TOSSIM System Description

Philip Levis

January 30, 2002

System Overview

TOSSIM is a discrete event simulator for TinyOS. It theoretically scales with $O(n \cdot log(n))$ time for n motes, and experimental results show it to scale well up to one thousand motes. TOSSIM is compiled directly from TinyOS code; compiling a TinyOS application only requires specifying a different target for `make` (typing `make pc` instead of `make rene` or `make mica`). Compiling a TinyOS application to native code allows a user to take advantage of traditional programming tools such as debuggers. TOSSIM has an extensive external communication system so testers can monitor packets being transmitted as well as dynamically inject packets into the simulated network. TOSSIM also allows finely grained configuration of debugging output at runtime.

Current general purpose network simulators are inadequate for TinyOS. Traditional network simulators are usually focused on protocol simulation for large area networks in which there is a large connectivity disparity in regions of the network (e.g. backbones vs. LANs). For example, `ns-2` simulates at a packet granularity and has detailed node and link specification. In contrast, TOSSIM is intended for simulating homogeneous sensor networks, in which each mote runs the same program. TOSSIM simulates at bit granularity and has an extensible but so far simple network connectivity model. This is because a good deal of research using TinyOS looks at data link level mechanisms, whether for power conservation or more efficient use of the available bandwidth.

Currently, TOSSIM has three network connectivity models: simple connectivity (all the

motes are in one cell), static connectivity (the network graph is established at startup and never changes), and space connectivity (the motes move around randomly in a square area, and potentiometer settings change their transmission range). All of these models are very simplistic: e.g. mote transmissions never cancel one another (they are always in phase), and bits are perfectly transmitted.

Unless one uses an external program to read in data produced by TOSSIM (e.g. the TOSSIM Java GUI), very little information is provided. Use of `dbg` settings by the `DBG` environment variable allow for some information on the state of the simulation (see `system/include/dbg_modes.h`). However, this information can be produced at a tremendous rate (e.g. if monitoring radio clock interrupts) and therefore might only be useful for post-run analysis.

Getting Started

TOSSIM is automatically compiled when you compile an application. Applications are compiled by entering an application directory (e.g. `nest/apps/blink`) and typing `make`. Alternatively, when in an application directory, you can type `make pc`, which will only compile a TOSSIM version of the program.

The TOSSIM executable is named `main`, and resides in `binpc`. It has the following usage:

```
main [-h|--help] [-r <static|simple|space>] [-l] [-e <file>] [-kb <val>] [-p <sec>]
<num_nodes>
```

The `-h` or `--help` options print out a more verbose usage message, specifying which dbg modes are enabled in the simulator, etc.

The `-r` option specifies the radio model to use. None of the current radio models take into account interference or noise; signals are perfectly transmitted and received. The `simple` model places all of the motes in a single radio cell; every mote can hear every other mote. The `static` model reads in a file at startup (`cells.txt`) that specifies the which motes can hear each-other. The format of this file is defined at the end of this document. The `space` model is a rudimentary model of 2D space. Motes are randomly placed at startup, then move in a random, Brownian motion-like fashion. Signal reception is based on distance between motes, with potentiometer settings mapping to a distance table that has no basis in reality.

The default model is `simple`.

The `-l` option enables invocation logging. Every event signaled, command called, and task posted by mote 0 (and only mote 0) will be written out to a TCP socket. These writes are synchronous; after writing out a message, the simulator will block until it reads a single byte in reply (the byte itself is unimportant). The invocation logging data is provided on port 10583. Using telnet to connect to port 10583 allows a user to slowly read the output; the return or enter key can be used to send 2-byte replies, causing the simulator to advance.

The `-p` option allows simulation text output to be more easily understood. If it is specified, then the simulation pauses at every system clock (not radio clock) event for the specified number of seconds. Since system clock events are often major instigators of mote behavior, this allows a user to make the simulation output be broken up enough to be read, instead of the standard (far too fast to read) constant stream.

The `-kb` options specifies the radio throughput. The default is 10 Kb/s (rene motes). It can be set to 10, 25, or 50 Kb/s. One can therefore simulate a network as if it were using a high-speed mica-specific stack; one still uses the 10 Kb stack and sets its throughput to 50 Kb/s.

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. If `-e` is specified, the simulator uses the specified file. This allows logger data to be persistent across invocations of the simulator, and also allows algorithms based on logger data to be easily compared. The file grows as needed, and is initially a sparse file (if mote 1000 writes a single byte to its last EEPROM line in an otherwise empty EEPROM, only one disk block is allocated). The TOSSIM logger component resembles the mica hardware; it is 512KB in size.

