Programming Sensor Networks Using Abstract Regions

Matt Welsh and Geoff Mainland

Harvard University

Division of Engineering and Applied Sciences

{mdw,mainland}@eecs.harvard.edu

Sensor Net Programming Challenges

Developing sensor network applications is notoriously difficult

- Bandwidth and energy limitations force in-network processing
- Infeasible to send all data to central location
- Requires complex distributed algorithms to be implemented in the network itself

Often requires coordination within **local regions** of the network

- Coordinated detection of localized phenomena
- Aggregation of sensor readings for bandwidth reduction

Examples of spatial coordination:

- Finding average or max sensor reading amongst a group of nodes
- Propagating data up a spanning tree to a base station
- Comparing sensor values to nearby neighbors

Sensor network programming is a nascent area of research

- Not much work on general-purpose programming models in this environment

Accuracy/Overhead Tradeoff

Our focus is on extremely resource-constrained devices

- MICA2 “mote:” 7.3 MHz CPU, 4 KB RAM, 128 KB ROM, 38.4 Kbps radio
- Powered by 2 AA batteries
- TinyOS: Event-driven OS for mote-class devices

Inherent tradeoff between resource consumption and accuracy

- More messages → increased energy and bandwidth → greater precision

But, sensor nodes have limited energy budget

- Cannot consume arbitrary energy to achieve reliable communication

Apps must deal with lossy communication, imperfect results

- Limited energy and bandwidth budget mandates statistical design

Macroprogramming Goals

Develop an **aggregate programming model** for sensor networks

- Current programming models are node centric and low level
- Scientists don't want to think about gronky details of radios, timers, battery life, etc.
- Like implementing Linux by toggling switches on a PDP-11

Requires flexible communication primitives

- Reduce programming effort to construct applications
- Abstract low-level details of local coordination
- Focus on spatial computation within local neighborhoods
- Neighborhood maintenance, routing, and collective communication

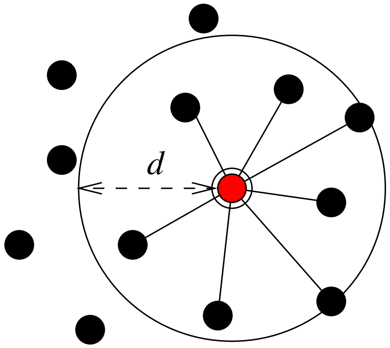
Allow application to tune resource/accuracy tradeoff

- Application must have control over resource usage
- Don't hide settings of complex parameters inside lower layer
- Provide feedback to applications:
  - ▷ *Timeouts on communication operations*
  - ▷ *Accuracy and completeness of collective operations*
- Feedback used to adapt to changing network conditions

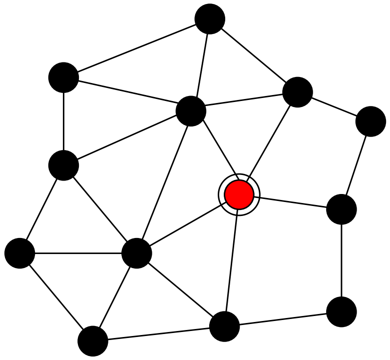
Abstract Regions

Group of nodes with some geographic or topological relationship

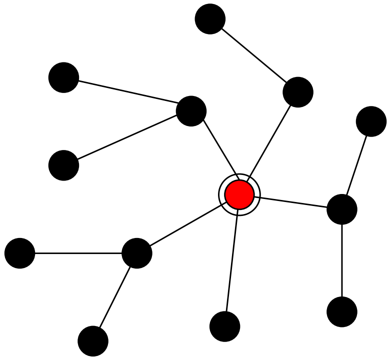
- e.g., All nodes within distance d from node k
- Neighbors forming a planar mesh based on radio connectivity
- Spanning tree rooted at node k



Geographic neighborhood



Planar mesh



Spanning tree

Regions capture common idioms in sensor net programming

- Flexible addressing of “local” nodes
- Sharing state across groups of nodes
- Efficient aggregation of data across a region

