Programming Primitives for Wireless Sensor Networks

Matt Welsh

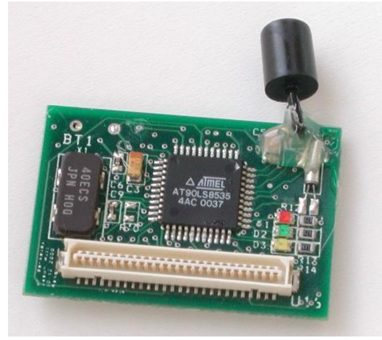
Harvard University

Division of Engineering and Applied Sciences

mdw@eecs.harvard.edu

Sensor networks are here!

WeC (1999)



René (2000)

DOT (2001)

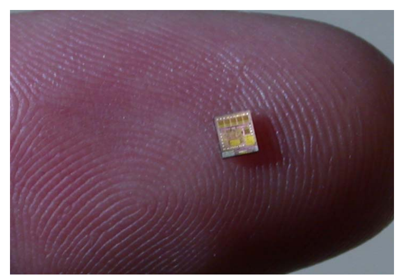
Exciting emerging domain of deeply networked systems

- 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

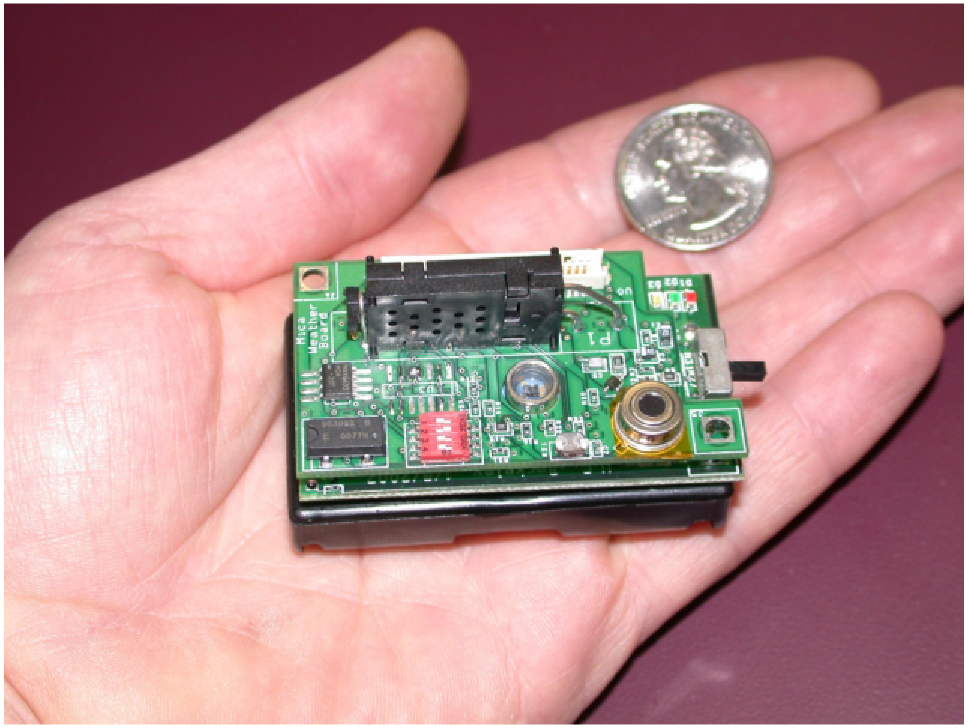
- Eventual goal - complete systems in 1 mm^3, MEMS sensors
- Family of Berkeley motes as COTS experimental platform

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

Thousands produced, used by over 150 research groups

- Get yours at `www.xbow.com` (or `www.ebay.com`)
- About $150 a pop

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)

Outline

- Sensor network applications and challenges
- nesC: A component-oriented dialect of C for embedded systems
- nesC concurrency model and static race detection
- Macroprogramming: High-level programming for entire sensor nets
- Abstract regions: A communication primitive for macroprogramming
- Application examples and evaluation
- Conclusion

