Sensor Networks and Macroprogramming

Matt Welsh

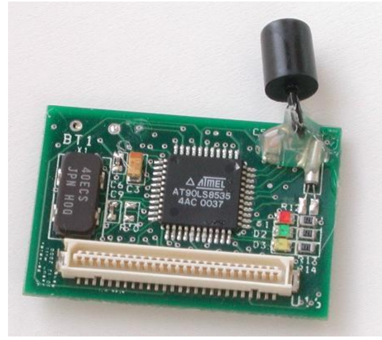
Harvard University

Division of Engineering and Applied Sciences

mdw@eecs.harvard.edu

Sensor networks

WeC (1999)



René (2000)

DOT (2001)

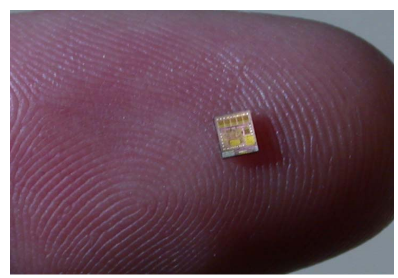
Exciting emerging domain of deeply networked systems

- Low-power, wireless “motes” with tiny amount of CPU/memory
- Large federated networks for high-resolution sensing of environment

Drive towards miniaturization and low power

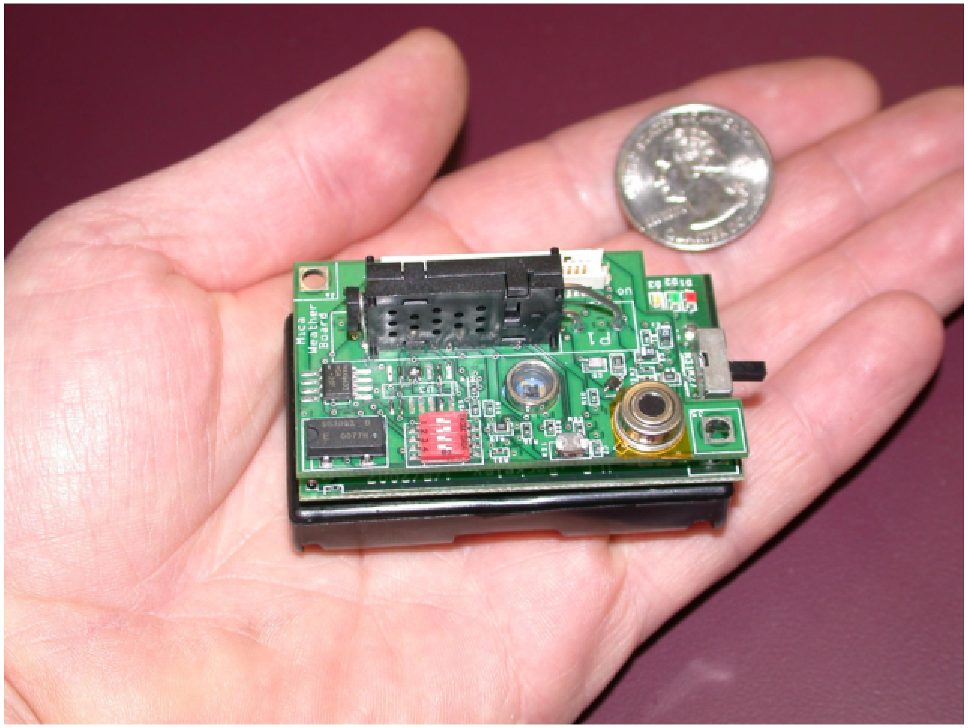
- Eventual goal - complete systems in 1 mm^3, MEMS sensors
- Family of Berkeley motes as COTS experimental platform

MICA (2002)



Speck (2003)

The Berkeley Mica mote



- ATMEGA 128L (4 MHz 8-bit CPU)
- 128KB code, 4 KB data SRAM
- 512 KB flash for logging
- 916 MHz 40 Kbps radio (100' max)
- Sandwich-on sensor boards
- Powered by 2AA batteries

Several thousand produced, used by over 150 research groups worldwide

- Get yours at `www.xbow.com` (or `www.ebay.com`)
- About $100 a pop (maybe more)

Great platform for experimentation (though not particularly small)

- Easy to integrate new sensors/actuators
- 15-20 mA active (5-6 days), 15 μA sleeping (21 years, but limited by shelf life)

