

Sensor Network OS

Outline

- Why?
- TinyOS solution
 - Concept
 - Execution model
 - Hardware abstraction architecture
 - Others
- nesC Programs

Outline

- Why?
- TinyOS solution
 - Concept
 - Execution model
 - Hardware abstraction architecture
 - Others
- nesC Programs

Why need a new OS?

- Big !
- Multi-threaded architecture
 - Large number of processes/threads => large memory footprint
- I/O model
 - Blocking I/O (most common)
- Kernel and user space separation
- Typically no energy constraints
- Ample available resources

Sensor Hardware Constraints

- Power
- Limited memory
- Slow CPU
- Size
- Limited hardware parallelisms
- Communication using radio
 - Low-bandwidth
 - Short range



Two AA battery
10KB RAM
255kpbs radio

Desired OS Properties

- Single purpose
- Small memory footprint
- Efficient in power and computation
- Communication is fundamental
- Meet specific timing requirements
- Inexpensive to mass-produce

by comparison ...

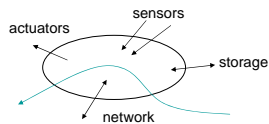
- Highly Constrained resources
 - processing, storage, bandwidth, power
- Applications spread over many small nodes
 - self-organizing Collectives
 - highly integrated with changing environment and network
 - communication is fundamental
- Concurrency intensive in bursts
 - streams of sensor data and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

Therefore ...

- Provide a framework for:
 - Resource-constrained concurrency
 - Defining boundaries
 - Application-specific processing allow abstractions to emerge

Characteristics of Network Sensors

- Small physical size and low power consumption
- node synchronous 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 advices
 - => 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



Outline

- Why?
- TinyOS solution
 - Concept
 - Execution mode
 - Hardware abstraction architecture
 - Others
- nesC Programs

TinyOS Solution

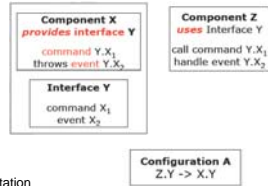
- Concurrency: uses event-driven architecture
- Modularity
 - Application composed of a graph of *components*
 - OS + Application compiles into single executable
- Code communication
 - Uses event/command model; translated to function calls
 - FIFO and non pre-emptive scheduling
- No kernel/application boundary (*security problem*)

Tiny OS Concepts

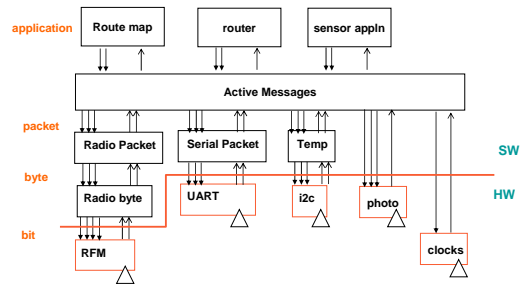
- Scheduler + Graph of Components
 - constrained two-level scheduling model: threads + events
- Component:
 - Commands,
 - Event Handlers
 - Frame (storage)
 - Tasks (concurrency)
- Constrained Storage Model
 - Frame per component, shared stack, no heap
- Very lean multithreading
- Efficient Layering

TinyOS & nesC Concepts

- New Language: **nesC**.
Basic unit of code = **Component**
- Component
 - Process **Commands**
 - Throws **Events**
 - Has a **Frame** for storing local state
 - one per component
 - statically allocated
 - fixed size
 - Uses **Tasks** for concurrency, computation
- Components **provide interfaces**
 - Used by other components to communicate with this component
- Components are **wired** to each other in a **configuration** to connect them

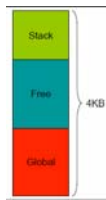


Application = Graph of Components



TinyOS Memory Model

- **STATIC** memory allocation!
 - No heap (malloc)
 - No function pointers
- **Global variables**
 - Available on a per-frame basis
- **Local variables**
 - Saved on the stack
 - Declared within a method

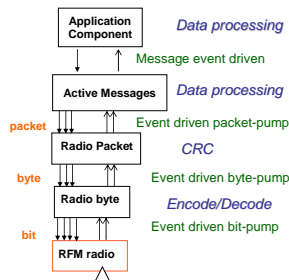


Outline

- Why?
- TinyOS solution
 - Concept
 - Execution model
 - Hardware abstraction architecture
 - Others
- nesC Programs

Execution Model

- commands request action
 - ack/hack at every boundary
 - call command or post task
- events notify occurrence
 - HW interrupt at lowest level
 - may signal events
 - call commands
 - post tasks
- Tasks provide logical concurrency
 - preempted by events
- Migration of HW/SW boundary



Commands/Events/Tasks

- **Commands**
 - Should be non-blocking
 - *i.e.* take parameters start the processing and return to app; postpone time-consuming work by posting a task
 - Can call commands on other components
- **Events**
 - Can call commands, signal other events, post tasks but cannot be signal-ed by commands
 - Asynchronous event Pre-empt tasks, not vice-versa
- **Tasks**
 - FIFO scheduling
 - Non pre-emptable by other task, pre-emptable by interrupt
 - Used to perform computationally intensive work
 - Can be posted by commands and/or events

Async and sync

- Functions declared with keyword “async” are *asynchronous* functions. Otherwise, they are synchronous function.
 - Commands and events without “async”, C-like functions are synchronous.
 - Asynchronous functions can interrupt each other or task.
 - Async command and event can only call/signal async command and event

Tasks

- FIFO scheduling
- Non-preemptive by other task, preemptable by interrupt events (“*async*”)
- Perform computationally intensive work
- Handling of multiple data flows:
 - a sequence of non-blocking command/event through component graph
 - post task for computational intensive work
 - preempt the running task, to handle new data

