nesC: A component-oriented language for networked embedded systems

Matt Welsh

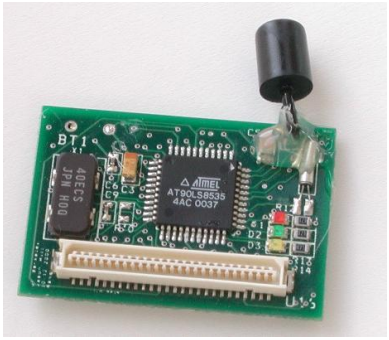
Intel Research Berkeley and Harvard University

mdw@eecs.harvard.edu

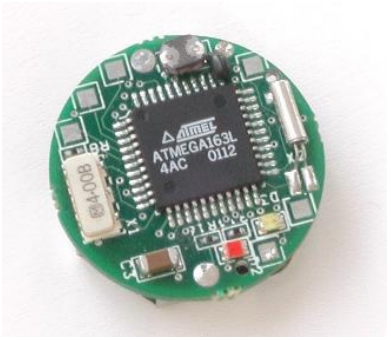
with **David Gay**, P. Levis, R. von Behren, E. Brewer, and D. Culler

Sensor networks are here!

WeC (1999)



René (2000)



DOT (2001)

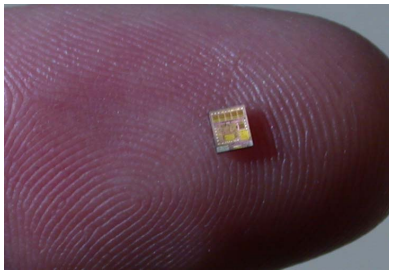
Exciting emerging domain of deeply networked systems

- 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

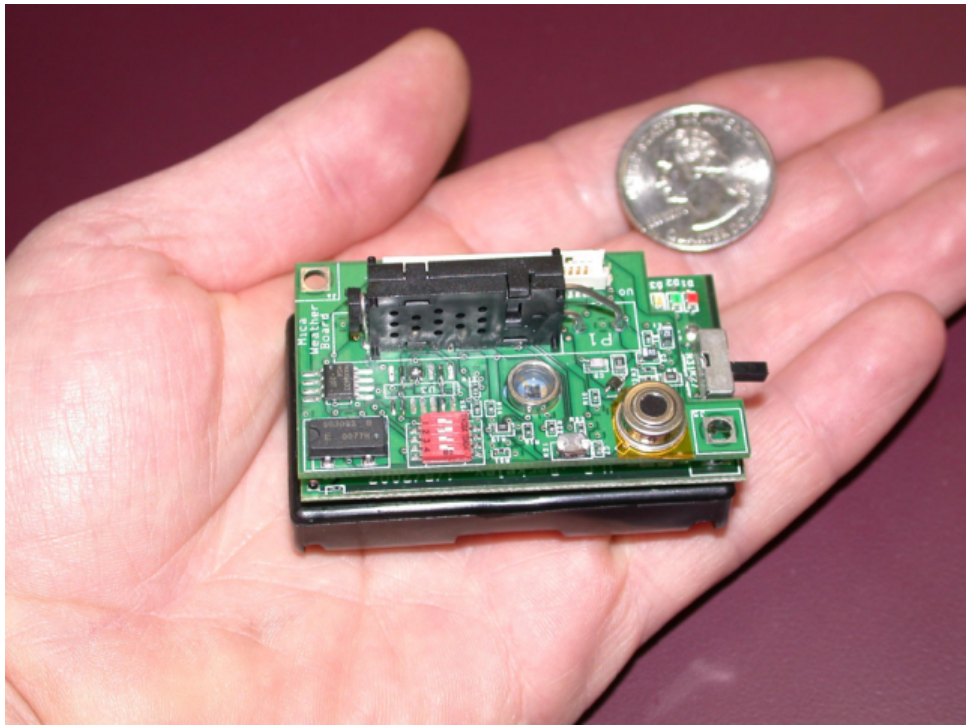
- Eventual goal - complete systems in 1 mm^3, MEMS sensors
- Family of Berkeley motes as COTS experimental platform

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)