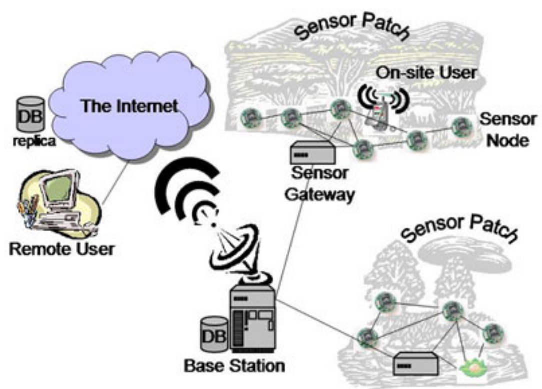
Typical applications

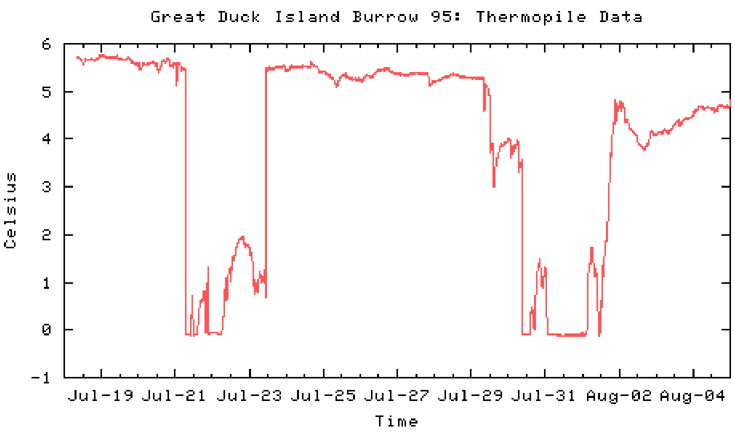
Object tracking

- Sensors take magentometer readings, locate object using centroid of readings
- Communicate using geographic routing to base station
- Robust against node and link failures

Great Duck Island - habitat monitoring

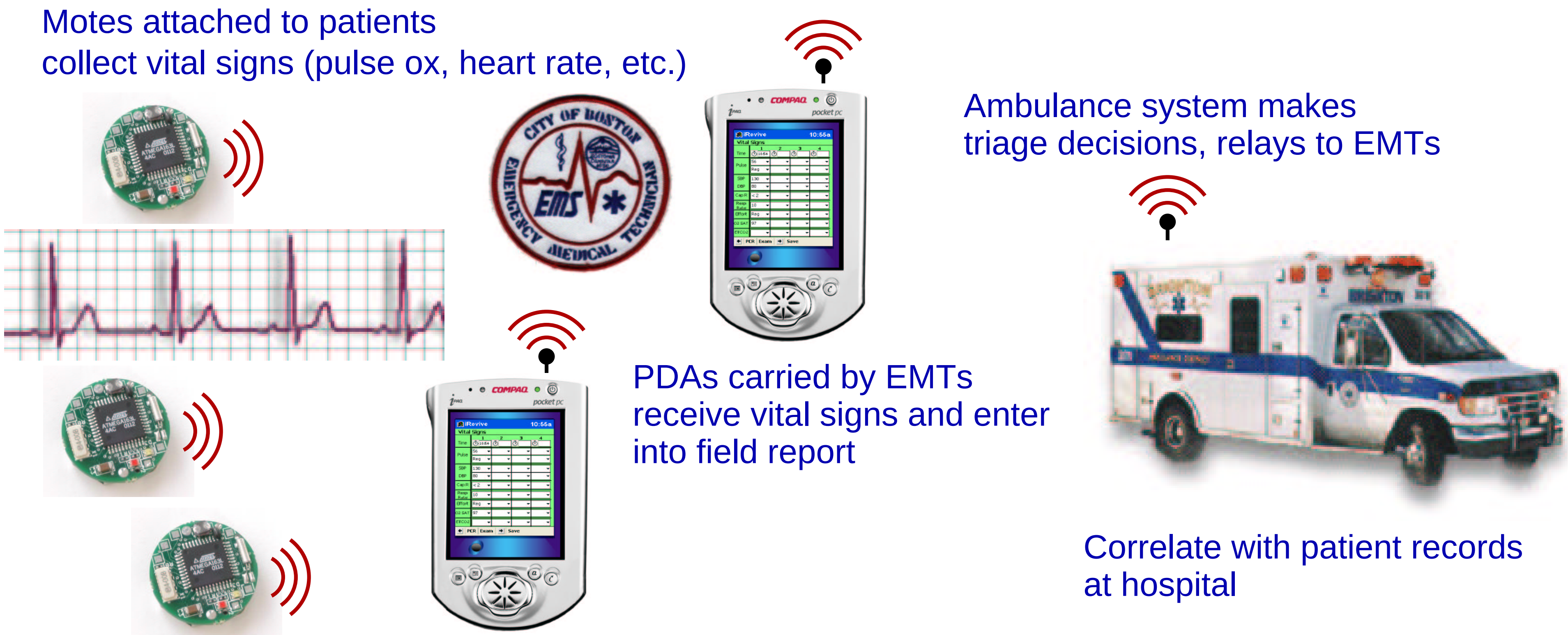
- Gather temp, humidity, IR readings from petrel nests
- Determine occupancy of nests to understand breeding/migration behavior
- Live readings at `www.greatduckisland.net`





