TinyOS and NesC
Traditional Systems

- Well established layers of abstractions
- Strict boundaries
- Ample resources
- Independent apps at endpoints communicate pt-pt through routers
- Well attended

User

System

Network Stack

Threads

Application

Address Space

Transport

Files

Network

Data Link

Drivers

Physical Layer

Slides from Culler, Levis, TinyOS website
by comparison ...

- Highly Constrained resources
  - processing, storage, bandwidth, power
- Applications spread over many small nodes
  - self-organizing Collectives
  - highly integrated with changing environment and network
  - communication is fundamental
- Concurrency intensive in bursts
  - streams of sensor data and network traffic
- Robust
  - inaccessible, critical operation

=> Provide a framework for:
  - Resource-constrained concurrency
  - Defining boundaries
  - Appl’n-specific processing allow abstractions to emerge
Characteristics of Network Sensors

- Small physical size and low power consumption
- Concurrency-intensive operation
  - multiple flows, not wait-command-respond
    => never poll, never block
- Limited Physical Parallelism and Controller Hierarchy
  - primitive direct-to-device interface
  - Asynchronous and synchronous devices
    => interleaving flows, events, energy management
- Diversity in Design and Usage
  - application specific, not general purpose
  - huge device variation
    => efficient modularity
    => migration across HW/SW boundary
- Robust Operation
  - numerous, unattended, critical
    => narrow interfaces
Tiny OS Concepts

- Scheduler + Graph of Components
  - constrained two-level scheduling model: threads + events

- Component:
  - Commands,
  - Event Handlers
  - Frame (storage)
  - Tasks (concurrency)

- Constrained Storage Model
  - frame per component, shared stack, no heap

- Very lean multithreading

- Efficient Layering
Application = Graph of Components

- Route map
- router
- sensor appln
  - Active Messages
  - Radio Packet
  - Serial Packet
  - Temp
  - photo
  - UART
  - ADC
  - clocks

- RFM
- bit
- byte
- packet
- application

SW

HW
TOS Execution Model

- commands request action
  - ack/nack at every boundary
  - call cmd or post task
- events notify occurrence
  - HW intrpt at lowest level
  - may signal events
  - call cmds
  - post tasks
- Tasks provide logical concurrency
  - preempted by events
- Migration of HW/SW boundary
TinyOS

- application = scheduler + graph of components
- event-driven architecture
- single shared stack
- NO kernel, process/memory management, virtual memory
Components

- A component has:
  - Frame (internal state)
  - Tasks (computation)
  - Interface (events, commands)

- Frame:
  - one per component
  - statically allocated
  - fixed size

- Commands and Events are function calls
- Application: linking/glueing interfaces (events, commands)
Commands/Events

- **commands:**
  - deposit request parameters into the frame
  - are non-blocking
  - need to return status => postpone time consuming work by posting a task
  - can call lower level commands

- **events:**
  - can call commands, signal events, post tasks, can not be signaled by commands
  - preempt tasks, not vice-versa
  - interrupt trigger the lowest level events
  - deposit the information into the frame
Scheduler

- two level scheduling: events and tasks
- scheduler is simple FIFO
- a task can not preempt another task
- events preempt tasks (higher priority)

```c
main {
    ...
    while(1) {
        while(more_tasks)
            schedule_task;
        sleep;
    }
}
```
Tasks

- FIFO scheduling
- non-preemptable by other task, preemtable by events
- perform computationally intensive work
- handling of multiple data flows:
  - a sequence of non-blocking command/event through the component graph
  - post task for computational intensive work
  - preempt the running task, to handle new data
Programming Environment

- OS: cygwin/Win2000 or gcc/Linux
- Software: atmel tools, java, perl

![Image of programming board, mote, and mote-PC comms with Code download label]
Programming Environment

- download, install and build:
  - cygwin (http://www.cygwin.com)
  - WinAVR (http://winavr.sourceforge.net)
  - nesC (http://nesc.sourceforge.net)
  - Java JDK (http://java.sun.com/j2se/1.4.1)
  - tinyOS distribution (http://sourceforge.net/projects/tinyos)

- build your application
  - code your components
  - $ make mica2 install.1

- debug your application with TOSSIM simulator:
  - $ make pc
  - $ build/pc/main.exe 25
nesC: A programming language for sensor networks

- Supports concurrency model of TinyOS
  - Small (396 bytes minimum) OS
  - Non-blocking operations - *split phase*
  - Very low overhead, no threads
- Dialect of C with support for components
  - Components *provide* and *require* interfaces
  - Create applications by wiring together components using configurations
- Whole program compilation and analysis
  - Aggressive cross-component inlining
  - Static data-race detection
- Optimization approaches include
  - No function pointers
  - No dynamic memory allocation
  - No dynamic component instantiation/destruction
nesC

- the nesC model:
  - interfaces:
    - uses
    - provides
  - components:
    - modules
    - configurations
- application: = graph of components
- Why is this a good choice for a sensor net language?
nesC

- naming conventions:
  - nesC files suffix: .nc
  - C stands for Configuration (Clock, ClockC)
  - M stands for Module (Timer, TimerC, TimerM)

- clarifications:
  - “C” distinguishes between an interface and the component that provides it
  - “M” when a single component has both: a configuration, a module
Interfaces

- used for grouping functionality, like:
  - split-phase operation (send, sendDone)
  - Non blocking operation
  - standard control interface (init, start, stop)
- describe bidirectional interaction:
  ```
  interface Clock {
      command result_t setRate (char interval, char scale);
      event result_t fired ();
  }
  ```

