Low-Power Sensor Networks

A Case Study in Seeking Distributed Predictability

Philip Levis (and John Regehr in abesntia)
Stanford University and the University of Utah
NSF HCSP-CPS Workshop
November 30, 2006
Alexandria, VA
Predictable

- TinyOS and the specter of low-power
  - Limited resources and communication
  - Black box operation
- Systems are easy; predictable/dependable systems are hard
  - Large numbers, distributed through space
- Failures are inevitable: isolating them is paramount
  - Systems approach: TinyOS, TinyOS 2.0/T2
  - Networking approach: MNet
- This talk has nothing to say about real-time
  - More on why later
Outline

- A brief history of: 1.0, 1.1 and 2.0 (T2)
- T2 core structure, language/OS co-design
- MNet architecture
- Real Time?
Sensor networks are on the horizon...
... but what are they going to do?
  - What problems will be important?
  - What will communication look like?
  - What will hardware platforms look like?
Having an operating system is nice...
... but how do you design one with these uncertainties?
The TinyOS Goals
(ASPLOS 2000)

- Allow high concurrency
- Operate with limited resources
- Adapt to hardware evolution
- Support a wide range of applications
- Be robust
- Support a diverse set of platforms
1999

2000
TinyOS 0.4
perl scripts + C
WeC platform (30)

2001
TinyOS 0.63
perl scripts + C
WeC + rene (1000)

2002
TinyOS 1.0
nesC 1.0
rene + mica

2003
TinyOS 1.1
nesC 1.1
>20 platforms

2006
TinyOS 2.0
nesc 1.2
9 platforms
TinyOS Basics

- A program is a set of components
  - Components can be easily developed and reused
    - Adaptable to many application domains
  - Components can be easily replaced
  - Components can be hardware or software
    - Allows boundaries to change unknown to programmer
- Hardware has internal concurrency
  - Software needs to be able to have it as well
- Hardware is non-blocking
  - Software needs to be so as well
TinyOS Basics
(2000)

Non-volatile Storage

Hardware Crypto

Component

Non-volatile Storage

Software Crypto

Component

Tasks

Software

Encrypt

Encrypt

Finished

Component

Non-volatile Storage

Non-volatile Storage
TinyOS Basics, Continued
(2002, nesC)
The TinyOS Goals
(A Scorecard, 2005)

- Allow high concurrency (A)
- Operate with limited resources (A–)
- Adapt to hardware evolution (B)
- Support a wide range of applications (B)
- Be robust (D)
- Support a diverse set of platforms (B–)
Robustness Drives Design

- Allow high concurrency (A)
- Operate with limited resources (A–)
- Adapt to hardware evolution (B)
- Support a wide range of applications (B)
- Be robust (D)
- Support a diverse set of platforms (B–)
TinyOS 0.6 -> TinyOS 1.0

- Introduce nesC language instead of perl + C
- Compilation benefits
  - Pre-nesC linked compiled components into an executable
  - The nesC compiler generates a single C file
    - Whole program analysis
    - Whole program optimization (code the native compiler likes)
    - Dead code elimination
- Interfaces
  - Establish programming abstraction as a language abstraction
  - Prevent bugs
TinyOS 1.0 -> TinyOS 1.1

- Major addition: async keyword
- Synchronous code: tasks (non-preemptive)
- async code is safe to call outside a task
  - Interrupt handlers are all async (preemptive code)
- To call sync code, async code must post a task
  - sync examples: start a ms timer, send a packet
  - async examples: start a 32kHz alarm, send a byte over a bus
Async vs. Sync

- Async code can preempt sync code
  - Might cause data races, atomic statements
- Sync code is written assuming no preemption
  - Sync code executes atomically with respect to other sync code
  - Simple, easy to write, no data races
- Tasks are the interface which transforms async to sync
- The explicit sync/async distinction allows nesC to detect all data races at compile time
- Fixed >100 data races in TinyOS (6 races/1000 lines)
Outline

