

# Tython: Scripting TOSSIM

Michael Demmer and Philip Levis

Version 0.2  
December 12, 2003

# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Getting Started</b>	<b>3</b>
2.1	Pausing and Resuming . . . . .	3
2.2	Invoking SimDriver . . . . .	4
2.3	Some Basic Commands . . . . .	4
2.4	Exploring Objects . . . . .	5
2.5	Importing and Instantiating Java . . . . .	5
<b>3</b>	<b>The Caffeinated Snake</b>	<b>6</b>
<b>4</b>	<b>Tython, SimDriver, plugins, and the radio</b>	<b>7</b>
4.1	Position-based Radio Models . . . . .	7
4.2	GUI Labels . . . . .	8
<b>5</b>	<b>Tython Data Structures</b>	<b>8</b>
5.1	Packets . . . . .	8
5.2	Motes . . . . .	9
<b>6</b>	<b>Complex Scripting</b>	<b>9</b>
6.1	Event Handlers . . . . .	10
6.2	Future Actions . . . . .	10
6.3	Periodic Actions . . . . .	11
6.4	Behavior Objects . . . . .	12
<b>7</b>	<b>Conclusions</b>	<b>13</b>
<b>8</b>	<b>Appendix A: Design Ideology</b>	<b>14</b>
<b>9</b>	<b>Appendix B: Tython Reference</b>	<b>15</b>
9.1	simcore package . . . . .	15
9.2	simutil package . . . . .	18

# 1 Introduction

Experimenting with and testing sensor networks is hard. TOSSIM, a TinyOS simulator, allows users to run and test algorithms, protocols, and applications in a controlled, reproducible environment. However, by itself a TOSSIM simulation is static. Instead of modeling behaviors such as motion or changing sensor readings, TOSSIM provides a socket-based command API for other programs to do so. On one hand, this keeps TOSSIM simple and efficient; on the other, it puts the burden of writing complex real-world models on the user.

One solution to this problem is TinyViz, a GUI that communicates with TOSSIM over the socket API. With TinyViz, users can interact with a simulation through a GUI panel, by dragging motes and setting options. These actions can be difficult to reproduce exactly (e.g., dragging a mote). Additionally, TinyViz can (as its name suggests) visualize what goes on in the network. Users can write TinyViz “plugins” in Java to extend the GUI’s functionality and issue commands to TOSSIM. However, because users must precompile their plugins, this only allows limited interactivity.

Tython (or, Tinython) complements TinyViz’s visualization by adding a scripting interface to TOSSIM. Users can interact with a running simulation through TinyViz, a Tython console, or both simultaneously. Tython is based on Jython, a Java implementation of the Python language. In addition to being a complete scripting language (with plenty of documentation, literature, tutorials, etc.), Jython makes it very easy to import and use Java classes within Python. This allows users to access the entire TinyOS Java tool chain, including packet sources, MIG-generated messages, and TinyViz.

This document explains how to get started with Tython, and explains the library functions that can be used to access TOSSIM or TinyViz. It does not contain an in-depth Python tutorial: many of those can be found at [www.python.org](http://www.python.org).

## 2 Getting Started

TinyViz and Tython sit on top of `SimDriver`, a java application that manages interactions with TOSSIM. The simplest way to use Tython is to start a TOSSIM simulation with the `-gui` option, then run `SimDriver` with the `-console` option:

```
java net.tinyos.sim.SimDriver -console
```

The Jython environment may take a little while to come up the first time; it scans all of the TinyOS jar files, but caches this information for later invocations. After booting, Tython will give you a script prompt:

```
>>>
```

### 2.1 Pausing and Resuming

If you used the `-gui` option when invoking TOSSIM (or used the `-run` option), then the simulation will be paused when Tython boots. The `sim` object allows you to pause and resume TOSSIM. Additionally, hitting the escape key and hitting return will pause a running simulation. For example:

```
>>> sim.resume()
>>> sim.pause()
```

You can either type in scripts (useful for examining data) at the prompt, or execute script files with the `execfile` command, specifying the file name:

```
>>> execfile("script.py")
```

Name	Effect
<code>sim.pause()</code>	Pause TOSSIM
<code>sim.resume()</code>	Resume TOSSIM
<code>motes[i].turnOn(time)</code>	Turn mote <i>i</i> on at time <i>time</i>
<code>motes[i].turnOff(time)</code>	Turn mote <i>i</i> off at time <i>time</i>
<code>motes[i].moveTo(x, y)</code>	Move mote <i>i</i> to <i>x,y</i>
<code>motes[i].move(x,y)</code>	Move mote <i>i</i> <i>x,y</i> from its current position
<code>comm.waitUntil(time)</code>	Make the script block until <i>time</i> in TOSSIM Note script will hang if TOSSIM is paused.
<code>comm.setSimRate(rate)</code>	Set the simulator rate to <i>rate</i> This is identical to TOSSIM's <code>-l</code> option.
<code>comm.sendRadioMessage(mote, time, message)</code>	Deliver message to mote over radio
<code>comm.sendUARTMessage(mote, time, message)</code>	Deliver message to mote over UART