Region Operations

Neighbor discovery identifies nodes

- Continuous background process, can be terminated or restarted
- Each node is notified of changes to region membership
- e.g., Nodes moving, joining, or leaving network

Shared variables support inter-node coordination

- Tuple-space like programming model:
- *get(k,n)* retrieves value of v from node n
- *put(v,l)* stores value l in variable v locally
- Implementation may broadcast, pull requested data, or gossip

Reductions support aggregation of shared variables

- Combine shared variables in region to a single value
- *reduce(op,v,d)* reduces variable v using operator op and stores in shared variable d
- Example operators: min, max, average, count, etc.

Radio and Geographic Neighborhoods

Nodes within n radio hops, k-nearest neighbors, etc.

Node discovery implementation

- Nodes emit periodic beacons with node ID and (optionally) location
- Filter received beacons to determine neighbors (e.g., k nearest nodes)
- Application can tune rate and number of beacons

Shared variable implementation

- *put()* operation stores value in local hashtable
- Fixed number of keys can be stored per node
- *get()* operation sends a fetch message to corresponding node
- Alternate implementation: *put()* broadcasts, while *get()* is local

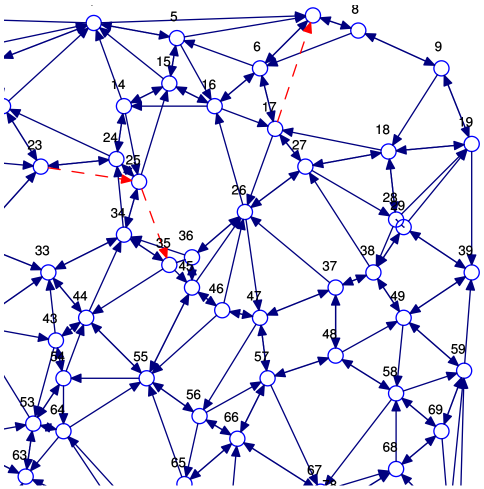
Reduction implementation

- Broadcast *get()* request for all values of shared variable
- Collect replies and perform reduction after all responses received
- Application can specify *timeout* for shared variable and reduction operations

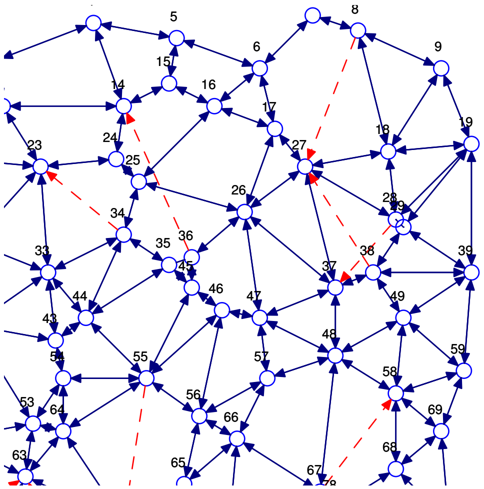
Approximate Planar Mesh

Useful construct for spatial computing

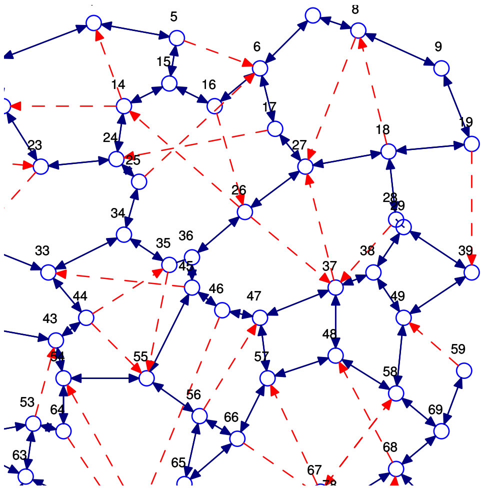
- Divide space into nonoverlapping cells
- Also used for geographic routing (e.g., GPSR): send message to node closest to given geographic location
- Different planarity tests yield different graphs:



Pruned Yao



Gabriel



RNG

True planarity is difficult to achieve

- Requires information on location and edges from all nearby neighbors
- Rather, strive for approximate planarity: allow some crossed edges
- Number of crossed edges depends on accuracy of neighborhood determination

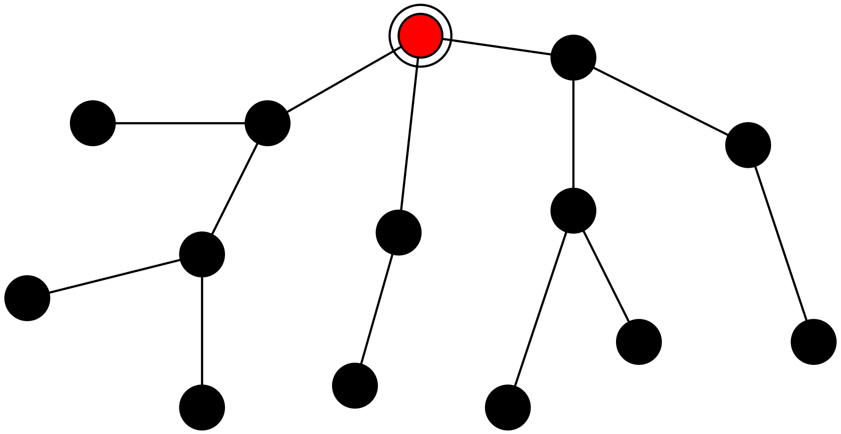
Adaptive Spanning Tree

Useful for aggregating data to a single point in the network

- Nodes continually evaluate link quality to neighbors and select ideal *parent node*
- Responds rapidly to changes in network topology
- Demonstrates **layering**: Spanning tree implemented on top of radio neighborhood

Shared variable and reduction semantics:

- *put()* at the root floods data to all nodes in tree
- *get()* at root fetches data from specific child node
- Reductions always store resulting value at the root



Quality feedback and tuning

Region operations are inherently statistical

- Reduction may time out or contact subset of nodes
- Collective operations report **yield**: fraction of nodes that responded to a request
- Each operation also provides a **timeout**