Vital Dust: Emergency Medical Triage

with S. Moulton, M.D., Boston Medical Center and
M. Gaynor, Boston University



- Patient motes form ad-hoc wireless network with EMT PDAs
- Enables rapid, continuous survey of patients in field
- Requires secure, reliable communications

Sensor network programming challenges

Driven by interaction with environment

- Data collection and control, not general purpose computation
- Reactive, event-driven programming model
- Exploit locality of communication in network

Extremely limited resources

- Very low cost, size, and power consumption
- Typical embedded OSs consume hundreds of KB of memory
- Battery lifetime is the critical resource

Many nodes with sparse and error-prone connectivity

- Reliable communication too expensive
- Ad-hoc formation of communication paths
- Applications must be robust to individual node failure

TinyOS

Very small “operating system” for sensor networks

- Core OS requires 396 bytes of memory

Component-oriented architecture

- Set of reusable system components: sensing, communication, timers, etc.
- No binary kernel - build *app specific* OS from components

Concurrency based on **tasks** and **events**

- **Task:** deferred computation, runs to completion, no preemption
- **Event:** Invoked by module (upcall) or interrupt, may preempt tasks or other events
- Very low overhead, no threads

Split-phase operations

- No blocking operations
- Long-latency ops (sensing, comm, etc.) are **split phase**
- Request to execute an operation returns immediatety
- Event signals completion of operation

nesC: A Programming Language for Sensor Networks

With D. Gay, P. Levis, R. von Behren, E. Brewer, D. Culler

Supports concurrency model of TinyOS

- Small (396 bytes minimum) OS for sensor networks
- No blocking operations - operations are **split phase**
- Very low overhead, no threads

Dialect of C with support for *components*

- Components **provide** and **require** interfaces
- Create application by wiring together components using **configurations**

Whole-program compilation and analysis

- Allows aggressive cross-component inlining
- Static data-race detection

Important restrictions permit extensive optimization

- No function pointers (makes whole-program analysis difficult)
- No dynamic memory allocation
- No dynamic component instantiation/destruction

nesC interfaces

nesC interfaces are bidirectional

- **Command:** Function call from one component requesting service from another
- **Event:** Function call indicating completion of service by a component
- Grouping commands/events together makes inter-component protocols clear

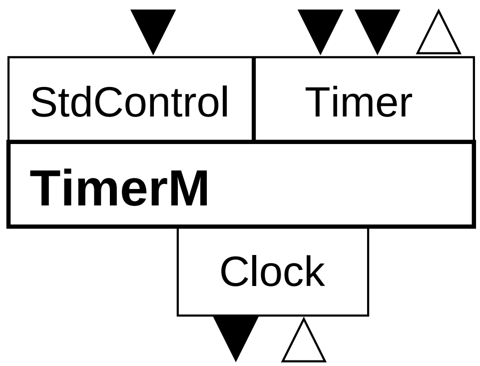
```
interface Timer {
  command result_t start(char type, uint32_t interval);
  command result_t stop();
  event result_t fired();
}

interface SendMsg {
  command result_t send(TOS_Msg *msg, uint16_t length);
  event result_t sendDone(TOS_Msg *msg, result_t success);
}
```

