CS 450
Network Embedded Systems
Fall 2008
TinyOS Tutorial

Chieh-Jan (Mike) Liang
Computer Science Department
Johns Hopkins University
Part I – TinyOS Basics
Sensor Network Characteristics

- **Event-driven**
  - e.g. message arrival, sensor acquisition

- **Concurrency-intensive operation**
  - Multiple flows, not wait-command-respond

- **Unattended Operation**
  - Reduce run-time errors

- **Limited resources**
  - Restricted by mote size, cost, and power consumption

- **Diversity in Design and Usage**
  - Application specific, not general purpose
TinyOS

- An operating system
- A collection of drivers for hardware components
- A programming environment
  - Language (nesC)
  - Set of services
- Design philosophy: Sleep as much as possible, and only wake up to react to events
  - Event-driven
TinyOS Concepts

• Component-based
  – Separate construction and composition
  – Applications are sets of components
  – Benefits?

• Two types of components:
  – Module implements the behavior
  – Configuration specifies wiring

• Configuration wires interface user module to interface provider module

• User can call provider's command, and provide event handler to provider
  – Two-way communication
  – As specified by the interface
Example Application

- Forwarding of photo / temp sensor readings over ad-hoc, multi-hop routing tree
Split-Phase Abstraction

- All long operations are split-phase: operation request and completion event are separated
- Higher level (SW) processes don’t wait or poll
  - Allows automated power management
  - Allows the program to continue while waiting for operation to complete
- Events are generally associated with hardware interrupts
- Examples:
  - Radio: `command send(...)`, `event sendDone(...)`
  - Sensor reading: `command read(...)`, `event readDone(...)`
Event-Driven Example

```c
event void Boot.booted() {
    call Timer.startPeriodic(200);
}
event void Timer.fired() {
    call sensor.read();
}
event void sensor.readDone(error_t error, uint16_t data) {
    if (error == SUCCESS)
        writeToFlash(data)
}
...
```

- Timer event handler initiates sensing
- Sensor event handle signals sensing data ready
  - An example of split-phase abstraction
- Device sleeps or handles other activity while waiting
- Note: SUCCESS is a TinyOS keyword
configuration ExampleAppC {} 
Implementation {
    component ExampleP, LedsC; 
    ExampleP.Leds -> LedsC.Leds; 
    ...
}

module ExampleP {
    uses interface Leds;
    uses interface Boot;
} 
implementation {
    event void Boot.booted() {
        call Leds.led0On();
    }
    ...
}

module LedsP {
    provides {
        interface Leds;
        ...
    }
    uses {
        interface GeneralIO as Led0;
        ...
    }
} 
implementation {
    async command Leds.led0On() {...}
}

configuration LedsC {
    provides interface Leds;
}
implementation {
    components LedsP, PlatformLedsC;
    ...
    LedsP.Led0 -> PlatformLedsC.Led0;
}

module PlatformLedsC {
    provides interface GeneralIO as Led0;
    ...
}

...
Component Hierarchy

- **Command**
  - Flows downward
  - Control return to caller

- **Event**
  - Flows upward
  - Control return to signaler

- **Event can call command, but not vice versa**
  - Why?
  - Post *task* instead (more on next slide)
Part II – More TinyOS
Tasks

• Another TinyOS execution thread model beside events
  – Provide concurrency internal to a component
• Tasks are scheduled to run in the future
  – Different from events
  – Suitable for non time-critical functions
• May call commands and signal events
• Not preempted by another task
  – Only one task runs at any time
  – A task can post itself
• Can be preempted by an async event
task void readSensor() {
    if (sensor.getData() != SUCCESS)
        post readSensor();    // Retries until success
}

event void Boot.booted() {
    call Timer.startOneShot(200);    // Samples after 200 ms
}

event void Timer.fired() {
    post readSensor();    // Posts a task
}

event void sensor.readDone(error_t error, uint16_t data) {
    if (error == SUCCESS) {
        writeToFlash(data)
        call Timer.startOneShot(200);    // Samples again after 200 ms
    } else
        post readSensor();    // Retries until success
}
...

- Extending previous event-driven example to handle failed events
Concurrency

• Synchronous code (SC):
  – Only reachable from scheduled tasks
• Asynchronous code (AC):
  – Reachable from at least one interrupt handler
• SC is atomic with respect to each other
  – Ex. Only one task runs at any time, and tasks run to completion
• Any update to variables shared with AC is potentially a race condition
  – What can we do?
Race-Free Invariant

- Posting a task allows you to go from AC to SC
- Atomic section is a code block that will run without interruptions from interrupts.
- “Any update to shared state is either not in a potential race condition (i.e. accessed through SC only), or should be within an atomic section”
Active Messages

- Interface to communication
  - Multiplexing radio service
- Each message contains an ID that specifies the event handler at the receiver side
  - Similar to the concept of port number in IP networks
Active Messages (cont.)

