TOSSIM

From TinyOS Wiki

This lesson introduces the **TOSSIM** simulator. You will become familiar with how to compile TOSSIM and use some of its functionality. You will learn how to:

- Compile TOSSIM.
- Configure a simulation in Python and C++.
- Inspect variables.
- Inject packets.

Note: This tutorial is for TOSSIM in TinyOS 2.0.1. If you are using TinyOS 2.0.0, it has a slightly different tutorial. The principal difference between the two is how you specify noise in RF simulation.

Contents

Introduction

TOSSIM simulates entire TinyOS applications. It works by replacing components with simulation implementations. The level at which components are replaced is very flexible: for example, there is a simulation implementation of millisecond timers that replaces `HilTimerMilliC`, while there is also an implementation for atmega128 platforms that replaces the HPL components of the hardware clocks. The former is general and can be used for any platform, but lacks the fidelity of capturing an actual chip's behavior, as the latter does. Similarly, TOSSIM can replace a packet-level communication component for packet-level simulation, or replace a low-level radio chip component for a more precise simulation of the code execution. TOSSIM is a discrete event simulator. When it runs, it pulls events of the event queue (sorted by time) and executes them. Depending on the level of simulation, simulation events can represent hardware interrupts or high-level system events (such as packet reception). Additionally, tasks are simulation events, so that posting a task causes it to run a short time (e.g., a few microseconds) in the future.

TOSSIM is a library: you must write a program that configures a simulation and runs it. TOSSIM supports two programming interfaces: Python and C++. Python allows you to interact with a running simulation dynamically, like a powerful debugger. However, since the interpretator can be a performance bottleneck when obtaining results, TOSSIM also has a C++ interface. Usually, transforming code from one to the other is very simple.

TOSSIM currently does not support gathering power measurements.

Compiling TOSSIM

TOSSIM is a TinyOS library. Its core code lives in `tos/lib/tossim (http://www.tinyos.net/tinyos-2.x/tos/lib/tossim)` . Every TinyOS source directory has an optional `sim` subdirectory, which may contain simulation implementations of that package. For example, `tos/chips/atm128/timer/sim (http://www.tinyos.net/tinyos-2.x/tos/chips/atm128/timersim)` contains TOSSIM implementations of some of the Atmega128 timer abstractions.

To compile TOSSIM, you pass the `sim` option to make:

```
$ cd apps/Blink
$ make micaz sim
```

Currently, micaz is the only platform supported by TOSSIM. You should see output similar to this:

```
mkdir -p build/micaz
placing object files in build/micaz
writing XML schema to app.xml
 compiling BlinkAppC to object file sim.o
ncc -c -fPIC -o build/micaz/sim.o -g -O0 -tossim -fnesc-nido-tosnodes=1000
-fnesc-simulate -fnesc-nido-motenumber=sim_node\(\)   -finline-limit=100000 -Wall
-Wshadow -DDEF_TOS_AM_GROUP=0x7d -Wnesc-all -target=micaz -fnesc-cfile=build/micaz/app.c
-board=micasb  -Wno-nesc-data-race BlinkAppC.nc   -fnesc-dump=components -fnesc-dump=variables
-fnesc-dump=constants -fnesc-dump=typedefs -fnesc-dump=interfacedefs -fnesc-dump=tags -fnesc-dumpfile=app.xml
  compiling Python support into pytossim.o and tossim.o
g++ -c -shared -fPIC -o build/micaz/pytossim.o -g -O0  /home/pal/src/tinyos-2.x/tos/lib/tossim/tossim_wrap.cxx
-I/usr/include/python2.3 -I/home/pal/src/tinyos-2.x/tos/lib/tossim -DHAVE_CONFIG_H
g++ -c -shared -fPIC -o build/micaz/tossim.o -g -O0  /home/pal/src/tinyos-2.x/tos/lib/tossim/tossim.c
-I/usr/include/python2.3 -I/home/pal/src/tinyos-2.x/tos/lib/tossim
  linking into shared object ./_TOSSIMmodule.so
g++ -shared build/micaz/pytossim.o build/micaz/sim.o build/micaz/tossim.o -lstdc++ -o _TOSSIMmodule.so
  copying Python script interface TOSSIM.py from lib/tossim to local directory
```

Depending on what OS you are using and what packages are installed, TOSSIM may not properly compile on the first try. Appendix A addresses some of the common causes and gives possible solutions.

Several different tasks are perfomed by the `make micaz sim` command. For details see Appendix B.

Running TOSSIM with Python

Go into the `RadioCountToLeds` application and build TOSSIM:

```
$ cd tinyos-2.x/apps/RadioCountToLeds
$ make micaz sim
```

We'll start with Python in interactive mode. To start the Python interpreter, type:

```
$ python
```

You should see a prompt like this:

```
Python 2.3.4 (#1, Nov  4 2004, 14:13:38)
[GCC 3.4.2 20041017 (Red Hat 3.4.2-6.fc3)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

The version may be different, depending on your installation.

The first thing we need to do is import TOSSIM and create a TOSSIM object. Type:

```
>>> from TOSSIM import *
>>> t = Tossim([])
```

The square brackets are an optional argument that lets you access variables in the simulation. We'll use those later. In this case, we're telling TOSSIM that we don't want to look at any variables. The way you run a TOSSIM simulation is with the `runNextEvent` function. For example:

```
>>> t.runNextEvent()
0
```

When you tell TOSSIM to run the next event, it returns 0. This means that there was no next event to run. In our case, there's no next event because we haven't told any nodes to boot. This snippet of code will tell mote 32 to boot at time 45654 (in simulation ticks) and run its first event (booting):

```
>>> m = t.getNode(32)
>>> m.bootAtTime(45654)
>>> t.runNextEvent()
1
```

Segmentation faults: If trying to do this causes TOSSIM to throw a segmentation violation (segfault), then chances are you are using a version of gcc that does not work well with the dynamic linking that TOSSIM is doing. In particular, it has been verified to work properly with 4.0.2 and 3.6, but some people have encountered problems with gcc 4.1.1.

Instead of using raw simulation ticks, you can also use the call `ticksPerSecond()`. However, you want to be careful to add some random bits into this number: having every node perfectly synchronized and only different in phase in terms of seconds can lead to strange results.