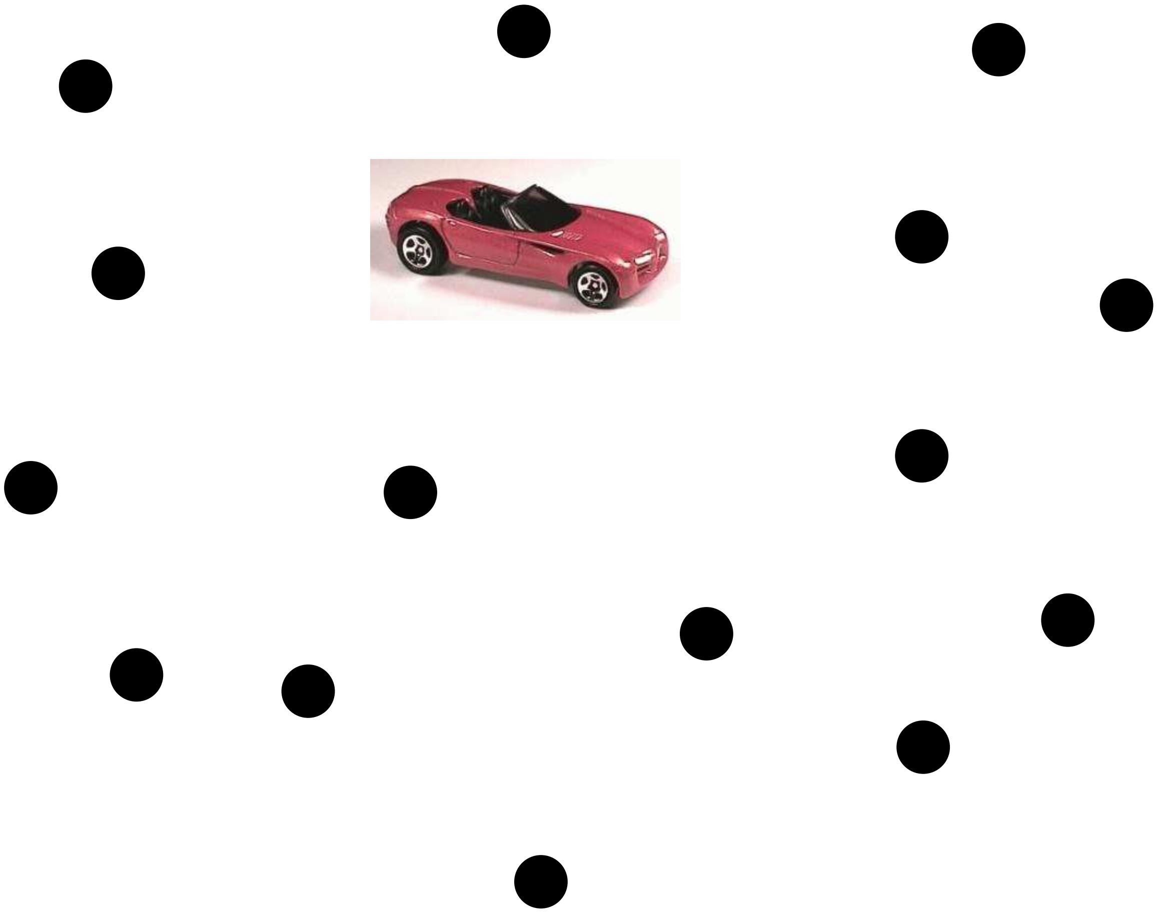
Programmer can tune many parameters affecting resource usage:

- Maximum number of retransmission attempts
- Delay between retransmissions
- Maximum number of neighbors to consider in region formation
- Frequency of beacons for radio region formation
- Number of beacons to send during region formation
- Threshold used to remove neighbor from set
- Timeout for various region operations

Tuning parameters support **adaptive applications**

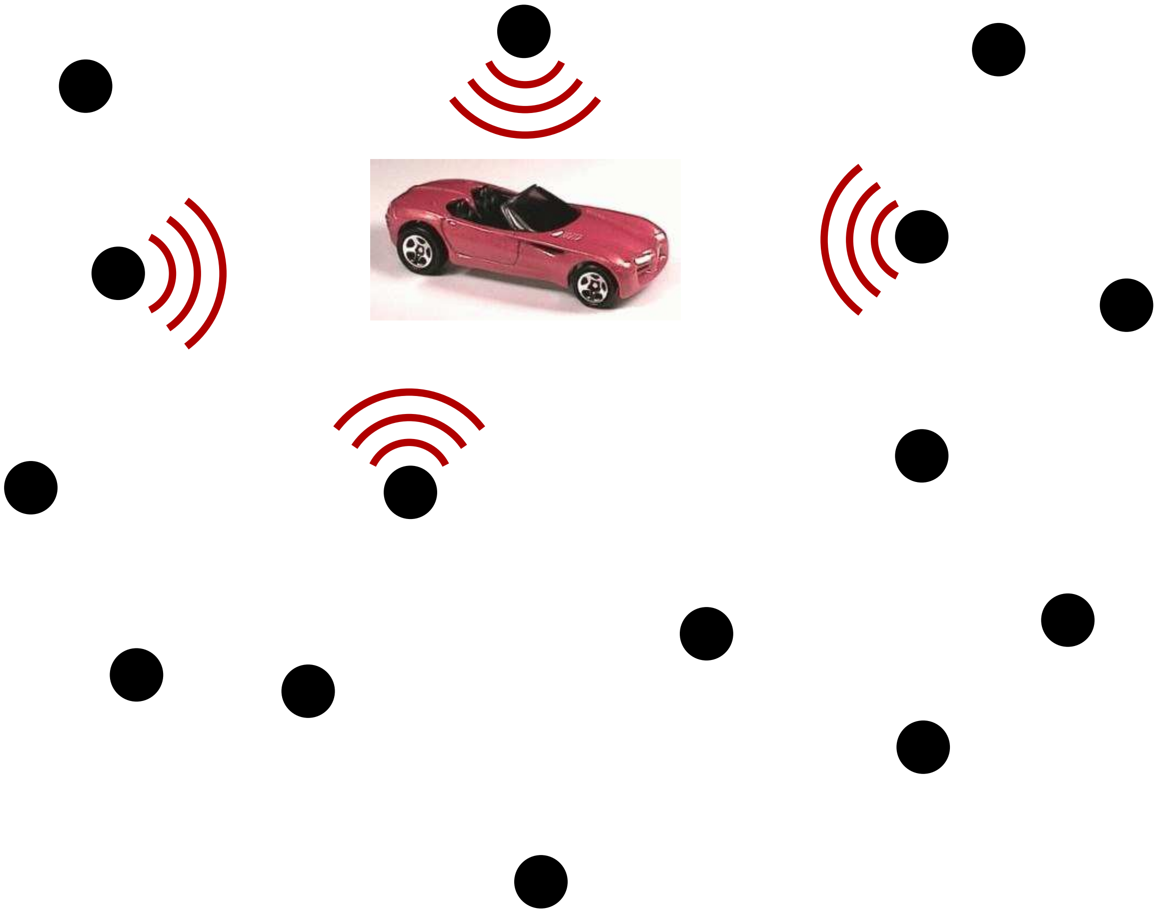
- Applications can turn knobs to meet targets of latency, accuracy, or lifetime