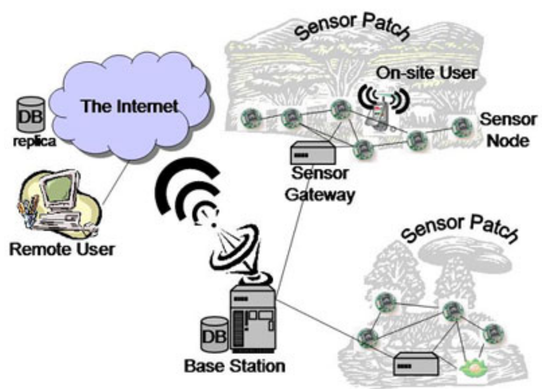
Typical applications

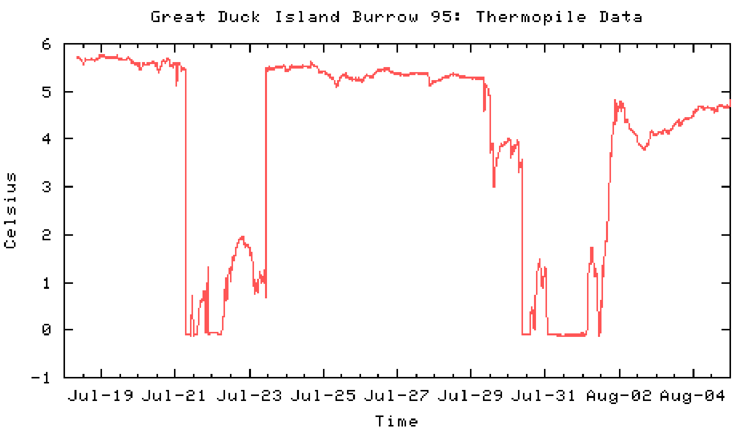
Object tracking

- 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`

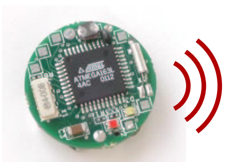


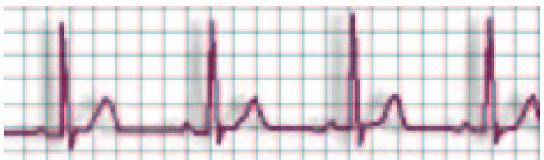


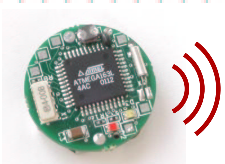
Vital Dust: Emergency Medical Triage

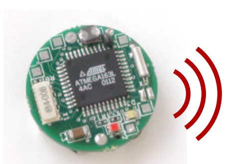
with S. Moulton, M.D., Boston Medical Center and
M. Gaynor, Boston University

Motes attached to patients
collect vital signs (pulse ox, heart rate, etc.)









PDAs carried by EMTs
receive vital signs and enter
into field report

Ambulance system makes
triage decisions, relays to EMTs

Correlate with patient records
at hospital

- 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

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

nesC: A Programming Language for Sensor Networks

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

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 data-race 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

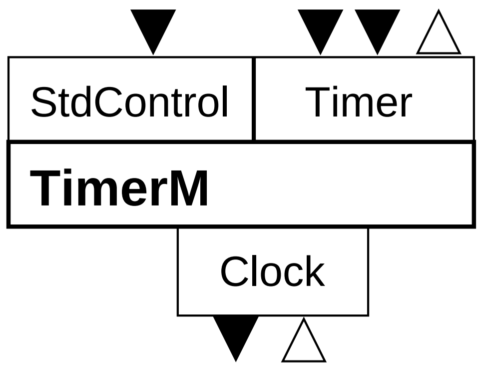
```
interface Timer {
  command result_t start(char type, uint32_t interval);
  command result_t stop();
  event result_t fired();
}

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

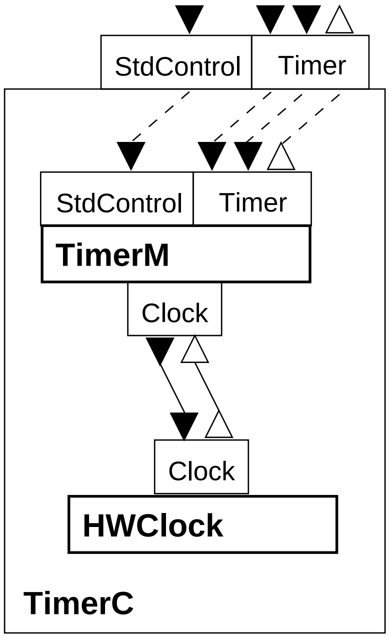
- **Modules** contain implementation code
- **Configurations** wire other components together
- An application is defined with a single top-level configuration

