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

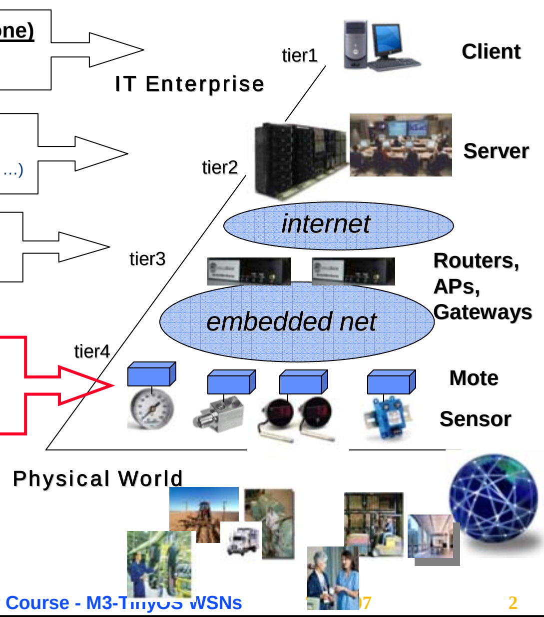
Technology Perspective

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

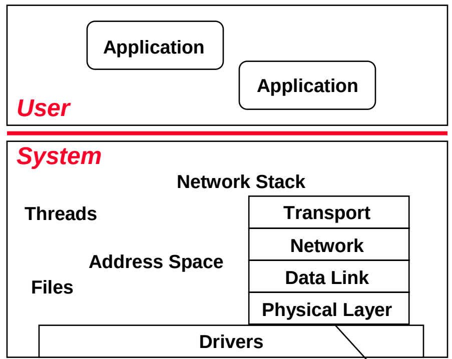
Server Tier:
- Unix (Linux, Solaris, AIX, HPux), Windows
- App Servers (Axis, J2EE, Weblogic, SAP, Oracle, …)

Router/Gateway Tier:
- Linux, Linux, Linux

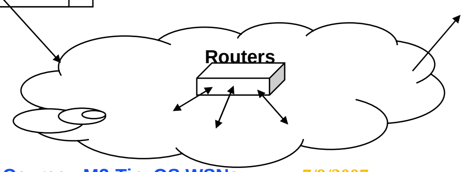
Embedded Tier: (mote)