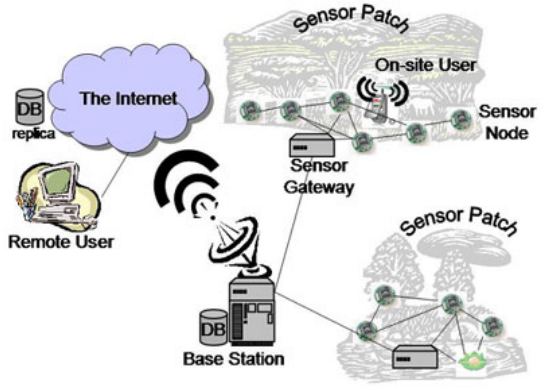
Typical applications

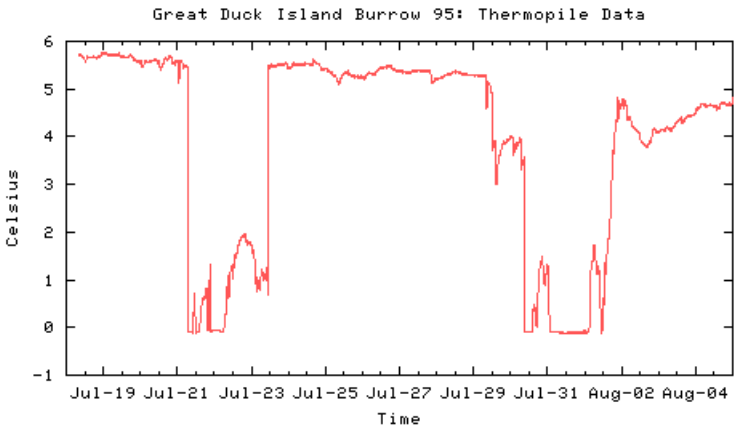
Object tracking

- 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`





Sensor network programming challenges

Driven by interaction with environment

- Data collection and control, not general purpose computation
- Reactive, event-driven programming model

Extremely limited resources

- Very low cost, size, and power consumption
- Typical embedded OSs consume hundreds of KB of memory

Reliability for long-lived applications

- Apps run for months/years without human intervention
- Reduce run time errors and complexity

Soft real-time requirements

- Few time-critical tasks (sensor acquisition and radio timing)
- Timing constraints through complete control over app and OS

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

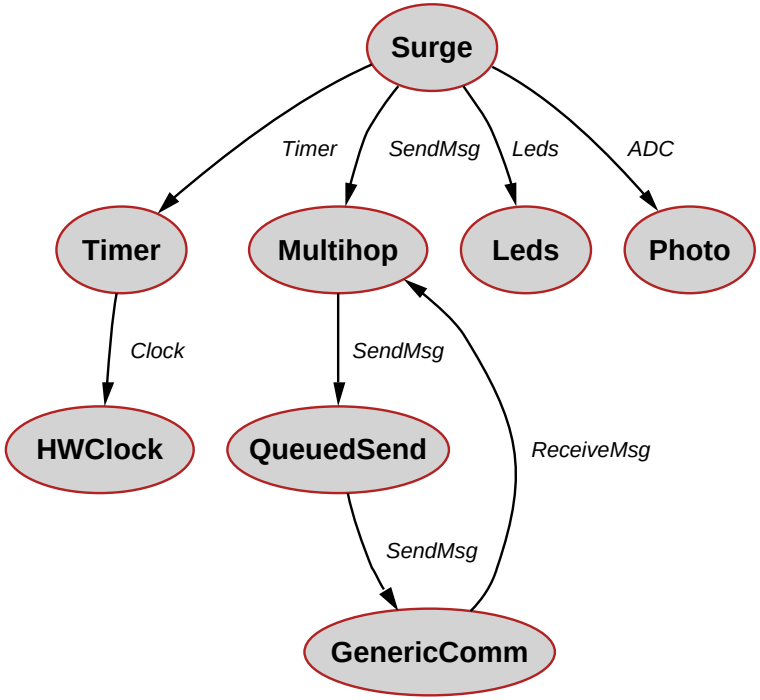
Example Application: multi-hop data collection

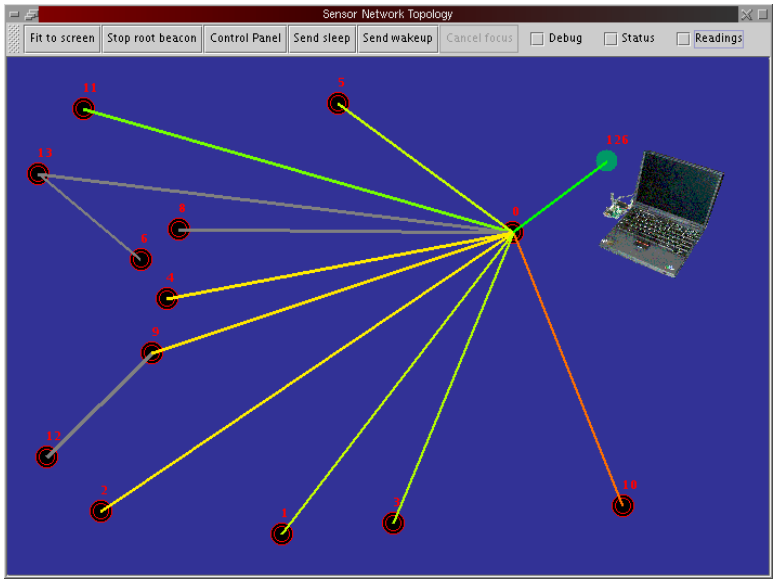
Periodically collect sensor readings and route to base

