Wireless Embedded Systems and Networking

Foundations of IP-based Ubiquitous Sensor Networks

Operating Systems for Communication-Centric Devices
TinyOS-based IP-WSNs

David E. Culler
University of California, Berkeley
Arch Rock Corp.
July 9, 2007

Technology Perspective

Client Tier: (desk, lap, PDA, MP3, phone)
- Windows, Wince, Symbian, Linux/Java

Server Tier:
- Unix (Linux, Solaris, AIX, HPux), Windows
- App Servers (Axis, J2EE, Weblogic, SAP, Oracle, …)

Router/Gateway Tier:
- Linux, Linux, Linux

Embedded Tier: (mote)
Traditional Systems

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

User

System

Network Stack

Transport

Network

Data Link

Physical Layer

Threads

Address Space

Files

Drivers

Drivers

Routers

by comparison, WSNs ...

- 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
- Unclear where the boundaries belong
  - even HW/SW will move

=> Provide a framework for:
- Resource-constrained concurrency
- Defining boundaries
- Appl’n-specific processing and power management
  allow abstractions to emerge
TinyOS

- New operating system built specifically for wireless sensor networks
  - Small, robust, communication centric design
  - Resource-constrained concurrency
  - Structured Event-driven SW architecture
  - Tool for protocols and dist. Algorithms
- Designed for synthesis and verification
  - Eg. Ptolemy, Metropolis, …
  - Whole-system compile-time analysis
- Rich set of services and development environment
- World-wide adoption
  - Open source, lead by UCB / Intel
  - Corporate and academic (1000s)
  - Dozen of platforms
  - de facto sensor net standard

A worldwide community

<table>
<thead>
<tr>
<th>TinyOS</th>
<th>Storage</th>
<th>Wireless</th>
<th>Processing</th>
<th>Sensors</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Ubimom</td>
<td>Luxoff Lab</td>
<td>Crossbow</td>
<td>NEST</td>
</tr>
<tr>
<td></td>
<td>ADURA</td>
<td>DUST</td>
<td>WRL</td>
<td>Streetline</td>
</tr>
<tr>
<td></td>
<td>Ember</td>
<td>Sensics</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Wireless Sensor

- CC2420
- Radio byte
- Radio Packet
- UART
- Serial Packet
- ADC
- Temp
- Active Messages
- Clocks
- Route map
- Router
- Sensor appln
- SW
- HW
- Photo
- Router
- Sensor appln
- Application

Country/Region (%)
- United States: 41.6%
- Europe: 26.2%
- Asia: 15.6%
- Other: 16.4%
Modern Mote Tier TinyOS Architecture

- Embedded applications built on a rich set of node services.
  - Timing, sensor streams, storage
  - Local processing
  - Reliable, low-power communication
  - Platform independent + extensions

Abstractions Emerge from Experience

- Communication Centric
- Resource-Constrained
- Event-driven Execution
Stack Library Alternative

- Link & networks protocols buried in block-box library
  - Ember, Figure8, . . .
- No execution model or storage model
- Arbitrary system/user code must TiCKle it “sufficiently often”
- Undefined call duration
- No system services
- Difficult to validate

- Same hardware, but a very different approach

TinyOS from First Principles
Characteristics of Network Sensors

- Small physical size and low power consumption
- **Concurrency-intensive operation**
  - multiple flows, not wait-command-respond
- **Limited Physical Parallelism and Controller Hierarchy**
  - primitive direct-to-device interface
  - Asynchronous and synchronous devices
- **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

Classical RTOS approaches

- **Responsiveness**
  => Provide some form of user-specified interrupt handler
    » User threads in kernel, user-level interrupts
    - Guarantees?
- **Deadlines / Controlled Scheduling**
  - Static set of tasks with prespecified constraints
    » Generate overall schedule
    => Doesn't deal with unpredictable events, especially communication
  - Threads + synchronization operations
    => Complex scheduler to coerce into meeting constraints
    - Priorities, earliest deadline first, rate monotonic
    - Priority inversion, load shedding, live lock, deadlock
    » Sophisticated mutex and signal operations
- **Communication among parallel entities**
  - Shared (global) variables: ultimate unstructured programming
  - Mail boxes (msg passing)
  => external communication considered harmful
  - Fold in as RPC
- **Requires multiple (sparse) stacks**
  - Preemption or yield
Alternative Starting Points

- Event-driven models
  - Easy to schedule handfuls of small, roughly uniform things
    » State transitions (but what storage and comm model?)
  - Usually results in brittle monolithic dispatch structures

- Structured event-driven models
  - Logical chunks of computation and state that service events via execution of internal threads

- Threaded Abstract machine
  - Developed as compilation target of inherently parallel languages
    » vast dynamic parallelism
    » Hide long-latency operations
  - Simple two-level scheduling hierarchy
  - Dynamic tree of code-block activations with internal inlets and threads

- Active Messages
  - Both parties in communication know format of the message
  - Fine-grain dispatch and consume without parsing