Object tracking using regions



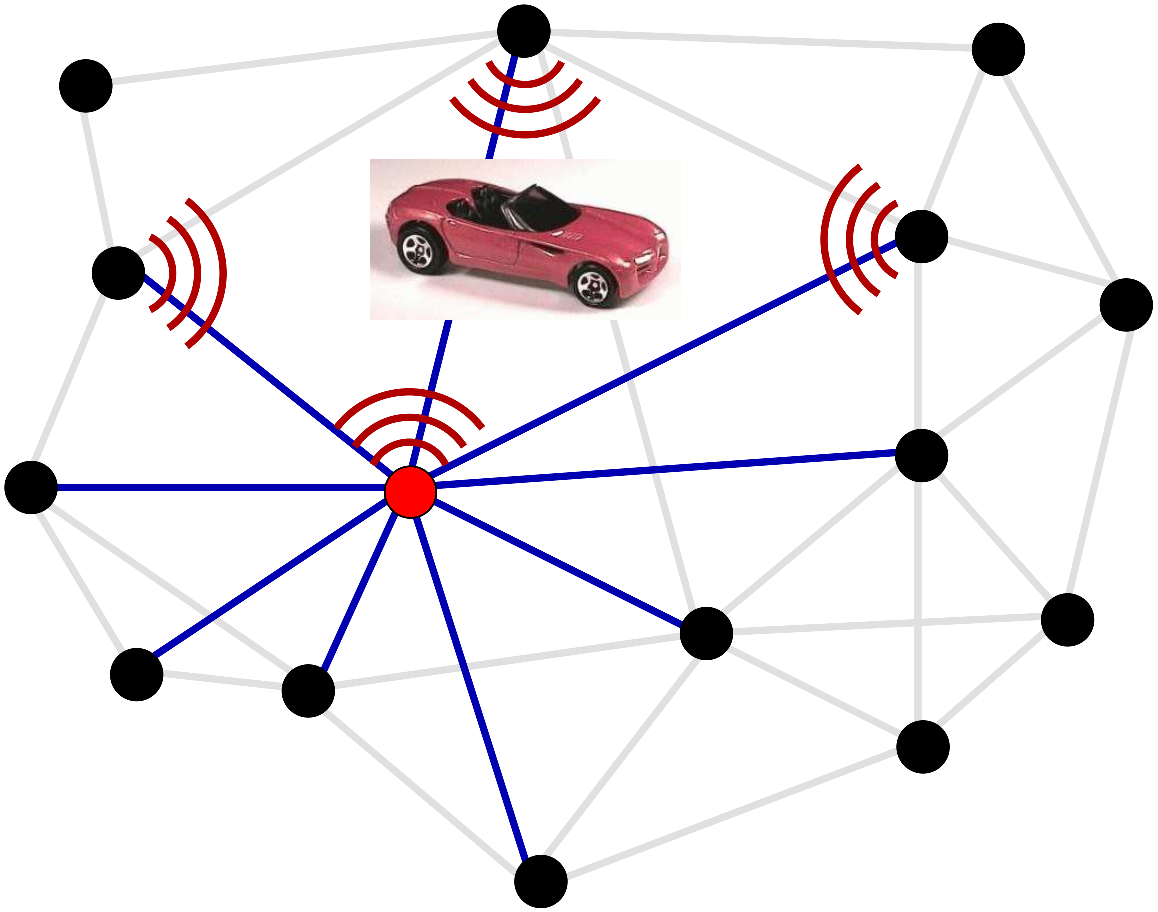
Track location of vehicle using magentometers attached to sensors