```
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 data race

- 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 data races by walking call graph from interrupt handlers

Two ways to fix a data race

- 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

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

Macroprogramming

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

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

Developing a high-level language to express **aggregate programs** across an entire field of motes

- 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
- Evidence: Huge engineering effort for each demo

Macroprogramming goals

Develop a set of communication and coordination primitives

- Goals somewhat akin to MPI
- Abstract the details of underlying communication
- Provide enough structure to permit optimizations

Expose these primitives in a global, high-level language

- Simple syntactic structures for performing spatiotemporal aggregation
- Automatically compile down to low-level behavior of individual nodes
- Compiler-directed optimization for energy and bandwidth usage

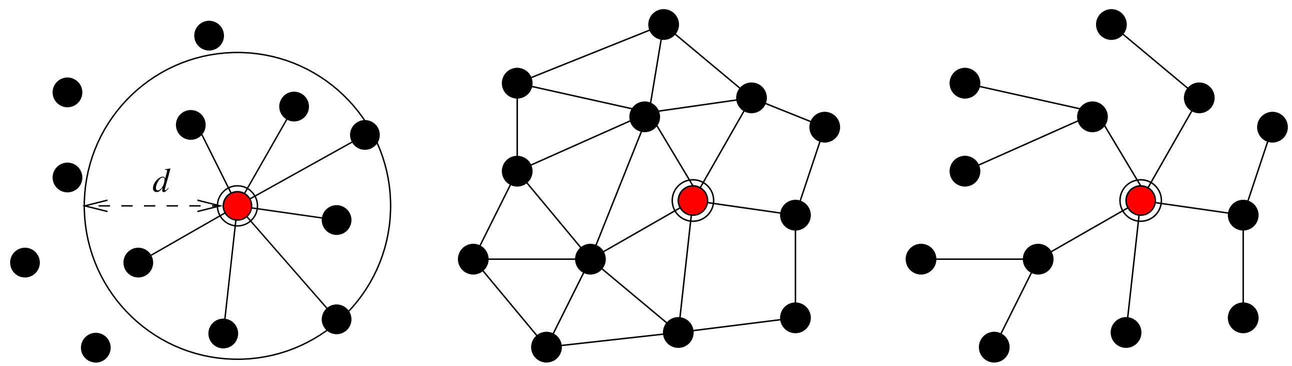
Expose the tradeoff between resource usage and accuracy

- Support “lossy programming”
- Programmer can tune resource usage of the communication layer
- Each collective operation reports its **yield**

Abstract Regions: A Macroprogramming Primitive

A *region* is a group of nodes with some geographic or

- e.g., All nodes within N radio hops from node k
- All nodes within distance d from node k
- All nodes in a spanning tree rooted at node k



