Principles of Wireless Sensor Networks

https://www.kth.se/social/course/EL2745/

Lecture 2

Introduction to Programming WSNs

Carlo Fischione
Associate Professor of Sensor Networks
e-mail:carlofi@kth.se
http://www.ee.kth.se/∼carlofi/

KTH Royal Institute of Technology
Stockholm, Sweden

September 8, 2014

Course content

- Part 1
  - Lec 1: Introduction to WSNs
  - Lec 2: Introduction to Programming WSNs
- Part 2
  - Lec 3: Wireless Channel
  - Lec 4: Physical Layer
  - Lec 5: Medium Access Control Layer
  - Lec 6: Routing
- Part 3
  - Lec 7: Distributed Detection
  - Lec 8: Static Distributed Estimation
  - Lec 9: Dynamic Distributed Estimation
  - Lec 10: Positioning and Localization
  - Lec 11: Time Synchronization
- Part 4
  - Lec 12: Wireless Sensor Network Control Systems 1
  - Lec 13: Wireless Sensor Network Control Systems 2
  - Lec 14: Summary and Project Presentations

WSN Programming

Introduction

Programming WSNs

- What do we need to write software for WSNs?
  (or: for any system, like your laptop, cell phone?)
  - Programming language
    - With compiler, etc.
  - OS/runtime libraries
    - Access to system resources
    - APIs: Communication, sensors, etc.
- Put Windows 7 on a sensor node?
  - Or Linux, Android, ...
  - Use Java, Python, ... as programming languages?
  - WSNs are embedded systems
    - Resource constraints

CPU	16bit, 8 MHz
RAM	10kB
ROM	48kB
Flash	1MB
Batteries	2xAA

Today's Topics

- NesC
  - Programming Language for WSNs
- TinyOS
  - WSN OS/runtime
  - Note: many other OS exist (Contiki, Mantis, LiteOS, Nano-RK, ...)
    - TinyOS: widespread, especially in academia
- Examples
- Hands-on
  - Your turn!

TinyOS and NesC

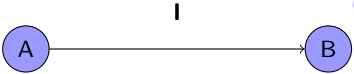
Component Model

TinyOS Components

- TinyOS and its applications are written in NesC
  - C dialect with extra features: mainly Components
- Basic unit of NesC code is a Component
- **Components** connect via **Interfaces**
  - Connections called “wiring”
  - Interfaces:
    - Access to the functions of component
    - Provide functionality to the outside
    - Similar to Interfaces in Java etc.

TinyOS Components

B provides functionality
Example:
-Turn on LED
-Send packet



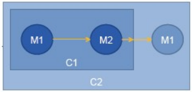
A uses functionality provided by **B**
Example: To send a packet

Interface **I** describes the "API"

TinyOS Components

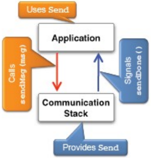
Two types of components

- Modules (M):
  - Base type
  - Contains functionality / implementations: "The real code"
- Configurations (C):
  - Contain multiple sub-components Modules or Configurations: Nesting
  - Describe connections Which module uses the function
  - Parameterize sub-components: Options, etc.
- Very similar to classes in Java etc.
  - But static
  - No instantiation at runtime (new "new()")



TinyOS Components

- **Components**
  - *use* or *provide* interfaces
  - *Use* : component uses func. of others
  - *Provide* : provide func. to others
- **Interfaces** contain
  - The API
  - Contain *commands* and *events*



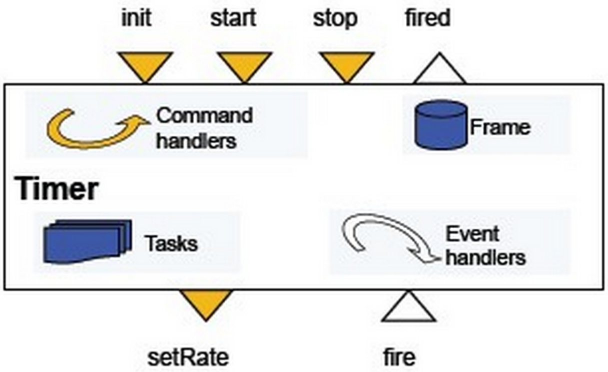
Function calls	**Commands**	**Events**
Use	Call Command	Implement Event Handler
Provide	Implement Command Body	Signal Event