```
>>> m = t.getNode(32)
>>> m.bootAtTime(4 * t.ticksPerSecond() + 242119)
>>> t.runNextEvent()
1
```

Now, `runNextEvent` returns 1, because there was an event to run. But we have no way of knowing whether the node has booted or not. We can find this out in one of two ways. The first is that we can just ask it:

```
>>> m.isOn()
1
>>> m.turnOff()
>>> m.isOn()
0
>>> m.bootAtTime(560000)
>>> t.runNextEvent()
0
>>> t.runNextEvent()
1
```

Note that the first `runNextEvent` returned 0. This is because when we turned the mote off, there was still an event in the queue, for its next timer tick. However, since the mote was off when the event was handled in that call, `runNextEvent` returned 0. The second call to `runNextEvent` returned 1 for the second boot event, at time 560000.

If it still shows the mote is not on, some simple code will run events until it boots:

```
>>> while not m.isOn():
>>>    t.runNextEvent()
```

A Tossim object has several useful functions. In Python, you can generally see the signature of an object with the `dir` function. For example:

```
>>> t = Tossim([])
>>> dir(t)
['__class__', '__del__', '__delattr__', '__dict__', '__doc__', '__getattr__',
'__getattribute__', '__hash__', '__init__', '__module__', '__new__',
'__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__str__',
'__swig_getmethods__', '__swig_setmethods__', '__weakref__', 'addChannel',
'currentNode', 'getNode', 'init', 'mac', 'newPacket', 'radio', 'removeChannel',
'runNextEvent', 'setCurrentNode', 'setTime', 'this', 'thisown', 'ticksPerSecond', 'time', 'timeStr']
```

Methods with double-underline, "_ _", in front and back, are, generally, internal functions that you probably will not use. For instance, __init__ is called internally during the creation of an object.

The most common utility functions are:

- **`currentNode()`**: returns the ID of the current node.
- **`getNode(id)`**: returns an object representing a specific mote
- **`runNextEvent()`**: run a simulation event
- **`time()`**: return the current time in simulation ticks as a large integer
- **`timeStr()`**: return a string representation of the current time
- **`init()`**: initialize TOSSIM
- **`mac()`**: return the object representing the media access layer
- **`radio()`**: return the object representing the radio model
- **`addChannel(ch, output)`**: add *output* as an output to channel *ch*
- **`removeChannel(ch, output)`**: remove *output* as an output to channel *ch*
- **`ticksPerSecond()`**: return how many simulation ticks there are in a simulated second

You've already used `ticksPerSecond()` above. The next section discusses the previous two.

Debugging Statements

The second approach to know whether a node is on is to tell it to print something out when it boots. TOSSIM has a debugging output system, called `dbg`. There are four `dbg` calls:

- `dbg`: print a debugging statement preceded by the node ID.
- `dbg_clear`: print a debugging statement which is not preceded by the node ID. This allows you to easily print out complex data types, such as packets, without interspersing node IDs through the output.
- `dbgerror`: print an error statement preceded by the node ID
- `dbgerror_clear`: print an error statement which is not preceded by the node ID

Modify the `Boot.booted` event in `RadioCountToLedsC` to print out a debug message when it boots, such as this:

```
event void Boot.booted() {
  call Leds.led0On();
  dbg("Boot", "Application booted.\n");
  call AMControl.start();
}
```

`dbg()` takes two or more parameters. The first parameter ("Boot" in the above example) defines the output *channel*. An output channel is a string. The second and subsequent parameters are the message to output and variable formatting. `dbg()` is identical to a sprintf statement in C++. For example, `RadioCountToLedsC` has this call:

```
event message_t* Receive.receive(message_t* bufPtr, void* payload, uint8_t len) {
  dbg("RadioCountToLedsC", "Received packet of length %hhu.\n", len);
  ...
}
```

which prints out the length of received packet as an 8-bit unsigned value (%hhu).

In order to print out the simulation time, you can use:

```
  dbg("RadioCountToLedsC", "Time: %s\n", sim_time_string());
```

Once you have added the debugging statement to the event, recompile the application with `make micaz sim` and start up your Python interpreter. Load the TOSSIM module and schedule a mote to boot as before:

```
>>> from TOSSIM import *
>>> t = Tossim([])
>>> m = t.getNode(32)
>>> m.bootAtTime(45654)
```

TOSSIM's debugging output can be configured on a per-channel basis. So, for example, you can tell TOSSIM to send the "Boot" channel to standard output, but another channel, "RadioCountToLedsC", to a file. By default, a channel has no destination and messages to it are discarded.

In this case, we want to send the Boot channel to standard output. To do this, we need to import the `sys` Python package, which lets us refer to standard out. We can then tell TOSSIM to send Boot messages to this destination:

```
>>> import sys
>>> t.addChannel("Boot", sys.stdout);
1
```

The return value shows that the channel was added successfully -- although no return value seems to also indicate the channel was successfully added. Run the first simulation event, and the mote boots:

```
>>> t.runNextEvent()
DEBUG (32): Application booted.
1
```

If no message is shown, you may have to run events until that occurs:

```
>>> while not m.isOn():
>>>    t.runNextEvent()
```

The only difference between debug and error functions is the string output at the beginning of a message. Debug statements print `DEBUG (n)`, while error statements print `ERROR (n)`.

A debugging statement can have multiple output channels. Each channel name is delimited by commas:

```
event void Boot.booted() {
  call Leds.led0On();
  dbg("Boot,RadioCountToLedsC", "Application booted.\n");
  call AMControl.start();
}
```

If a statement has multiple channels and those channels share outputs, then TOSSIM only prints the message once. For example, if both the Boot channel and RadioCountToLedsC channel were connected to standard out, TOSSIM will only print one message. For example, this series of debug statements

```
event void Boot.booted() {
  call Leds.led0On();
  dbg("Boot,RadioCountToLedsC", "Application booted.\n");
  dbg("RadioCountToLedsC", "Application booted again.\n");
  dbg("Boot", "Application booted a third time.\n");
  call AMControl.start();
}
```

when configured like this:

```
>>> import sys
>>> t.addChannel("Boot", sys.stdout)
>>> t.addChannel("RadioCountToLedsC", sys.stdout)
```

will print this, after the appropriate number of *runNextEvent()'s*:

```
DEBUG (32): Application booted.
DEBUG (32): Application booted again.
DEBUG (32): Application booted a third time.
```

A channel can have multiple outputs. For example, this script will tell TOSSIM to write `RadioCountToLedsC` messages to standard output, but to write `Boot` messages to both standard output and a file named `log.txt`:

```
>>> import sys
>>> f = open("log.txt", "w")
>>> t.addChannel("Boot", f)
>>> t.addChannel("Boot", sys.stdout)
>>> t.addChannel("RadioCountToLedsC", sys.stdout)
```

There is no central list of all the debug channels used by code in TinyOS distribution. To debug an existing module or a block of code, first read the relevant code and look for `dbg()` statements to find out all the debug channels that are used. Then use the method described in this section to look at the ouput of those debug statements.

Configuring a Network

When you start TOSSIM, no node can communicate with any other. In order to be able to simulate network behavior, you have to specify a *network topology*. Internally, TOSSIM is structured so that you can easily change the underlying radio simulation, but that's beyond the scope of this tutorial. The default TOSSIM radio model is signal-strength based. You provide a set of data to the simulator that describes the propagation strengths. You also specify noise floor, and receiver sensitivity. There are some very early results that describe current sensor platforms (e.g., the mica2) in these terms. Because all of this is through a scripting interface, rather than provide a specific radio model, TOSSIM tries to provide a few low-level primitives that can express a wide range of radios and behavior.

You control the radio simulation through a Python *radio* object:

```
>>> from TOSSIM import *
>>> t = Tossim([])
>>> r = t.radio()
>>> dir(r)
['__class__', '__del__', '__delattr__', '__dict__', '__doc__',
'__getattr__', '__getattribute__', '__hash__', '__init__',
'__module__', '__new__', '__reduce__', '__reduce_ex__',
'__repr__', '__setattr__', '__str__', '__swig_getmethods__',
'__swig_setmethods__', '__weakref__', 'add', 'connected',
'gain', 'remove', 'setNoise', 'this', 'thisown',
]
```

The important ones are at the end. They are:

- **`add(src, dest, gain)`**: Add a link from *src* to *dest* with *gain*. When *src* transmits, *dest* will receive a packet attenuated by the *gain* value.
- **`connected(src, dest)`**: Return whether there is a link from *src* to *dest*.
- **`gain(src, dest)`**: Return the gain value of the link from *src* to *dest*.
- **`threshold()`**: Return the CCA threshold.
- **`setThreshold(val)`**: Set the CCA threshold value in dBm. The default is -72dBm.

The default values for TOSSIM's radio model are based on the CC2420 radio, used in the micaZ, telos family, and imote2. It uses an SNR curve derived from experimental data collected using two micaZ nodes, RF shielding, and a variable attenuator.

In addition to the radio propagation model above, TOSSIM also simulates the RF noise and interference a node hears, both from other nodes as well as outside sources. It uses the Closest Pattern Matching (CPM) algorithm. CPM takes a noise trace as input and generates a statistical model from it. This model can capture bursts of interference and other correlated phenomena, such that it greatly improves the quality of the RF simulation. It is not perfect (there are several things it does not handle, such as correlated interference at nodes that are close to one another), but it is much better than traditional, independent packet loss models. For more details, please refer to the paper "Improving Wireless Simulation through Noise Modeling," by Lee et al.

To configure CPM, you need to feed it a noise trace. You accomplish this by calling `addNoiseTraceReading` on a Mote object. Once you have fed the entire noise trace, you must call `createNoiseModel` on the node. The directory `tos/lib/tossim/noise` contains sample noise traces, which are a series of noise readings, one per line. For example, these are the first 10 lines of meyer-heavy.txt, which is a noise trace taken from Meyer Library at Stanford University:

```
-39
-98
-98
-98
-99
-98
-94
-98
-98
-98
```

If you look at the file, you can see the hardware noise floor is around -98 dBm, but there are spikes of interference around -86dBm and -87dBm.

This piece of code will give a node a noise model from a noise trace file. It works for nodes 0-6: you can change the range appropriately:

```
noise = open("meyer-heavy.txt", "r")
lines = noise.readlines()
for line in lines:
    str1 = line.strip()
    if str1:
        val = int(str1)
        for i in range(7):
            t.getNode(i).addNoiseTraceReading(val)

for i in range(7):
    t.getNode(i).createNoiseModel()
```

CPM can use a good deal of RAM: using the entire Meyer-heavy trace as input has a cost of approximately 10MB per node. You can reduce this overhead by using a shorter trace; this will of course reduce simulation fidelity. The trace must be at least 100 entries long, or CPM will not work as it does not have enough data to generate a statistical model.

The Radio object only deals with physical-layer propagation. The MAC object deals with the data link layer, packet lengths, and radio bandwidth. The default TOSSIM MAC object is for a CSMA protocol. You get a reference to the MAC object by calling `mac()` on a Tossim object:

```
>>> mac = t.mac()
```

The default MAC object has a large number of functions, for controlling backoff behavior, packet preamble length, radio bandwidth, etc. All time values are specified in terms of radio symbols, and you can configure the number of symbols per second and bits per symbol. By default, the MAC object is configured to act like the standard TinyOS 2.0 CC2420 stack: it has 4 bits per symbol and 64k symbols per second, for 256kbps. This is a subset of the MAC functions that could be useful for changing backoff behavior. Every accessor function has a corresponding set function that takes an integer as a parameter. E.g., there's `int initHigh()` and `void setInitHigh(int val)`. The default value for each parameter is shown italicized in parentheses.

- **initHigh**: The upper bound of the initial backoff range. *(400)*
- **initLow**: The lower bound of the initial backoff range. *(20)*
- **high**: The upper bound of the backoff range. This is multiplied by the exponent base to the nth power, where n is the number of previous backoffs. So if the node had its initial backoff, then the upper bound is high * base, while if it is after the second backoff then the upper bound is high * base * base. *(160)*
- **low**: The lower bound of the backoff range. This is multiplied by the exponent base to the nth power, where n is the number of previous backoffs. So if the node had its initial backoff, then the upper bound is low * base, while if it is after the second backoff then the upper bound is low * base * base. *(20)*
- **symbolsPerSec**: The number of symbols per second that the radio can transmit. *(65536)*
- **bitsPerSymbol**: The number of bits per radio symbol. Multiplying this by the symbols per second gives the radio bandwidth. *(4)*
- **preambleLength**: How long a packet preamble is. This is added to the duration of transmission for every packet. *(12)*
- **exponentBase**: The base of the exponent used to calculate backoff. Setting it to 2 provides binary exponential backoff. *(0)*.

- **maxIterations**: The maximum number of times the radio will back off before signaling failure, zero signifies forever. *(0)*.
- **minFreeSamples**: The number of times the radio must detect a clear channel before it will transmit. This is important for protocols like 802.15.4, whose synchonrous acknowledgments requires that this be greater than 1 (you could have sampled in the dead time when the radios are changing between RX and TX mode). *(2)*
- **rxtxDelay**: The time it takes to change the radio from RX to TX mode (or vice versa).*(32)*
- **ackTime**: The time it takes to transmit a synchonrous acknowledgment, not including the requisite RX/TX transition. *(34)*

Any and all of these configuration constants can be changed at compile time with #define directives. Look at `tos/lib/tossim/sim_csma.h`.

Because the radio connectivity data can be stored in a flat file, you can easily create topologies in files and then load the file using a Python script and store them into the radio object. For example, this script will load a file which specifies each link in the graph as a line with three values, the source, the destination, and the gain, for example:

```
1  2 -54.0
```

means that when 1 transmits, 2 hears it at -54 dBm. Create a file `topo.txt` that looks like this:

```
1  2 -54.0
2  1 -55.0
1  3 -60.0
3  1 -60.0
2  3 -64.0
3  2 -64.0
```

This script will read the file and store the data in the radio object:

```
>>> f = open("topo.txt", "r")
>>> for line in f:
...   s = line.split()
...   if s:
...     print " ", s[0], " ", s[1], " ", s[2];
...     r.add(int(s[0]), int(s[1]), float(s[2]))
```

Now, when a node transmits a packet, other nodes will hear it. This is a complete script for simulating packet transmission with RadioCountToLedsC. Save it as a file `test.py`:

```
#! /usr/bin/python
from TOSSIM import *
import sys

t = Tossim([])
r = t.radio()
f = open("topo.txt", "r")

for line in f:
  s = line.split()
  if s:
    print " ", s[0], " ", s[1], " ", s[2];
    r.add(int(s[0]), int(s[1]), float(s[2]))

t.addChannel("RadioCountToLedsC", sys.stdout)
t.addChannel("Boot", sys.stdout)

noise = open("meyer-heavy.txt", "r")
for line in noise:
  str1 = line.strip()
  if str1:
    val = int(str1)
    for i in range(1, 4):
      t.getNode(i).addNoiseTraceReading(val)

for i in range(1, 4):
  print "Creating noise model for ",i;
  t.getNode(i).createNoiseModel()

t.getNode(1).bootAtTime(100001);
t.getNode(2).bootAtTime(800008);
t.getNode(3).bootAtTime(1800009);
```

```
for i in range(100):
  t.runNextEvent()
```

Run it by typing `python test.py`. You should see output that looks like this:

```
1  2 -54.0
2  1 -55.0
1  3 -60.0
3  1 -60.0
2  3 -64.0
3  2 -64.0
DEBUG (1): Application booted.
DEBUG (1): Application booted again.
DEBUG (1): Application booted a third time.
DEBUG (2): Application booted.
DEBUG (2): Application booted again.
DEBUG (2): Application booted a third time.
DEBUG (3): Application booted.
DEBUG (3): Application booted again.
DEBUG (3): Application booted a third time.
DEBUG (1): RadioCountToLedsC: timer fired, counter is 1.
DEBUG (1): RadioCountToLedsC: packet sent.
DEBUG (2): RadioCountToLedsC: timer fired, counter is 1.
DEBUG (2): RadioCountToLedsC: packet sent.
DEBUG (3): RadioCountToLedsC: timer fired, counter is 1.
DEBUG (3): RadioCountToLedsC: packet sent.
DEBUG (1): Received packet of length 2.
DEBUG (3): Received packet of length 2.
DEBUG (2): Received packet of length 2.
DEBUG (1): RadioCountToLedsC: timer fired, counter is 2.
DEBUG (1): RadioCountToLedsC: packet sent.
DEBUG (2): RadioCountToLedsC: timer fired, counter is 2.
DEBUG (2): RadioCountToLedsC: packet sent.
DEBUG (3): RadioCountToLedsC: timer fired, counter is 2.
DEBUG (3): RadioCountToLedsC: packet sent.
DEBUG (1): Received packet of length 2.
```

If you set node's clear channel assessment to be at -110dBm, then nodes will never transmit, as noise and interference never drop this low. You'll see something like this:

```
1  2 -54.0
2  1 -55.0
1  3 -60.0
3  1 -60.0
2  3 -64.0
3  2 -64.0
DEBUG (1): Application booted.
DEBUG (1): Application booted again.
DEBUG (1): Application booted a third time.
DEBUG (2): Application booted.
DEBUG (2): Application booted again.
DEBUG (2): Application booted a third time.
DEBUG (3): Application booted.
DEBUG (3): Application booted again.
DEBUG (3): Application booted a third time.
DEBUG (1): RadioCountToLedsC: timer fired, counter is 1.
DEBUG (1): RadioCountToLedsC: packet sent.
DEBUG (2): RadioCountToLedsC: timer fired, counter is 1.
DEBUG (2): RadioCountToLedsC: packet sent.
DEBUG (3): RadioCountToLedsC: timer fired, counter is 1.
DEBUG (3): RadioCountToLedsC: packet sent.
DEBUG (1): RadioCountToLedsC: timer fired, counter is 2.
DEBUG (2): RadioCountToLedsC: timer fired, counter is 2.
DEBUG (3): RadioCountToLedsC: timer fired, counter is 2.
```

Because the nodes backoff perpetually, they never transmit the packet, and so subsequent attempts to send fail. Although it only takes a few simulation events to reach the first timer firings, it takes many simulation events (approximately 4000) to reach the second timer firings. This is because the nodes have MAC backoff events. If you want to simulate in terms of time, rather than events, you can always do something like this, which simulates 5 seconds from the first node boot:

```
t.runNextEvent();
time = t.time()
while time + 50000000000 > t.time():
  t.runNextEvent()
```

TOSSIM allows you to specify a network topology in terms of gain. However, this raises the problem of coming up with a topology. There are two approaches you can take. The first is to take data from a real world network and input this into TOSSIM. The second is to generate it from applying a theoretical propagation model to a physical layout. The standard file format is:

```
gain src dest g
```

where each statement is on a separate line. The *gain* statement defines a propagation gain *g* when *src* transmits to *dest*. This is a snippet of Python code that will parse this file format:

```
f = open("15-15-tight-mica2-grid.txt", "r")

for line in f:
  s = line.split()
  if (len(s) > 0):
    if (s[0] == "gain"):
      r.add(int(s[1]), int(s[2]), float(s[3]))
```

TOSSIM has a tool for the second option of generating a network topology using a theoretical propagation model. The tool is written in Java and is `net.tinyos.sim.LinkLayerModel`. The tool takes a single command line parameter, the name of a configuration file, e.g.:

```
java net.tinyos.sim.LinkLayerModel config.txt
```

The format of a configuration file is beyond the scope of this document: the tool has its own documentation (http://www.tinyos.net/tinyos-2.x/doc/html/tutorial/usc-topologies.html) . TOSSIM has a few sample configuration files generated from the tool in `tos/lib/tossim/topologies`. Note that the tool uses random numbers, these configuration files can generate multiple different network topologies. Network topology files generated from the tool follow the same format as `15-15-tight-mica2-grid.txt`. If you have topologies measured from real networks, we would love to include them in the TOSSIM distribution.

Variables

TOSSIM allows you to inspect variables in a running TinyOS program. Currently, you can only inspect basic types. For example, you can't look at fields of structs, but you can look at state variables.

When you compile TOSSIM, the make system generates a large XML file that contains a lot of information about the TinyOS program, including every component variable and its type. If you want to examine the state of your program, then you need to give TOSSIM this information so it can parse all of the variables properly. You do this by instantiating a Python object that parses the XML file to extract all of the relevant information. You have to import the Python support package for TOSSIM to do this. First, set your PYTHONPATH environment variable to point to `tinyos-2.x/support/sdk/python`. This tells Python where to find the TOSSIM packages. Then, in an interpreter type this:

```
from tinyos.tossim.TossimApp import *

n = NescApp()
```

Instantiating a `NescApp` can take quite a while: Python has to parse through megabytes of XML. So be patient (you only have to do it once). NescApp has two optional arguments. The first is the name of the application being loaded. The second is the XML file to load. The default for the latter is `app.xml`, which is the name of the file that the make system generates. The default for the former is "Unknown App." So this code behaves identically to that above:

```
from tinyos.tossim.TossimApp import *

n = NescApp("Unknown App", "app.xml")
```

You fetch a list of variables from a NescApp object by calling the function `variables` on the field `variables`:

```
vars = n.variables.variables()
```

To enable variable inspection, you pass this list to a Tossim object when you instantiate it:

```
t = Tossim(vars)
```

The TOSSIM object now knows the names, sizes, and types of all of the variables in the TinyOS application. This information allows the TOSSIM support code to take C variables and properly tranform them into Python variables. This currently only works for simple types: if a component declares a structure, you can't access its fields. But let's say we want to read the counter in `RadioCountToLedsC`. Since each mote in the network has its own instance of the variable, we need to fetch it from a specific mote:

```
m = t.getNode(0)
v = m.getVariable("RadioCountToLedsC.counter")
```

The name of a variable is usually *C.V*, where *C* is the component name and *V* is the variable. In the case of generic components, the name is *C.N.V*, where *N* is an integer that describes which instance. Unfortunately, there is currently no easy way to know what *N* is from nesC source, so you have to root through `app.c` in order to know.

Once you have a variable object (`v` in the above code), you can fetch its value with the `getData()` function:

```
counter = v.getData()
```

Because `getData()` transforms the underlying C type into a Python type, you can then use its return value in Python expressions. For example, this script will start a simulation of five nodes (actually four nodes since range(0,4) yields values 0 through 3) and run it until node 0's counter reaches 10:

```
from sys import *
from random import *
from TOSSIM import *
from tinyos.tossim.TossimApp import *

n = NescApp()
t = Tossim(n.variables.variables())
r = t.radio()

f = open("topo.txt", "r")
for line in f:
  s = line.split()
  if (len(s) > 0):
    if (s[0] == "gain"):
      r.add(int(s[1]), int(s[2]), float(s[3]))

noise = open("meyer-heavy.txt", "r")
for line in noise:
  s = line.strip()
  if s:
    val = int(s)
    for i in range(4):
      t.getNode(i).addNoiseTraceReading(val)

for i in range (4):
  t.getNode(i).createNoiseModel()
  t.getNode(i).bootAtTime(i * 2351217 + 23542399)

m = t.getNode(0)
v = m.getVariable("RadioCountToLedsC.counter")



while v.getData() < 10:
  t.runNextEvent()

print "Counter variable at node 0 reached 10."
```

Remember to change `topo.txt` in a suitable manner: since we are now considering nodes from 0 to 4, we have to specify a proper topology for the network, such as:

```
0  2 -66.0
2  0 -67.0
1  2 -54.0
2  1 -55.0
1  3 -60.0
```

```
3  1 -60.0
2  3 -64.0
3  2 -64.0
```

The TOSSIM examples (http://www.tinyos.net/tinyos-2.x/tos/lib/tossim/examples) subdirectory also has an example script, named `variables.py`.

Injecting Packets

TOSSIM allows you to dynamically inject packets into a network (if serial packets are intended, refer to TOSSIM Live). Packets can be scheduled to arrive at any time. If a packet is scheduled to arrive in the past, then it arrives immediately. Injected packets circumvent the radio stack: it is possible for a node to receive an injected packet while it is in the midst of receiving a packet from another node over its radio.

TinyOS 2.0 has support for building Python packet objects. Just like the standard Java toolchain, you can build a packet class based on a C structure. The packet class gives you a full set of packet field mutators and accessors. If an application has a packet format, you can generate a packet class for it, instantiate packet objects, set their fields, and have nodes receive them.

The `RadioCountToLeds` application Makefile has an example of how to do this. First, it adds the Python class as a dependency for building the application. Whenever you compile the app, if the Python class doesn't exist, make will build it for you:

```
BUILD_EXTRA_DEPS = RadioCountMsg.py RadioCountMsg.class
```

The Makefile also tells make how to generate RadioCountMsg.py:

```
RadioCountMsg.py: RadioCountToLeds.h
	mig python -target=$(PLATFORM) $(CFLAGS) -python-classname=RadioCountMsg RadioCountToLeds.h RadioCountMsg -o $@
```

The rule says to generate RadioCountMsg.py by calling mig with the python parameter. The Makefile also has rules on how to build Java class, but that's not important for TOSSIM. Since we've been using RadioCountToLeds so far, the Python class should be there already.

RadioCountMsg.py defines a packet format, but this packet is contained in the data payload of another format. If a node is sending a `RadioCountMsg` over AM, then the `RadioCountMsg` structure is put into the AM payload, and might look something like this:

AM Header	RadioCountMsg	AM Footer

If it is sending it over a routing protocol. the packet is put in the routing payload, and might look something like this:

AM Header	Routing Header	RadioCountMsg	AM Footer

If you want to send a `RadioCountMsg` to a node, then you need to decide how to deliver it. In the simple AM case, you place the `RadioCountMsg` structure in a basic AM packet. In the routing case, you put it in a routing packet, which you then put inside an AM packet. We'll only deal with the simple AM case here.

To get an AM packet which you can inject into TOSSIM, you call the `newPacket` function on a Tossim object. The returned object has the standard expected AM fields: *destination*, *length*, *type*, and *data*, as well as *strength*.

To include support for a packet format, you must import it. For example, to include `RadioCountMsg`, you have to import it:

```
from RadioCountMsg import *
```

This snippet of code, for example, creates a `RadioCountMsg`, sets its counter to 7, creates an AM packet, stores the `RadioCountMsg` in the AM packet, and configures the AM packet so it will be received properly (destination and type):