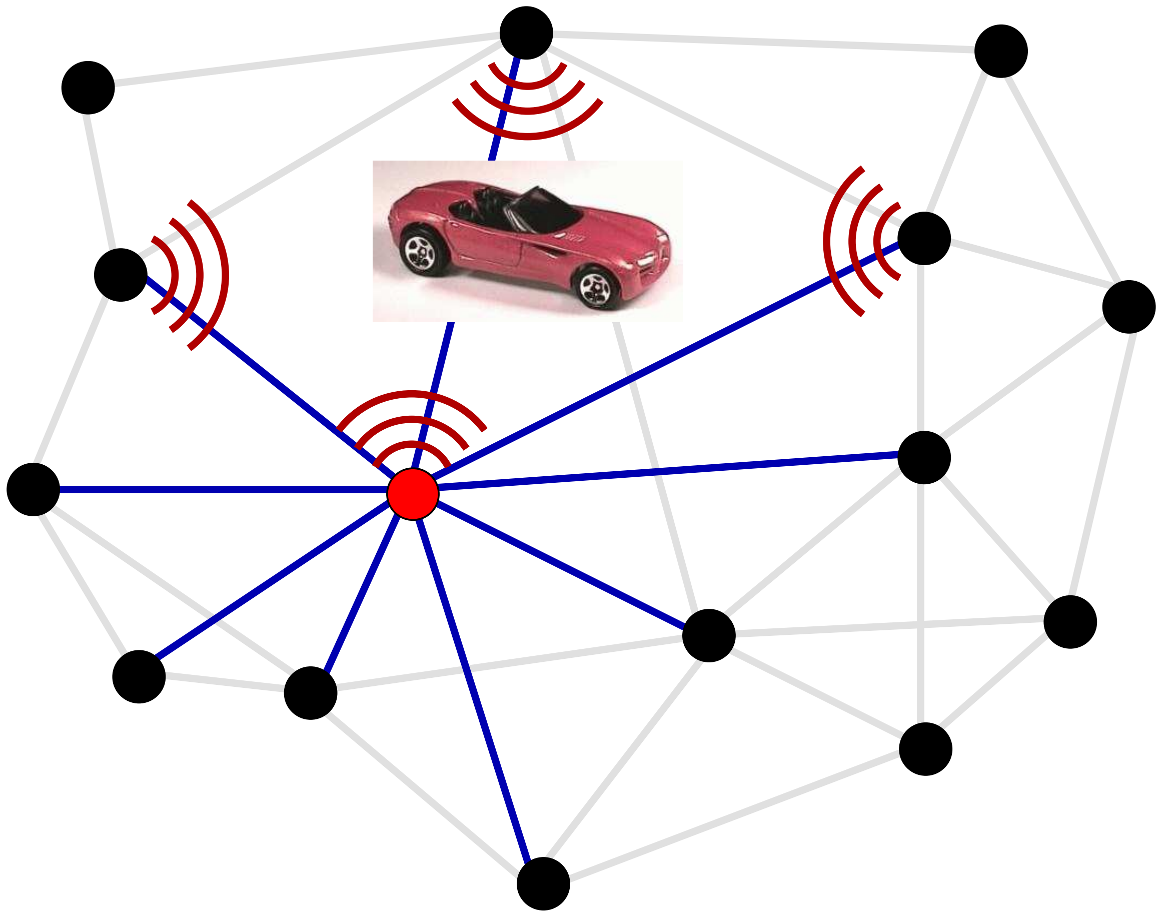
Shared variables support coordination

- Tuple-space like programming model within regions
- Implementation may broadcast, pull requested data, or gossip