- Concurrent Data-structures
  - Non-blocking, lock-free (Herlihy)

TinyOS design goals

- Simple framework for resource constrained concurrency
  - Single stack

- Flexible hardware/software and system boundary

- Expressive enough to build sophisticated, application specific system structures
  - Avoid arbitrary constraints on optimization

- Communication is integral to execution
  - Asynchrony is first class

- Promote robustness
  - Modular
  - Static allocation
  - Explicit success/fail at all interfaces
  - Reuse

- Ease of interpositioning
Embedded System Design: **Hardware Abstraction**

- Abstract a **hardware unit** for convenient software access.
- Datasheet describes set of interfaces (pins, wires, busses) and operations
  - Commands that can be asserted or issued to it
  - Events that it will signal or raise
  - Interfaces to other hardware units that it is attached to
- Internally the unit has state and computational processes that operate in parallel with other units.

![Diagram of hardware unit](image)

**Embedded System Design: Data Acquisition**

- Configure and command ADC to sample external I/O attached to sensor.
  - Either directly or over a bus protocol
- Obtain readings upon notification by polling or handling interrupts
  - One short or periodic
- Perform processing on the readings (smoothing, thresholding, transformation) and possibly signal higher level notification
- Similar for DAC to actuator

![Diagram of data acquisition](image)
Embedded System Design: Protocol Implementation

- For
  - Bus Protocols within a node,
  - Link Protocols between two nodes in direct communication,
  - Network Protocols between possibly widely separate node.
- Each has
  - Set of operations that it issues
  - Set responses that it receives
    » synchronous or asynchronous,
  - state it maintains,
  - state-transition diagram that it implements
- And various commands and events that define its interface above and below
  - Exceptions, etc.

Tiny OS Concepts

- System = Scheduler + graph of Components
  - Hierarchical
- Component:
  - Set of bidirectional Command/Event Interfaces
  - Commands Handlers
  - Event Handlers
  - Frame (storage)
  - Tasks (concurrency)
- Constrained two-level scheduling model
  - tasks + events
- Constrained Storage Model
  - frame per component,
  - Single shared stack,
  - no heap
- Structured event-driven processing
- Very lean multithreading
- Efficient Layering
  - Events can signal events
- Extremely modular construction
  - Separates creation and composition of functional elements
**Application = Graph of Components**

- Modular construction of Protocols.
- Graph of cooperating state machines on shared stack
- Execution driven by interrupts

* Early TinyOS 0.x component graph going all the way down to modulating the RF channel in software.

**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**

- **data processing**
  - message-event driven
  - event-driven packet-pump
  - event-driven byte-pump
  - event-driven bit-pump
TinyOS Execution Contexts

- Events generated by interrupts preempt tasks
- Tasks do not preempt tasks
- Both essential process state transitions

Dynamics of Events and Threads

- Bit event filtered at byte layer
- Bit event => end of byte => end of packet => end of msg send
- Thread posted to start send next message
- Radio takes clock events to detect recv
Programming TinyOS - nesC

- TinyOS 1.x and TinyOS 2.x are written in an extension of C, called nesC
- Applications are too!
  - just additional components composed with the OS components
- Provides syntax for TinyOS concurrency and storage model
  - commands, events, tasks
  - local frame variables
- Rich Compositional Support
  - separation of definition and linkage
  - robustness through narrow interfaces and reuse
  - interpositioning
- Whole system analysis and optimization
- Platform independent data types and structure
  - because packets are sent between different kinds of processors!

Composition

- A component specifies a set of interfaces by which it is connected to other components
  - provides a set of interfaces to other components
  - uses a set of interfaces provided by other components
- Interfaces are bi-directional
  - include commands and events
- Interface methods form the external namespace of the component
  - Composition by “wiring”
Split-phase abstraction of HW

- Command synchronously initiates action
- Device operates concurrently
- Signals event(s) in response
  - ADC
  - Clock
  - Send (UART, Radio, ...)
  - Recv – depending on model
  - Coprocessor
- Higher level (SW) processes don’t wait or poll
  - Allows automated power management
- Higher level components behave the same way
  - Tasks provide internal concurrency where there is no explicit hardware concurrency
- Components (even subtrees) replaced by HW and vice versa

TASKS

- provide concurrency internal to a component
  - longer running operations
  - are preempted by events
  - able to perform operations beyond event context
  - may call commands
  - may signal events
  - not preempted by tasks
- Simple (pluggable) Scheduler
  - Composition exercises substantial control over scheduling

```
{ ...
  post TskName();
  ...
}
task void TskName { ...
  }
```
Typical application use of tasks

- event driven data acquisition
- schedule task to do computational portion

```c
event result_t sensor.dataReady(uint16_t data) {
    putdata(data);
    post processData();
    return SUCCESS;
}

void processData() {
    int16_t i, sum=0;
    for (i=0; i < maxdata; i++)
        sum += (rdata[i] >> 7);
    display(sum >> shiftdata);
}
```