```
from RadioCountMsg import *

msg = RadioCountMsg()
msg.set_counter(7)
pkt = t.newPacket()
pkt.setData(msg.data)
pkt.setType(msg.get_amType())
pkt.setDestination(0)
```

The variable `pkt` is now an Active Message of the AM type of `RadioCountMsg` with a destination of 0 that contains a `RadioCountMsg` with a counter of 7. You can deliver this packet to a node with the `deliver` function. The `deliver` function takes two parameters, the destination node and the time to deliver:

```
pkt.deliver(0, t.time() + 3)
```

This call delivers pkt to node 0 at the current simulation time plus 3 ticks (e.g., 3ns). There is also a `deliverNow`, which has no time parameter. Note that if the destination of `pkt` had been set to 1, then the TinyOS application would not receive the packet, as it was delivered to node 0.

Taken all together, the following script starts a simulation, configures the topology based on topo.txt, and delivers a packet to node 0. It can also be found as `packets.py` in the TOSSIM examples (http://www.tinyos.net/tinyos-2.x/tos/lib/tossim/examples/) subdirectory.

```
#! /usr/bin/python
import sys
from TOSSIM import *
from RadioCountMsg import *

t = Tossim([])
m = t.mac()
r = t.radio()

t.addChannel("RadioCountToLedsC", sys.stdout)
t.addChannel("LedsC", sys.stdout)

for i in range(0, 2):
  m = t.getNode(i)
  m.bootAtTime((31 + t.ticksPerSecond() / 10) * i + 1)

f = open("topo.txt", "r")
for line in f:
  s = line.split()
  if s:
    if s[0] == "gain":
      r.add(int(s[1]), int(s[2]), float(s[3]))

noise = open("meyer-heavy.txt", "r")
for line in noise:
  s = line.strip()
  if s:
    val = int(s)
    for i in range(4):
      t.getNode(i).addNoiseTraceReading(val)

for i in range(4):
  t.getNode(i).createNoiseModel()

for i in range(60):
  t.runNextEvent()

msg = RadioCountMsg()
msg.set_counter(7)
pkt = t.newPacket()
pkt.setData(msg.data)
pkt.setType(msg.get_amType())
pkt.setDestination(0)

print "Delivering " + str(msg) + " to 0 at " + str(t.time() + 3);
pkt.deliver(0, t.time() + 3)


for i in range(20):
  t.runNextEvent()
```

C++

Python is very useful because it is succinct, easy to write, and can be used interactively. Interpretation, however, has a significant cost: a Python/C transition on every event is a significant cost (around 100%, so it runs at half the speed). Additionally, it's often useful to step through code with a standard debugger. TOSSIM also has support for C++, so that it can be useful in these circumstances. Because many of the Python interfaces are merely wrappers around C++ objects, much of the scripting stays the same. The two major exceptions are inspecting variables and injecting packets.

In a C++ TOSSIM, there is no variable inspection. While it is possible to request memory regions and cast them to the expected structures, currently there is no good and simple way to do so. The Python support goes through several steps in order to convert variables into Python types, and this gets in the way of C++. However, as the purpose of C++ is usually to run high performance simulations (in which inspecting variables is a big cost) or debugging (when you have a debugger), this generally isn't a big problem.

There is a C++ `Packet` class, which the Python version is a simple wrapper around. In order to inject packets in C++, however, you must build C support for a packet type and manually build the packet. There currently is no support in mig with which to generate C/C++ packet structures, and since most packets are nx_struct types, they cannot be parsed by C/C++. Furthermore, as many of the fields are nx types, they are big endian, while x86 processors are little endian. Still, if you want to deliver a packet through C++, you can do so.

Usually, the C++ and Python versions of a program look pretty similar. For example (note that this program will use a lot of RAM and take a long time to start due to its noise models):

Python

```
import TOSSIM
import sys
import random

from RadioCountMsg import *

t = TOSSIM.Tossim([])
r = t.radio()

for i in range(999):
  m = t.getNode(i)
  m.bootAtTime(5000003 * i + 1)

  for j in range(2):
    if j != i:
      r.add(i, j, -50.0)

  # Create random noise stream
  for j in range (500):
    m.addNoiseTraceReading(int(random.random() * 20) - 70)
  m.createNoiseModel()

for i in xrange(1000000):
  t.runNextEvent()
```

C++

```
#include <tossim.h>
#include <stdlib.h>


int main() {
  Tossim* t = new Tossim(NULL);
  Radio* r = t->radio();

  for (int i = 0; i < 999; i++) {
    Mote* m = t->getNode(i);
    m->bootAtTime(5000003 * i + 1);
    for (int j = 0; j < 2; j++) {
      if (i != j) {
        r->add(i, j, -50.0);
      }
    }
    for (int j = 0; j < 500; j++) {
       m->addNoiseTraceReading((char)(drand48() * 20) - 70);
    }
    m->createNoiseModel();
  }


  for (int i = 0; i < 1000000; i++) {
    t->runNextEvent();
  }
}
```

To compile a C++ TOSSIM, you have to compile the top-level driver program (e.g, the one shown above) and link it against TOSSIM. Usually the easiest way to do this is to link it against the TOSSIM objects rather than the shared library. Often, it's useful to have a separate Makefile to do this with. E.g., `Makefile.Driver`:

```
all:
	make micaz sim
	g++ -g -c -o Driver.o Driver.c -I../../tos/lib/tossim/
	g++ -o Driver Driver.o simbuild/micaz/tossim.o simbuild/micaz/sim.o simbuild/micaz/c-support.o
```

Using gdb

Since Driver is a C++ program, you can use gdb on it to step through your TinyOS code, inspect variables, set breakpoints, and do everything else you can normally do. Unfortunately, as gdb is designed for C and not nesC, the component model of nesC means that a single command can have multiple providers; referring to a specific command requires specifying the component, interface, and command. For example, to break on entry to the `redOff` command of the `Leds` interface of `LedsC`, one must type:

```
$ gdb Driver
GNU gdb Red Hat Linux (6.0post-0.20040223.19rh)
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.

This GDB was configured as "i386-redhat-linux-gnu"...Using host libthread_db library "/lib/tls/libthread_db.so.1".