Table 1: Basic Tython Commands

## 2.2 Invoking SimDriver

`SimDriver` can be invoked with a variety of options. For example, `-console` has `SimDriver` start a scripting console, while `-gui` has it start a GUI interface (`TinyViz`). The `-run` option will automatically invoke a TOSSIM simulation, while `-args` allows you to pass arguments to the TOSSIM invocation. For example,

```
# java net.tinyos.sim.SimDriver -console -gui -args "-l=1.0 -b=1" -run main.exe 20
```

Will start `TinyViz`, provide a console, and start a TOSSIM simulation (in this case, `main.exe`) of 20 motes. The TOSSIM arguments specify that it should run in real time if it can (`-l=1.0`), and the motes should boot in the first second (`-b=1`). The `-h` flag option will print the full set of `SimDriver` command line options. For TOSSIM options, refer to the TOSSIM reference manual.

When `SimDriver` invokes TOSSIM, it always enables the TOSSIM lossy radio model.

## 2.3 Some Basic Commands

In addition to a basic Jython environment, Tython has several objects that provide functions to interact with TOSSIM. Section 2.1 showed how the `sim` object can be used to pause and resume a simulation.

For all commands, specifying a time in the past will cause the command to be executed as soon as possible (i.e., at current time in TOSSIM). An easy way to have something happen immediately is to specify a time of 0. Time is measured in simulator ticks, which are at 4MHz. So, a time of one minute is 240,000,000, or 60 seconds of four million ticks each. Some basic commands are shown in Table 1. Appendix B contains a reference description of the Tython object model.

The send commands have a message as a parameter; this message must be an instance of a subclass of `net.tinyos.message.Message`. In the common case, this is a message class created with the MIG tool. For example, to send a basic AM packet to mote 2, with an address field of 2, AM type of 12, and an empty payload:

```
from net.tinyos.message import TOSMsg
msg = TOSMsg()
msg.set_addr(2)
msg.set_type(12)
msg.set_length(0)
comm.sendRadioMessage(2, 10 * 4000000, msg)
```

This packet is scheduled to arrive at mote 2, ten seconds into simulation (each second has four million ticks).

## 2.4 Exploring Objects

The Tython environment has a few predefined Python objects specific to a TOSSIM simulation. For example, the `sim` object has functions for pausing and resuming simulation. Use the `dir()` function to see the set of functions an object provides. First, type the name of the object:

```
>>> sim
net.tinyos.sim.script.reflect.Sim@5f3a5f53
```

This tells you that the `sim` object is actually an instance of the `net.tinyos.sim.script.reflect.Sim` Java class. To see the functions a `Sim` object provides, type `dir(Sim)`.

```
>>> dir(Sim)
['dumpDBG', 'exit', 'pause', 'resume', 'setSimDelay', 'simDelay', 'stop', 'stopDBGDump']
```

`dir()` can be used on both Java and Python objects. For example, `dir(Sim)` returns a Python list of strings, with each element being a method of the `Sim` class. So, `dir(dir(Sim))` lists the functions a Python list object provides:

```
>>> dir(dir(sim))
['append', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
```

Note that `dir()`'s semantics are slightly different for Python and Java. One can call `dir()` on a Python object to learn its functions, but for Java, one must call it on the class. Specifically:

```
>>> sim
net.tinyos.sim.script.reflect.Sim@5f3a5f53
>>> dir(sim)
[]
>>> dir(Sim)
['dumpDBG', 'exit', 'pause', 'resume', 'setSimDelay', 'simDelay', 'stop', 'stopDBGDump']
>>> m = dir(Sim)
>>> dir(m)
['append', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
```

Calling `dir()` with no parameters lists the set of variables bound in the Jython environment:

```
>>> dir()
['Commands', 'Interp', 'Location', 'Mote', 'Motes', 'PacketType', 'Packets', 'Radio', 'Sim',
'SimBindings', 'SimReflect', '__doc__', '__driver__', '__name__', 'comm', 'interp', 'location',
'motes', 'packets', 'radio', 'sim', 'simcore', 'sys']
```

## 2.5 Importing and Instantiating Java

Jython allows you to import Java classes with Python's import syntax. For example, to import `java.util.Enumeration`, or the entire `java.util` package:

```
>>> from java.util import Enumeration
>>> from java.util import *
```

Python is a dynamically typed language. Therefore, variables do not have type specifiers. While instantiating a `StringBuffer` in Java looks like this<sup>1</sup>,

```
StringBuffer buffer = new StringBuffer();
```

in Jython it looks like this,

```
buffer = StringBuffer()
```

Note, however, that Jython does not automatically import `java.lang.*`.