- interface provider must implement commands
- interface user must implement events
## Interfaces

- **Interfaces are bi-directional**
  - **Command:** function call requiring service
  - **Event:** Function call notifying completion of service

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

```c
interface Timer {
    command result_t start (char type,
                            uint32_t interval);
    command result_t stop ();
    event result_t fired ();
}
```

```c
interface SendMsg {
    command result_t send (uint16_t addr,
                           uint8_t len,
                           TOS_MsgPtr p);
    event result_t sendDone ();
}
```

```c
interface ReceiveMsg {
    event TOS_MsgPtr receive (TOS_MsgPtr m);
}
```
Modules

- implements a component’s specification with C code:

```c
module MyComp {
    provides interface X;
    provides interface Y;
    uses interface Z;
}
implementation {
    ...// C code
}
```

- a thread of control crosses components only through their specifications
Modules

- parameterised interfaces:

```plaintext
module GenericComm {
  provides interface SendMsg [uint8_t id];
  provides interface ReceiveMsg [uint8_t id];

  ...
}
implementation {...
}
```

- i.e., it provides 256 instances of SendMsg andRecvMsg interfaces
- they are not strictly necessary – the handler ID can be passed as an argument to the send method
Modules

- implementing the specification:
  - simple interfaces, (e.g. interface Std of type StdControl):

```plaintext
module DoesNothing {
  provides interface StdControl as Std;
}
implementation {
  command result_t Std.init() {
    return SUCCESS;
  }
  command result_t Std.start() {
    return SUCCESS;
  }
  command result_t Std.stop() {
    return SUCCESS;
  }
}
```

DoesNothing.nc
Modules

- calling commands and signaling events
  - simple interface:
    - In addition to stop, put the start.

```plaintext
module TimerM {
  provides interface StdControl;
  provides interface Timer[uint8_t id];
  uses interface Clock;...
}

implementation {
  command result_t StdControl.stop() {
    call Clock.setRate(TOS_I1PS, TOS_S1PS);
  }
  ...
}
```

TimerM.nc
Modules

- posting tasks:

```plaintext
module BlinkM {...
}
implementation {...
    task void processing () {
        if(state) call Leds.redOn();
        else call Leds.redOff();
    }

    event result_t Timer.fired () {
        state = !state;
        post processing();
        return SUCCESS;
    }
}...
```
Configurations

- connected elements must be compatible (interface-interface, command-command, event-event)

- 3 wiring statements in nesC:
  - `endpoint_1 = endpoint_2`
  - `endpoint_1 -> endpoint_2`
  - `endpoint_1 <-> endpoint_2` (equivalent: `endpoint_2 -> endpoint_1`)
Example

- **BlinkM** module:

```plaintext
module BlinkM {
    provides interface StdControl;
    uses interface Clock;
    uses interface Leds;
}

implementation {
    bool state;

    command result_t StdControl.init() {
        state = FALSE;
        call Leds.init();
        return SUCCESS;
    }

    command result_t StdControl.start() {
        return call Clock.setRate(128, 6);
    }

    command result_t StdControl.stop() {
        return call Clock.setRate(0, 0);
    }

    event result_t Clock.fire() {
        state = !state;
        if (state) call Leds.redOn();
        else call Leds.redOff();
    }
}
```
Summary

- small memory footprint +
- concurrency intensive application, event-driven architecture +
- power conservation +
- modular, easy to extend +
- good OS race conditions support. +
- simplistic FIFO scheduling -> no real-time guarantees -
- bounded number of pending tasks -
- no process management -> resource allocation problems -
The Problem

- Your TinyOS application doesn’t work
  - Is the network so messy that routing fails?
  - Is there a bug in your routing algorithm?
  - How do you tell the difference?

- Test TinyOS code
- Reproducible, controlled experiments
- Interaction with network
- Experiment visualization
Solution: TOSSIM and TinyViz

- TOSSIM, TinyOS mote simulator
- Add more realistic radio models to TOSSIM
  - Based on empirical data
  - Incorporate tools to generate loss rates
- TinyViz: visualization and actuation tool
  - Customizable for specific applications
TOSSIM

- TinyOS mote simulator
- Scales to thousands of nodes
- Compiles directly from TinyOS source
- Simulates network at bit level
Empirical Radio Models

- Based on empirical data set
- Extrapolate bit error from packet loss rates
  - Independent bit errors
  - Generate loss graph from physical topologies
- TOSSIM simulates per-link bit errors

![Graph showing loss rate vs. distance]
Model Strengths and Limits

- Many loss topologies for a physical topology
- Repeatable loss rates
- Asymmetric links
- Signal strength not considered
References


TinyViz Goals

- **Visualization**
  - Sensor readings, leds, radio links

- **Actuation: affecting a run of Tossim**
  - Changing underlying radio model, sensor readings

- **Extensibility**
  - Application specific visualization
TinyViz Architecture

Communication

SerialForwarder

TOSSIM

Event Bus

Drawing

Commands
Visualizing Simulation

SerialForwarder

Communication

Event Bus

TOSSIM
Actuating Simulation

SerialForwarder

Communication

Event Bus

TOSSIM
Visualizing Real World

Communication

SerialForwarder

Event Bus
Dynamics of Events and Threads

- bit event =>
- end of byte =>
- end of packet =>
- end of msg send

bit event filtered at byte layer

thread posted to start
send next message

radio takes clock events to detect recv

timer interrupt

sampling for packet start symbol every 50us