(gdb) break *LedsP$Leds$led0Toggle
Breakpoint 1 at 0x804f184: file LedsP.nc, line 73.
```

nesC translates component names to C names using $. $ is a legal but almost-never-used character in some versions of C, so nesC prohibits it and uses it internally. The leading * is necessary so dbg can parse the $s. With the above breakpoint set, gdb will break whenever a mote toggles led0.

Variables have similar names. For example, to inspect the packet of RadioCountToLedsC in the RadioCountToLeds application,

```
(gdb) print RadioCountToLedsC$packet
$1 = {{header = {{data = ""}, {data = ""}, {data = ""}, {data = ""}, {
        data = ""}}, data = {{data = ""} }, footer = {{
        data = ""}, {data = ""}}, metadata = {{data = ""}, {data = ""}, {
        data = ""}, {data = ""}, {data = ""}}} }
```

For those who know gdb very well, you'll recognize this as a print of an array, rather than a single variable: there are more than 1000 instances of the message_t struct. This is because TOSSIM simulates many motes; rather than there being a single RadioCountToLedsC$packet, there is one for every node. To print the packet of a specific node, you have to index into the array. This, for example, will print the variable for node 6:

```
(gdb) print RadioCountToLedsC$packet[6]
$2 = {header = {{data = ""}, {data = ""}, {data = ""}, {data = ""}, {
      data = ""}}, data = {{data = ""} }, footer = {{
      data = ""}, {data = ""}}, metadata = {{data = ""}, {data = ""}, {
      data = ""}, {data = ""}, {data = ""}}}
```

If you want to print out the variable for the node TOSSIM is currently simulating, you can do this:

```
(gdb) print RadioCountToLedsC$counter[sim_node()]
$4 = 0
```

You can also set watchpoints (although, as to be expected, they are *slow*:

```
(gdb) watch CpmModelC$receiving[23]
Hardware watchpoint 2: CpmModelC$receiving[23]
```

This variable happens to be an internal variable in the packet-level network simulation, which keeps track of whether the radio thinks it is receiving a packet. So setting the above watchpoint will cause gdb to break whenever node 23 starts receiving a packet or returns to searching for packet preambles.

Generic components add another wrinkle. Since they use a code-copying approach, each instance of a generic has its own separate functions and variables (this is mostly due to the fact that you can pass types to them). Take, for example, `AMQueueImplP`, which is used in both the radio AM stack and the serial AM stack. If you use gdb on an application that uses both serial and radio communication and try to break on its Send.send, you'll see an error:

```
(gdb) break *AMQueueImplP$Send$send
No symbol "AMQueueImplP$Send$send" in current context.
```

nesC gives each generic a unique number. So if you have an application in which there is a single copy of AMQueueImplP, its name will actually be AMQueueImplP$0. For example, in RadioCountToLeds, this will work:

```
(gdb) break *AMQueueImplP$0$Send$send
Breakpoint 5 at 0x8051b29: file AMQueueImplP.nc, line 79.
```

If you have multiple instances of a generic in a program, there is unfortunately no easy way to figure out each one's name besides looking at the source code or stepping into them. E.g., if you application uses serial and radio communication, knowing which stack has AMQueueImpl$0 and which has AMQueueImplP$1 requires either stepping through their send operation or looking at their `app.c` files.

Conclusions

This lesson introduced the basics of the TOSSIM simulator. It showed you how to configure a network, how to run a simulation, how to inspect variables, how to inject packets, and how to compile with C++.

< **Previous Lesson** | **Top** | **Next Lesson** >

Appendix A: Troubleshooting TOSSIM compilation

TOSSIM is a C/C++ shared library with an optional Python translation layer. Almost all of the problems encountered in compiling TOSSIM are due to C linking issues. If you don't know what a linker is (or have never linked a C program), then chances are the rest of this appendix is going to be cryptic and incomprehensible. You're best off starting with learning about linkers (http://en.wikipedia.org/wiki/Linker) , why they are needed (http://www.iecc.com/linker/linker01.html) , and how you use the gcc/g++ compilers (http://www.linuxjournal.com/article/6463) to link code.

Generally, when compiling TOSSIM using `make micaz sim`, one of four things can go wrong:

1. You are using Cygwin but the `sim` compilation option can't figure this out.
2. You do not have the needed Python support installed.
3. You have Python support installed, but the make system can't find it.
4. You have Python support installed, but it turns out to be incompatible with TOSSIM.
5. You have a variant of gcc/g++ installed that expects slightly different compilation options than the normal installation.

We'll visit each in turn.

You are using Cygwin but the `sim` compilation option can't figure this out

It turns out that the Cygwin and Linux versions of gcc/g++ have different command-line flags and require different options to compile TOSSIM properly. For example, telling the Linux compiler to build a library requires `-fPIC` while the Cygwin is `-fpic`. If you're using Cygwin and you see the output look like this:

```
ncc -c -shared -fPIC -o build/micaz/sim.o ...
```

rather than

```
ncc -c -DUSE_DL_IMPORT -fpic -o build/micaz/sim.o ...
```

then you have encountered this problem. The problem occurs because Cygwin installations do not have a consistent naming scheme, and so it's difficult for the compilation toolchain to always figure out whether it's under Linux or Cygwin.

Symptom: You're running cygwin but you see the `-fPIC` rather than `-fpic` option being passed to the compiler.

Solution: Explicitly set the OSTYPE environment variable to be `cygwin` either in your `.bashrc` or when you compile. For example, in bash:

```
$ OSTYPE=cygwin make micaz sim
```

or in tcsh

```
$ setenv OSTYPE cygwin
$ make micaz sim
```

Note that often this problem occurs in addition to other ones, due to using a nonstandard Cygwin installation. So you might have more problems to track down.

You do not have the needed Python support installed

If when you compile you see lots of errors such as "undefined reference to" or "Python.h: No such file or directory" then this might be your problem. It is a subcase of the more general problem of TOSSIM not being able to find needed libraries and files.

Compiling Python scripting support requires that you have certain Python development libraries installed. First, check that you have Python installed:

```
$ python -V
Python 2.4.2
```

In the above example, the system has Python 2.4.2. If you see "command not found" then you do not have Python installed. You'll need to track down an RPM and install it. TOSSIM has been tested with Python versions 2.3 and 2.4. You can install other versions, but there's no assurance things will work.

In addition to the Python interpreter itself, you need the libraries and files for Python development. This is essentially a set of header files and shared libraries. If you have the `locate` command, you can type `locate libpython`, or if you don't, you can look in `/lib`, `/usr/lib` and `/usr/local/lib`. You're looking for a file with a name such as `libpython2.4.so` and a file named `Python.h`. If you can't find these files, then you need to install a `python-devel` package.

Symptom: Compilation can't find critical files such as the Python interpreter, `Python.h` or a Python shared library, and searching your filesystem shows that you don't have them.

Solution: Installed the needed files from Python and/or Python development RPMS.

If you have all of the needed files, but are still getting errors such as "undefined reference" or "Python.h: No such file or directory", then you have the next problem: they're on your filesystem, but TOSSIM can't find them.

You have Python support installed, but the make system can't find it

You've found libpython and Python.h, but when TOSSIM compiles it says that it can't find one or both of them. If it can't find Python.h then compilation will fail pretty early, as g++ won't be able to compile the Python glue code. If it can't find the python library, then compilation will fail at linking, and you'll see errors along the lines of "undefined reference to __Py...". You need to point the make system at the right place.

Open up `support/make/sim.extra`. If the make system can't find Python.h, then chances are it isn't in one of the standard places (e.g., /usr/include). You need to tell the make system to look in the directory where Python.h is with a `-I` option. At the top of sim.extra, under the PFLAGS entry, add