By importing Java classes, one can access the TinyOS Java toolchain:

```
from net.tinyos.message import *
from net.tinyos.packet import *

moteIF = MoteIF()
msg = BaseTOSMsg()
msg.set_addr(0)
msg.set_type(5)
moteIF.send(0, msg)
```

The above code will send a message of AM type 5 to mote 0 over the default MoteIF interface.

Access to the toolchain includes GUI-based programs. For example, one can instantiate a `SerialForwarder`:

```
from net.tinyos.sf import *
s = SerialForwarder([])
```

Note, however, that some classes (`SerialForwarder`'s GUI among them) expect to be stand-alone applications. This means that when you tell them to quit, with the quit button, for example, they may cause the entire Jython environment to quit.

### 3 The Caffeinated Snake

Jython's Java support allows a user to basically use it as a Java scripting language. However, being dynamically typed, it loses many of the correctness benefits of the Java compilation process. Essentially, there is a lot of redundancy between Java and Python, and moving back and forth between the two can be a little difficult at times.

For objects such as strings, lists, and maps, it's often easier to use Python's primitives instead of Java's, as they are syntactically built into the language. For example, compare the use of a Python map and a Java hashtable:

```
# Java
from java.util import *
h = new Hashtable()
h.put("Seed", getPacket())
print(h.get("Seed"))

# Python
h = {}
h["Seed"] = getPacket()
print h["Seed"]
```

Similarly, one can use either Java or Python to write a file:

---

<sup>1</sup>From now on, Jython code will be shown as if it were in a file, that is, without a command line prompt. Each line could be just as easily typed into a console.

```

# Java
from java.io import *
file = FileWriter("output.txt")
file.write(packets[0])

# Python
file = open("output.txt", "w")
file.write(packets[0])

```

We’ve found that, for data collection and processing, Python is much easier to use than Java. Among other things, its data structure primitives end up being a lot simpler and less error prone. However, Java allows you to interact with all of the existing structure a TinyOS distribution provides, such as packet sources, message class generation, and interaction with TinyViz.

## 4 Tython, SimDriver, plugins, and the radio

The core of SimDriver is an event bus. Java plugins can connect to this event bus, receiving TOSSIM events and sending TOSSIM commands; many of the Tython abstractions are built on top of SimDriver plugins. TinyViz, the visualization environment, also uses plugins, so users can write new visualizations. Correspondingly, there are two classes of SimDriver plugins: GUI and non-GUI.

Examples of GUI plugins include TinyViz’s radio quality visualization, and its communication visualization. Examples of non-GUI plugins include the radio model plugin and the packet logger plugin. GUI plugins are only loadable through TinyViz, while non-GUI plugins are accessible through Tython, and are used to actuate TOSSIM in response to network changes. SimDriver maintains a list of mote objects, which store state pertinent to the scripting environment, such as position and LED values.

For example, calling `mote.moveTo()` (see Table 1) modifies that mote’s position. This can change radio connectivity. The mote object uses the radio model plugin to calculate bit error rates based on new distances. The plugin will then automatically communicate the new link qualities to TOSSIM, which is unaware of mote position.

### 4.1 Position-based Radio Models

Tython and TinyViz provide two position-based radio models: disc and empirical. The first models the network using “perfect disc” connectivity, at three different disc radii (“disc10”, “disc100”, and “disc1000”). Connectivity is perfect in that links are either error-free or not present, depending on whether the target is in or outside of the disc. Although links are perfect, collisions can still occur, which will corrupt a packet.

The second model, empirical, uses an error distribution based on empirical data. Essentially, for any particular distance there is a distribution of possible bit error rates: two links of identical length can have different rates. These bit errors can cause a packet to be corrupted so that a receiver does not recover it properly. Links are directed: the error rate from A to B is distinct from the rate from B to A. Roughly speaking, motes closer than 16 units will have good connectivity, motes between 16 and 28 units will have highly variable and possibly asymmetric connectivity, and motes 30 units and higher away will have poor connectivity.

These position models boil down to a simple function: given a distance between two motes, produce a bit error rate. For disc-based models, this is a simple cutoff, which results in symmetric and perfect links. For the empirical model, the function returns a sample from the distribution corresponding to the distance.

Scripts can also explicitly specify connectivity qualities with the `radio` object. `setLossRate(src, dest, prob)` sets the bit error rate from ID `src` to ID `dest` to be probability `prob`. `comm.packetLossToBitError(prob)` calculates bit error rates from a packet loss rate<sup>2</sup>. Calling this function does not update TOSSIM; that must be done with `radio.publishModel()`. Calling this updates the entire graph to TOSSIM, which is  $O(n^2)$ .

