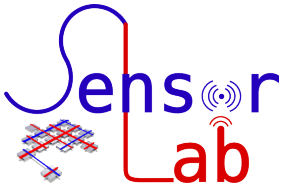
Faculty of Mathematics and Computer Science

– Georg-August University Göttingen –

Telematics Group

Lab 3 - Simulation in TinyOS with TOSSIM

Sensorlab



Contents

1 Introduction **1**

2 TOSSIM Compilation **1**

3 Python programming interface in TOSSIM **2**

4 Use of Debugging Statements **5**

5 Creating the Network Topology **7**

6 Your own application **12**

6.1 Assignment 3 (50 Points) 12

1 Introduction

TOSSIM (TinyOS SIMulator) is a simulation platform for TinyOS applications which can be used to run entire TinyOS applications without the physical involvement of the sensor motes on a regular PC. Since debugging sensor applications can be a tricky task due to lacking or harder to use output methods, it is often beneficial to simulate an application before actually testing it on sensor nodes itself.

TOSSIM is a TinyOS library and works mainly by replacing components with simulation implementations. Therefore you have to write a program that configures a simulation and runs it. TOSSIM supports two programming interfaces: Python and C++. Python allows us to interact with a running simulation dynamically, like a powerful debugger. However, since the interpreter can be a performance bottleneck when obtaining results, TOSSIM also has a C++ interface. During this lab, we will only use the python interface for the sake of simplicity.

Further information

- `http://tinyos.stanford.edu/tinyos-wiki/index.php/TOSSIM`

2 TOSSIM Compilation

The core code of TOSSIM lives in `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` contains TOSSIM implementations of some of the Atmega128 timer abstractions. In this lab, we are using the `Blink` application which boots up the node, toggles the red led and goes to sleep. We are using the iris platform in this case, since it is also available in the laboratory. Other platforms like micaZ, mica2 etc. can also be used for simulation. To build an application for simulation, we pass the sim option to make:

```
cp -a /opt/tinyos-2.1.2/apps/Blink $HOME/BlinkSim
cd $HOME/BlinkSim
make iris sim
```

The output should be similar to the following (do **not** type this in, the build system does it for you!):

```
mkdir -p simbuild/iris
  placing object files in simbuild/iris
```

```
  writing XML schema to app.xml
  compiling BlinkAppC to object file sim.o
ncc -c -shared -fPIC -o simbuild/iris/sim.o -g -O0 -tossim -fnesc-nido-tosnodes=1000 -fnesc
    -simulate -fnesc-nido-motenumber=sim_node\(\) -fnesc-gcc=gcc -Wall -Wshadow -Wnesc-all
    -target=iris -fnesc-cfile=simbuild/iris/app.c -board=micasb -DDEFINED_TOS_AM_GROUP=0x22
     --param max-inline-insns-single=100000 -DIDENT_APPNAME=\"BlinkAppC\" -DIDENT_USERNAME
    =\"sl-pc\" -DIDENT_HOSTNAME=\"SensorLabPC03\" -DIDENT_USERHASH=0x31cb7cbbL -
    DIDENT_TIMESTAMP=0x4def175dL -DIDENT_UIDHASH=0xedeb41deL -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 and C libraries into pytossim.o, tossim.o, and c-support.o
g++ -c -shared -fPIC -o simbuild/iris/pytossim.o -g -O0 -DIDENT_APPNAME=\"BlinkAppC\" -
    DIDENT_USERNAME=\"sl-pc\" -DIDENT_HOSTNAME=\"SensorLabPC03\" -DIDENT_USERHASH=0
    x31cb7cbbL -DIDENT_TIMESTAMP=0x4def175dL -DIDENT_UIDHASH=0xedeb41deL /opt/tinyos-2.1.1/
    tos/lib/tossim/tossim_wrap.cxx -I/usr/include/python2.5 -I/opt/tinyos-2.1.1/tos/lib/
    tossim -DHAVE_CONFIG_H
/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
g++ -c -shared -fPIC -o simbuild/iris/tossim.o -g -O0 -DIDENT_APPNAME=\"BlinkAppC\" -
    DIDENT_USERNAME=\"sl-pc\" -DIDENT_HOSTNAME=\"SensorLabPC03\" -DIDENT_USERHASH=0
    x31cb7cbbL -DIDENT_TIMESTAMP=0x4def175dL -DIDENT_UIDHASH=0xedeb41deL /opt/tinyos-2.1.1/
    tos/lib/tossim/tossim.c -I/usr/include/python2.5 -I/opt/tinyos-2.1.1/tos/lib/tossim
g++ -c -shared -fPIC -o simbuild/iris/c-support.o -g -O0 -DIDENT_APPNAME=\"BlinkAppC\" -
    DIDENT_USERNAME=\"sl-pc\" -DIDENT_HOSTNAME=\"SensorLabPC03\" -DIDENT_USERHASH=0
    x31cb7cbbL -DIDENT_TIMESTAMP=0x4def175dL -DIDENT_UIDHASH=0xedeb41deL /opt/tinyos-2.1.1/
    tos/lib/tossim/hashtable.c -I/usr/include/python2.5 -I/opt/tinyos-2.1.1/tos/lib/tossim
  linking into shared object ./_TOSSIMmodule.so
g++ -shared -fPIC simbuild/iris/pytossim.o simbuild/iris/sim.o simbuild/iris/tossim.o
    simbuild/iris/c-support.o -lstdc++ -o _TOSSIMmodule.so
  copying Python script interface TOSSIM.py from lib/tossim to local directory

*** Successfully built iris TOSSIM library.
```

