**2 Aim: Exploring and understanding TinyOS computational concept: Events,Commands and task.**

–**nesC model**

–**nesC Components**

**TinyOS Lab Exercise in Ad Hoc and Sensor Networks**

* **Sensor network programming in a nutshell**

– Read ‘Getting started with TinyOS’ (at home)

– Solve two Lab-style exercises on real hardware

– Teams of two to three students are ideal

– One lab working place is available in ETL F29

– Reservation system on the course website

– Expected time needed for all tasks: 3-4 hours

* **Shockfish TinyNode**

– Slow CPU

– 8 MHz Texas Instruments MSP430 microcontroller

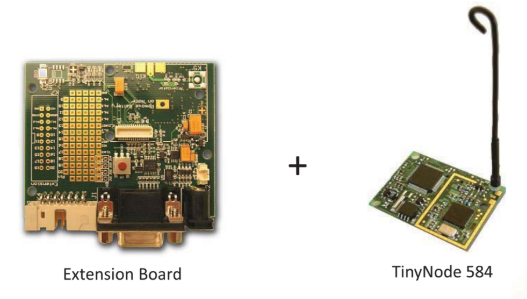
– Little memory

– 10 KByte RAM, 48 KByte ROM, 512 Kbyte external flash

– Short-range radio

– 868 MHz Xemics XE1205 ultra-low power wireless transceiver

– Light sensor, temperature and humidity sensors



* **Exchange of a sensor data**

– Two sensor nodes are used for this task

– One node periodically samples its light sensor and broadcasts the sensor reading over its radio

– The other node listens for radio messages and signals if it is getting brighter or darker

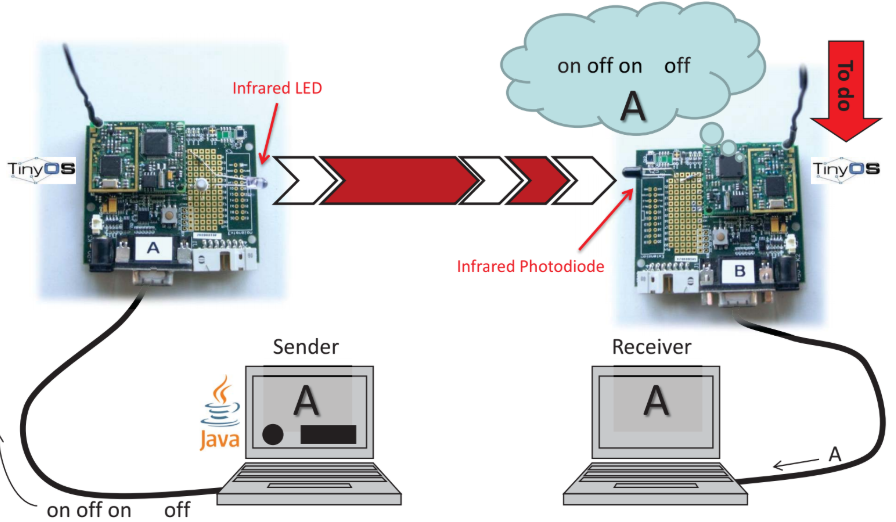
– Brighter → The green LED of the receiver is set

– Darker → The red LED of the receiver is set

– No significant change → The yellow LED is set



**Optical Communication using Morse Codes**



**TinyOS**

• TinyOS is an operating system for sensor nodes

– Open source project with a strong academic background

– Hardware drivers, libraries, tools, compiler

• TinyOS applications are written in nesC

– C dialect with extra features

– nesC compiler converts your application into plain C code



**Why using a new Operating System?**

• Measure real-world phenomena

–Event-driven architecture

• Resource Contraints

–Hurry up and sleep!