---

<sup>2</sup>This is actually a non-trivial calculation. It makes several assumptions about the data link encoding (the mica stack, with SecDed based encoding and a 9 bit start symbol) and packet length. Specifically, the packet error rate represents a 36-byte packet. As packets grow longer, their loss rate increases, as there is a set bit error rate.

Therefore, if a large number of links are updated, it's best to only publish the model once, after all of the updates.

## 4.2 GUI Labels

When using Tython with TinyViz, scripts can add annotations to mote visualizations. The `setLabel()` method of the mote class takes three parameters: a string, an X offset, and a Y offset. For example,

```
motes[2].setLabel("queue depth: " + depth, 10, 0)
```

will put a small label by mote 2 in the TinyViz visualization panel, stating what its queue depth is (assuming the variable `depth` has been set to this value previously).

## 5 Tython Data Structures

Section 2.3 described a few of the commands that Tython provides to interact with TOSSIM. In addition to these functions, Tython provides data structures to access simulator events. For example, the `packets` variable is a dictionary of every packet that has been sent over the radio, indexed by mote ID.

```
print packets[3]
```

will print all of the packets mote 3 has sent. Section 2.3 mentioned the `motes` variable, which is a list of motes.

```
print motes[3]
```

will display the current state of mote 3.

### 5.1 Packets

The `packets` variable allows scripts to access all of the packets that motes have transmitted. The variable is a dictionary of lists, keyed by mote ID. Each list contains all of the packets a given mote has sent, in temporal order (the first packet is the first element of the list). Each element is a Java object, an instance of `net.tinyos.sim.event.RadioMsgSentEvent`. This class is used because it stores not only the message sent, but also metadata, such as time. The time represents when the packet send *completed*, not when it began.

`RadioMsgSentEvent` has three basic accessor functions: `getMessage()`, which returns a `net.tinyos.message.TOSMsg`, `getMoteID()`, which returns the sender's ID, and `getTime()`, which returns the send time.

`packets` is an immutable object; unlike normal dictionaries, a script cannot assign to it. If a script needs to transform the packet dictionary, it can make a copy with `packets.copy()`, which is a normal Python dictionary and can be modified accordingly.

In addition to normal dictionary operations, `packets` has an extra function to support TinyOS messages, `addPacketType(Message message)`. This function can create packet type specific reflections of `packets`. For example,

```
from net.tinyos.message import IntMsg
packets.addPacketType(IntMsg())
```

will create a new variable in the `/name` environment, `IntMsgs`, which contains all of the messages in `packets` that are of type `IntMsg`, as defined by active message type. This variable, `IntMsgs`, can be indexed and accessed just as `packets` is.

Note, however, that each element in the `packets` lists contains a `TOSMsg`; this means that in the above example, `IntMsgs` also has `TOSMsgs`; you cannot easily manipulate the elements as if they were actually instances of `IntMsg`, by accessing their message type specific fields, etc.

In order to transform the `IntMsgs` into a dictionary of `net.tinyos.message.IntMsg`, you must call `IntMsgs.downCast()`. For example:

Name	Effect
int getID()	Returns the mote's ID
String getCoords()	A string representation of its x,y position
double getYCoord()	The x coordinate of its position
double getXCoord()	The y coordinate of its position
double getDistance(int otherMoteID)	The euclidean distance from another mote
void turnOn()	Turn this mote on
void turnOff()	Turn this mote off
boolean isOn()	Is this mote on?
void move(double x, double y)	Mote mote (x,y) from its current position
void moveTo(double x, double y)	Move mote to (x,y)
byte[] getBytes(String variable)	Get a variable
long getLong(String variable)	Get a variable
int getInt(String variable)	Get a variable
short getShort(String variable)	Get a variable

Table 2: Mote Methods

```

from net.tinyos.msg import IntMsg
packets.addPacketType(IntMsg())

# Get a dictionary of IntMsgs, instead of TOSMsgs
myMsgs = IntMsgs.downCast()
# Print the value in the first IntMsg that mote 0 transmitted
print myMsgs[0].pop().get_val()

```

Using a `downCast()` dictionary allows you to easily access message fields. However, it discards the additional metadata (such as transmit time) that a `RadioMsgSentEvent` contains.

The variable defined by `addPacketType()` will update as more motes send packets; it represents a filter on top of `packets`. The dictionary returned by `downCast()`, however, is static; it does not update as more motes send packets.

## 5.2 Motes

The `motes` variable is a list of `Mote` objects, which provide the following methods:

The final four functions (`getBytes()`, `getLong()`, `getInt()`, and `getShort()`) allow you to request variable values from TOSSIM. The string parameter is the variable's C name. For nesC components, this takes the form of `< componentname >${< variablename >}`. For example, this code fetches some variables from the `TimerM` component of mote 2,:

```

tail = motes[2].getBytes("TimerM$queue_tail")
head = motes[2].getBytes("TimerM$queue_head")
print "head: ", head, "tail: ", tail

```

The `getBytes()` function can be used to fetch entire structures. Currently, the variable parameter does not support accessing structure fields, pointer traversals, or array elements. Therefore, while `getBytes("TimerM$mTimerList")` will return all of the timer structures as an array of bytes, `getBytes("TimerM$mTimerList[1]")` and `getBytes("TimerM$mTimerList[1].ticks")` do not work.

