Programming Platform for Sensor Networks: TinyOS

Jun Yang
CPS 296.1, Spring 2007
Sensor Data Processing
With contents from D.-O. Kim

Announcements (Feb. 27)
- Course project milestone 1 this Thursday
  - I need to know your team members and project ideas
  - 10% of total grade
- Reading for next Tuesday: MauveDB (review due)
- Next Thursday (Mar. 8): project proposal talk
  - 15 minutes per group; 20% of total grade
  - What is it? Why do we care? Hasn’t it been done before? Plans, thoughts, and preliminary results?

Challenges in programming sensors
- WSN usually has severe power, memory, and bandwidth limitations
- WSN must respond to multiple, concurrent stimuli
  - At the speed of changes in monitored phenomena
- WSN are large-scale distributed systems

Traditional embedded systems
- Event-driven execution and real-time scheduling
- General-purpose layers are often bloated → microkernel
- Strict layering often adds overhead → expose hardware controls

Node-level methodology and platform
- Traditional design methodologies are node-centric
- Node-level platforms
  - Operating system
    - Abstracts the hardware on a sensor node
    - Provides services for apps such as, traditionally, file management, memory allocation, task scheduling, device drivers, networking…
  - Language platform
    - Provides a library of components to programmers

TinyOS
- Started out as a research project at Berkeley
- Now probably the de facto platform
- Overarching goal: conserving resources
- No file system
- No dynamic memory allocation
- No memory protection
- Very simple task model
- Minimal device and networking abstractions
- Application and OS are coupled—composed into one image
  - Both are written in a special language nesC
TinyOS components

- Components: reusable building blocks
- Each component is specified by a set of interfaces
  - Provide "hooks" for wiring components together
- A component $C$ can provide an interface $I$
  - $C$ must implement all commands available through $I$
  - Commands are methods exposed to an upper layer
    - An upper layer can call a command
- A component $C$ can use an interface $J$
  - $C$ must implement all events that can be signaled by $J$
  - These are methods available to a lower layer
    - By signaling an event, the lower layer calls the appropriate handler
- Components are then wired together into an application

Component specification

```
module TimerModule {
    provides {
        interface StdControl;
        interface Timer01;
    }
    uses interface Clock as Clk;
}
```

```
interface StdControl {
    command result_t init();
    command result_t start();
    command result_t stop();
}
```

```
interface Timer01 {
    command result_t start(char type, uint32_t interval);
    command result_t stop();
    event result_t timer0Fire();
    event result_t timer1Fire();
}
```

```
interface Clock {
    command result_t setRate(char interval, char scale);
    event result_t fire();
}
```

Module vs. configurations

- Two types of components
  - Module: implements the component specification (interfaces) with application code
  - Configuration: implements the component specification by wiring existing components

Module implementation

```
module TimerModule {
    provides { interface StdControl; interface Timer01; } uses interface Clock as Clk;
    implementation {
        bool eventFlag;
        command result_t StdControl.init() {
            eventFlag = 0;
            return call Clk.setRate(128, 4); // 4 ticks per sec
        }
        event result_t Clk.fire() {
            eventFlag = !eventFlag;
            if (eventFlag) signal Timer01.timer0Fire();
            else signal Timer01.timer1Fire();
            return SUCCESS;
        }
        ...
    }
}
```

Configuration implementation

```
configuration TimerConfiguration {
    provides {
        interface StdControl;
        interface Timer01;
    }
    implementation {
        components TimerModule, HWClock;
        StdControl = TimerModule.StdControl;
        Timer01 = TimerModule.Timer01;
        TimerModule.Clock => HWClock.Clock;
    }
```

Commands vs. events

```
command cmdName(args) {
    ...
    status = call UsedInterfaceName.cmdName(args);
    ...
    return status;
}
```

```
event UsedInterfaceName.eventName(args) {
    ...
    status = signal ProvidedInterfaceName.eventName(args);
    ...
```
FieldMonitor example

Concurrency model

Two types of execution contexts

- Tasks
  - Longer running jobs
  - Time flexible
  - (Currently) simple FIFO scheduling
  - Atomic w.r.t. other tasks, i.e., single-threaded
  - But can be preempted by events

- Events (an overloaded term)
  - More precisely, hardware interrupt handlers
  - Time critical
  - Shorten duration as much as possible
    - By issuing tasks for later execution
  - LIFO semantics; can preempt tasks and earlier events

Tasks

- A task is always posted for later execution; control returns to poster immediately
- Scheduler supports a bounded queue of pending tasks
  - Node sleeps when the queue is empty
  - For simplicity, tasks don’t take args and don’t return values
- Typical use
  - Event necessitates further processing
    - Wrap it up in a task
  - Event handler simply posts the task and can return immediately

Execution example

A more complex example

- Timer01.Timer0Fire() triggers data acquisition (through ADC) and transmission to base station (through Send)
Split-phase operation

- Data acquisition doesn’t take place immediately (why?)
- How does a traditional OS accomplish this?

Posting task in interrupt handler

```c
bool localBusy;
atomic { localBusy = busy; busy = TRUE; }
if (!localBusy) { call ADC.getData(); return SUCCESS; }
else { return FAILED; }
```

```c
bool busy;
norace unit16_t sensorReading;
```

Ensures at most one outstanding ADC request at all times

Discussion

- Provides framework for concurrency and modularity
- Interleaves flows, events, energy management
- Never poll, never block
- Trade off flexibility for more optimization opportunities

Race conditions

- Because of preemption, race conditions may arise on shared data
  - nesC compiler can detect them, but with false positives
- In case of false positive, declare shared variable with norace keyword
- In case of real race conditions, use atomic to make code blocks non-preemptible