Object tracking using regions



Nodes near the vehicle detect high magnetometer value

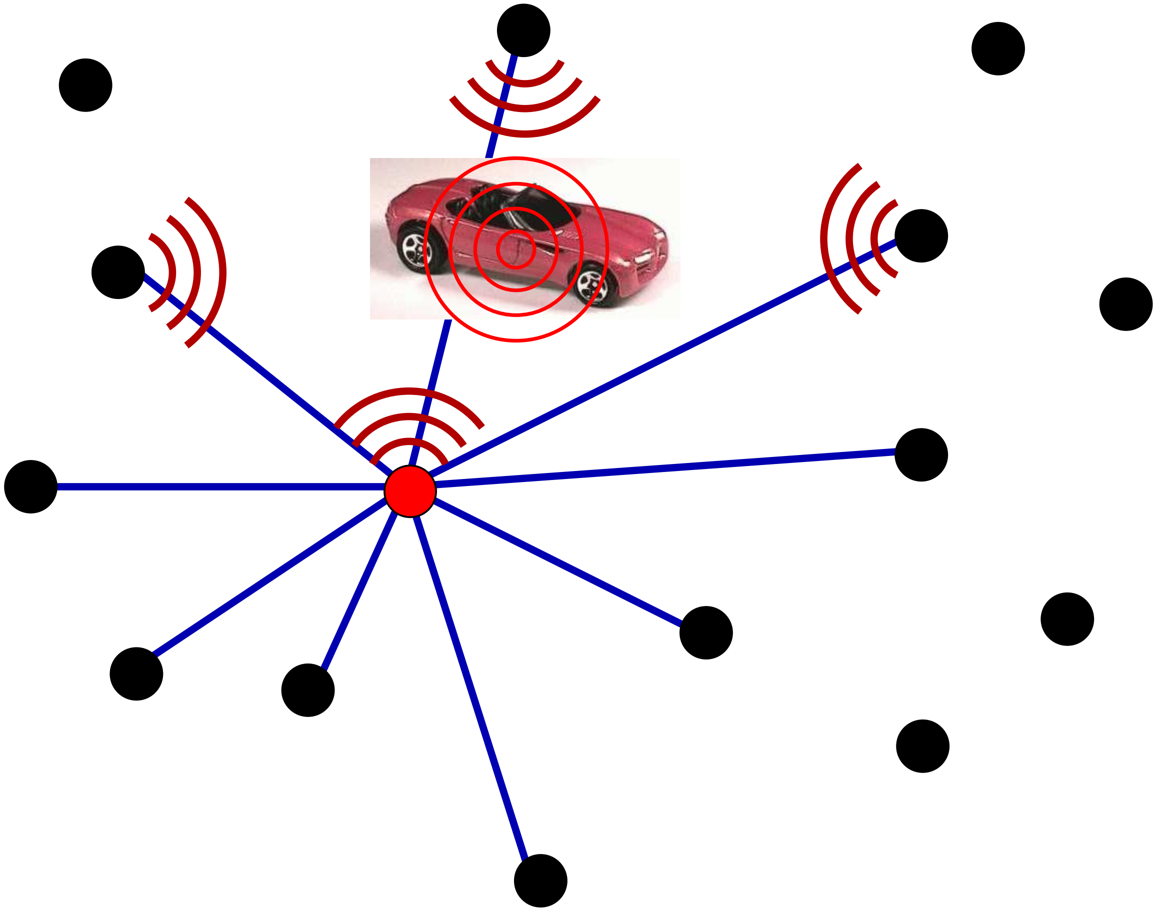
Object tracking using regions