Reductions support aggregation of shared variables

- Combine shared variables in region to a single value

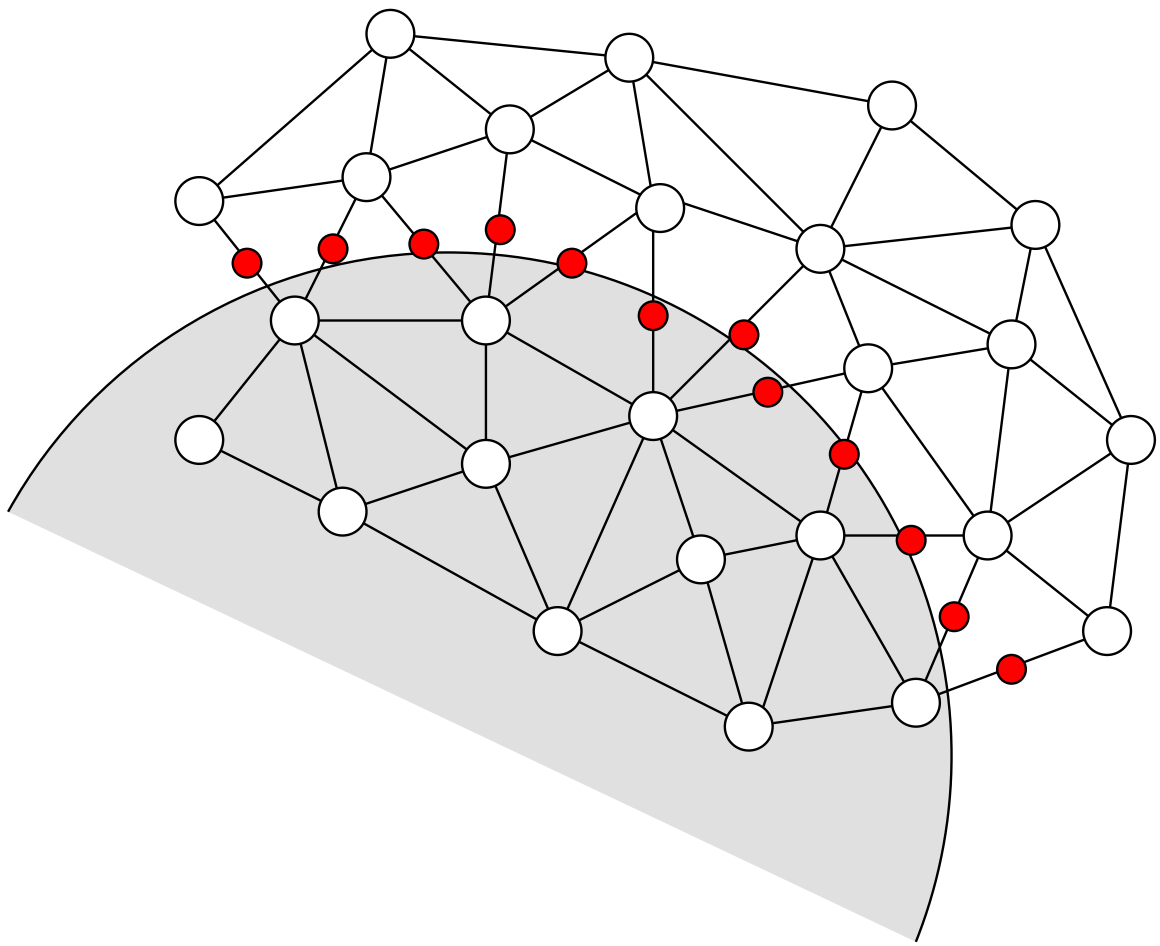
Object tracking using regions



Localize vehicle in a sensor field using magentometer readings

- Nodes store local sensor reading as shared variable
- Reduction used to determine node with the largest value
- Max node performs sum-reductions to determine centroid of sensor readings

Contour finding



Determine location of threshold between sensor readings

- Construct approximate planar mesh of nodes
- Nodes above threshold compare values with neighbors
- Contour defined as midpoints of edges crossing threshold