nesC components

Two types of components

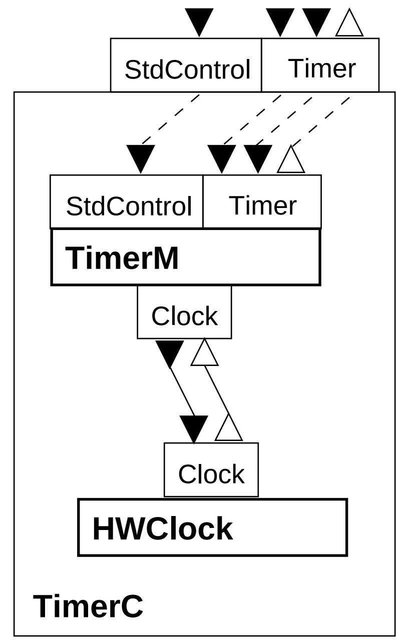
- **Modules** contain implementation code
- **Configurations** wire other components together
- An application is defined with a single top-level configuration

```
module TimerM {
  provides {
    interface StdControl;
    interface Timer;
  }
  uses interface Clock;

} implementation {

  command result_t Timer.start(char type, uint32_t interval) { ... }
  command result_t Timer.stop() { ... }
  event void Clock.tick() { ... }
}
```



Configuration example

- Allow aggregation of components into "supercomponents"




```
configuration TimerC {
  provides {
    interface StdControl;
    interface Timer;
  }

} implementation {

   components TimerM, HWClock;

   // Pass-through: Connect our "provides" to TimerM "provides"
   StdControl = TimerM.StdControl;
   Timer = TimerM.Timer;

   // Normal wiring: Connect "requires" to "provides"
   TimerM.Clock -> HWClock.Clock;
}
```

nesC Optimizations

Application	Size		Reduction
	optimized	*unoptimized*	
Base TinyOS	396	646	41%
Runtime	1081	1091	1%
Habitat monitoring	11415	19181	40%
Surge	14794	20645	22%
Object tracking	23525	37195	36%
Maté	23741	25907	8%
TinyDB	63726	71269	10%

Inlining benefit for 5 sample applications.

Cycles	optimized	unoptimized	Reduction
Work	371	520	29%
Boundary crossing	109	258	57%
Total	**480**	**778**	**38%**

Clock cycles for clock event handling, crossing 7 modules.

Inlining and dead code elimination saves both space and time

- Elimination of module crossing code (function calls)
- Cross-module optimization, e.g., common subexpression elim

Macroprogramming

How do you program a system composed of a large number of distributed, volatile, error-prone systems?

- Initial focus is on sensor networks, natürlich
- Approach applies to many other domains:
- Distributed systems, protocol design, and P2P to name a few

High-level language for aggregate programs

- Examples: contour finding, object tracking, distributed control
- TinyDB [Madden et al.] is one step in this direction

Current programing models are node centric

- NesC focuses entirely on individual nodes, rather than the aggregate
- Want to program the "whole system"

Current programing models are too low-level

- Scientists don't want to think about gronky details of radios, timers, battery life, etc.
- Like writing Linux by toggling switches on a PDP-11

Macroprogramming goals

Queries over "virtual" coordinate spaces

- Define overlay set in space, e.g., grid, disc-neighborhood, Voronoi diagrams
- Allow query to arbitrary point in that space
- E.g., "sensor reading at $(30.5, 42.6)$"
- Runtime interpolates across nearby sensor values

Temporal operations and aggregation

- Triggers and event-driven operations
- Sample and aggregate over time
- Specify timeouts, periodic execution, etc.

Express fidelity and uncertainty of data

- Main goal: support "lossy programming"
- Errors/uncertainty (i.e., interpolation) exposed to programmer
  - ▷ *Programmer can supply* **timeout** *for aggregate operations*
  - ▷ *Operation reports* **yield**

nesCscript - Simplifying sensor programming

Eliminate split-phase operations in favor of blocking

- Supported by lean **fibers** library
- Abstract common operations and communication
- Much easier than NesC, where every blocking operation is a separate task!

```
// Track an object through the sensor field
while (1) {
    my_reading = get_reading();
    if (my_reading > THRESHOLD) {
        foreach (n in neighbors) {
            // Read data from neighboring nodes
            neighbor_reading[n] := remote_reading[n];
        }
    }
    object_location = compute_centroid(my_reading, neighbor_readings);
    send_to_base(object_location);
}
```

