing 16 motes communicating in an ad-hoc network. 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 you 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.


ENG
File
Plugins
Layout
On/off
Delay
0 ms
ADC Readings
Set breakpoint
Contour points
Debug messages
Set location
Sent radio packets
Radio links
Radio model
Selected motes only
Highlight
Clear
Simulation paused
0 port 8:0x223 port 6:0x1fa
1 port 8:0x31b port 6:0x398
2 port 8:0x312 port 6:0x256
3 port 8:0x187 port 6:0x2b8
4 port 8:0x9f port 6:0x3b7
5 port 8:0xd5 port 6:0x3a5
6 port 8:0x27d port 6:0x20f
7 port 8:0x29e port 6:0x37
8 port 8:0x234 port 6:0x3e1
9 port 6:0xcf
10 port 6:0xb2
11 port 6:0x15e
12 port 8:0x97 port 6:0x341
13 port 6:0x181
14 port 6:0x72
15 port 8:0x14f port 6:0x1fc

```
[Mote 2] Message <TOSMsg>   [addr=0x1]   [type=0x66]   [group=0x7d]   [length=0x13]   [data=0x0 0x2 0x0 0x0 0x0 0x0
[Mote 4] Message <TOSMsg>   [addr=0xadde]   [type=0x66]   [group=0x7d]   [length=0x13]   [data=0x0 0x1 0x0 0x2 0x0 0x0
[Mote 2] Message <TOSMsg>   [addr=0x1]   [type=0x66]   [group=0x7d]   [length=0x13]   [data=0x0 0x3 0x0 0x0 0x0 0x0
[Mote 1] Message <TOSMsg>   [addr=0xadde]   [type=0x66]   [group=0x7d]   [length=0x13]   [data=0x0 0x1 0x0 0x0 0x0 0x0
[Mote 1] Message <TOSMsg>   [addr=0xadde]   [type=0x66]   [group=0x7d]   [length=0x13]   [data=0x0 0x3 0x0 0x0 0x0 0x0
[Mote 6] Message <TOSMsg>   [addr=0x1]   [type=0x66]   [group=0x7d]   [length=0x13]   [data=0x0 0x2 0x0 0x1 0x0 0x0
[Mote 12] Message <TOSMsg>   [addr=0xadde]   [type=0x66]   [group=0x7d]   [length=0x13]   [data=0x0 0x0 0x0 0x3 0x0
[Mote 12] Message <TOSMsg>   [addr=0xadde]   [type=0x66]   [group=0x7d]   [length=0x13]   [data=0x0 0x0 0x0 0x3 0x0
[Mote 4] Message <TOSMsg>   [addr=0xadde]   [type=0x66]   [group=0x7d]   [length=0x13]   [data=0x0 0x0 0x0 0x1 0x0 0x0
[Mote 12] Message <TOSMsg>   [addr=0xadde]   [type=0x66]   [group=0x7d]   [length=0x13]   [data=0x0 0x0 0x0 0x3 0x0
[Mote 13] Message <TOSMsg>   [addr=0xa]   [type=0x66]   [group=0x7d]   [length=0x13]   [data=0x0 0x1 0x0 0x3 0x0 0x0
[Mote 14] Message <TOSMsg>   [addr=0xa]   [type=0x66]   [group=0x7d]   [length=0x13]   [data=0x0 0x2 0x0 0x3 0x0 0x0
[Mote 1] Message <TOSMsg>   [addr=0xadde]   [type=0x66]   [group=0x7d]   [length=0x13]   [data=0x0 0x3 0x0 0x0 0x0 0x0
[Mote 7] Message <TOSMsg>   [addr=0x2]   [type=0x66]   [group=0x7d]   [length=0x13]   [data=0x0 0x3 0x0 0x1 0x0 0x0
[Mote 1] Message <TOSMsg>   [addr=0xadde]   [type=0x66]   [group=0x7d]   [length=0x13]   [data=0x0 0x3 0x0 0x0 0x0 0x0
[Mote 1] Message <TOSMsg>   [addr=0xadde]   [type=0x66]   [group=0x7d]   [length=0x13]   [data=0x0 0x1 0x0 0x0 0x0 0x0
[Mote 1] Message <TOSMsg>   [addr=0xadde]   [type=0x66]   [group=0x7d]   [length=0x13]   [data=0x0 0x2 0x0 0x1 0x0 0x0
[Mote 4] Message <TOSMsg>   [addr=0xadde]   [type=0x66]   [group=0x7d]   [length=0x13]   [data=0x0 0x0 0x0 0x1 0x0 0x0
[Mote 1] Message <TOSMsg>   [addr=0xadde]   [type=0x66]   [group=0x7d]   [length=0x13]   [data=0x0 0x2 0x0 0x1 0x0 0x0
[Mote 4] Message <TOSMsg>   [addr=0xadde]   [type=0x66]   [group=0x7d]   [length=0x13]   [data=0x0 0x0 0x0 0x1 0x0 0x0
[Mote 1] Message <TOSMsg>   [addr=0xadde]   [type=0x66]   [group=0x7d]   [length=0x13]   [data=0x0 0x2 0x0 0x1 0x0 0x0
[Mote 4] Message <TOSMsg>   [addr=0xadde]   [type=0x66]   [group=0x7d]   [length=0x13]   [data=0x0 0x0 0x0 0x1 0x0 0x0
[Mote 1] Message <TOSMsg>   [addr=0xadde]   [type=0x66]   [group=0x7d]   [length=0x13]   [data=0x0 0x3 0x0 0x0 0x0 0x0
[Mote 10] Message <TOSMsg>   [addr=0x5]   [type=0x66]   [group=0x7d]   [length=0x13]   [data=0x0 0x2 0x0 0x2 0x0 0x0
[Mote 4] Message <TOSMsg>   [addr=0xadde]   [type=0x66]   [group=0x7d]   [length=0x13]   [data=0x0 0x0 0x0 0x1 0x0 0x0
[Mote 1] Message <TOSMsg>   [addr=0xadde]   [type=0x66]   [group=0x7d]   [length=0x13]   [data=0x0 0x2 0x0 0x1 0x0 0x0
[Mote 5] Message <TOSMsg>   [addr=0x4]   [type=0x66]   [group=0x7d]   [length=0x13]   [data=0x0 0x1 0x0 0x1 0x0 0x0
[Mote 8] Message <TOSMsg>   [addr=0x4]   [type=0x66]   [group=0x7d]   [length=0x13]   [data=0x0 0x0 0x0 0x2 0x0 0x0
[Mote 1] Message <TOSMsg>   [addr=0xadde]   [type=0x66]   [group=0x7d]   [length=0x13]   [data=0x0 0x3 0x0 0x0 0x0 0x0
[Mote 9] Message <TOSMsg>   [addr=0x4]   [type=0x66]   [group=0x7d]   [length=0x13]   [data=0x0 0x1 0x0 0x2 0x0 0x0
```