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.

While knowing these details is not required, they might come in handy at some point.

3 Python programming interface in TOSSIM

In this section we will the `RadioCountToLeds` application which maintains a 4Hz counter, broadcasting its value in an AM packet every time it gets

updated. A `RadioCountToLeds` node that receives a counter, displays the bottom three bits on its LEDs. This application is a useful test to demonstrate that basic AM communication and the corresponding timer work.

Let us now look at the `RadioCountToLeds` application and build TOSSIM:

```
cp -a /opt/tinyos-2.1.2/apps/RadioCountToLeds $HOME/RadioCountToLeds
cd $HOME/RadioCountToLeds
make iris sim
```

We'll start with running a simulation in Python. We can either write a script and just tell Python to run it, or we can use Python interactively. To create a script, the first line in the file will be:

```
#!/usr/bin/python
```

This specifies the location of the python interpreter. Now, before we continue writing the script, we'll start python in interactive mode first, to explore some of TOSSIM's functionality. To do this, start the interpreter by executing:

```
python
```

NOTE: All following commands can also be copied to a script file and execute all together instead of typing them into the interactive Python shell one after another. The complete simulation script can be found at the end of section 5.
Output:

```
Python 2.6.5 (r265:79063, Apr 16 2010, 13:09:56)
[GCC 4.4.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

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

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

The square brackets are may contain optional arguments that let us access variables in the simulation. In this case, however we're telling TOSSIM that we don't want to look at any internal variables.

In order to run a TOSSIM simulation, the `runNextEvent` method of the TOSSIM object is used. For example:

```
>>> t.runNextEvent()
False
```

When we tell TOSSIM to run the next event, it returns False. This means that there was no next event to run. In our case, there's no next event since we did not instruct 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()
True
```

Instead of using raw simulation ticks, we can also use the call `ticksPerSecond()`. However, we need 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()
True
```

Now, `runNextEvent` returns True, since there was an event to run. But we have no way of knowing whether the node has booted or not. We can figure that out in the following two manners:

```
>>> m.isOn()
True
>>> m.turnOff()
>>> m.isOn()
False
>>> m.bootAtTime(560000)
>>> t.runNextEvent()
False
>>> t.runNextEvent()
True
```

Note that the first `runNextEvent` returned False. 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 False. The second call to `runNextEvent` returned True for the second boot event, at time 560000. If it still shows the mote is not on, some simple code can be used to keep running events until it boots:

```
>>> while (m.isOn()==False):
>>>   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']
```

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

4 Use of 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

We can make a copy of the `RadioCountToLeds` application and 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. In this regard, dbg() is identical to a printf 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 %hu.\n", len);
  ...
}
```

which prints out the length of received packet as an 8-bit unsigned value (%hu). In order to print out the simulation time, you can use:

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

Once we have added the debug statement, recompile the application with make iris sim and start up the Python interpreter. Now we follow the same steps as before

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

TOSSIM's debugging output has to be configured on a per-channel basis. So, we 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 forward the Boot channel to the standard output. To do this, we need to import the sys package in Python, which provides us a handle to the standard out. We can then use this to tell TOSSIM to write messages sent to the Boot channel to the standard output:

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

The return value indicates that the channel was added successfully – although no return value seems to also indicate the channel was successfully added. We run the first simulation event, and the mote boots:

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

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

```
>>> while (m.isOn()==False):
>>>   t.runNextEvent()
```

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 debug 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 the standard output, TOSSIM will only print out 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 (second message).\n");
  dbg("Boot", "Application booted (third message).\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 (second message).
DEBUG (32): Application booted (third message).
```

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)
```

5 Creating the Network Topology

By default, in TOSSIM no nodes can communicate with each other. In order to be able to simulate network behavior, we have to specify a network topology. The default TOSSIM radio model is signal-strength based. We provide a set of data to the simulator that describes the propagation strengths, specifies the noise floor, and receiver sensitivity. We control the radio simulation through the radio Python object. Let us take a look at the provided methods:

```
>>> 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*.

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.

Further information

- For more details, please refer to the paper "Improving Wireless Simulation through Noise Modeling," by Lee et al. (`http://geometry.stanford.edu/papers/lcl-iwsnm-07/lcl-iwsnm-07.pdf`)

In order to configure CPM, you have to feed it a noise trace, which is accomplished by calling `addNoiseTraceReading` on a Mote object. Once you have fed the entire noise trace, you can call `createNoiseModel` on the node. The directory `$TOSDIR/lib/tossim/noise` contains sample noise traces, which are represent series of noise readings, one per line. For example, these are the first 10 lines of meyer-heavy.txt, which is a noise trace:

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

The following piece of code will give you a noise model from a noise trace file. It works for nodes 0-6. We can change the range as we please:

```
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(0, 7):
            t.getNode(i).addNoiseTraceReading(val)

for i in range(0, 7):
    t.getNode(i).createNoiseModel()
```

The CPM can use a good deal of RAM: Using the entire Meyer-heavy trace as input has a cost of approximately 10MB per node and increases simulation start up time a good bit. We 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 since it might not have enough data to generate a statistical model. In this lab, we will use the first 100 lines of the noise trace, since high fidelity noise modeling is not required at the moment.

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. We 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 access function has a corresponding set function that takes an integer as a parameter. E.g., there's int initHigh() and void setInitHigh(int val).

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

Since the radio connectivity data can be stored in a flat file, we can easily create topologies in files and then load the file using Python and store them as part of the radio object. For example, the following script loads 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
```

This means that when 1 transmits, 2 hears it at -54 dBm. Now we create a file topo.txt as following:

```
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 reads the file and stores the data in the radio object:

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

Now, as soon as a node transmits a packet, other nodes will receive it. In the following you can find the complete script for simulating packet transmission with `RadioCountToLedsC`. Save it as a file runSim.py:

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

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

lines = f.readlines()
for line in lines:
  s = line.split()
  if (len(s) > 0):
    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")
lines = noise.readlines()
for line in lines:
  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(0, 100):
  t.runNextEvent()
```

Execute it by typing `python runSim.py`. The output should be similar to 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 (second message).
DEBUG (1): Application booted (third message).
DEBUG (2): Application booted.
DEBUG (2): Application booted (second message).
DEBUG (2): Application booted (third message).
DEBUG (3): Application booted.
DEBUG (3): Application booted (second message).
DEBUG (3): Application booted (third message).
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 we set node's clear channel assessment to be at -200 dBm, 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 (second message).
DEBUG (1): Application booted (third message).
DEBUG (2): Application booted.
DEBUG (2): Application booted (second message).
DEBUG (2): Application booted (third message).
DEBUG (3): Application booted.
DEBUG (3): Application booted (second message).
DEBUG (3): Application booted (third message).
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.
```

6 Your own application

6.1 Assignment 3 (50 Points)

1. Modify a copy of `RadioCountToLeds`. When executed in the TOSSIM environment, it should display messages (including the simulation time) whenever a packet is sent or received. Make sure that the sender's node ID is displayed as part of the output of received packets. You can use all functions from the `AMPacket` interface to do so. Take a look at the API documentation. Include output in your report (35 Points).

2. Also modify your program and show that packets can be dropped, given a different, appropriate noise model. Include output in your report (10 Points).

3. Add your modified source code of `RadioCountToLeds` to your report's source archive, but explain your modifications in the report (5 Points).