- 128 Hz sampling rate
- simple FIR filter
- dynamic software tuning for centering the magnetometer signal (1208 bytes)
- digital control of analog, not DSP
- ADC (196 bytes)

Tasks in low-level operation

- transmit packet
  - send command schedules task to calculate CRC
  - task initiated byte-level data pump
  - events keep the pump flowing
- receive packet
  - receive event schedules task to check CRC
  - task signals packet ready if OK
- i2c component
  - i2c bus has long suspensive operations
  - tasks used to create split-phase interface
  - events can proceed during bus transactions
- Timer
  - Post task in-critical section, signal event when current task complete

Make SW look like HW
Structured Events vs Multi-tasking

- **Storage**
- **Control Paradigm**
  - Always block/yield – rely on thread switching
  - Never block – rely on event signaling
- **Communication & Coordination among potentially parallel activities**
  - Threads: global variables/mailboxes, mutex, signaling
  - Preemptive – handle many potential races
  - Non-preemptive
    » All interactions protected by costs system synch ops
  - Events: signaling

- **Scheduling:**
  - Complex threads require sophisticated scheduling
  - Collections of simple events ??

Modern TinyOS Service Architecture

- Domain-Specific Application Components
  - Service Interface
  - Persistent Attributes & Event Streams
  - Int/Boot
  - Messages
  - Commands
  - Attributes
  - Events
  - Discovery
  - Network Collection, Dissemination, & Routing
  - Links
  - Flash
  - Radio / Serial
  - Sensor / Actuator
  - Microcontroller Core, Timers, Buses, Onboard ADCs
  - Management & Power
  - Device Attributes & Event Streams
    - Device
    - Domain-Specific
    - Device Drivers
  - Domain-Specific
  - Application Components
  - Hardware
    - TelosB
    - MicaZ
    - Intel Mote2
Embedded Networking Requirements

- Reliable Dissemination
- Data Collection and Aggregation
- Point-to-point Transfers

- Reliably over lossy links
- At low power
  - Idle listening, management, monitoring
- Adapting to changing conditions
- Scalar and Bulk Versions
**Neighbor Communication**

![Diagram of neighbor communication]

**Multihop Routing**

- Upon each transmission, one of the recipients retransmits
  - determined by source, by receiver, by ...
  - on the 'edge of the cell'
Power to Communicate

Route-Free Dissemination
Data Collection

TinyOS 2x Embedded IP Architecture

Higher Level Embedded Web Services

UDP/TCP L4
IP route L3
IP 6LowPAN L2
Basic Health & Mgmt Services
Basic Configuration Services

Low-Power 802.15.4
Flash Storage
Virtual ms Timer
RTC
Scheduler
Pwr Mgr
Sensor Drivers

SPI, i2c, UART
µs Timer
Ext. INT
GPIO Pins
ADC

TinyOS 2x Embedded IP Architecture
Canonical SensorNet Network Architecture

Typical IP Network
WSNs in an IP context

Stand-alone embedded networks

Internally connected embedded networks

Router / Gateway Architecture

Just another TinyOS Application / System "RockBridge"
TEP - TinyOS Enhancement Proposals

- TEP 1: TEP Structure and Key Words [HTML]
- TEP 2: Hardware Abstraction Architecture [HTML]
- TEP 3: Coding Standards [HTML]
- TEP 101: ADC [HTML]
- TEP 102: Timers [HTML]
- TEP 103: Storage [HTML]
- TEP 106: Schedulers and Tasks [HTML]
- TEP 107: Boot Sequence [HTML]
- TEP 108: Resource Arbitration [HTML]
- TEP 109: Sensorboards [HTML]
- TEP 111: message_t [HTML]
- TEP 112: Microcontroller Power Management [HTML]
- TEP 113: Serial Communication [HTML]
- TEP 114: SiDs: Source and Sink Independent Drivers [HTML]
- TEP 115: Power Management of Non-Virtualized Devices [HTML]
- TEP 116: Packet Protocols [HTML]
- TEP 117: Low-Level I/O [HTML]
- TEP 118: Dissemination [HTML]
- TEP 119: Collection [HTML]
- TEP 123: Collection Tree Protocol (CTP) [HTML]
- TEP 124: Link Estimation Exchange Protocol (LEEP) [HTML]
- TEP 125: TinyOS 802.15.4 Frames [HTML]
- TEP 126: CC2420 Radio Stack [HTML]

TinyOS Execution Philosophy

- Sleep almost all the time.
- Wake-up (quickly) when there is something to do.
- Process it and all other concurrent or serial activities as rapidly as possible.
  - Structured, event driven concurrency
- Never wait!!!
- Automatically go back to sleep
TinyOS Structured Design Philosophy

- Think hard about components
  - Well-define behavior, well-define interfaces
- Compose components into larger components
- Flexible structured design of entire system
  - And application
- Dealing with distributed system of many resource-constrained devices embedded in hard to reach places and coping with noise, uncertainty and variation.
- Make the node, the network, and the system as robust as possible.
- KEEP IT SIMPLE!