Figure 3: 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 green arrows and corresponding labels represent link probabilities for mote 15, and the magenta arrows indicate packet transmission.

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.

6.1 TinyViz 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.

7 Using `gdb`

The binary executable produced by typing `make pc` in any app directory can be debugged using GDB. Developers of TinyOS code can step through programs they have written, debugging deterministic, logical aspects of their code before loading programs onto motes.

When accessing variables and functions in TOSSIM under GDB, specific prefixes precede them, to distinguish variables from different components that have ths same name. The nesC compiler takes each component's fields and functions and renames them with unique identifiers. This renaming for fields of a component is done by taking the field name, and preceding it by the component name and a `$` sign. Functions are renamed by taking the name of the function and preceding it by the name of the component it is defined in, followed by a `$` sign, followed by the interface name, followed by another `$` sign.

The maximum number of nodes that can be simulated is determined at compile time. This value, as mentioned in section `TOSSIM Architecture` specifies the size of the array for each component's fields during code generation of the compilation process. When compiling for the `pc`, each declaration of a component field is followed by `[`, the value of `tossim_num_nodes`, and `]`, where tossim_num_nodes is specified by the flag `-fnesc-tossim_tosnodes=` in `Makerules`, or by default, set to 1000. This is the way TOSSIM simulates multiple motes. TOSSIM keeps an array for each variable of state, and references each mote's corresponding state when running code for that mote.

During code generation for the `pc`, all references to component fields are followed by `[`, `tos_state.current_node` and `]`, where `tos_state.current_node` refers to the moteID of the mote TOSSIM is currently executing code for.