Traditional Systems



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



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

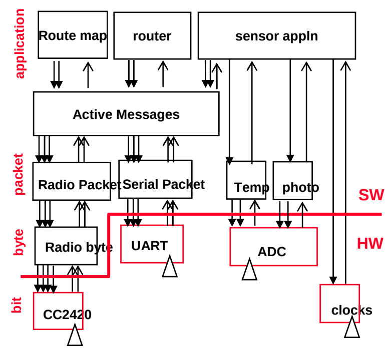
• Defining boundaries

• Appl'n-specific processing and power management

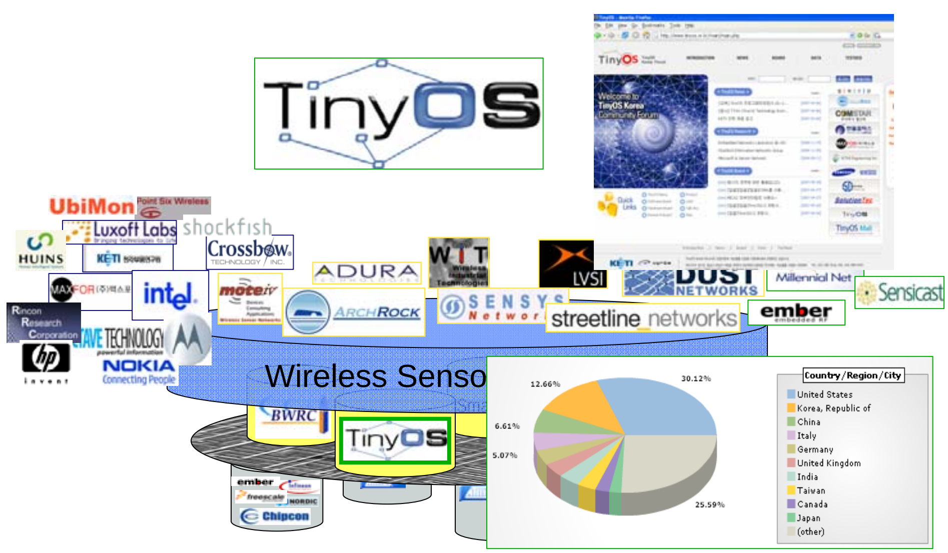
allow abstractions to emerge

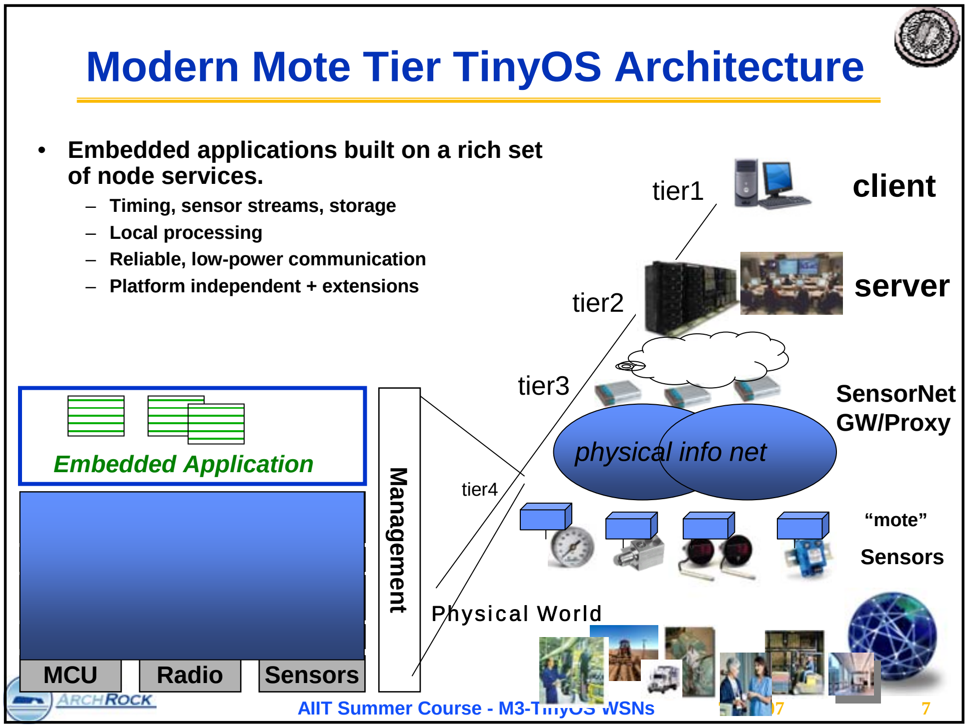
TinyOS

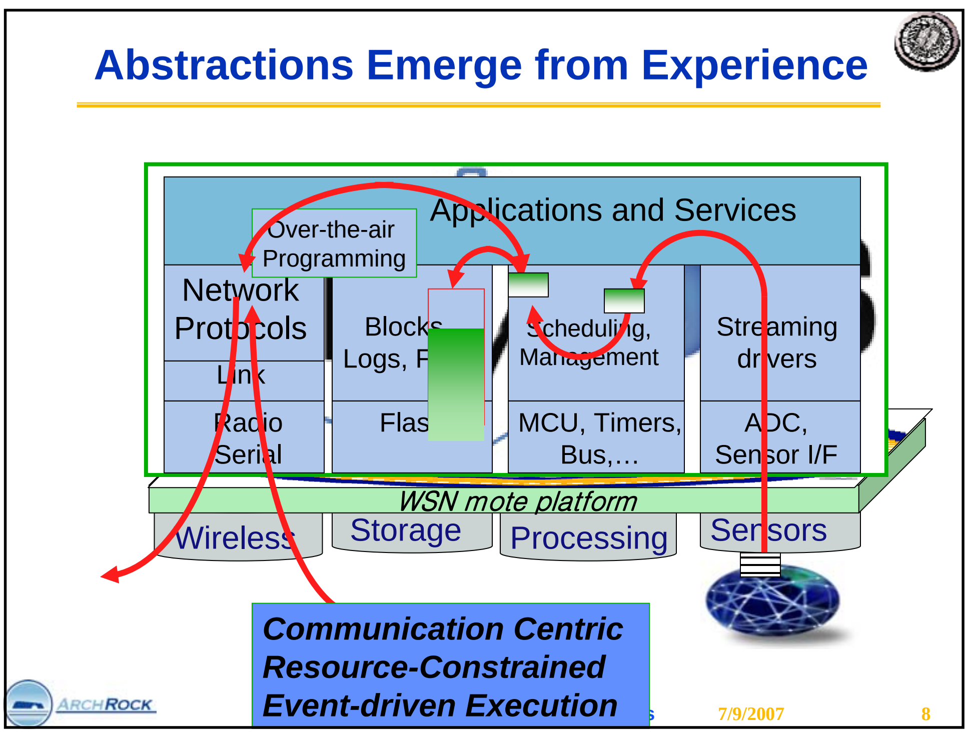
- New operating system built specifically for wireless sensor networks
  - Small, robust, communication centric design
  - Resource-constrained concurrency
  - Structured Event-driven SW architecture
  - Tool for protocols and dist. Algorithms
- Designed for synthesis and verification
  - Eg. Ptolemy, 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

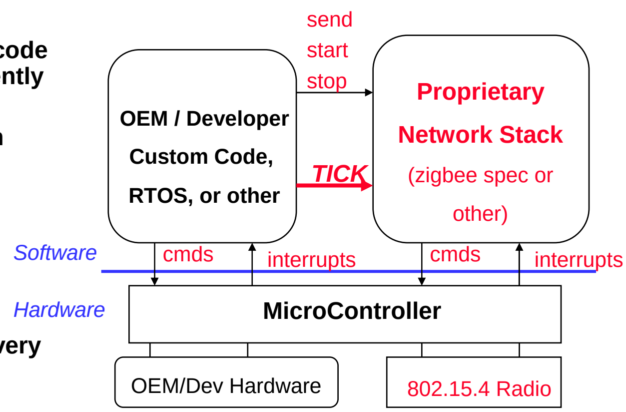



Modern Mote Tier TinyOS Architecture
• Embedded applications built on a rich set of node services.
– Timing, sensor streams, storage
– Local processing
– Reliable, low-power communication
– Platform independent + extensions
tier1
client
tier2
server
tier3
SensorNet GW/Proxy
physical info net
tier4
"mote"
Sensors
Physical World
Embedded Application
Management
MCU
Radio
Sensors
ARCHROCK
AIIT Summer Course - M3-TinyOS WSNs
7