• Adapt to changing technologies

–Modularity & re-use

• Applications spread over many small nodes

–Communication is fundamental

• Inaccessible location, critical operation

–Robustness

**NesC/TinyOS Programming Model**

• Programs are built out of components

• Two types of components:

–Modules: Implement program logic

–Configurations: Wire components together

• Components use and provide interfaces

• Components are wired together by connecting interface users with interface providers

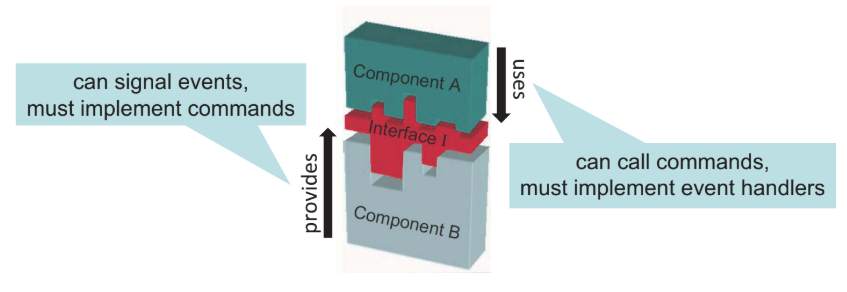
**Programming Model**

• Interfaces contain definitions of

–Commands

– Events

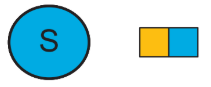
• Components implement the event handlers they use and the commands they provide



**Concurrency Model**

• Coarse-grained concurrency only

–Implemented via tasks



• Tasks are executed sequentially by the TinyOS scheduler

–no threads

–Atomic with respect to other tasks (single threaded)

–Longer background processing jobs

• Events (interrupts)

–Time critical

–Preempt tasks

–Short duration (hand off computation to tasks if necessary)

**Memory Model**

• Static memory allocation

–No heap (malloc)

–No function pointers

• Global variables

–One namespace per component

• Local variables

–Declared within a function

–Saved on the stack

• Conserve memory

• Use pointers, don‘t copy buffers

**nesC – Hello World**

module BlinkC {

uses interface Timer<TMilli>

as BlinkTimer;

uses interface Leds;

uses interface Boot;

}

implementation{

event void Boot.booted() {

call BlinkTimer.startPeriodic(1000);

}

event void BlinkTimer.fired() {

call Leds.led0Toggle();

}

}

• Blink the red LED every second

• On boot start a 1 second timer

• On timer fire (countdown at 0)

–Toggle the state of the red LED

–Reset the timer to 1 second

**nesC – Hello World**

interface Timer<precision\_tag> {

event void fired();

command void startPeriodic(...);

command void startOneShot(...);

command void stop();

…

}

configuration BlinkAppC{

}

implementation {

components MainC, BlinkC,

LedsC;

components new TimerMilliC()

as Timer0;

BlinkC.Boot -> MainC.Boot;

BlinkC.BlinkTimer -> Timer0;

BlinkC.Leds -> LedsC.Leds;

}

module BlinkC {

uses interface Timer<TMilli>

as BlinkTimer;

uses interface Leds;

uses interface Boot;

}

implementation {

event void Boot.booted() {

call BlinkTimer.startPeriodic(1000);

}

event void BlinkTimer.fired() {

call Leds.led0Toggle();

}

}

**3 Aim: Understanding TOSSIM for**

* **Mote-mote radio communication**
* **Moto-PC serial communication**

**Introduction**

TOSSIMisadiscreteeventsimulatorforTinyOSsensornetworks. InsteadofcompilingaTinyOSapplication for a mote, users can compile it into the TOSSIM framework, which runs on a PC. This allows users to debug, test, andanalyzealgorithmsinacontrolledandrepeatableenvironment. AsTOSSIMrunsonaPC,userscan examine their TinyOS code using debuggers and other development tools. This document brieﬂy describes the design philosophy of TOSSIM, its capabilities, its structure. It also provides a brief tutorial on how to use TOSSIM for testing or analysis. TOSSIM’s primary goal is to provide a high ﬁdelity simulation of TinyOS applications. For this reason, it focuses on simulating TinyOS and its execution, rather than simulating the real world. While TOSSIM can be used to understand the causes of behavior observed in the real world, it does not capture all of them, and should not be used for absolute evaluations. TOSSIM is not always the right simulation solution; like any simulation, it makes several assumptions, focusing on making some behaviors accurate while simplying others. One of the most common questions aboutTOSSIM is whetheritcan“simulate X”or whetherit“has anaccurate X model.” Inhope ofanswering most of such questions, here is a brief summmary of its characteristics:

• Fidelity: By default, TOSSIM captures TinyOS’ behavior at a very low level. It simulates the network at the bit level, simulates each individual ADC capture, and every interrupt in the system

• Time: While TOSSIM precisely times interrupts (allowing things like bit-level radio simulation), it does not model execution time. From TOSSIM’s perspective, a piece of code runs instantaneously.

• Models: TOSSIM itself does not model the real world. Instead, it provides abstractions of certain real-world phenomena (such as bit error). With tools outside the simulation itself, users can then manipulate these abstractions to implement whatever models they want to use.

**Compiling and Running a Simulation**

TOSSIM is automatically built when you compile an application. Applications are compiled by entering an application directory (e.g. /apps/Blink) and typing make. Alternatively, when in an application directory, you can type make pc, which will only compile a simulation of the application. There are several compilation options to ncc when compiling for TOSSIM, including the maximum number of motes that can be simulated. The default options in the TinyOS 1.1 makeﬁle should ﬁt almost any need; refer to the nesC manual for further information on the options available. The TOSSIM executable is named main.exe, and resides in build/pc. It has the following usage:

Usage: ./build/pc/main.exe [options] num\_nodes [options] are: -h, --help Display this message. -gui pauses simulation waiting for GUI to connect -a=<model> specifies ADC model (generic is default) options: generic random -b=<sec> motes boot over first <sec> seconds (default: 10) -ef=<file> use <file> for eeprom; otherwise anonymous file is used -l=<scale> run sim at <scale> times real time (fp constant) -r=<model> specifies a radio model (simple is default) options: simple static lossy -rf=<file> specifies file input for lossy model (lossy.nss is default) -s=<num> only boot <num> of nodes -t=<sec> run simulation for <sec> virtual seconds num\_nodes number of nodes to simulate

The -h or --help options prints out the above usage message, and some additional information. The -a option speciﬁes the ADC model to use. TOSSIM currently supports two models: generic and random. Section 4 describes these models. The -b option speciﬁes the interval over which motes boot. Their boot times are uniformly distributed over this interval. The default value is ten seconds. The -e option is for named EEPROM ﬁles. If -e isn’t speciﬁed, the logger component stores and reads data, but this data is not persistent across simulator invocations: it uses an anonymous ﬁle. Section 5 describes how the EEPROM in TOSSIM works. The -l option is for making TOSSIM run at a rate representative of real time. The scale argument speciﬁes what relative rate should be used. For example, -l=2.0 means twice as fast as real time (two virtual seconds run in one real second), while -l=0.1 means one tenth of real time (one virtual seconds runs in ten real seconds.). TOSSIM can only run so fast; specifying it to run faster than it can will cause it to run as quickly as possible. Using this option imposes a signiﬁcant performance overhead; it shouldn’t be used when trying to run simulations quickly. The -r option speciﬁes the radio model to use. TOSSIM currently supports two models: simple and lossy. Earlier versions also supported a “static” model, but this has been subsumed by the lossy model. Section 3 describes these models The -s option tells TOSSIM to only boot a subset of the number of nodes speciﬁed. This is useful if you want some to boot later, in response to user input. If the -s option is speciﬁed, TOSSIM boots mote IDs 0-(num - 1). The -t option tells TOSSIM to run for a speciﬁed number of virtual seconds. After sec seconds have passed, TOSSIM exits cleanly.