TinyOS Modules

- Specification
  - List of interfaces the component
    - Provides:
      Functionality it provides to others
    - Uses:
      Functionality of others it uses
  - Alias interfaces as new name
- Implementation
  - Commands of provided interfaces
  - Events of used interfaces

```
module FooM {
  //Specification
  provides {
    interface Foo;
  }
  uses {
    interface Poo as PooFoo;
    interface Boo;
  }
}
//Implementation
implementation {
  //Command handlers
  command result_t Foo.comm{
    ...
  }
  //Event handlers
  event void Boo.event{
    ...
  }
}
```

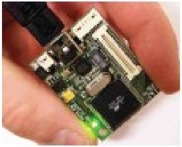
TinyOS Modules



TinyOS and NesC

Example: Hello World

Example: “Hello World”



- No screen, we cannot print “Hello World” ...
- ...So, let them periodically blink the LEDs!

- Ingredients:
  - A timer to trigger the periodic blinking
    - TOS provides a Timer library
  - A way to turn the LEDs on and off
    - TOS provides a LedsC component to control the LEDs

"Hello World"

Called to control a led

```
interface Leds {
    command void led0On();
    command void led0Off();
    command void led0Toggle();
    // ...
}
```

Can be TMilli, TMicro, T32Khz

```
interface Timer <precision> {
    command void startPeriodic(uint32_t dt);
    command void startOneShot(uint32_t dt);
    event void fired();
    // ...
}
```

Called to start a timer

Signaled when the timer expires

"Hello World"

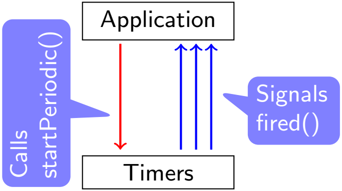
Timer with millisecond precision

```
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();
    }
}
```

Signaled when hw ready



"Hello World"

No interfaces used or provided

```
configuration BlinkAppC {}

implementation {
    components MainC,
               BlinkC,
               LedsC;
    components new TimerMilliC as Timer0;

    BlinkC -> MainC.Boot;
    BlinkC.BlinkTimer -> Timer0;
    BlinkC.Leds -> LedsC;
}
```

Provides Boot

TimerMilliC is a generic component: can be instantiated multiple times!

Wirings

- Instantiation is done at compile-time
  - very different from OO programming

"Hello World"

Cross-compile for a specific WSN platform

```
$ make telosb
    // ...
    compiled BlinkAppC to build/telosb/main.exe
                2648 bytes in ROM
                  54 bytes in RAM
    msp430-objcopy –output-target=ihex build/telosb/
    main.exe build/telosb/main.ihex
        writing TOS image
$ make telosb install, 0 bsl,/dev/ttyUSB0
    // ...
```

Node id

USB port the node is attached to

- **Use "motelist" command to determine USB port**

Programming (cont.)

TinyOS and NesC

Asynchronous Programming Model

```
event void Boot.booted() {
    call BlinkTimer.startPeriodic(1000);
}
event void BlinkTimer.fired() {
    call Leds.led0Toggle();
}
```

```
void main() {
    while() {
        sleep(1);
        printf('hello');
    }
}
```

- TinyOS:
  - Asynchronous programing (split phase)
  - No blocking calls
    - Not synchronous
  - Result:
    - Memory efficient concurrency
      If execution is neither in booted nor in fired, it will “return” to the OS
      No context switch required
    - Users must avoid endless loops etc.

Tasks

- TinyOS has a single stack:
  - Lon-running computation can reduce responsiveness
- Tasks: mechanism to defer computation
  - Tells TinyOS "do this later"
- Tasks run until completion
  - Tasks do not interrupt each other
  - TinyOS scheduler runs them one by one in the order of their post
  - Keep them short!
- Interrupts run on stack, can post tasks
  - Interrupts can interrupt tasks
    - Interrupts: low level hardware events
  - Interrupts can interrupt each other
    - See interrupt priority

```
// ...
task void compute () {
    // do something;
}

event void Timer.fired ()
{
    post compute()
}
// ...
```

TOSSIM

- TinyOS mote SIMulator
  - Allows NesC code to run on a standard machine
    - Linux etc.
  - Includes models for
    - Communication
    - Sensing
    - ...
  - Compiles directly from TinyOS source
    - Run the same code in simulator and sensor node
    - Good for testing before deployment
- See TinyOS website for details

Code examples

- TinyOS has an “apps” folder
  - Many good examples
- Other sources of information
  - TinyOS Book
    - TinyOS Programming by Philip Levis, David Gay
    - In part: csl.stanford.edu/∼ pal/pubs/tinyos-programming.pdf
  - TinyOS web book
    - csl.stanford.edu/∼ pal/pubs/tos-programming-web.pdf
  - TinyOS tutorial
    - IPSN 2009 (enl.usc.edu/talks/cache/tinyos-ipsn2009.ppt)
  - TinyOS website: http://www.tinyos.net/
  - TinyOS mailing list (and its archives)
    - On the TinyOS website

Programming (cont.)