- Timer component fires event to read from ADC
- ADC completion event initiates communication

Multihop routing using adaptive spanning tree

- Nodes route messages to parent in tree
- Parent selection based on link quality estimation
- Snoop on radio messages to find better parent





nesC overview

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

- nesC compiles entire application into a single C file
- Compiled to mote binary by back-end C compiler (e.g., gcc)
- Allows aggressive cross-component inlining
- Static race condition detection

Important restrictions

- No function pointers (makes whole-program analysis difficult)
- No dynamic memory allocation
- No dynamic component instantiation/destruction
  - ▷ *These static requirements enable analysis and optimization*

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

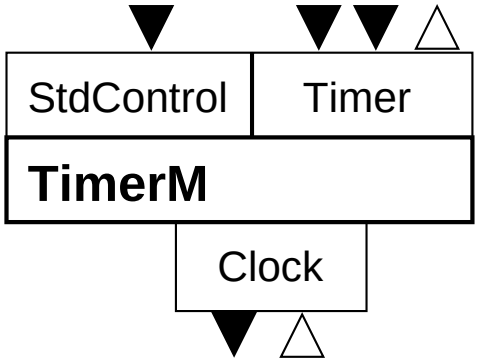
```
interface Timer {
  command result_t start(char type, uint32_t interval);
  command result_t stop();
  event result_t fired();
}

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

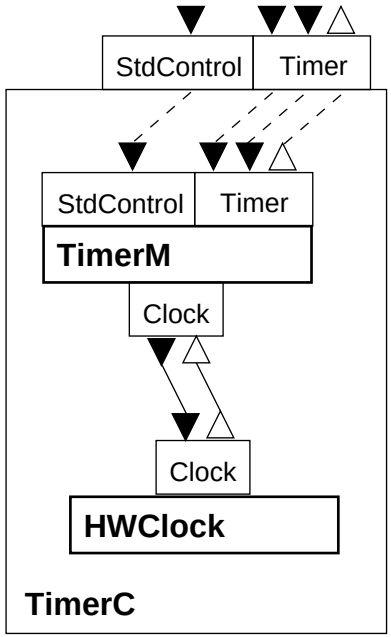
- **Modules** contain implementation code
- **Configurations** wire other components together
- An application is defined with a single top-level configuration

```
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;
}
```

Concurrency model

Tasks used as deferred computation mechanism

```
// Signaled by interrupt handler
event void Receive.receiveMsg(TOS_Msg *msg) {
  if (recv_task_busy) {
    return; // Drop!
  }
  recv_task_busy = TRUE;
  curmsg = msg;
  post recv_task();
}

task void recv_task() {
  // Process curmsg ...
  recv_task_busy = FALSE;
}
```

- Commands and events cannot block
- Tasks run to completion, scheduled non-preemptively
- Scheduler may be FIFO, EDF, etc.

Race condition detection

All code is classified as one of two types:

- **Asynchronous code (AC):** Code reachable from at least one interrupt handler
- **Synchronous code (SC):** Code reachable only from tasks

Any update to shared state from AC is a potential race condition

- SC is atomic with respect to other SC (no preemption)
- Race conditions are shared variables between SC and AC, and AC and AC
- Compiler detects race conditions by walking call graph from interrupt handlers

Two ways to fix a race condition

- Move shared variable access into tasks
- Use an *atomic section:*

```
atomic {
  sharedvar = sharedvar+1;
}
```

- Short, run-to-completion atomic blocks
- Currently implemented by disabling interrupts

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]( ... );
```

Also used for multi-client services (e.g., `TimerM`)

- Single `TimerM` component shared by all clients
- Each client wires to a unique interface instance
- Internal state of each "instance" partitioned by hand
- This really needs some work...

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)

Race condition analysis on TinyOS tree

- Original code: 156 potential 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)

Inlining and dead code elimination

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

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 races

Future work

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)

Conclusion

Sensor networks raise a number of programming challenges

- Much more restrictive than typical embedded systems
- Reactive, concurrent programming model

nesC language features

- “Components for free” – compile away overhead
- Bidirectional interfaces
- Lightweight, event-driven concurrency
- Static race condition detection

Efficient whole-program optimization

- Code size savings of 8% – 40% for a range of apps
- Significant reduction in boundary crossing overhead
- Inlining allows cross-module optimizations

Code available for download at:

`http://www.tinyos.net`

`http://nescc.sourceforge.net`