Abstractions Emerge from Experience
Applications and Services
Over-the-air Programming
Network Protocols
Link
Radio Serial
Blocks, Logs, F
Flas
Scheduling, Management
MCU, Timers, Bus,…
Streaming drivers
ADC, Sensor I/F
WSN mote platform
Wireless
Storage
Processing
Sensors
Communication Centric
Resource-Constrained
Event-driven Execution
ARCHROCK
7/9/2007
8

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

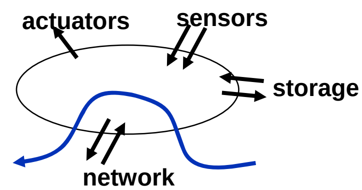


- Same hardware, but a very different approach

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

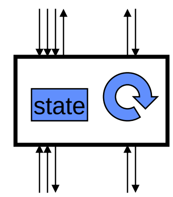
- Event-driven models
  - Easy to schedule handfuls of small, roughly uniform things
    - » State transitions (but what storage and comm model?)
  - Usually results in brittle monolithic dispatch structures
- Structured event-driven models
  - Logical chunks of computation and state that service events 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

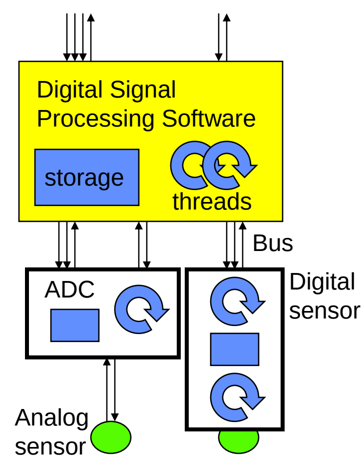
Embedded System Design: Hardware Abstraction

- 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.



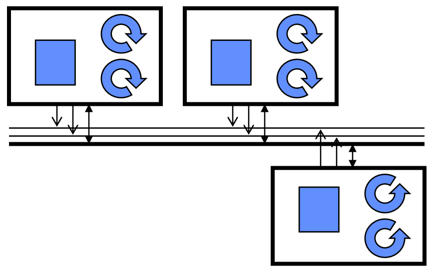
Embedded System Design: Data Acquisition

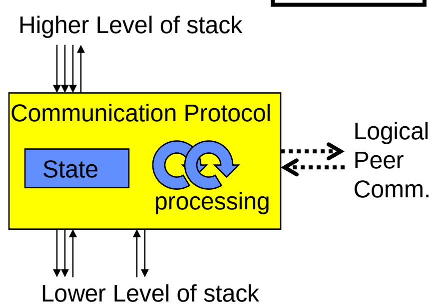
- Configure and command ADC to sample external I/O attached to sensor.
  - Either directly or over a bus protocol
- Obtain readings upon notification by polling or handling interrupts
  - One short or periodic
- Perform processing on the readings (smoothing, thresholding, transformation) and possibly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



Embedded System Design: Protocol Implementation

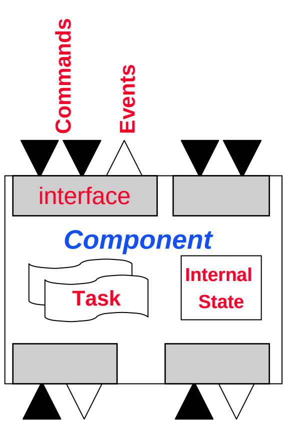
- For
  - Bus Protocols within a node,
  - Link Protocols between two nodes in direct communication,
  - Network Protocols between possibly widely separate node.
- Each has
  - Set of operations that it issues
  - Set responses that it receives
    - » synchronous or asynchronous,
  - state it maintains,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





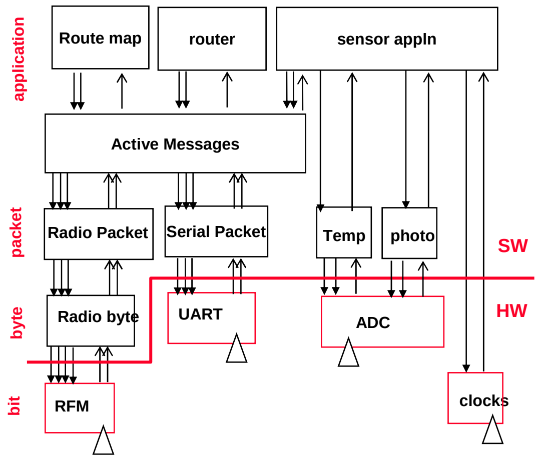
Tiny OS Concepts

- 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.