For, for example, the `Counter` component has a variable named `state` of type `uint_16t`. This translates to the C name `Counter$state`. When compiled for TOSSIM, it becomes `uint16_t Counter$state[MAX_NODES]`. If a component has a variable that's an array (e.g., `uint8_t buffer[8]`, it becomes an array of arrays (e.g., `uint8_t buffer[8][MAX_NODES]`). Laying out module variables in this way means that overrun and underrun bugs will have different effects on real motes and in TOSSIM. On a real mote, such a bug will corrupt that mote's state; in TOSSIM, it will corrupt other motes' state.

Using the application `Blink` as an example, type `make pc` in the `/apps/Blink/` directory to compile the application for TOSSIM. Run `gdb` on the executable produced by typing `gdb build/pc/main.exe`. Examining `BlinkM.td`, to break at the function Clock.fire(), type `break *BlinkM$Clock$fire`. `break` (or `b`) specifies a breakpoint in GDB, and the `*` tells the GDB command line parser to examine the entire block of text following it as a single expression. It is currently necessary to specify the `*` before the function name for GDB to parse it properly. `BlinkM` denotes the component the function is defined in, and `Clock` specifies the interface the function `fire` belongs to.

To run `Blink` with one mote, type `run 1` or `r 1`. `r` is the GDB command to run the current executable, and all values after the `r` are passed as arguments to the executable. Therefore the `1` specifies the number of motes to simulate to TOSSIM. MoteIDs are assigned starting at 0. In this current example the moteID of the single mote being simulated is 0.

Upon hitting the breakpoint at `fire`, to print the value of `BlinkM`'s `state`, type
`print BlinkM$state[tos_state.current_node]` (or `p` for `print`). `p` is the print command in GDB. `BlinkM` corresponds to the component the field `state` belongs to. Since the application was compiled for the pc, it is necessary to specify which mote's `BlinkM$state` to access, as `[tos_state.current_node]` following the field name does.

8 Concurrency Model

TOSSIM captures the TinyOS event-driven concurrency model at interrupt and task granularity. For a description of the TinyOS concurrency model, refer to the TinyOS documentation. This document does not describe the model, merely how TOSSIM implements it.

TOSSIM models each TinyOS interrupt as a simulation event. Each event is associated with a specific mote. Simulator events run atomically with respect to one another. Therefore, unlike on real hardware, interrupts cannot pre-empt one another. After each simulator event executes, TOSSIM checks the task

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 timesta mp 3987540.
	The interrupt handler completes; the clock event re-enqueues itself fo r 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 b its 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 4: Sample Execution

queue for any pending tasks, and executes all of them in FIFO scheduling order. In TOSSIM, interrrupts do not pre-empt tasks. This method of task execution means that spinning tasks (e.g., tasks that enqueue themselves in a spin-lock fashion) will cause TOSSIM to execute that task indefinitely. Using tasks in this way is contrary to the event-driven TinyOS programming model.

Once a simulator event calls an interrupt handler, the handler executes TinyOS code through commands and events.

8.1 Sample Execution

Figure 4 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 4 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.

9 Internals

Currently, TOSSIM compiles by using alternative implementations of some core system components (e.g. `Main.nc`, `HPLUART.nc`) and incorporating a few extra files (e.g. `event_queue.c`).

The basic TOSSIM structure definitions exist in `platform/pc/nido.h`. These include the global simulation state (event queue, time, etc.) and individual node state. Hardware settings that affect multiple components (such as the potentiometer setting) currently must be modeled as global node state. `nido.h` also includes a few useful macros, such as `NODE_NUM` and `THIS_NODE`; these should be used whenever possible, so minimal code modifications will be necessary if the data representation changes.

`Nido.nc` is where the TOSSIM global state is defined. The `main()` function parses the command line options, initializes the TOSSIM global state, initializes the external communication functionality (defined in `external_comm.c, PCRadio.h`), then initializes the state of each of the simulated motes with `call StdControl.init()` and `call StdControl.start()`. It then starts executing a small loop that handles events and schedules tasks.

9.1 TOSSIM Architecture

TOSSIM replaces a small number of TinyOS components, the components that handle interrupts and the `Main` component. Interrupts are modeled as simulatgor discrete events. Normally, the core TinyOS loop that runs on motes is this:

```
while(1){
  TOSH_run_task();
}
```

`TOSH_run_task` runs tasks until the task queue is empty, at which point it puts the CPU to sleep. An interrupt will wake the mote. If the interrupt has caused a task to be scheduled, then that task will be run. While that task is running, interrupts can be handled.