Each node forms k-nearest-neighbor region

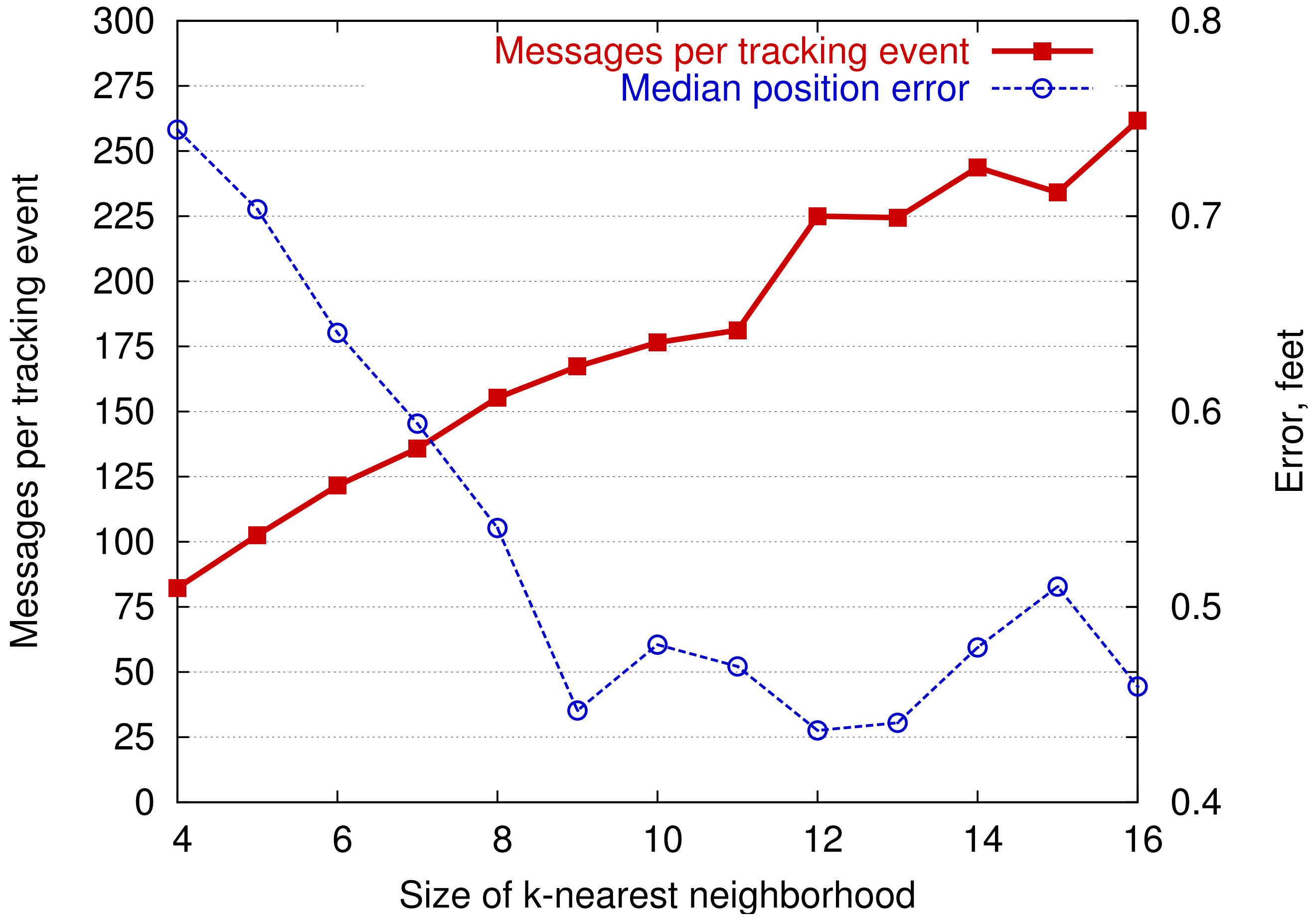
Store local sensor reading as shared variable