Using LossyBuilder LossyBuilder assumes each mote has a transmission radius of 50 feet. Combined with the bit error rate, this means each mote transmits its signal in a disc of radius 50 feet, with the bit error rate increasing with distance from the center. LossyBuilder can read in or generate physical topologies ((x,y) coordinates), and generate loss topologies from physical topologies by sampling from the model of the empirical distribution in Figure 1(a). Its usage is:

usage: java net.tinyos.sim.LossyBuilder [options] options: -t grid: Topology (grid only and default) -d <m> <n>: Grid size (m by n) (default: 10 x 10) -s <scale>: Spacing factor (default: 5.0) -o <file>: Output file -i <file>: Input file of positions -p: Generate positions, not error rates

Running LossyBuilder will, by default, generate a loss topology for a 10 by 10 grid with a grid spacing of ﬁve feet (45’ by 45’), and print it to standard out. It prints the loss topology in the form that TOSSIM expects. The -i option allows you to specify an input ﬁle of (x,y) positions, which has the format

x y x y x y ...

where the coordinate system is in feet. The ﬁrst (x,y) pair is the position of mote 0, the second is the position of mote 1, etc. For example,

java net.tinyos.sim.LossyBuilder -d 20 20 -s 10 -o 20x20-10.nss

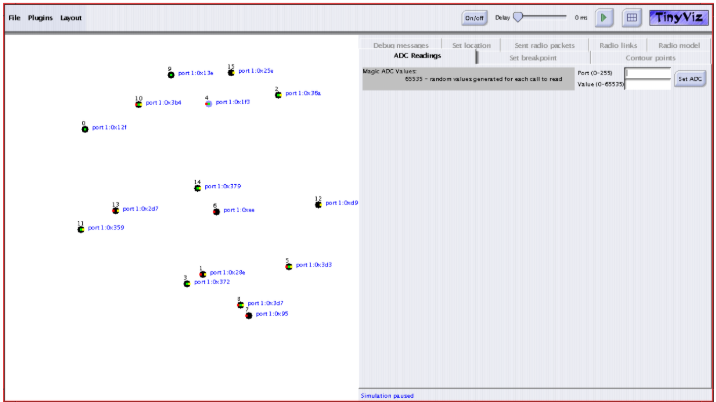
will output a loss topology of a 20 by 20 grid, with ten foot spacing, and write it to the ﬁle “20x20-10.nss”.

Bit Errors and Packet Errors The forumula for calculating packet error rates (Ep) from bit error rates (Eb) for the mica RFM 40Kb stack with SecDed encoding is:

Ep = 1−(Ss ·(Se)d) Ss = (1−Eb)9 Se = (1−Eb)8 +(8·Eb ·(1−Eb)12) Ep = 1−((1−Eb)9 ·((1−Eb)8 +(8·Eb ·(1−Eb)12))d) where Ss is the start symbol success probability, Se the probability a packet byte is uncorrupted (zero or one bit errors), and d is the number of bytes in the packet. The simple model can be represented in the lossy model: it is a fully connected graph, in which every edge has a loss probability of zero. However, doing so requires specifying the entire graph to TOSSIM; the simple model is an easy shortcut (and has an internal representation that’s more eﬃcient).

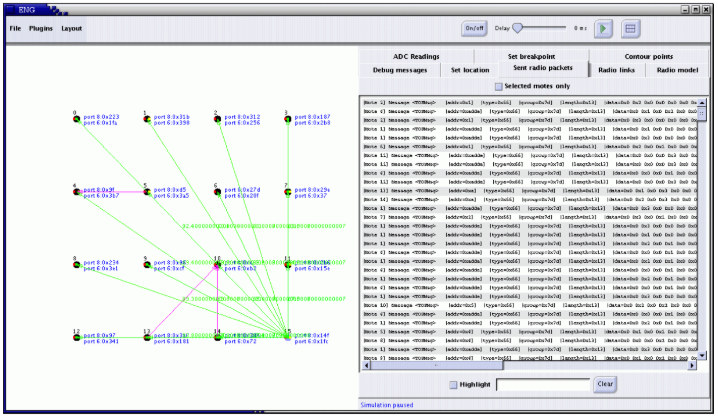
**EEPROM**

TOSSIM models the EEPROM at the line (16-byte block) level. TOSSIM models it with a large, memorymapped ﬁle. By default, this ﬁle is anonymous, and disappears when a simulation ends. However, with the