Scheduler

- Two level scheduling: interrupt events and tasks
- Scheduler is simple FIFO
- a task can not pre-empt another task
- Interrupt events can pre-empt tasks and interrupt (higher priority)

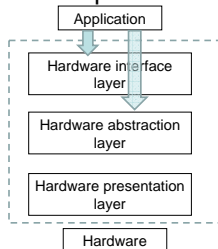


Outline

- Why?
- TinyOS solution
 - Concept
 - Execution model
 - Hardware abstraction architecture
 - Others
- nesC Programs

Hardware Abstraction Architecture (HAA)

- Increase portability and hide hardware implementation from application development



Example

- HIL: DemoSensorC temperature sensors
- HAL: Msp430Adc and Atml128Adc
- HPL: HplAdcMSP430 and HplAtml128

what do I really need to know?

- No dynamic memory (*TinyOS-2.1 now supports dynamic allocation*)
- Single process execution; event-driven
- *commands* and *events* should do little work
- Post a *task* to do long processing

- **Your entire code should be a state-machine (arrgh !)**

i.e. Code should be *split-phase*, for example...

Fn exists to read a *single byte* at a time from flash
Write a wrapper to read *multiple bytes* together from flash

Example

- X calls your component Y to read some bytes from flash using a command `Y.multiRead()`
- Y posts a task to read the first byte calling `Flash.read()`
- Return to caller with status OK
- When `Flash.readDone()` returns, post task A to read 2nd byte
- When `Flash.readDone()` returns, post task A to read 3rd byte
- ...
- When all bytes are read, signal event `Y.multiReadDone()`
- If error was encountered, signal event `Y.multiReadDone()` passing an error value

Summary

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

Other Sensor OSs

- Contiki (SICS)
- SOS (UCLA)
- Nano-RK (CMU)
- Pixies (Harvard)
- Mantis (Colorado)

- Contiki provides a different way than TinyOS
 - Multithread via stackless protothread
 - With a tiny IP stack
 - Run-time reprogramming, loadable code
 - Applications
 - Libraries/components
 - Device drivers

Outline

- Why?
- TinyOS solution
 - Concept
 - Execution model
 - Hardware abstraction architecture
 - Others
- nesC Programs

TinyOS Practical Information

Gaurav Mathur (gmathur@cs.umass.edu)
Sensors Lab, UMass-Amherst

nesC: Naming conventions

- nesC files extension: `.nc`
- `Clock.nc`: either an interface or a configuration
- `ClockC.nc`: a configuration
- `ClockM.nc`: a module
- C stands for Configuration (`Clock`, `ClockC`)
 - "C" distinguishes between an interface and the component that provides it
- M stands for Module (`Timer`, `TimerC`, `TimerM`)
 - "M" when a single component has both: a configuration, a module

changed since tinyos 2.0

In tinyos 2.0

- All public components should be suffixed with 'C'.
- All private components should be suffixed with 'P'.

Interfaces

- Provides the inter-connect
- Fabric between components

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

```
interface X
{
  command result_t doSomething();
  event result_t doSomethingDone();
}
```

Modules

- Implement one or more interfaces
- Can use one or more other interfaces

```
module Provider
{
  provides interface StdControl;
  provides interface X;
  uses interface Z;
}
implementation
{
  // C code
  ....
}
```

MyComp.nc

Modules

```
module Provider
{
  provides interface StdControl;
  provides interface X;
  uses interface Z;
}
implementation {
  command result_t StdControl.init()
  { return SUCCESS; }
  command result_t StdControl.start()
  { return SUCCESS; }
  command result_t StdControl.stop()
  { return SUCCESS; }
  task void signaler()
  {
    signal X.doSomethingDone();
  }
  command result_t
  X.doSomething()
  {
    post signaler();
    return SUCCESS;
  }
}

module User
{
  uses interface X;
}
implementation {
  ....
  task void A()
  {
    res = call X.doSomething();
  }
  ....
  event result_t
  X.doSomethingDone()
  {
    // Yay! doSomething returned ok
    return SUCCESS;
  }
}
```

Implementor User Component

Modules

- Interfaces can also be parameterized
 - Multiple instances can be instantiated and used

```
module Provider
{
  ...
  provides interface X[uint8_t id];
}
implementation {
  uint8_t id;
  task void signaler()
  {
    signal X.doSomethingDone[id]();
  }
  command result_t X.doSomething
  [uint8_t id] ()
  {
    id = id;
    post signaler();
    return SUCCESS;
  }
}
```

Configurations

- Two components are linked together in nesC by **wiring** them
- Interfaces on *user* component are wired to the same interface on the *provider* component
- 3 wiring statements in nesC:
 - `endpoint1 = endpoint2`
 - `endpoint1 -> endpoint2`
 - `endpoint1 <- endpoint2` (equivalent: `endpoint2 -> endpoint1`)

```
Application.nc
configuration Application {
  implementation {
    components Main, Provider, User, SomeComp;
    Main.StdControl -> Provider.StdControl;
    User.X -> Provider.X;
    Provider.Z -> SomeComp.Z;
  }
}
```

Component that **uses** an interface is on the *left*, and the component **provides** the interface is on the *right*.

Compile & Run

- Compiler processes nesC files converting them into a gigantic C file
 - Has both your application & the relevant OS components you are using
- Then platform specific compiler compiles this C file
 - Becomes a single executable
- Loader installs the code onto the Mote (Mica2, Telos, etc.)

```
make telosb install
make telosb reinstall
make telosb install,1
make telosb install bsl,/dev/ttyUSB0
```