Tossim教程

Tossim简介

TOSSIM（TinyOS simulator）是TinyOS自带的一个仿真工具，可以支持大规模的网络仿真。由于TOSSIM运行和传感器硬件相同的代码，所以仿真编译器能直接从TinyOS应用的组件表编译仿真程序。通过替换TinyOS 下层部分硬件相关的组件，TOSSIM把硬件中断换成离散仿真事件，由仿真器事件抛出的中断来驱动上层应用，其他的TinyOS组件尤其是上层的应用组件都无须更改。
Tossim是一个库，你必须写程序配置仿真运行，TOSSIM支持两种编程接口：Python和C++。Python允许你动态的和仿真交互，就像一个强大的debugger,但是Python的解释器本身就是一个性能瓶颈，执行效率没有c++高。这两种接口各有优缺点。下面我们只介绍Python这种接口。

编译TOSSIM

TOSSIM是TinyOS系统的一个程序库，其核心代码位于tos/lib/tossim。在每个TinyOS源代码目录里，都可能有一个对应的sim子目录，其中包含改代码的仿真实现。例如，tos/chips/atm128/timer/sim含有Atmega128定时器的仿真组件。
为了编译TOSSIM，你应该在make命令后面加上可选的“sim”选项：

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

现在TOSSIM只支持micaz平台。如果TOSSIM编译通过，会得到下面的输出信息：

```
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
```

基于Python的仿真

下面将以**RadioCountToLeds**应用程序为例，演示TOSSIM仿真的一般步骤。
RadioCountToLeds应用程序有一个4Hz的计数器，每当数值更新时，就把数值无线广播出去。当另一个RadioCountToLeds节点接收到技术值，就将其低3位显示在LED灯上。
用cd命令切换到RadioCountToLeds（在apps目录下）程序所在目录：

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

我们使用python的交互模式，开启Python解释器：

```
$ python
```

你应该会得到类似下面的输出：

```
Python 2.7.6 (default, Jun 22 2015, 17:58:13)
[GCC 4.8.2] on linux2
Type "help", "copyright", "credits" or "license" for more informati
on.
>>>
```

创建TOSSIM对象。导入TOSSIM仿真器，并创建一个TOSSIM对象。输入如下命令：

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

运行TOSSIM仿真，利用t.runNextEvent命令可以运行一个仿真器事件。t就是刚才创建的TOSSIM对象，runNextEvent是TOSSIM对象的一个函数，输入函数格式如下：

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

此时输入t.runNextEvent，返回0。表示没有事件需要运行，因为当前还没有启动任何节点。下面这段代码将在45654时间点（仿真时间标记点，相当于仿真中的最小时间单位）启动32号节点，然后运行它的第一个事件，即Boot.booted事件。此时，执行runNextEvent返回1，因为它有一个启动事件并成功运行了该事件，代码如下：

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

另外，可以调用tickPerSecond函数来表示一秒的仿真时间点总和，调用函数的代码如下：

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

仿真运行中，一共有两种方法判断节点是否启动，其中一种方法就是利用节点对象中的isOn命令直接查询，代码如下：

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

注意到，节点关闭并再重新开启后执行的第一个runNextEvent函数返回0，这是因为当节点关掉时，队列里仍然有一个事件。但是，当轮到处理这个事件时，节点仍处于关闭状态，runNextEvent就返回0。第二个runNextEvent命令针对于560000时间点的启动事件，故返回1。
TOSSIM对象还有很多功能函数。在Python里，一般可用dir函数查看某对象的功能函数，如下：

```
>>> t = Tossim([])
>>> dir(t)
['__class__', '__del__', '__delattr__', '__dict__', '__doc__', '__g
etattr__',
'__getattribute__', '__hash__', '__init__', '__module__', '__new_
_',
'__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__str_
_',
'__swig_getmethods__', '__swig_setmethods__', '__weakref__', 'addCh
annel',
'currentNode', 'getNode', 'init', 'mac', 'newPacket', 'radio', 'rem
oveChannel',
'runNextEvent', 'setCurrentNode', 'setTime', 'this', 'thisown', 'ti
cksPerSecond', 'time', 'timeStr']
```