**TinyViz**

TinyViz is a Java visualization and actuation environment for TOSSIM. The main TinyViz class is a jar ﬁle, tools/java/net/tinyos/sim/tinyviz.jar. TinyViz can be attached to a running simulation. Also, TOSSIM can be made to wait for TinyViz to connect before it starts up, with the -gui ﬂag. This allows users to be sure that TinyViz captures all of the events in a given simulation. TinyViz is not actually a visualizer; instead, it is a framework in which plugins can provide desired functionality. By itself, TinyViz does little besides draw motes and their LEDs. However, it comes with a few example plugins, such as one that visualizes network traﬃc. Figure 3 shows a screenshot of the TinyViz tool. The left window contains the simulation visualization, showing 16 motes communicating in an ad-hoc network. The right window is the plugin window; each plugin is a tab pane, with conﬁguration controls and data. The second element on the top bar is the Plugin menu, for activating or de-activating individual plugins. Inactive plugins have their tab panes greyed out.



**Using gdb**

The binary executable produced by typing make pc in any app directory can be debugged using GDB. Developers of TinyOS code can step through programs they have written, debugging deterministic, logical aspects of their code before loading programs onto motes. When accessing variables and functions in TOSSIM under GDB, speciﬁc preﬁxes precede them, to distinguish variables from diﬀerent components that have ths same name. The nesC compiler takes each

component’s ﬁelds and functions and renames them with unique identiﬁers. This renaming for ﬁelds of a component is done by taking the ﬁeld name, and preceding it by the component name and a $ sign. Functions are renamed by taking the name of the function and preceding it by the name of the component it is deﬁned in, followed by a $ sign, followed by the interface name, followed by another $ sign.

**TOSSIM Architecture**

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

while(1){ TOSH\_run\_task(); }

TOSH run task runs tasks until the task queue is empty, at which point it puts the CPU to sleep. An interrupt will wake the mote. If the interrupt has caused a task to be scheduled, then that task will be run. While that task is running, interrupts can be handled. The core TOSSIM loop is slightly diﬀerent:

for (;;) {

while(TOSH\_run\_next\_task())

{}

if (!queue\_is\_empty(&(tos\_state.queue)))

{

tos\_state.tos\_time =

queue\_peek\_event\_time(&(tos\_state.queue));

queue\_handle\_next\_event(&(tos\_state.queue));

}

Anotionofvirtualtime(storedasa64-bitinteger)iskeptinthesimulator(storedintos state.tos time), and every event is associated with a speciﬁc mote. Most events are emulations of hardware interrupts. For example, when the clock in hpl.c is initialized to certain counter values, TOSSIM’s implementation enqueues a clock interrupt event, whose handler calls the function normally registered as the clock interrupt handler; from this point, normal TinyOS code takes over (in terms of events being signaled, etc.) In addition, the event enqueues another clock event for the future, with its time (for the event queue) being the current time plus the time between interrupts.

TOSSIM Implementations To load a component, the nesC compiler uses a directory search path. The default search path is:

• The application directory

• tos/platform/xxx (the selected platform)

• tos/sensorboards/xxx (the selected sensor board(s))

• tos/system

**RFM**

A TOSSIM network model has the following structure:

typedef struct {

void(\*init)();

void(\*transmit)(int, char);

// int moteID, char bit void(\*stop\_transmit)(int);

// int moteID char(\*hears)(int);

// char bit, int moteID bool(\*connected)(int,int);

// int moteID1, int moteID2 link\_t\*(\*neighbors)(int);

// int moteID } rfm\_model;

init, transmit, stop transmit and hears are straightforward; they’re used to transmit and receive bits. The connected and neighbors functions are used for some speciﬁc low-level emulations of the radio channel during bit synchronization.

• tos/lib Entries can be prepended to the search path with the -I option to the compiler. Using the -I option therefore allows you to use alternative implementations of components provided by TinyOS. For example, one can write a TOSSIM implementation of the microphone, and put it in a separate directory, then include that directory with -I. This will cause the compiler to load the new TOSSIM implementation instead of the default one in the sensor board directory.