Application

send[AM=20](…)

AM

Radio Stack

Handler (20)

receive[AM=20](…)

AM

Radio Stack

Handler (21)

receive[AM=21](…)

send[AM=20](…) receive[AM=20](…) receive[AM=21](…)
int main() __attribute__((C, spontaneous)) {
    atomic {
        platform_bootstrap();
        call Scheduler.init();
        call PlatformInit.init();
        while (call Scheduler.runNextTask());
        call SoftwareInit.init();
        while (call Scheduler.runNextTask());
    }
    _nesc_enable_interrupt();
    signal Boot.booted();
    call Scheduler.taskLoop();
    return -1;
}
Part III – Programming
nesC Design

• An extension of C
  – Taking advantage of C’s efficiency and low-level features necessary for accessing hardware.

• Provides syntax for various abstractions in TinyOS

• Static whole system analysis
  – Possible because of the small code size and single-C-output file design
  – Eliminate unreachable code with application call-graph

• Static allocation only
  – Developers should know what they need and allocate at compile-time
nesC Compilation Stages

- SenseP.nc
- SenseAppC.nc
- DriverP.nc

nesC Compiler → app.c → C Cross Compiler → Object Code
Programming the Mote

- Makefile specifies the top-level configuration file

```
mike@sprite:~/local/src/tinyos-2.x/apps/Null$ ls
   build  Makefile  NullAppC.nc  NullC.nc  README.txt

mike@sprite:~/local/src/tinyos-2.x/apps/Null$ cat Makefile
COMPONENT=NullAppC
include $(MAKERULES)
```
Programming the Mote

- `make telosb (re)install [,id] [bsl ,device_port]`
  - `id`: node ID
  - `device_port`: Select the device to program. e.g. `/dev/ttyUSB0`
    - `install` compiles and pushes the object code to mote
    - `reinstall` only pushes the object code to mote

- Use `motelist` command to find the device port of connected telosb motes
TOSSIM

• Why simulation?
  – Controlled, reproducible testing environment
  – Cost-effective alternative
  – Means to explore and improve design space

• A discrete-event simulator
  – It pulls events of the event queue (sorted by time) and executes them

• Replace components with simulation-implementation
  – Ex. Radio stack

• Replace hardware with simulation-components
  – Ex. Hardware timer
  – Support for micaZ
TOSSIM (cont.)

• Compile directly from the TinyOS sources
  – Simulate and deploy the same code

• Supports two programming interfaces:
  – Python: Easy to write, and allows variable inspection
  – C++: Faster simulation, especially in large networks

• Incorporates environment noise model that captures complex temporal dynamics

• Compile code to TOSSIM: `make micaz sim`
from TOSSIM import *

t = Tossim([])

r = t.radio()
n0 = t.getNode(0)
n1 = t.getNode(1)

n0.bootAtTime(long(0.5 * t.ticksPerSecond()))
n1.bootAtTime(long(0.5 * t.ticksPerSecond()))

for j in range(100):
    node0.addNoiseTraceReading(-100)
    node1.addNoiseTraceReading(-100)

node0.createNoiseModel()
node1.createNoiseModel()

r.add(n0, n1, -20)
r.add(n1, n0, -20)

while (t.time() < (100 * t.ticksPerSecond())):  
    t.runNextEvent()
TinyOS 2 Directory Structure

- **tinyos-2.x/apps/**
  - Example applications (*Useful for learning TinyOS!*)

- **tinyos-2.x/support/**
  - **make/**: TinyOS make system

- **tinyos-2.x/tos/**
  - **chips/**: Drivers for microprocessors, radio chips, and flash chips
  - **interfaces/**: TinyOS core interface, e.g. Leds
  - **lib/net/**: Network abstractions, e.g. dissemination and collection
  - **platforms/**: Platform-specific modules, e.g. telosb, micaz, ...
Module RadioSendP {
    uses interface AMSend;
    ...
}
implementation {
    typedef nx_struct data {
        nx_uint32_t counter;
    } data_t;
    message_t msg;

    void sendMsg(uint32_t val) {
        data_t *msg_payload = (data_t *) call AMSend.getPayload(&msg, sizeof(data_t));
        msg_payload->counter = val;

        if (call AMSend.send(AM_BROADCAST_ADDR, &msg, sizeof(data_t)) != SUCCESS) {
            // Send failed, so sendDone event will not be signaled
        }
    }

    event void AMSend.sendDone(message_t* buf, error_t error) {
        if (buf == &msg) {
            if (error == SUCCESS) { /* Send succeeded, and the packet is in the air */ }
            else { /* Send failed */ }
        }
    }

    ...
}
Configuration RadioSendAppC {}
typedef nx_struct data {
    nx_uint32_t counter;
} data_t;

event message_t* Receive(message_t* buf, void* payload, uint8_t len) {
    data_t *msg_payload = (data_t *)payload;
    // display(msg_payload->counter);
    
    return buf
}
RadioRecvAppC.nc

Configuration RadioRecvAppC {}

implementation {
    components RadioRecvP,
        new AMReceiverC(0x00);

    RadioRecvP.Receive -> AMReceiverC.Receive;
    ...
}

Active Message ID