## 6 Complex Scripting

The previous sections described how to control and monitor TOSSIM through Tython. However, the examples given were fairly simple. By using some of Python's expressive power, one can write much more intricate and complex scripts.

Name	When?
ADCDataReadyEvent	When a mote receives an ADC reading
DebugMsgEvent	Each enabled <code>dbg</code> statement
RadioMsgSentEvent	When a mote finishes sending a radio packet
SimulationPausedEvent	In response to a pause command; refer to Section 6.2
TossimInitEvent	When Tython connects to TOSSIM
UARTMsgSentEvent	When a mote finishes sending a UART packet
VariableResolveEvent	In response to a variable resolve command
VariableValueEvent	In response to a variable request command

Table 3: TOSSIM Events

## 6.1 Event Handlers

Many Tython primitives depend on being able to handle events sent by TOSSIM, which update the known state about the simulation. For example, every time a mote transmits a packet, TOSSIM sends an event to `SimDriver`. Tython subscribes to packet event notifications and uses them to update the `packets` variable (discussed in Section 5.1).

Scripts can also register event handlers through the `interp` object. Event handlers must be functions with a single parameter, the event handled. Scripts register handlers with `interp.addEventHandler()`. The first parameter is the event handling function, the second, optional parameter is an instance of an event type. Passing an event type will register the handler for only events of that type; omitting it will have it handle all event types. For example:

```
def print_handler(event):
    print event

# Print all events
interp.addEventHandler(print_handler)

# Print only LED events
from net.tinyos.sim.event import LedEvent
interp.addEventHandler(print_handler, LedEvent())
```

The `net.tinyos.sim.event` package contains all of the events that TOSSIM generates (as well as a few which TinyViz uses internally). The TOSSIM events are:

Event handlers can be useful for waiting for a condition, or for logging information. For example, if you want to examine why a mote sends a specific (perhaps malformed) packet, you can register a handler for send events, and when the handler detects the packet, pause TOSSIM.

## 6.2 Future Actions

Most of the events described in Table 3 are issued when a mote performs a certain action. TOSSIM also provides an event for external programs to receive a callback in the future, the `SimulationPausedEvent`. TOSSIM issues a `SimulationPausedEvent` in response to a `SimulationPauseCommand`, which one can issue with `comm.pauseInFuture()`. This function takes two parameters; the first is time (in 4MHz simulator ticks), while the second is an integer ID. The integer ID allows a program to distinguish pause events when it has issued several concurrently. Scripts can obtain globally unique pause IDs with `comm.getPauseID()`.

Here is an example use of these pause events, where a user wants to print out mote state one minute into simulation:

```
from net.tinyos.sim.event import SimulationPausedEvent

id = comm.getPauseID()

def time_handler(event):
```

```

global id
if event.get_id() == id:
    print "Printing notes 1-20"
    for i in range(0, 20):
        print notes[i]

interp.addEventHandler(time_handler, SimulationPausedEvent())
comm.pauseInFuture(60 * 4000000, id)

```

To help manage some of the event and id bookkeeping, the `simutil` package provides a simpler python function interface to accomplish the same task:

```

import simutil

def time_handler():
    print "Printing notes 1-20"
    for i in range(0, 20):
        print notes[i]

simutil.call_in(60 * 4000000, time_handler);

```

Note that all the pause and event identifiers are managed by the wrapper function. In addition, note that the first parameter to `simutil.call_in` is an offset from the current time, so the above call will occur 60 seconds in the future. This is in contrast to both `comm.pauseInFuture` and the simple wrapper `simutil.call_at` which both take a absolute time value.

### 6.3 Periodic Actions

`SimulationPausedEvents` allow a script to take actions in the future without having to block using `comm.waitUntil()`; this way, they can easily be doing multiple things simultaneously. Event handlers can be made to reschedule themselves, for periodic actions:

```

def time_handler(event):
    global id
    if event.get_id() == id:
        print "Printing notes 1-20"
        for i in range(0, 20):
            print notes[i]
        # Reschedule to happen again in one minute
        comm.pauseInFuture(event.getTime() + 60 * 4000000, id)

```

This can get the job done, but is clunky, verbose and difficult. Changing the period from once a minute to twice a minute requires a new function. In order to just print out mote state every thirty seconds, a user has to write a function, request a pause ID, register a handler, and schedule an event to go off. The `simutil` package provides a convenience class, `Periodic` for exactly this reason:

```

from simutil import Periodic

def time_handler(event):
    print "Printing notes 1-20"
    for i in range(0, 20):
        print notes[i]

p = Periodic(60 * 4000000, time_handler)