Runtime primitives for macroprogramming

Neighborhood management

- Build graph of connected nodes
- Various definitions - radio, geographic, interest sets
- Approximate planar mesh construction - pruned Yao graph

Neighborhood data reflection

- Nodes publish local variables that can be read by neighbors
- No attempt at consistency
- Any read can fail or timeout (one aspect of aggregate “yield”)

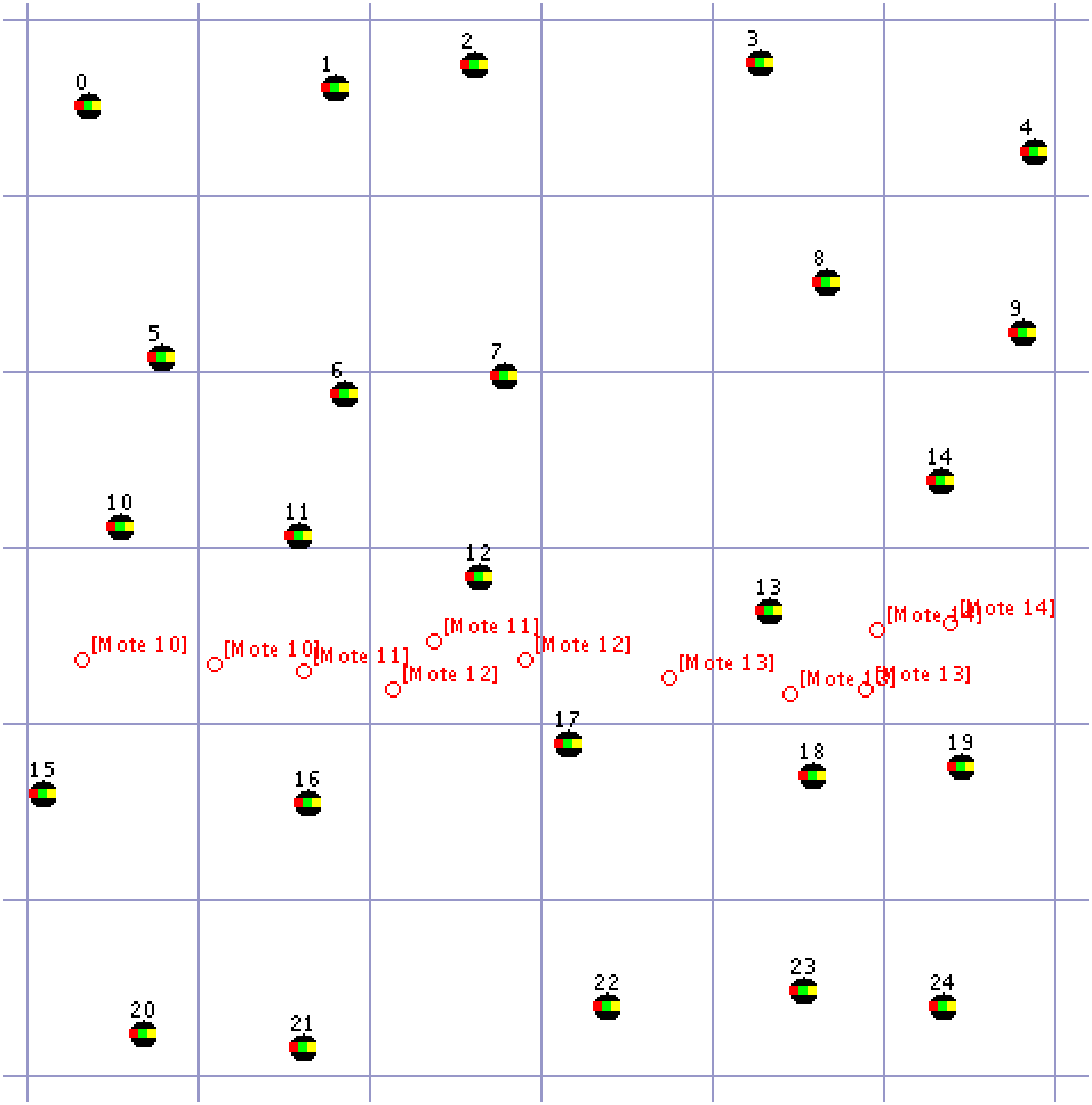
Generalized tree-based reductions

- Construct spanning tree rooted at some node
- Perform aggregation (sum, average, etc.) up the tree