常用函数有：

```
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 inte
ger
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 s
imulated second
```

调试语句

TOSSIM有一个称为dbg的调试系统。
dbg：打印调试信息，以节点ID开头。
修改RadioCountToLedsC里的**Boot.booted**事件，当启动时打印一条debug信息，例如：

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

dbg()有两个或者更多的参数，第一个参数(“Boot”)，定义了输出通道，输出通道是一个字符串，dbg命令和C++语言中的sprintf语句十分相似。例如，RadioCountToLedsC有如下调用：

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

输出通道需要利用TOSSIM对象中的addChannel函数将它与具体输出设备绑定，为了这些，我们需要导入Python的sys包，告诉TOSSIM将Boot信息输出到指定地方。

```
>>> from TOSSIM import *
>>> t = Tossim([])
>>> m = t.getNode(32)
>>> m.bootAtTime(45654)
>>> import sys
>>> t.addChannel("Boot", sys.stdout);
1
```

返回信息表示绑定成功，运行第一个仿真事件，节点启动：

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

如果没有信息输出，你可能需要运行事件直到它输出：

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

如果一条语句有多个通道并且这些通道共享输出，TOSSIM只打印这些信息一次，例如，Boot通道和RadioCountTOLedsC通道连接到标准输出，将只会打印出一条信息。例如：

```
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();
}
```

在python里输入如下命令：

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

运行runNextEvent之后，将会打印出如下结果：

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

网络配置

仿真刚启动时，节点之间不能互相通信。只有配置好网络拓扑结构，TOSSIM仿真才能模拟网络行为。TOSSIM中默认的无线电模型是以信号强度为基础的，需要程序员通过脚本接口提供一组描述网络传播强度、底噪声和接收灵敏度的数据。TOSSIM已经提供了一组基于CC2420芯片的数据。
通过Python中的radio对象可以仿真无线电行为，利用dir命令还可以查看radio对象的具体函数，其代码如下：

```
>>> 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',
]
```

TOSSIM不但可以仿真无限电的传播模型，还可以利用就近匹配算法（CPM）来模拟射频噪声、节点间的相互干扰以及外部信号源节点的干扰。CPM算法利用噪声道文件产生一个统计模型。通过调用节点对象的addNoiseTraceReading命令创建噪声模型。在tos/lib/tossim/noise目录下有几个简单的噪声文件。例如，meyer-heavy.txt文件保存的是斯坦福大学Meyer图书馆的噪声道数据，它的前十行如下：

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

下面是一段根据噪声道数据分别为节点0-7创建噪声模型的代码：

```
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()
```

因为无线电的链路信息可以存在文本文件里，所以TOSSIM允许创建一个描述拓扑结构的文件，然后用Python脚本加载该文件，并把这些拓扑信息保存到radio对象里。在文本文件中，3个数值指明一条链路：源节点、目标节点和增益。例如：

```
1  2 -54.0
```

表示节点1发送消息，节点2以-54dBm的增益接收。
创建一个拓扑文件topo.txt，其内容如下：

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

以下脚本代码会读取该文件，并把数据存到radio对象里：

```
>>> 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]))
```

此时，当一个节点发送数据包，其邻居节点就能接收到数据包。下面是RadioCountToLedsC实例的完整仿真脚本代码，另存为test.py文件：

```
#! /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()
```

运行该脚本，你应该能看到如下输出：

```
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.
```

这些调试输出语句，都对应着RadioCountToLedsC.nc文件里的dbg调试语句，前面打印的网络拓扑语句是在test.py文件里

```
print " ", s[0], " ", s[1], " ", s[2];
```

这条语句输出的。

> **注意**
> 同学们可以试着修改调试语句的输出或者添加调试语句，能够更加清楚该程序的流程。但是每次修改完RadioCountToLedsC.nc文件都要重新执行make micaz sim这条命令才会使之生效。