- A brief history of: 1.0, 1.1 and 2.0 (T2)
- T2 core structure, language/OS co-design
- MNet architecture
- Real Time?
TinyOS Evolution

- TinyOS 1.x improved component dependability
  - Adding language mechanisms for better checking
  - Low-level system code (few writers, many users)
  - OK to trade verbosity for dependability
  - Push checks to compile-time when possible
- TinyOS 2.0 takes the next step: system predictability
Failures of Implementation

- Components intended to be independent
- Unforeseen interactions
  - “The ADC hangs when I send packets!”
  - “Time synchronization gives crazy readings!”
  - “When I turn off the radio my application hangs!”
  - “When I boot with flash support the radio stops working!”
Failures of Structure

- TinyOS 1.x has no resource management
- Most operations can fail at any time (busy)
  - Packet transmission
  - Bus access
  - ADC sampling
- Depends on higher-level retries
  - Global “done” events (e.g., GenericComm.sendDone)
  - Fan-out has deterministic scheduling
- No component isolation
TinyOS has always followed a static allocation policy
  - Argument: dynamic allocation leads to dynamic failures
- One major 1.x exception: the task scheduler
  - Major source of failures
  - Inherent inter-component dependency
Concurrency Model

○ T2 has the same basic concurrency model
  ○ Tasks, sync vs. async

○ T2 changes the task semantics
  ○ TinyOS 1.x: post() can return FAIL, can post() multiple times (shared slots)
  ○ T2: post returns FAIL iff the task is already in the queue (single reserved slot per task)
Static Binding

- Run-time vs. compile time parameters

```c
interface CC2420Register {
    command uint16_t read(uint8_t reg);
    command uint8_t write(uint8_t reg, uint16_t val);
    command uint8_t strobe();
}
component CC2420C {
    provides interface CC2420Register;
}

interface CC2420StrobeReg {
    command uint8_t strobe();
}
component CC2420C {
    provides interface CC2420StrobeReg as SNOP;
    provides interface CC2420StrobeReg as STXONCCA;
    ....
}
```
Static Allocation

- You know what you’ll need: allocate it at compile-time (statically)
- Depending on probabilities is a bet
  - I.e., “it’s very unlikely they’ll all need to post tasks at once” = “they will”
- You know what components will use a resource, can allocate accordingly
  - In some cases, static allocation can save memory
  - Less defensive programming/error handling
module Foo {
    bool busy;

    command result_t request() {
        if (!busy() &&
            post fooTask() == SUCCESS) {
            busy = TRUE;
            return SUCCESS;
        }
        else {
            return FAIL;
        }
    }
}

module Foo {
    bool busy;

    command result_t request() {
        return post fooTask();
    }
}

TinyOS 1.x

T2
The Power of Counting

- Basic language mechanism that TinyOS provides
- Ability to count elements in an application at compile time
  - unique(key): for each key, returns a unique number starting at 0
  - uniqueCount(key): returns number of calls to unique(key)
- Each needed service or abstraction can use its own key
  - Tasks: unique(“TinySchedulerC.BasicTask”), etc.

unique(...) = 1

unique(...) = 2

uniqueCount(...) = 8
Basic OS Requirement: QoS

- Flaw in many protocols: under load, routing fails
  - Data packets overflow queues
  - Control packets are lost, routes disintegrate
- Priorities are difficult: they can break promises
  - I’ve agreed to forward this data packet, but have to drop it now...
  - Defining priorities across many protocols can be difficult
- Want to promise a minimum quality of service
  - Control traffic receives at least k/n of the available bandwidth
  - A control packet has to wait for at most x packets
QoS Through an OS Interface

- Every component that needs to send a packet instantiates an instance of a packet sending service
  - Broadcast, collection, unicast, etc.
- Each instance of the service can have at most one outstanding packet at any time
- Like tasks, send fails if and only if a packet is already pending