Localization and time synchronization

- Much ongoing work in this area

Example application - contour finding



Matt Welsh
Harvard University

Opportunities and questions

Generating nodal programs from high-level specifications

- Holy grail of parallel languages community
- But, raw performance is (perhaps) not the goal here

Statically check properties of systems

- e.g., robustness to failure, real-time response, safety
- Tension between spec and implementation
- Solution: code in the same language

Simplify the lives of scientists and engineers

- Express programs in natural form
- Idealized structures mapped onto physical space
- Example: TinyDB - SQL interface to sensor nets

Conclusion

Distributed systems raise a host of new programming challenges

- Managing concurrency and communication in volatile environments
- Meeting complex specifications and resource requirements

Key: Use languages and compilers to help!

- Leverage static program analysis and transformation
- Present programmer with high-level view
- Regimented communication abstractions, not sockets
- Eventually - generate programs from high-level specs

Macroprogramming environments currently under development

- nesC: Component-oriented language for sensor nodes
- nesCscript: Lowering the bar for app developers
- Data-centric languages abstract behavior of nodes

For more information:

`http://www.eecs.harvard.edu/~mdw`

Backup slides follow

Parameterized interfaces

Components can provide multiple *instances* of an interface

- e.g., `SendMsg` with RPC handler ID

```
provides SendMsg[uint8_t handler];
// ...
command result_t SendMsg.send[uint8_t handler](...) { ... }
```

- Allow multiple independent wirings to component

```
MyApp.SendMsg -> RadioStack.SendMsg[MY_HANDLER_ID];
```

- Permits runtime dispatch (i.e., message reception based on RPC ID)

```
signal MyApp.ReceiveMsg[msg->handlerid]( ... );
```

Multi-client services

May only have a single instance of a component wired into the app

- Some components have multiple clients
- e.g., communication stack and timers

Currently use parameterized interfaces

```
/* State for each timer */
struct {
  uint8_t type;
  uint32_t time;
} timer_state[MAX_TIMERS]; // Constant value !

/* Manipulate per-client state */
command result_t Timer.start[uint8_t id]( ... ) {
  timer_state[id].type = ...
}
```

One solution: Abstract components

- Allow multiple instances of a component to be wired in
- Need way to name individual instances in wiring graph
- Still need to share state across instances

Evaluation

TinyOS component census

- Core TinyOS: 401 components (235 modules, 166 configurations)
- Average of 74 components per app
- Modules between 7 and 1898 lines of code (avg. 134)

Data race condition analysis on TinyOS tree

- Original code: 156 potential data races, 53 false positives
- Fixed by using `atomic` or moving code into tasks

Race condition false positives:

- Shared variable access serialized by another variable
- Pointer-swapping (no alias analysis)

nesC - future directions

Extend concurrency model to support blocking

- Prohibit blocking calls in atomic sections
- Use blocking operations as yield points for task scheduling
- Multiprocessor support and VM would require preemption

Various language enhancements

- Better support for multi-client services - *abstract components*
- Make the task scheduler another component, rather than built-in
- Allow `post` interface to be extended by application

Application to server platforms

- Support memory allocation
- Extend race detection mechanisms to handle dynamic dispatch and aliasing
- Threaded-style concurrency (with limitations)

Related work

nesC components and wiring are very different than OO languages

- Java, C++, etc have no explicit wiring or bidirectional interfaces
- Modula-2 and Ada module systems have no explicit binding of interfaces
- Module system is more similar to Mesa and Standard ML
- nesC's **static wiring** allows aggressive optimization

Lots of embedded, real-time programming languages

- Giotto, Esterel, Lustre, Signal, E-FRP
- Much more restrictive programming environment - not general-purpose languages
- VHDL, SAFL h/w description languages have similar wirings

Component architectures in operating systems

- Click, Scout, x-kernel, Flux OSKit (Knit), THINK
- Mostly based on dynamic dispatch, no whole-program optimization or bidirectional interfaces

Tools for whole-program race detection (ESC, LockLint, mcc, Eraser)

- Our approach is much simpler: restrict the language
- All of these systems (including nesC) only check single-variable data races