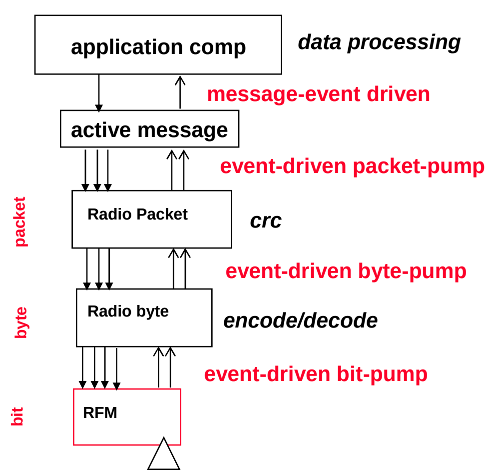
Graph of cooperating state machines on shared stack

Execution driven by interrupts

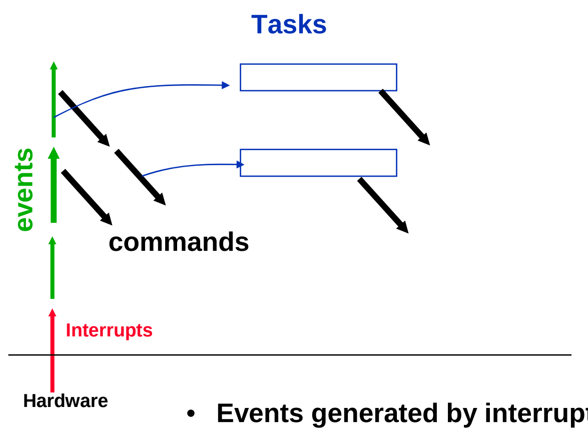
* Early TinyOS 0.x component graph going all the way down to modulating the RF channel in software.

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**

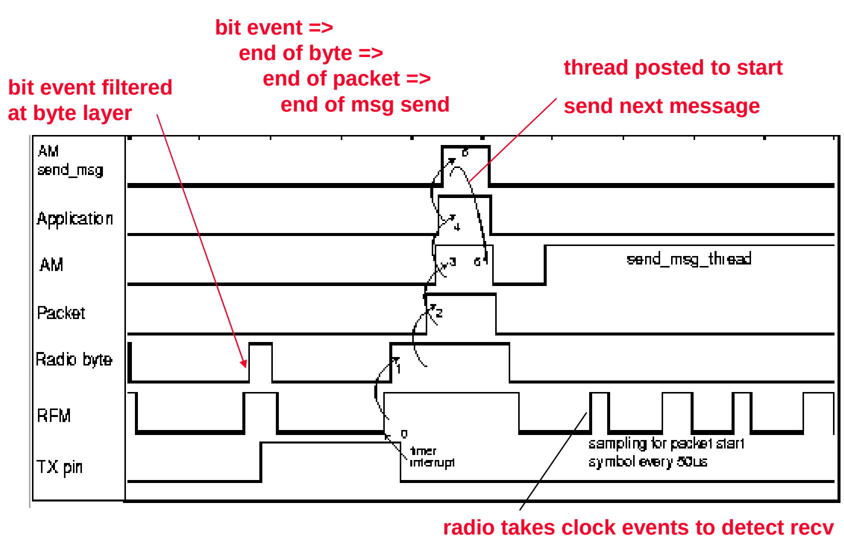


TinyOS Execution Contexts



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

Dynamics of Events and Threads



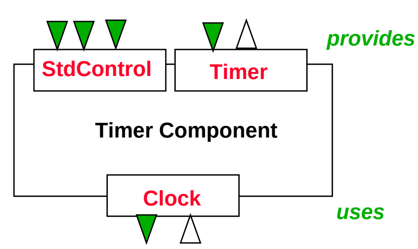
Programming TinyOS - nesC

- 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”

```
provides
    interface StdControl;
    interface Timer:
uses
    interface Clock
```



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();
…
}
```

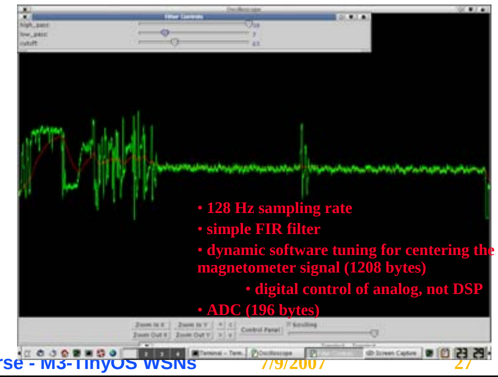
→

```
task void TskName {
…
}
```

Typical application use of tasks

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