![Diagram of QoS Through an OS Interface]

26
QoS Through Counting

- Each instance allocates a queue entry with unique(...)
- The service has a queue of length uniqueCount(...)
- Implementation scans through the queue for pending packets
Extending the Model

- A protocol can allocate more than one sender for a greater share
Extending the Model

- A protocol can allocate more than one sender for a greater share
- A protocol can introduce its own queue
- Still uses $k/n$ bandwidth
More Defensive Programming

- Interfaces allow programs to easily swap component implementations
  - Exchange SerialAMSenderC for AMSenderC
- Interfaces are weakly specified
  - Allow implementation differences
  - E.g., 1.x SendMsg vs 2.0 Send
- Weak specifications lead to defensive programming
  - More code -> more errors
  - Wastes resources
Interface Contracts

- Specify valid call patterns with annotations
  - Per-interface basis (heavy reuse)
  - Both sides of the interface
- Base case: hardware abstractions follow contracts
- Inductive static, dynamic, run-time checking
  - Run-time approach has detected several serious bugs in 1.x (which turn out to be impossible by design in 2.0)
Outline

- A brief history of: 1.0, 1.1 and 2.0 (T2)
- T2 core structure, language/OS co-design
- MNet architecture
- Real Time?
Sensornets Are Hard

- Sensor networks often fail/operate poorly
  - Great Duck Island network: median yield 58% [SenSys 2004]
  - Redwood network: median yield 40% [SenSys 2005]
  - Volcano network: median yield: 68% [OSDI 2006]

- Survey of causes
  - Protocol conflicts/interference
  - Collisions and congestion induced loss
  - Neighbor management (with layer 2 scheduling, e.g. TMAC)
  - Don’t know!

- Low-power, limited resources make complete logging prohibitively expensive...
Management

- Give operators a peek into the sensornet black box
- SNMS [EWSN 2005]: lightweight get/set
- Sympathy [SenSys 2005]: expert system
MNet Principle

- The difficulty in deploying and developing sensornets is part of the essence of this class of systems.
  - Large numbers, limited energy, distributed over space, different views of the environment, noise, local optimizations, etc.
  - This is more than an artifact.
- MNet principle: Improve visibility into the internal operation of the network.
  - Quantify: Minimize the energy required to identify the cause of network behavior.
- Case study: network protocols.
Goal
Inter-Protocol Interference

- Snooping is a common routing approach
  - Implicit acks, rate control, backpressure, etc.
- Vulnerable to inter-protocol interference
  - Reduces energy efficiency, can even cause failures
- One misbehaving protocol can prevent anyone else from performing well
Isolation

- Isolating behaviors simplifies reasoning.
  - Basic technique in systems: apply to networks
- If any protocol X, Y, Z can a protocol to fail, then we have a larger (more expensive) state space to explore
- We need a way to isolate protocols from one another, so they can operate concurrently but not interfere.
- Mechanism: grant-to-send (GTS)
Grant–To–Send

- A transmitter may embed a quiet time in a packet.
- No-one except the destination may transmit for the duration of the quiet time (including transmitter).
- Sending a packet grants the channel to the receiver.
Grant-To-Send

- A transmitter may embed a quiet time in a packet.
- No-one except the destination may transmit for the duration of the quiet time (including transmitter).
- Sending a packet grants the channel to the receiver.
Grant–To–Send

- A transmitter may embed a quiet time in a packet.
- No-one except the destination may transmit for the duration of the quiet time (including transmitter).
- Sending a packet grants the channel to the receiver.
Grant–To–Send

- A transmitter may embed a quiet time in a packet.
- No-one except the destination may transmit for the duration of the quiet time (including transmitter).
- Sending a packet grants the channel to the receiver.
Grant–To–Send

- A transmitter may embed a quiet time in a packet.
- No–one except the destination may transmit for the duration of the quiet time (including transmitter).
- Sending a packet grants the channel to the receiver.
Grant–To–Send