```

Because Python functions are first class values, we can use closures to dynamically generate functions with the parameters a user needs. Thus telling Tython to print out a range of mote states periodically can be done with a single function call:

```

from simutil import Periodic

def print_state(low_mote, high_mote, period):
    def time_handler(event):
        print "Printing motes ", low_mote, " to ", high_mote
        for i in range(low_mote, high_mote):
            print motes[i];
    return Periodic(period, time_handler);

p = print_state(3, 5, 60 * 4000000);

```

`print_state` returns the newly allocated `Periodic` object instance, whose `stop` method can be used to stop the recurring action. Once `print_state` is called, the Tython environment will print out the state of motes `low_mote` to `high_mote`, every `period` ticks.

Closures can also be used for periodic commands to TOSSIM. For example, a routing protocol may need periodic beacons from a PC to a mote over the UART:

```

from simutil import *
def uart_beacon(mote, period, msg):
    def sendMessage(event):
        comm.sendUARTMessage(mote, 0, msg)
    return Periodic(period, sendMessage)

```

## 6.4 Behavior Objects

Using closures as described in Section 6.3 allows scripts to start periodic actions in a concise and simple manner. One use of this is to implement mote behaviors. For example, to make a mote move in a random walk, all one needs is a handler that moves a mote a small random step, then schedule this event to go off periodically.

One problem that emerges is conflicting behaviors. For example, one could call `random_walk()` to make a mote move randomly, and also call `move_to()` to make it move towards a specific destination. By registering two event handlers, the script has specified two conflicting movement behaviors, which will interleave and come into conflict.

A clean solution to solve this problem is through the use of a Python object. The object has functions for each of the different behaviors, and stores state on what behavior a mote is currently following.

From an abbreviated example adapted from the `simutil` package:

```

class MoteMover:
    handlers = {}

    def moveTo(self, moteID, step, x, y, arrivedCallback = None):
        if (self.handlers.has_key(moteID)):
            raise IndexError, "Mote ID %d already on the move" % moteID

        dx = x - mote.getXCoord();
        dy = y - mote.getYCoord();
        distance = mote.getDistance(x, y);
        nsteps = distance / step;

        def callback(pauseEvent):
            distance = mote.getDistance(x, y);
            if (distance < step):
                mote.moveTo(x, y);
                self.stop(mote); # clear handlers, cancel event, etc
                if (arrivedCallback != None):
                    arrivedCallback(mote)
            else:
                mote.move(dx / nsteps, dy / nsteps);

        periodic = Periodic(rate, callback);

```

```

self.handlers[moteID] = (periodic, 'move_to');

def randomWalk(self, mote, step):
    moteID = mote.getID();

    if (self.handlers.has_key(moteID)):
        raise IndexError, "Mote ID " + moteID + " already on the move"

    r = Random();

    def callback(pauseEvent):
        x = r.nextDouble() * step * 2.0 - step;
        y = r.nextDouble() * step * 2.0 - step;
        motes[moteID].move(x, y)

    periodic = Periodic(rate, callback);
    self.handlers[moteID] = (periodic, 'random_walk')

def stop(self, mote):
    moteID = mote.getID()
    if (self.handlers.has_key(moteID) == False):
        raise IndexError, "Mote ID % not moving" % moteID

    (periodic, what) = self.handlers.get(moteID);
    periodic.stop();
    del self.handlers[moteID]

```

The class variable `handlers` is used to store the notion of whether or not a particular mote is moving. Conflicting movement patterns are therefore detected and an error is reported. Additional movement patterns can be easily added to the object as new functions that follow the same pattern to avoid conflict.

## 7 Conclusions

Tython is a new system that adds a powerful new tool to the sensor network developer's portfolio. Through both predefined scripts and interactive console sessions, Tython aids the tasks of developing, testing, and evaluating a new algorithm or application. The core architecture is extensible, allowing developers to write new python modules and `SimDriver` plugins to add new forms of interaction and manipulation.

## 8 Appendix A: Design Ideology

The primary goal of Tython is to offer the sensor network developer a simulation environment with dynamic interactivity, enabling both unattended simulation experiments, as well as interactive debugging and simulation control. The confluence of these two goals informs the major design decisions of our project, as well as the particular interfaces exposed by the Tython commands and classes.

In some senses, the first order design question relates to the value of a dynamic simulation environment. Indeed, TOSSIM alone is a valuable tool for aiding sensor network development, offering a scalable simulation of a network of sensor motes with a fairly realistic radio model and a rich debugging capability (gdb). Yet TOSSIM alone essentially just simulates tossing a set of motes into a field and letting them go, assuming a constant radio topology. On the other hand, the real world is a dynamic place; objects and motes can move, radio connectivity changes, motes can fail. An important tool in the developer's toolbox, therefore, is the ability to simulate these dynamic interactions and thereby engineer a program that can cope with these situations.