Hands-On

Your Tasks

1. System test
    - Install Blink on your sensor node
        - See apps/Blink
        - Compile, install (see prev. slides, see tutorial from TA)
2. Main task:
    - Program Anti-Theft App for your sensor node

Your Main Task

- Goal: write an anti-theft device
  - Detect when someone steels your sensor node
- Two parts:
  - Detecting theft
    - Assume: thieves put the motes in their pockets
    - So, a “dark” mote is a stolen mote
    - Every N ms check if light sensor is below some threshold
  - Reporting theft
    - Assume: bright flashing lights deter thieves
    - Theft reporting algorithm: light the red LED for a little while!
- What you will use
  - Basic components, interfaces, wiring
  - Essential system interfaces for startup, timing, sensor sampling
- Start from Blink
  - In apps/Blink
  - See apps/RadioSensoToLeds on sensor use
    - Replace “DemoSensorC” with “HamamatsuS10871ParC” or “HamamatsuS10871TsrC”

The Basics - Let's Get Started

```
module AntiTheftC {
    uses interface Boot;
    uses interface Timer<TMilli> as Check;
    uses interface Read<uint16_t>;
}
implementation {
    event void Boot.booted() {
        call Check.startPeriodic(1000);
    }
    event void Check.fired() {
        call Read.read();
    }
    event void Read.readDone(error_t
        if (ok == SUCCESS && val <
            theftLed();
    }
}
```

```
interface Boot {
    /* Signaled when OS booted */
    event void booted();
}
```

```
interface Timer< tag> {
    command void startOneShot(uint32_t period
    command void startPeriodic(uint32_t period)
    event void fired();
}
```

Components start with a *signature* specifying
- the interfaces *provided* by the component
- the interfaces *used* by the component

A module is a component implemented in C
- with functions implementing commmands and events
- and extensions to call commands, events

The Basics - Split-Phase Ops

```
module AntiTheftC {
    uses interface Boot;
    uses interface Timer<TMilli> as Check;
    uses interface Read<uint16_t>;
}
implementation {
    event void Boot.booted() {
        call Check.startPeriodic(1000);
    }
    event void Check.fired() {
        call Read.read();
    }
    event void Read.readDone(error_t ok, uint16_t val) {
        if (ok == SUCCESS && val < 200)
            theftLed();
    }
}
```

In TinyOS, all long-running operations are split-phase:
- A command starts the op: read
- An event signals op completion: readDone

```
interface Read< val_t > {
    command error_t read();
    event void readDone(error_t ok, val_t val);
}
```

The Basics - Split-Phase Ops

```
module AntiTheftC {
    uses interface Boot;
    uses interface Timer<TMilli> as Check;
    uses interface Read<uint16_t>;
}
implementation {
    event void Boot.booted() {
        call Check.startPeriodic(1000);
    }
    event void Check.fired() {
        call Read.read();
    }
    event void Read.readDone(error_t ok, uint16_t val) {
        if (ok == SUCCESS && val < 200)
            theftLed();
    }
}
```

In TinyOS, all long-running operations are split-phase:
- A command starts the op: read
- An event signals op completion: readDone

```
interface Read< val_t > {
    command error_t read();
    event void readDone(error_t ok, val_t val);
}
```

The Basics - Configurations

```
configuration AntiTheftAppC
implementation
{
    components AntiTheftC, MainC, LedsC;

    AntiTheftC.Boot -> MainC.Boot;
    AntiTheftC.Leds -> LedsC;

    components new TimerMilliC() as MyTimer;
    AntiTheftC.Check -> MyTimer;

    components new HamamatsuS10871ParC();
    AntiTheftC.Read -> HamamatsuS10871ParC;
}
```

```
generic configuration TimerMilliC() {
    provides interface Timer<TMilli>;
}
implementation { ...}
```

```
generic configuration PhotoC() {
    provides interface Read;
}
implementation { ...}
```

A configuration is a component built out of other components.
It *wires* "used" to "provided" interfaces.
It can instantiate *generic* components
It can itself provide and use interfaces

Thanks!

Questions?

- These slides are based on slides from
  - Olaf Landsiedel, Phil Levis, David Gay, David Culler, Luca Mottola, Hamad Alizai, and many others