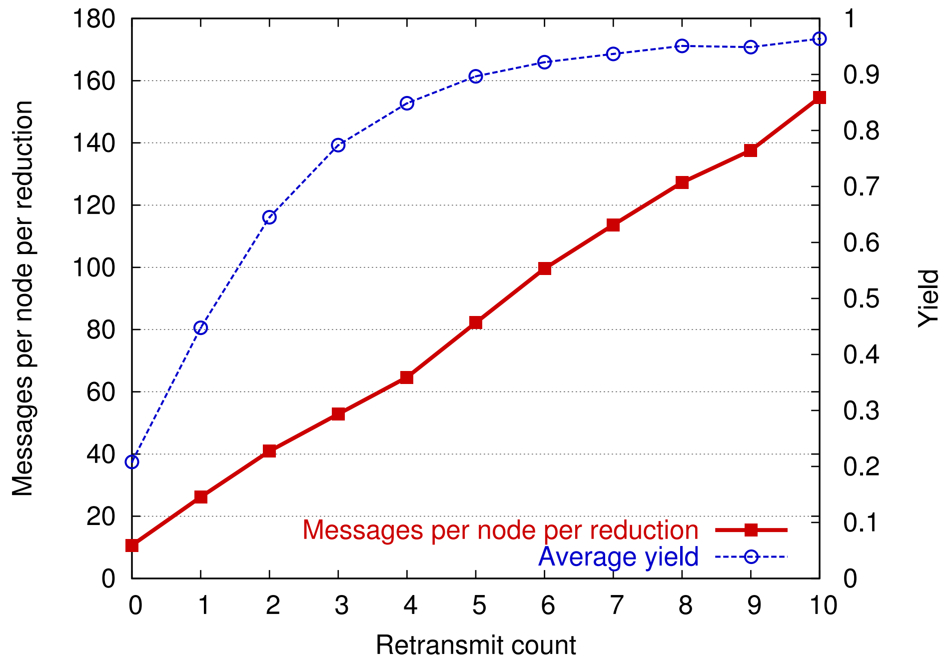
Quality feedback and tuning

Region operations are inherently statistical

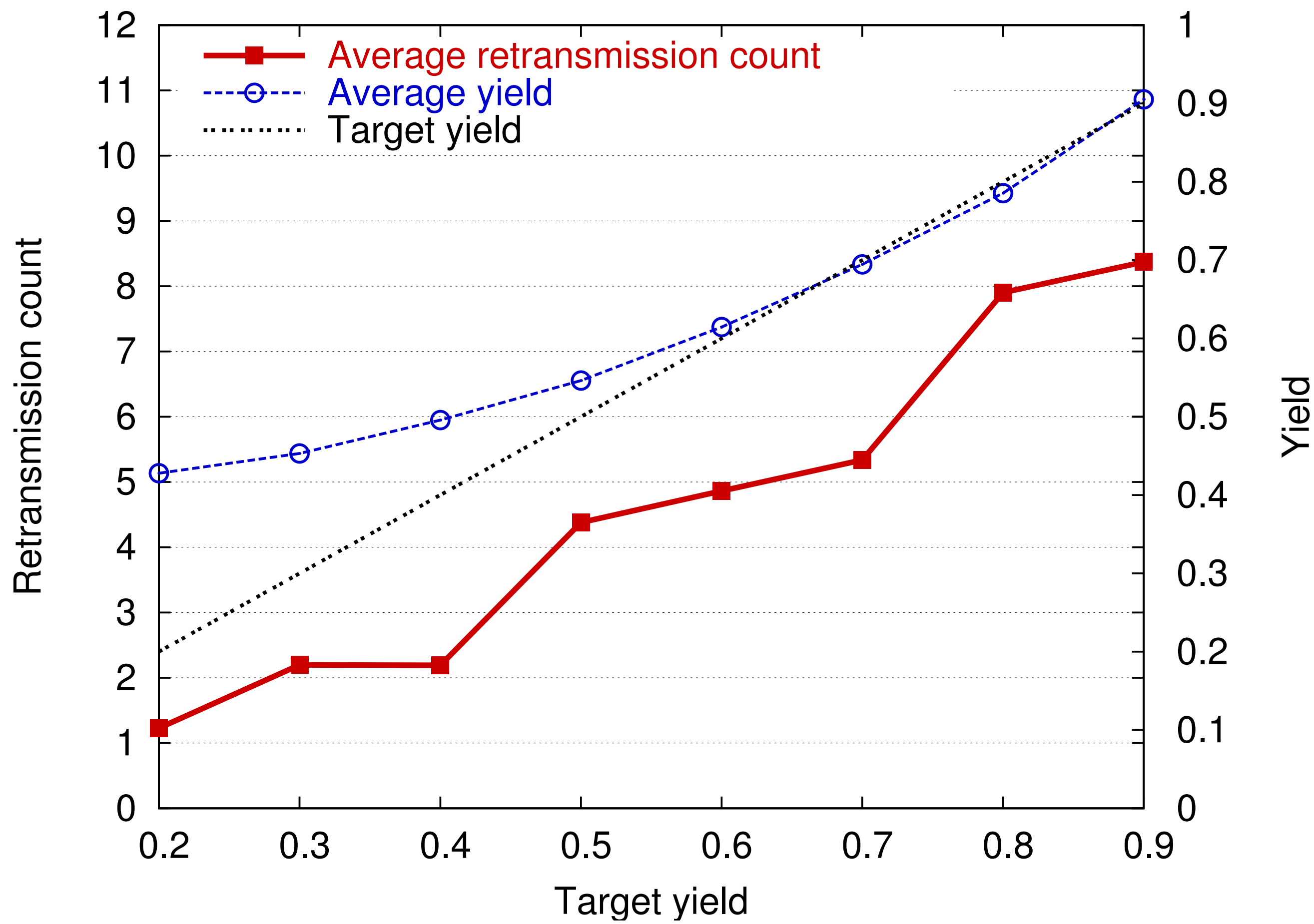
- Shared variable operations may timeout
- Reductions may contact subset of nodes
- Collective operations report *yield*

Regions provide control over overhead-accuracy tradeoff

- Programmer can tune retransmission count, timeouts, etc.
- Quality feedback can be used to drive adaptation to changing network conditions