```
event result_t sensor.dataReady(uint16_t data) {
    putdata(data);
    post processData();
    return SUCCESS;
 }
task void processData() {
    int16_t i, sum=0;
    for (i=0; i ‹ maxdata; i++)
        sum += (rdata[i] ›› 7);
    display(sum ›› shiftdata);
 }
```



Tasks in low-level operation

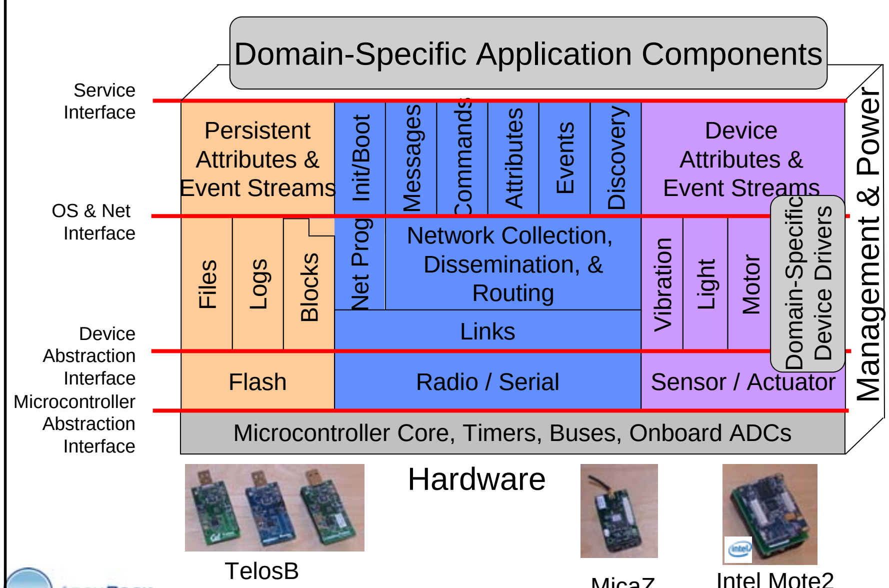
- transmit packet
  - send command schedules task to calculate CRC
  - task initiated byte-level data pump
  - events keep the pump flowing
- receive packet
  - receive event schedules task to check CRC
  - task signals packet ready if OK
- i2c component
  - i2c bus has long suspensive operations
  -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

Make SW look like HW

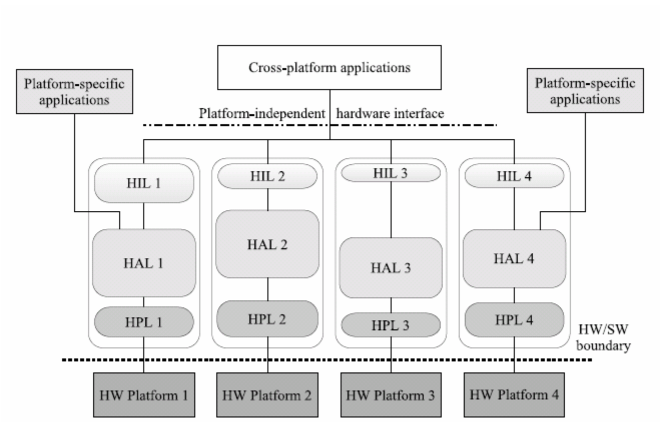
Structured Events vs Multi-tasking

- 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-premptive
    - » All interactions protected by costs system synch ops
  - Events: signaling
- Scheduling:
  - Complex threads require sophisticating scheduling
  - Collections of simple events ??

Modern TinyOS Service Architecture



TinyOS 2.0 – abstraction architecture



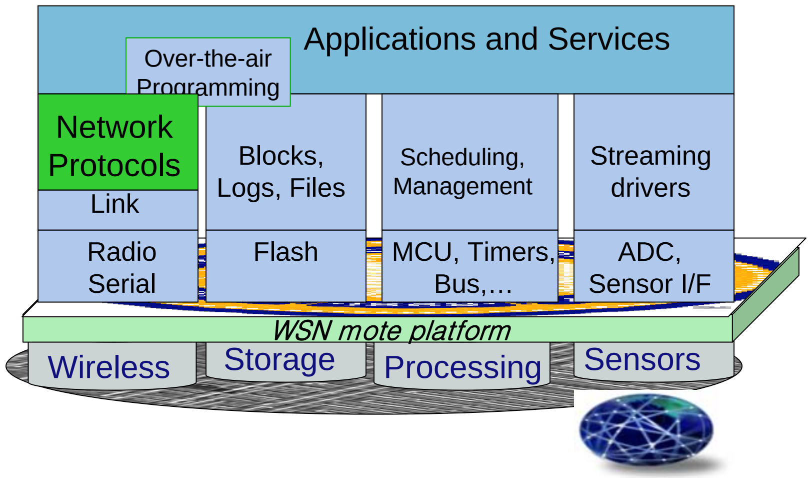
Flexible Hardware Abstraction for Wireless Sensor Networks, Vlado Handziski, Joseph Polastre, Jan-Hinrich Hauer, Cory Sharp, Adam Wolisz, David Culler,*In Proceedings of the Second European Workshop on Wireless Sensor Networks (EWSN '05), January 31-February 2, 2005.*

Sample of TinyOS Platforms

Platform	Year	Microcontroller	Radio	Flash	Sensor Interface
Rene2	2000	Atmel Atmega163, 8 KB prog, 1 KB RAM	RFM TR1000, 916 MHz, ASK, 40 Kbps, GPIO	24LC256,32 KB, EEPROM, I^2C	51-pin Expansion; UART, I^2C, SPI, GPIO
Mica	2001	Atmel Atmega128, 128 KB Prog, 4 KB RAM	RFM TR1000, 916 MHz, ASK, 40 Kbps, UART	AT45DB041B, 512 KB, SPI	same
Mica2	2002	Atmega128	ChipCon CC1000, 916 MHz, FSK, 38 Kbps, SPI	AT45DB041B	same