The `num_nodes` option specifies how many nodes should be simulated. A compile-time upper bound (`TOSNODES`) is specified in `TOSSIM.h`. The standard TinyOS distribution sets this value to be 1000.

dbg

TOSSIM provides configuration of debugging output at run-time. We have added debugging statements to most TinyOS system components. Every 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. For 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` will enable boot, routing and active message output.

This system allows a user to reconfigure the type of output displayed on each run of the simulator without needing to modify source files and recompile. There are several modes reserved for applications or temporary testing use, which can be safely used when writing new components. There are also modes that provide output on the internals of the simulator, such as memory allocation or the event queue. When TinyOS is compiled for mote hardware, all of the debug statements are removed through redefinition of the macro; we have verified that they add no instructions to the TinyOS code image.

The GUI

The TOSSIM visualization GUI is included in the default TinyOS release as a Java application. Currently, the simulation only supports radio packet messages (incoming and outgoing). It resides in the `tools/` directory of the default TinyOS release, and can be run by typing `java TossimGUI` when in that directory. It has a single optional parameter, `-r`, for specifying a file of radio messages to introduce to the simulation.

File Format

TossimGUI allows a file of radio packets to be specified at the command line. These packets are read in and then sent over port 10576. The file has the following format:

<time (decimal long long)> <mote (decimal short) <data0 (hex byte)> ... <data36 (hex byte)>.

Each value is separated from the other by whitespace. All whitespace is insignificant. A sample file exists at `tools/radio.txt`.

TOSSIM Internals

Currently, TOSSIM compiles by using alternative implementations of some core system components (e.g. `MAIN.c`, `CLOCK.c`) and incorporating a few extra files (e.g. `event_queue.c`). Also, TinyOS macros such as `VAR` are provided alternative definitions in `tos.h`.

The basic TOSSIM structure definitions exist in `system/tossim/tossim.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; the `static` nature of TinyOS frames prevents otherwise. `tossim.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. They will probably soon be transitioned to functions for this reason.

`MAIN.c` 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`), then initializes the state of each of the simulated motes with `MAIN_SUB_INIT` and `MAIN_SUB_START`. It then starts executing a small loop that handles events and schedules tasks. Currently, tasks are un-interruptible and execute instantaneously (in simulation time). We hope to model tasks more accurately soon.

TOSSIM catches SIGINT (control-C) to exit cleanly. This is useful when profiling code.

TOSSIM Architecture

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

```
while(1){
  while(!TOS_schedule_task()) { };
  sbi(MCUCR, SE);
  asm volatile ("sleep" ::);
  asm volatile ("nop" ::);
  asm volatile ("nop" ::);
}
```

If there are no tasks to run, the mote sleeps. An interrupt will wake the mote from sleep. 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:

```
while(!queue_is_empty(&(tos_state.queue))) {
  while(!TOS_schedule_task()) {}

  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 component 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.

TOSSIM redefines several of the TinyOS programming macros to account for a large number of motes. For example, the `TOS_FRAME` macro produces not a single structure, but an array of them. The `VAR` macro (used to access the fields of a frame) indexes into the array using the `current_node` field of the global `tos_state` structure. 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 1024. Since every event is associated with a specific mote, this field is set just before the event handler function is called, and all modifications to frame variables affect the correct mote's state. Since all TinyOS tasks enqueued by an event are run to completion before the next event is handled, the tasks all execute with the same node number and access the enqueuing mote's state.

RFM

Instead of requiring a certain level of radio simulation accuracy, TOSSIM uses an extensible radio model. By doing so, the simulation can be run with any level of accuracy a user wants (and is willing to write). While simple protocol correctness tests might be satisfied with a model that assumes perfect signal transmission, protocol performance tuning tests might want to introduce bit error and transient interference.

The interface to the radio model is defined in `rfm_model.h`:

```
typedef struct {
  void(*init)();
  void(*transmit)(int, char);
  void(*stop\_transmit)(int);
  char(*hears)(int);
  void* data;
} rfm_model;
```

The data pointer is provided so that radio models that require a large amount of state

can only allocate the memory if they're being used; otherwise, if the state is comprised of global variables in source files, the process uses up space for every model's bookkeeping.

The `int` arguments are mote IDs, while the `char` arguments are bits (0 or 1).

`init` is called at simulation startup. `transmit` is called whenever a mote calls the transmit command on the lowest level radio abstraction. `stop_transmit` is called on every radio interrupt to clear status if the mote was previously transmitting. `hears` is called on every radio interrupt when the radio is in receive state, and returns the value that is heard. The `data` pointer is provided so that large data structures can be allocated dynamically at `init`; otherwise, if the simulation includes many models, a lot of memory could be wasted.

We have implemented several radio models; we will describe a simple one that we call **static**. When the `init` method of the static model is called, the simulator tries to open a file named `cells.txt` that defines an undirected network connectivity graph in terms of mote IDs. `init` reads and builds this graph. If no file is found, every mote is made adjacent to every other. The model stores two pieces of state for every mote: whether it is currently **transmitting** a one, and a **radio_active** value. Both values are initialized to zero in `init`.

When a mote transmits a 1, its **transmitting** is set to true, and **radio_active** of every mote adjacent to it in the graph is incremented. If **transmitting** is true when `stop_transmit` is called, **radio_active** of every adjacent mote is decremented. When a mote `hears`, it tests **radio_active**; if greater than zero, the mote hears a bit. This assumes that two motes transmitting at the same time will never cancel each-other's signals. The increment/decrement element is necessary to determine when the last of the transmitters a mote can hear has finished.

The static model has many shortcomings; it assumes perfect signal propagation (there are no bit errors) and the connectivity graph doesn't change (hence the name). We have also implemented a simplistic spatial model that handles mote movement and transmit distances; motes begin at random positions and move in a random walk manner. Connectivity for a mote is recomputed at each radio clock event. Our intention is not to provide realistic world models that account for the complexities of RF propagation, instead, to provide a framework for users to develop models of complexity as fine or coarse as their work needs; we believe the models we have developed show the feasibility of more realistic models.

The only radio stack component that required modification for radio simulation was

RFM, the bit-level hardware interface. All other components in the radio stack could remain unchanged. We added a hooks in higher level components for the external communication system, but their logic is unchanged.

ADC and Spatial Models

TOSSIM also has extensible ADC and spatial models. The provided implementations of these models return identical data on every call; every ADC port has a value of zero, and all motes are always at the same coordinates. This is the interface to a spatial model:

```
typedef struct {
  double xCoordinate;
  double yCoordinate;
  double zCoordinate;
} point3D;

typedef struct {
  void(*init)();
  void (*get_position)(int, long long, point3D*);   // int moteID,
                                                    // long long time
                                                    // point3D* storage
  void* data;
} spatial_model;
```

This is the interface to an ADC model:

```
typedef struct {
  void(*init)();
  short (*read)(int, char, long long);  // int mote, char port, long long time
  void* data;
} adc_model;
```

The global `tos_state` structure holds a pointer to all of the three models: rfm, spatial and adc.

The spatial model is separate from the other two so that they can easily share state and can provide correlated readings.

External Communication

IO is a major limitation in debugging sensor networks. If a network has a complex topology (e.g. motes can be several network hops from one another), gathering real-time information on network traffic can be difficult. For example, a single mote might be producing packets with corrupted application data due to a bug. These corrupted packets could then have a ripple effect through the network. Neither of the two options to deal with this kind of bug are very appealing.

One can incorporate network traffic logging on the motes, storing packets for later retrieval. The difficulty arises when correlating the traffic observed by multiple motes; doing so requires global time synchronization, which is not incorporated into the motes by default, and adding it might make the bug disappear, just as adding logging might.

Alternatively, one could add a number of motes that solely listen and log traffic. These monitor motes could be connected to PCs, allowing real-time observation of traffic. However, if the network is composed of many transmit cells (many hops), one must use a large number of monitor motes, or at least move them around.

Instead, TOSSIM allows a global view of the network that is distinct from what the motes hear. It sends data on network and UART traffic over TCP sockets, including AM-level packet transmissions as well as bit transmissions. The former is useful for application debugging, while the latter can be used to debug data link level encoding and decoding. When TOSSIM starts, it attempts to connect to ports on the local machine, each of which is associated with a certain kind of IO data. If a connection attempts fails, TOSSIM does not retry to connect and doesn't output that data during its run.

TOSSIM also opens a server socket that runs in a separate thread. Connections made to this socket allow a user to dynamically inject packets into the network; each packet has a time stamp and a mote ID; that mote will receive the packet at that time, bypassing all of the protocol stack below the AM component (an event is enqueued whose handler signals the event that tells AM a packet has been received). If the timestamp is set for a point in the past, the packet is received at the current time in the simulator (the time of the next event in the queue).

In addition, TOSSIM has a command-line option that allows events, commands, and tasks enqueued to be monitored. If the `-l` option is specified, TOSSIM waits for a TCP connection request on a specified port before it commences the mote boot sequence – this synchronous behavior ensures that the commands issued at boot (a common source of bugs) will not be missed. The `CALL_COMMAND`, `SIGNAL_EVENT` and `POST_TASK` macros are redefined to also log each call for mote 0. Connecting to the simulator via telnet yields output such as this:

```
452497954 CLOCK_FIRE_EVENT
452497954 COUNTER_OUTPUT
452497954 INT_TO_LEDS_LEDy_off
452497954 INT_TO_LEDS_LEDg_off
452497954 INT_TO_LEDS_LEDr_off
452997922 CLOCK_FIRE_EVENT
452997922 COUNTER_OUTPUT
452997922 INT_TO_LEDS_LEDy_on
452997922 INT_TO_LEDS_LEDg_off
452997922 INT_TO_LEDS_LEDr_off
```

This output is from `cnt_to_leds`. The yellow LED is the lowest bit of the counter. The first value displayed is zero, while the second value displayed is 1. The blinks are 500,000 simulator time ticks apart, which means that in this program, the clock interrupt has been configured to interrupt at 4 Hz (as there are two million ticks in a second).

TOSSIM includes functionality for introducing and reading network traffic. This is achieved through the use of TCP sockets that are opened before the simulation begins. TOSSIM initiates a connection to localhost on a series of ports. Each of these ports, their use, and their communication format are detailed below. In addition, TOSSIM opens a single server (passive) port and waits for external connections; client sockets connecting to this port can inject messages into the network.

UART

Reading: Port 10580

The UART communication channel allows an external process to communicate with the motes over their UARTs. At startup, the TOSSIM process reads from the socket until the server side closes it. Data sent over the socket must have the following format:

Time (long long)	Mote ID (short)	Data(char)

These messages cannot be generated dynamically; they are all read in before the simulation starts. Once the server process has sent all of its messages, it closes the connection to TOSSIM.

UART communication is currently not implemented.

Writing: Port 10581

UART messages are written using the same format with which they are read. On startup, the simulator opens a connection to port 22577 on localhost and send messages over this channel.

UART communication is currently not implemented.

Radio

Reading: Port 10576

The radio communication channel allows an external process to communicate with the motes over the radio. At startup, the TOSSIM process reads from the socket until the server side closes it. Data sent over the socket must have the following format:

Time (long long)	Mote ID(short)	TOS_Msg (36 bytes)

These messages cannot be generated dynamically; they are all read in before the simulation starts. The simulation stores them as events. Sending messages in this manner bypasses the radio layers in the mote, directly calling `AM_RX_PACKET_DONE`.

Once the server has sent all of its messages, it closes the connection to TOSSIM.

This mechanism allows for the initialization of base stations at simulation startup. Since all motes must run the same code image, base station logic must be incorporated into the build of TinyOS. A message handler can be used to receive messages that switch a mote

from standard to base-station mode, and messages of this type can be introduced into the simulator through this mechanism.

Writing: Port 10577

This socket notifies an external program whenever a mote starts to transmit a packet. This notification occurs when the mote completes the transmission; if there is a lot of radio traffic, this might be significantly after it attempts to start transmitting. The packet is sent along the socket in this format:

Time (long long)	Mote ID(short)	Data (36 bytes)

Bit Writing: Port 10578

Radio messages are written to a connection made to this port. The messages have the following format:

Time (long long)	Mote ID(short)	Bit (byte)

The bit states whether the mote transmits a 1 or doesn't transmit (0). The mote is considered to be producing the same value until another message saying otherwise arrives.

This functionality is useful if debugging activity below the packet layer, such as encodings/start symbols.

Real-time Reading: Port 10579

TOSSIM spawns a thread at startup that opens a server socket and waits for connections on port 10579. Clients connecting to this port can inject messages dynamically into the network. Data sent over this socket should have the following format:

Time (long long)	Mote ID(short)	TOS_Msg (36 bytes)

If the time field specifies a time that has already passed in the simulation, the message is scheduled to be received immediately. If the time field specifies a time in the future, the message is scheduled to be received at the specified time.

If the client closes the connection, TOSSIM will wait for another connection request.

A GUI for injecting packets is included in the TinyOS release: `net.tinyos.tossim.NetworkInjector`. It uses the `net.tinyos.packet` package to provide a packet type specific GUI. If you want to add a new packet type, merely write a new packet class and include it in the `main` of `NetworkInjector`. This GUI sends all packets to mote 0.

cells.txt

There are currently two network connectivity models in TOSSIM. The simple model places every mote within a single cell; all of the motes can hear one another. This is the default network model. The second network model, the static model, uses a network connectivity graph that is determined at run-time before the simulation starts. The graph is specified by a file named `cells.txt` which must be in the directory where the program is executed. The file format is very simple.

```
0:1 1:2 4:5 9:1
```

defines a graph in which motes 0 and 1 can communicate, 2 and 1 can communicate, 4 and 5 can communicate, and 9 and 1 can communicate. Connectivity is bidirectional: `0:1` is the same as `1:0`. A sample file can be found in the root TinyOS directory.