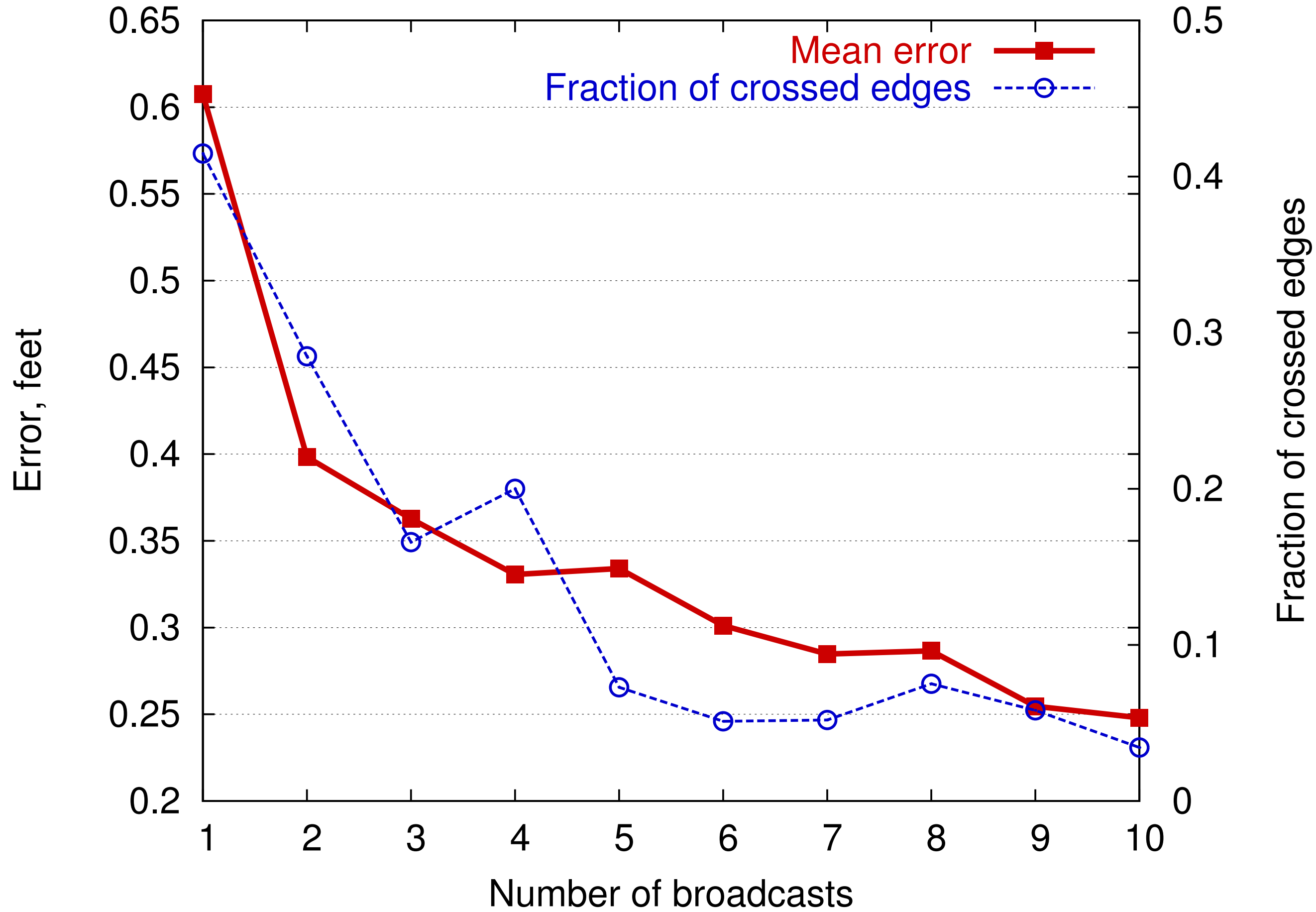
Adaptive Reduction



Tune message retransmission count to meet yield target

- Take EWMA of yield, adjust retransmission count using simple AIAD controller
- Very accurate for yield targets above 0.5
- For low targets, low retransmission count often achieves better than desired results