The core TOSSIM loop is slightly different:

```
for (;;) {
  while(TOSH_run_next_task()) {}
  if (!queue_is_empty(&(tos_state.queue))) {
    tos_state.tos_time =
       queue_peek_event_time(&(tos_state.queue));
    queue_handle_next_event(&(tos_state.queue));
}
```

A notion of virtual time (stored as a 64-bit integer) is kept in the simulator (stored in `tos_state.tos_time`), and every event is associated with a specific mote. Most events are emulations of hardware interrupts. For example, when the clock in `hpl.c` is initialized to certain counter values, TOSSIM's implementation enqueues a clock interrupt event, whose handler calls the function normally registered as the clock interrupt handler; from this point, normal TinyOS code takes over (in terms of events being signaled, etc.) In addition, the event enqueues another clock event for the future, with its time (for the event queue) being the current time plus the time between interrupts.

When compiling for TOSSIM, the nesC compiler modifies code generation for components such that fields declared in components result in arrays of each field. The maximum number of motes that can be simulated at once is set at compile time by the size of this array; the default value is 1000 and is specified in `/apps/Makerules` with the `-fnesc-tossim-tosnodes=` flag. Since every event is associated with a specific mote, the `tos_state.current_node` field is set just before the event handler function is called, so that all modifications to component fields affect the correct mote's state. Since all TinyOS tasks enqueued by an event are run to completion before the next event is handled, tasks all execute with the same node number and access the enqueuing mote's state.

9.2 TOSSIM Implementations

To load a component, the nesC compiler uses a directory search path. The default search path is:

- The application directory
- `tos/platform/xxx` (the selected platform)
- `tos/sensorboards/xxx` (the selected sensor board(s))
- `tos/system`

- tos/lib

Entries can be prepended to the search path with the -I option to the compiler. Using the -I option therefore allows you to use alternative implementations of components provided by TinyOS. For example, one can write a TOSSIM implementation of the microphone, and put it in a separate directory, then include that directory with -I. This will cause the compiler to load the new TOSSIM implementation instead of the default one in the sensor board directory.

The component model of TinyOS means that TOSSIM implementations are not constrained to hardware abstractions. For example, TOSSIM has an implementation of the EEPROM component, instead of the lower-level hardware interface. Similarly, instead of using a bit-level simulation, one can write a TOSSIM implementation of AMStandard, for a packet-level simulation.

9.3 RFM

A TOSSIM network model has the following structure:

```
typedef struct {
  void(*init)();
  void(*transmit)(int, char);   // int moteID, char bit
  void(*stop_transmit)(int);    // int moteID
  char(*hears)(int);            // char bit,   int moteID
  bool(*connected)(int,int);    // int moteID1, int moteID2
  link_t*(*neighbors)(int);      // int moteID
} rfm_model;
```

init, transmit, stop_transmit and hears are straightforward; they're used to transmit and receive bits. The connected and neighbors functions are used for some specific low-level emulations of the radio channel during bit synchronization.

To give a feel for how the TOSSIM RFM model works, we'll step through the lossy model. The variable names have been changed to make it a bit easier to explain.

In the lossy model, each mote has four variables: the value it currently hears, whether it is transmitting or not, and a structure defining its network links. Each network link has a bit error probability and stores the value currently being transmitted along that link (after error).

Every time a mote transmit()s (after error) a 1 to another mote, that mote's hears variable is incremented. When the transmitting mote calls stop_transmit(), the variable is decremented. When the receiver calls hears(), it receives a 1 if the hears variable is non-zero. Signals are therefore modeled as a logical or.

When a mote transmits a zero or one with transmit(), the function scans through the list of network connections (neighbors()). For each link, it flips the bit with the probability specified by that link, and stores the bit that's actually transmitted (this is needed for when stop_transmit() is called, so hears can be decremented properly).

References

[1] D. Ganesan, B. Krishnamachari, A. Woo, D. Culler, D. Estrin, and S. Wicker. An empirical study of epidemic algorithms in large scale multihop wireless networks. citeseer.nj.nec.com/ganesan02empirical.html, 2002. Submitted for publication, February 2002.

[2] P. Levis, N. Lee, M. Welsh, and D. Culler. TOSSIM: Accurate and Scalable Simulation of Entire TinyOS Applications To appear in *Proceedings of the First ACM Conference on Embedded Networked Sensor Systems (SenSys 2003).*