```
CFLAGS += -I/path
```

where `/path` is the path of the directory where Python.h lives. For example, if it is in `/opt/python/include`, then add `CFLAGS += -I/opt/python/include`.

If the make system can't find the python library for linking (causing "undefined reference") error messages, then you need to make sure the make system can find it. The sim.extra file uses two variables to find the library: `PYDIR` and `PYTHON_VERSION`. It looks for a file named libpython$(PYTHON_VERSION).so. So if you have Python 2.4 installed, make sure that PYTHON_VERSION is 2.4 (be sure to use no spaces!) and if 2.3, make sure it is 2.3.

Usually the Python library is found in `/usr/lib`. If it isn't there, then you will need to modify the `PLATFORM_LIB_FLAGS` variable. The -L flag tells gcc in what directories to look for libraries. So if libpython2.4.so is in `/opt/python/lib`, then add `-L/opt/python/lib` to the `PLATFORM_LIB_FLAGS`. Note that there are three different versions of this variable, depending on what OS you're using. Be sure to modify the correct one (or be paranoid and modify all three).

Symptom: You've verified that you have the needed Python files and libraries, but compilation is still saying that it can't link to them ("undefined reference") or can't find them ("cannot find -lpython2.4").

Solution: Change the sim.extra file to point to the correct directories using -L and -I flags.

You have Python support installed, but it turns out to be incompatible with TOSSIM.

Symptom: You see a "This python version requires to use swig with the -classic option" error message.

Symptom: You see a long string of compilation errors relating to SWIG and Python, e.g.:

```
/opt/tinyos-2.1.1/tos/lib/tossim/tossim_wrap.cxx: In function ‘void SWIG_Python_AddErrorMsg(const char*)’:
/opt/tinyos-2.1.1/tos/lib/tossim/tossim_wrap.cxx:880: warning: format not a string literal and no format arguments
```

Solution: Install SWIG and regenerate Python support with the sing-generate script in `tos/lib/tossim`, or install a different version of Python.

You have a variant of gcc/g++ installed that expects slightly different compilation options than the normal installation.

Symptom: g++ complains that it cannot find main() when you are compiling the shared library ("undefined reference to `_WinMain@16'").

Solution: There are two possible solutions. The first is to include a dummy main(), as described in this tinyos-help posting. (http://mail.millennium.berkeley.edu/pipermail/tinyos-help/2006-December/021719.html) The second is to add the -shared option, as described in this web page (http://curl.haxx.se/mail/archive-2003-01/0056.html) .

Hopefully, these solutions worked and you can get back to compiling, If not, then you should email tinyos-help.

You have an Import error when you use TossimApp.

Symptom: when you used the package tinyos.tossim.TossimApp in your simulation you have the error:

```
ImportError: No module named tinyos.tossim.TossimApp
```

Solution: type the command:

```
export PYTHONPATH=$PYTHONPATH:$TOSROOT/support/sdk/python
```

Appendix B: TOSSIM Compilation Steps

The make system for TOSSIM performs five basic steps:

1. Writing an XML Schema,
2. Compiling the TinyOS Application,
3. Compiling the Programming Interface,
4. Building the Shared Object, and
5. Copying Python Support.

Let's go through them one by one.

Writing an XML Schema

```
writing XML schema to app.xml
```

The first thing the TOSSIM build process does is use nesc-dump to produce an XML document that describes the application. Among other things, this document descibes the name and type of every variable.

Compiling the TinyOS Application

Besides introducing all of these new compilation steps, the `sim` option changes the include paths of the application. If the application has a series of includes:

```
-Ia -Ib -Ic
```

Then the sim option transforms the list to:

```
-Ia/sim -Ib/sim -Ic/sim -I%T/lib/tossim -Ia -Ib -Ic
```

This means that any system-specific simulation implementations will be used first, followed by generic TOSSIM implementations, followed by standard implementations. The `sim` option also passes a bunch of arguments to the compiler, so it knows to compile for simulation.

The product of this step is an object file, `sim.o`, which lives in the platform's build directory. This object file has a set of C functions which configure the simulation and control execution.

Compiling the Programming Interface

```
  compiling Python support into pytossim.o and tossim.o
g++ -c -shared -fPIC -o build/micaz/pytossim.o -g -O0 \
/home/pal/src/tinyos-2.x/tos/lib/tossim/tossim_wrap.cxx \
-I/usr/include/python2.3 -I/home/pal/src/tinyos-2.x/tos/lib/tossim \
-DHAVE_CONFIG_H
g++ -c -shared -fPIC -o build/micaz/tossim.o -g -O0 \
/home/pal/src/tinyos-2.x/tos/lib/tossim/tossim.c \
-I/usr/include/python2.3 -I/home/pal/src/tinyos-2.x/tos/lib/tossim
```

The next step compiles support for the C++ and Python programming interfaces. The Python interface is actually built on top of the C++ interface. Calling a Python object calls a C++ object, which then calls TOSSIM through the C interface. `tossim.o` contains the C++ code, while `pytossim.o` contains the Python support. These files have to be compiled separately because C++ doesn't understand nesC, and nesC doesn't understand C++.

Building the shared object

```
  linking into shared object ./_TOSSIMmodule.so
g++ -shared build/micaz/pytossim.o build/micaz/sim.o build/micaz/tossim.o -lstdc++ -o _TOSSIMmodule.so
```

The next to last step is to build a shared library that contains the TOSSIM code, the C++ support, and the Python support. If you are using tinyos-2.1.0 in **Ubuntu**, install **python2.5-dev** to avoid error while linking into shared object

Copying Python Support

```
  copying Python script interface TOSSIM.py from lib/tossim to local directory
```

Finally, there is the Python code that calls into the shared object. This code exists in `lib/tossim`, and the make process copies it into the local directory.

< **Previous Lesson** | **Top** | **Next Lesson** >