- A transmitter may embed a quiet time in a packet.
- No-one except the destination may transmit for the duration of the quiet time (including transmitter).
- Sending a packet grants the channel to the receiver.
Fairness

- Isolation is insufficient.
  - The simplest approach is to not let anyone do anything.
- Every protocol should receive its fair share of the network bandwidth.
- Wireless is inherently distributed
  - Different views of the channel
  - Perfect fairness is not always possible (but we can be close)
- Mechanism: fair queueing
  - GTS times represent channel utilization
  - Naturally fit into fair queueing
Fair Queueing
(Demers, Shenker, and Keshav)

- Send protocol which has lowest channel utilization.

P1

P2

P3
Fair Waiting Protocol

- Uses Grant–To–Send mechanism
- Sits between CSMA layer and network layer
- Fair queueing according to the channel occupation
  - Considers the grant duration as a channel occupation
Single-Hop Uniform Lossless Load

- Ideal case without collisions and packet losses
- Perfect fairness among nodes and protocols
  - CSMA allows all nodes to have equal chance of transmission
  - All nodes agree on channel usages of protocols, thus perfect fairness among protocols
- Perfect Isolation
  - Every node waits until the current quiet time expires
Loss

- Lost packets can cause inconsistent view of the channel occupation times of protocols
- Experimental Setting:
  - Five nodes in single-hop range
  - Three protocols with different quiet times (20ms / 40ms / 80ms)
- Normalized share of one node
- High channel fairness: 0.99 (Jain’s Fairness Index)
- However, individual nodes are servicing protocols unevenly
- "Ping-pong Effect"
Multihop Uniform Load

- Uniform loads on 40 nodes on motelab
  - 20 / 40 / 80 ms (Fig. 1)
  - 20 / 60 / 140 ms (Fig. 2)
- Plain (no decay)
  - Global fairness : 0.997
  - Poor transmit fairness
- Decaying every 500 ms
  - Global Fairness : 0.995
  - Best transmit fairness
- Understanding the decaying period better is a future work
Real Loads – ARC

- ARC: rate-limiting collection protocol [Mobicom 2001]
- Goodput and cost for two separate ARC instances running in the presence of two other protocols (PSFQ and Trickle)
- FWP increases ARC goodput by 23–30% and decreases cost by 5–10%
Network Protocols

- FWP isolates network protocols from each other
- How do we isolate causes within a protocol?
- Apply minimization principle to higher layers
Case Study: Collection

- 5 principal causes of packet loss

1. Retransmit timeout
2. Queue drop
3. False positive duplicate suppression
4. Reboot
5. Kaboom!

Origin sequence numbers, THL field
Decision Tree

No more packets?
- Death/Disconnect

Seq No. Break?
- Reboot

High THL?
- Suppression
- Queue Drop

Requires no querying of network state!
MNet Architecture

- Elevate management and visibility to an architectural principle and design goal
- Isolation of causes
- Fairness (protocol, node, application...)
- FWP as the narrow waist
Outline

- A brief history of: 1.0, 1.1 and 2.0 (T2)
- T2 core structure, language/OS co-design
- MNet architecture
- Real Time?
The Real-Time Tension

- Real-time is inherently unfair
  - Some people get to go first!
  - Understanding *why* something failed is hard (Mars Rover)
  - Necessitates local operations and internal decisions
  - Makes it more difficult to understand the internal operation of the system

- Optimal scheduling of a scarce resource
  - Uncommon in sensornets because utilization is so low...
  - Event-driven, not periodic workloads

- Wireless is an inherent challenge
  - Outside of your control
Predictability

○ Being able to assume things will behave in a certain way
○ Breaking outside current approaches
  ○ Language–OS co-design
  ○ Static, dynamic, run-time approaches
○ Predictable networks, not just systems
  ○ Network is increasingly cause of failure
  ○ Wireless more so...
Questions