MicaZ	2004	Atmega128	ChipCon CC2420, 2.4 GHz, QPSK, 250 Kbps, SPI	AT45DB041B	same
iMOTE	2004	Zeevo ZV4002 with ARM 7, 64 KB SRAM, 512 KB Flash	ZV4002, Bluetooth		
BTnode	2004	Atmega128	CC1000 & ZV4002, Bluetooth	SST39	
TelosB	2004	TI MSP 430, 60 KB program, 10 KB RAM	ChipCon CC2420	STM25P	10-pin header; Integrated temperature, light, and humidity
EyesIFX	2005	TI MSP 430	Infineon TDA5250	AT45DB041B	
Fleck	2005	Atmega128	Nordic nRF903, 433 MHz, GFSK, 76 Kbps,	512 KB	31-pin
Schockfish	2006	TI MSP430	Xemics XE1205 868 MHz 153 kbps	512 KB Flash	30-pin Molex
Mote2	2006	Intel PXA271, 256 KB SRAM, 32 MB DRAM	ChipCon CC2420	Intel Strata, 32 MB direct	31+21-pin top; UART, SPI, I^2C, SDIO; 40+20-pin bottom; GPIO, USB, JTAG, MSL, CIF

FIGURE 1. Comparison of TinyOS platforms

Wireless Embedded Networks

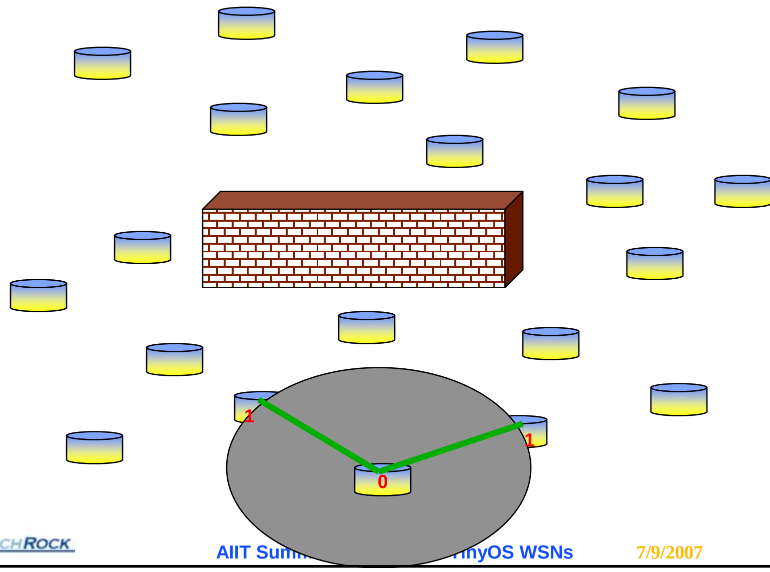


Embedded Networking Requirements

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

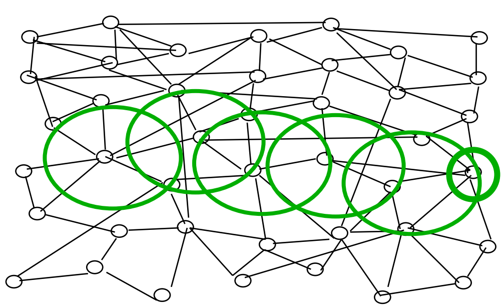