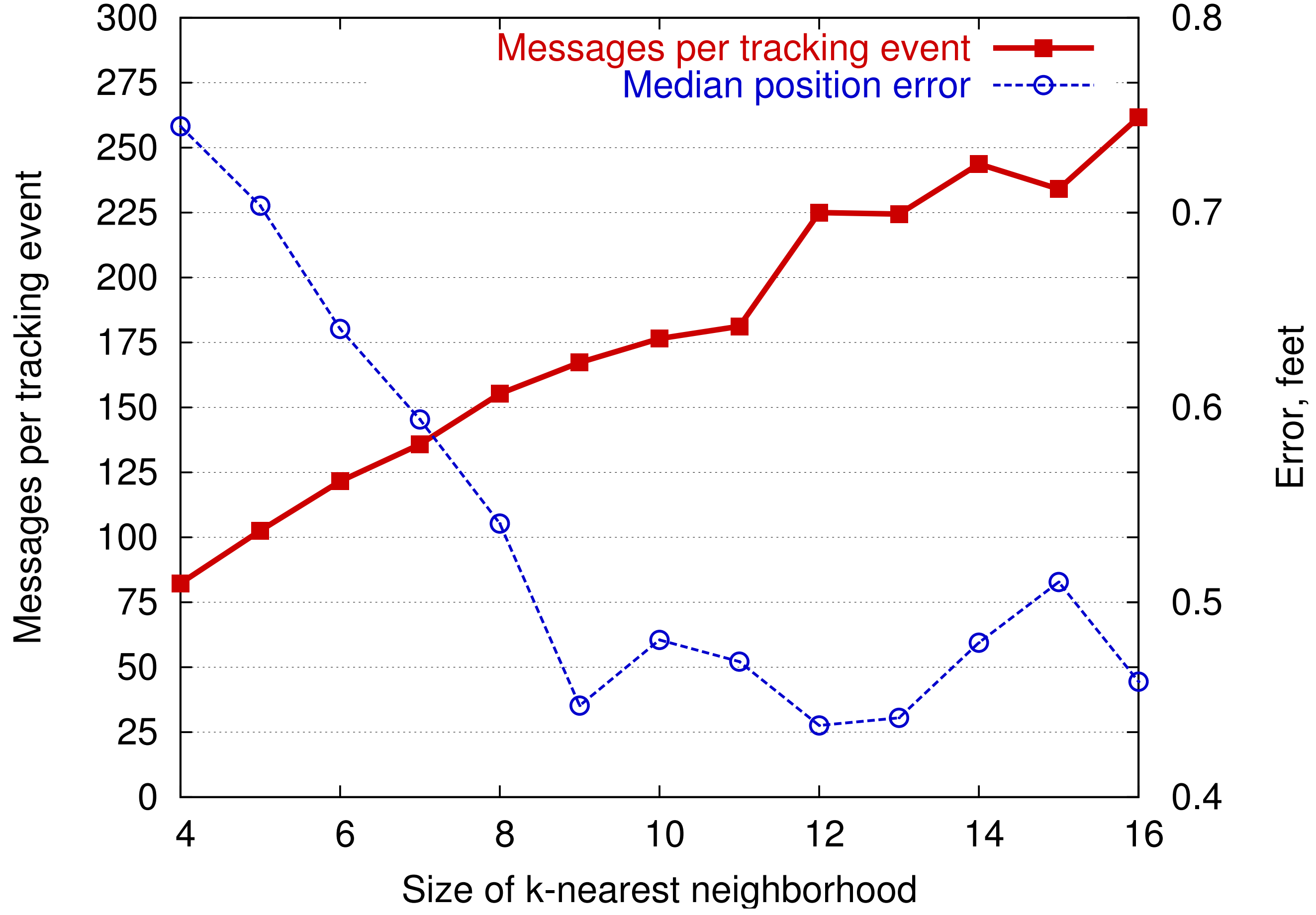
Contour detection accuracy and overhead



Contour finding accuracy a function of node advertisements

- Form approximate planar mesh region
- More advertisements $\longrightarrow$ fewer crossed edges
- Mean error directly correllated with mesh quality

Object tracking accuracy and overhead



- Object moving in circular path through sensor net
- Size of neighborhood increases accuracy and message overhead

Future Work

Ongoing work on abstract regions

- Generic region constructors
  - *Supply membership primtive and communication strategy*
- Feedback control for tuning resource usage for desired loss level or energy budget
- Exploirng language design based on region primitive

Spatial operations 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 $(30.5, 42.6)$”
- Translates into interpolation across nearby sensor values

Temporal operations and aggregation

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

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 data race detection

Macroprogramming and abstract regions

- Steps towards a global programming model for sensor nets
- Abstract details of low-level node coordination
- Provide control over resource/accuracy tradeoff

Code available for download at:

`http://www.tinyos.net`

`http://nescc.sourceforge.net`