Rather than integrating dynamically controlled interfaces to TOSSIM, the simulator instead implements a network protocol that allows interactive applications to connect to and control a running simulation. This separation allows the TOSSIM code base to remain insulated and relatively lean (maintaining performance capabilities), while more complicated calculations and interactivity can be implemented in a separate process.

TinyViz is an existing tool that enables developers to dynamically manipulate a simulation. The protocol between TOSSIM and TinyViz enables the GUI to introduce dynamics into a test application's execution. However, GUI elements do not address all the needs of a dynamic interaction. In general, a user's interaction with a GUI is non-deterministic, making repeated executions of a test case difficult if not impossible to reproduce. Furthermore, to run a set of experiments using manual manipulation of the simulation state is an extremely cumbersome task, limiting the developer to a simple cursory exploration of the potential interactivity parameter space.

The TinyViz plugin system is an available avenue to manipulate or visualize the parameters of an application. The plugin API allows a Java object to be loaded into the TinyViz GUI environment and to interact with the TOSSIM protocol. Through custom plugins, an application writer could fully express control over the dynamics of a particular experiment. In point of fact, much of the core Tython functionality is implemented using the TinyViz plugin system. However, writing custom plugins is an insufficient solution due not only to the cumbersome nature of writing experiments in a compiled language, but more importantly that it does not enable interactive code execution. In addition, being able to run an unattended simulation is a valuable convenience to developers.

Through a scripting environment, developers are able to control experiments through repeatable interactions. The Python and reflected Java commands/objects expose the key control elements of the simulation environment. These hooks can be used both in a controlled experiment framework and through console interaction with a running simulation. The interactive session is both useful for dynamic debugging and investigation of an experiment, but also as a prototyping arena for code that may become part of a longer experiment framework. The dynamic console functionality not only eases the burden of writing simulation scripts, but also enables interactive experimentation with the simulation itself. A developer can pause a simulation at a given time, use the variable resolution features to probe around (and potentially alter) the simulation state, then continue the simulation to observe the effects of the actions.

## 9 Appendix B: Tython Reference

### 9.1 simcore package

The `simcore` package is the base interface between scripts and Tython. It consists of the following reflected Java classes:

class <code>Command</code>		
void sendRadioMessage	(short moteID, long time, Message msg)	Send a fabricated <code>Radio Message</code> instance to the given <code>moteID</code> at the specified time.
void sendUARTMessage	(short moteID, long time, Message msg)	Send a fabricated <code>UART Message</code> instance to the given <code>moteID</code> at the specified time.
void turnMoteOn	(short moteID, long time)	Turns the specified <code>moteID</code> on at the given time
void turnMoteOff	(short moteID, long time)	Turns the specified <code>moteID</code> off at the given time
void setADCValue	(short moteID, long time, byte port, short value)	Sets the value of the ADC sensor at the given <code>port</code> of the specified <code>moteID</code> to a new <code>value</code>
void setSimRate	(double rate)	Adjusts the rate of simulation. Analogous to the <code>-1 TOSSIM</code> command line option.
void setLinkBitErrorProbability	(short src, long time, short dest, double loss)	Sets the radio link bit error rate from <code>src</code> to <code>dest</code> to the new value of <code>loss</code> . Note that <code>loss</code> is a bit error probability, not a packet loss probability.
double packetLossToBitError	(double packetLoss)	As per above, converts a desired packet loss rate into the appropriate bit error rate.
double distanceToPacketLoss	(double distance)	Returns the packet loss for a given distance from the current radio model.
void pauseInFuture	(long time, int pauseID)	Schedules a <code>SimulationPausedEvent</code> to be delivered at the specified <code>time</code> .
<code>VariableResolveEvent</code> resolveVariable	(short moteID, String name)	Returns an event with the address and length of the variable specified by <code>name</code> , or <code>{0, -1}</code> if it cannot be resolved.
<code>VariableValueEvent</code> requestVariable	(long addr, short length)	Returns <code>length</code> bytes from TOSSIM memory at <code>addr</code> .
void setDBG	(long dbg)	Enables the specified <code>dbg</code> flag.
int getPauseID()		Returns a unique identifier for use in <code>pauseInFuture</code> .

class Interp		
int addEventHandler	(PyFunction callback)	Register the specified <code>callback</code> to be called with events. Returns a unique identifier for the event handler.
int addEventHandler	(PyFunction callback, PyJavaClass eventclass)	Same as above, but only deliver events of the specified Java class <code>eventclass</code>
int removeEventHandler	(int id)	Remove a registered event handler with identifier <code>id</code> .