- **Reliably over lossy links**
- **At low power**
  - **Idle listening, management, monitoring**
- **Adapting to changing conditions**
- **Scalar and Bulk Versions**

Neighbor Communication

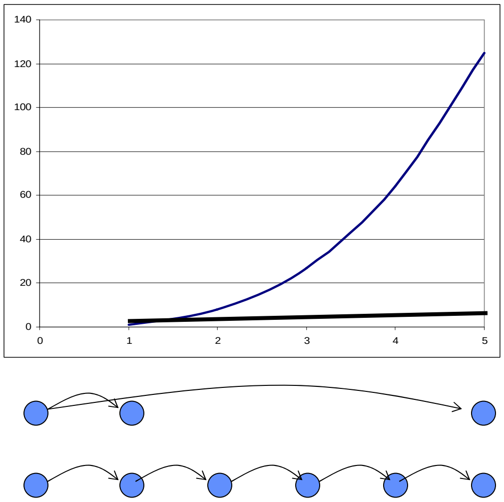


Multihop Routing

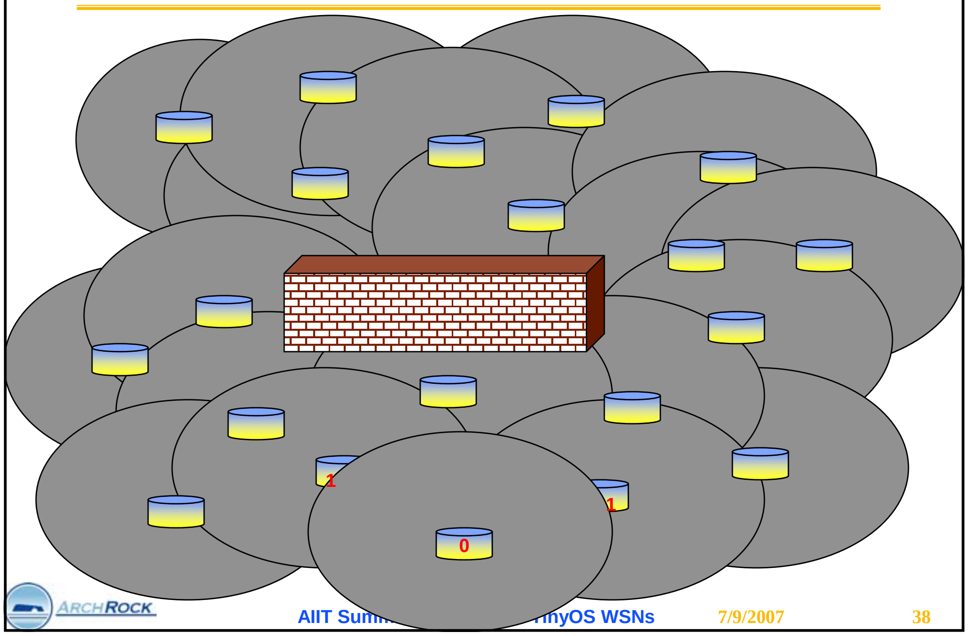
- Upon each transmission, one of the recipients retransmits
  - determined by source, by receiver, by …
  - on the 'edge of the cell'



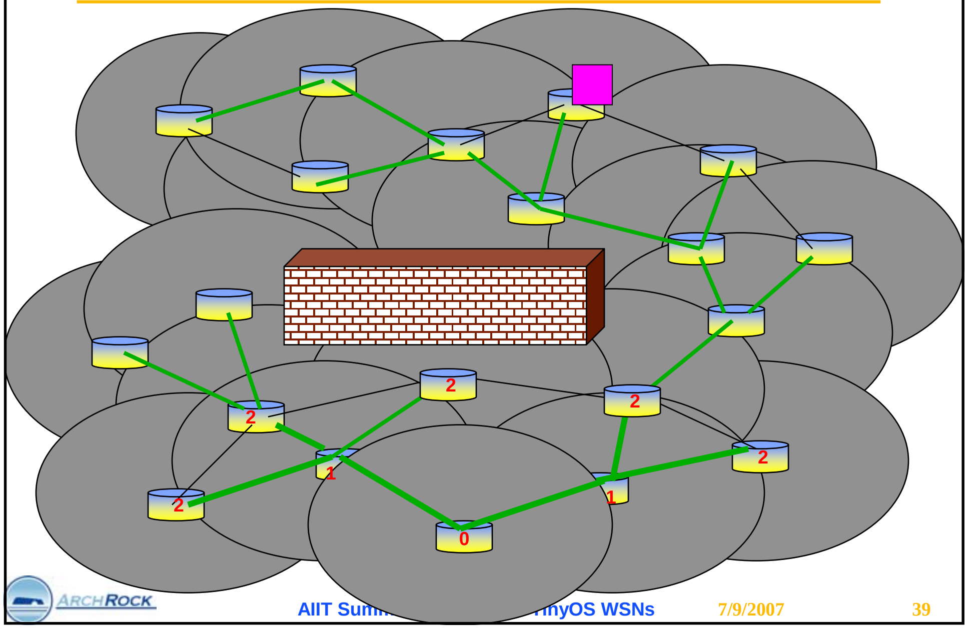
Power to Communicate



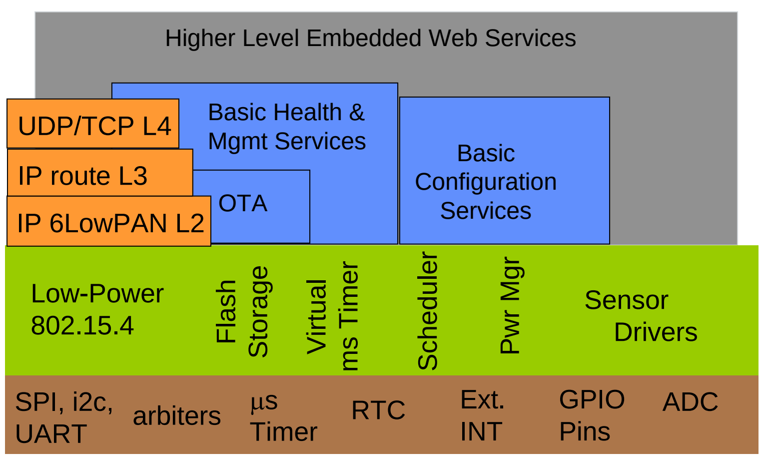
Route-Free Dissemination



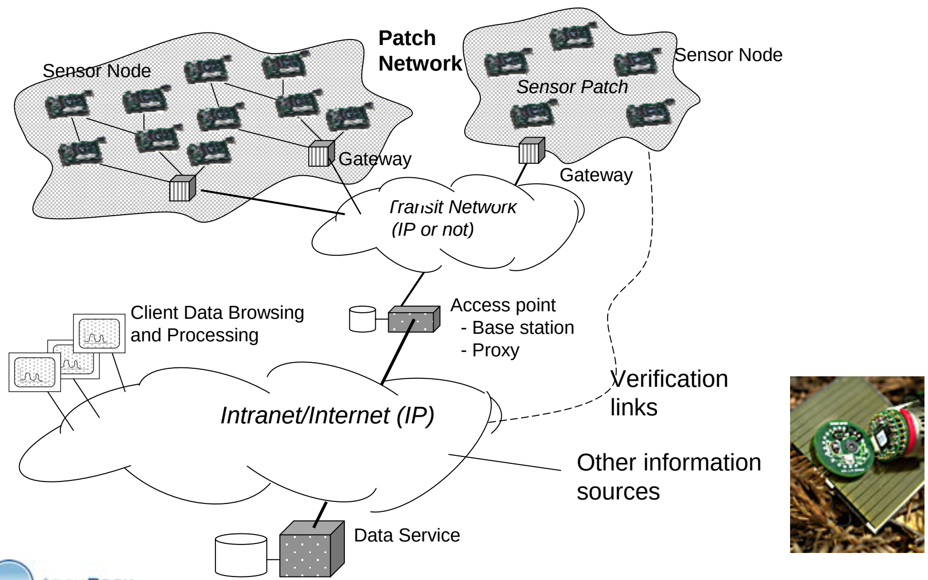
Data Collection



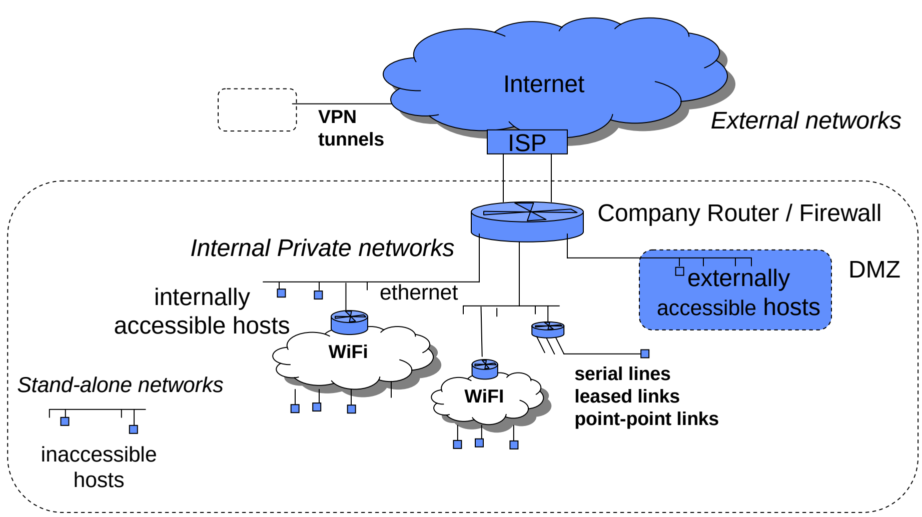
TinyOS 2x Embedded IP Architecture



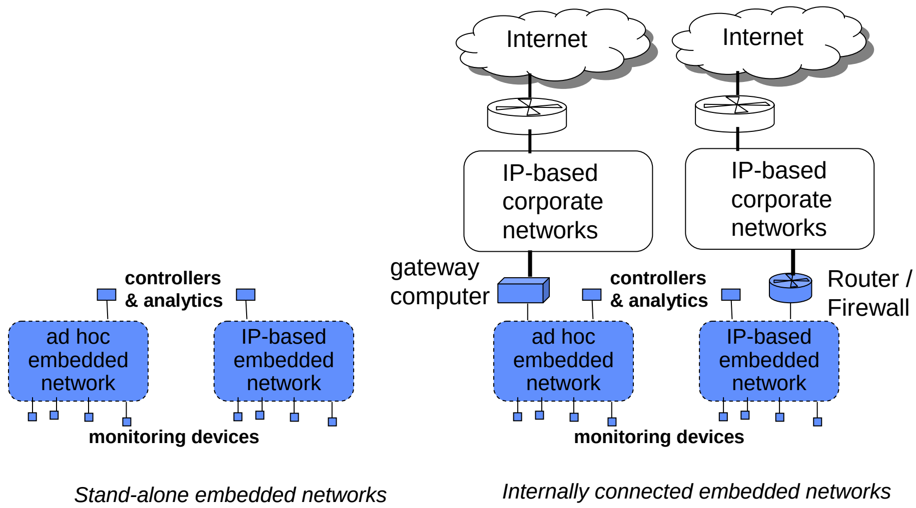
Canonical SensorNet Network Architecture



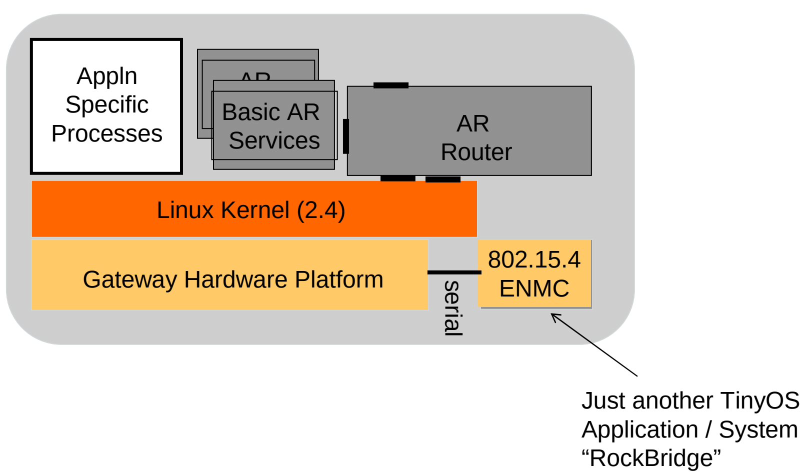
Typical IP Network



WSNs in an IP context



Router / Gateway Architecture



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!**