class Mote		
int getID()		Return the mote id.
String getCoord()		Return a string of the mote location in the form “(x, y)”.
double getXCoord()		Return the X coordinate of the mote.
double getYCoord()		Return the Y coordinate of the mote.
String toString()		Return a string depiction of the mote state.
double getDistance	(int moteID)	Return the distance to another mote, specified by <code>moteID</code> .
double getDistance	(double x, double y)	Return the distance to the specified location.
void turnOn()		Power up the mote.
void turnOff()		Power down the mote.
boolean isOn()		Return the state of whether or not the mote is on.
void setLabel	(String label, int xoff, int yoff)	Sets a descriptive string <code>label</code> to be displayed if the TinyViz GUI is running. <code>xoff</code> and <code>yoff</code> specify the coordinate offsets from the mote’s position that the string should be displayed.
void move	(double dx, double dy)	Move the mote by the given <code>dx</code> and <code>dy</code> offsets.
void moveTo	(double x, double y)	Move the mote to the absolute position given by <code>x</code> , <code>y</code>
byte[] getBytes	(String var)	Return the mote variable named by <code>var</code> as a byte array.
long getLong	(String var)	Return the mote variable named by <code>var</code> as a long integer.
int getInt	(String var)	Return the mote variable named by <code>var</code> as an integer.
short getShort	(String var)	Return the mote variable named by <code>var</code> as a short integer.
byte getByte	(String var)	Return the mote variable named by <code>var</code> as a byte.

<b>class Radio</b>	
String getCurModel()	Return the name of the current radio model.
void setCurModel (String modelname)	Set the current radio model.
void setScalingFactor (double scalingFactor)	Set the distance scaling factor.
double getLossRate (int senderID, int receiverID)	Get the packet loss rate between the specified motes.
void setLossRate (int senderID, int receiverID, double prob)	Get the packet loss rate between the specified motes.
void printLossRates()	Dump out the current loss rate graph.
void setAutoPublish (boolean autoPublish)	Set a flag indicating whether or not the radio should automatically propagate model changes due to motes moving to TOSSIM.
void updateModel()	Force an update of the internal radio model connectivity graph.
void publishModel()	Publish the current connectivity graph to TOSSIM.

<b>class Sim</b>	
void pause()	Pause the simulation.
void resume()	Resume the simulation.
void stop()	Stop the simulation.
long getTossimTime()	Return the “current” time in TOSSIM time units. This is actually the time that the last TOSSIM event was received by the system.
void exit (int errcode)	Exit the process.
void setSimDelay (long delay_ms)	Set the delay parameter.
void dumpDBG (String filename)	Start spooling TOSSIM DBG messages to the given <b>filename</b>
void stopDBGDump()	Stop dumping out DBG messages.

## 9.2 simutil package

The `simutil` package is a python package that implements a some richer functional additions to much of the `simutil` functionality. It provides the following python classes and functions:

Functions		
<code>call_at</code>	(when, callback, args = None)	Schedules an event to call <code>callback</code> with optional <code>args</code> at time <code>when</code> in the future. Returns the event ID.
<code>call_in</code>	(delay, callback, args = None)	Same as <code>call_at</code> , except the call takes place in <code>delay</code> ticks past the current time. Returns the event ID.
<code>call_cancel</code>	(id)	Cancels the previously scheduled call.

class <code>Periodic</code>		
<code>Periodic</code>	(interval, callback, args = None, call_immediate = 1)	The class constructor interacts with the event system to schedule <code>callback</code> to be called with optional <code>args</code> repeatedly every <code>interval</code> ticks. The <code>call_immediate</code> option controls whether the first call is at time 0 or waits for an interval.
<code>void</code> <code>stop()</code>		Stops the periodic call.
<code>boolean</code> <code>is_stopped()</code>		Returns whether or not the object is stopped.

class <code>MoteMover</code>		
<code>MoteMover()</code>		Creates a <code>MoteMover</code> instance.
<code>void</code> <code>moveTo</code>	(mote, step, x, y, callback = None, rate = -1)	Move the given <code>mote</code> object in increments to the (x, y) location. Call the <code>callback</code> (if any) when it gets there. The <code>step</code> parameter controls how far to move on each interval, The <code>rate</code> parameter controls the frequency of movement, -1 means use the class default.
<code>void</code> <code>randomWalk</code>	(mote, step, rate = -1)	Similar to <code>moveTo</code> , though move the mote in a random direction at each tick. The <code>step</code> and <code>rate</code> parameters have the same meaning.
<code>void</code> <code>stop</code>	(mote)	Stop the given mote's movement.
<code>boolean</code> <code>isMoving</code>	(mote)	Return whether or not the given mote is moving.
<code>void</code> <code>setDefaultRate</code>	(rate)	Set the default movement rate.