Object tracking using regions



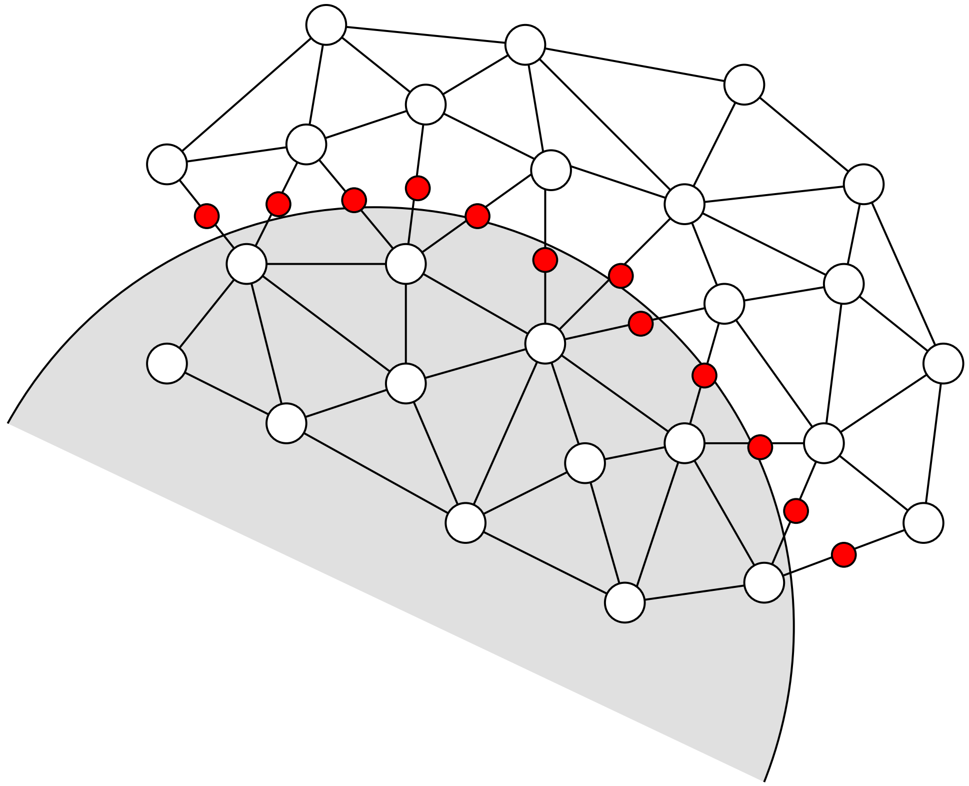
Node with highest sensor value calculates centroid of neighbor's readings

Object tracking accuracy and overhead



- TOSSIM sensor network simulator with realistic radio model
- Object moving in circular path through sensor net
- Tuning knob: Number of neighbors in k-nearest neighbor region
- Size of neighborhood increases both accuracy and message overhead

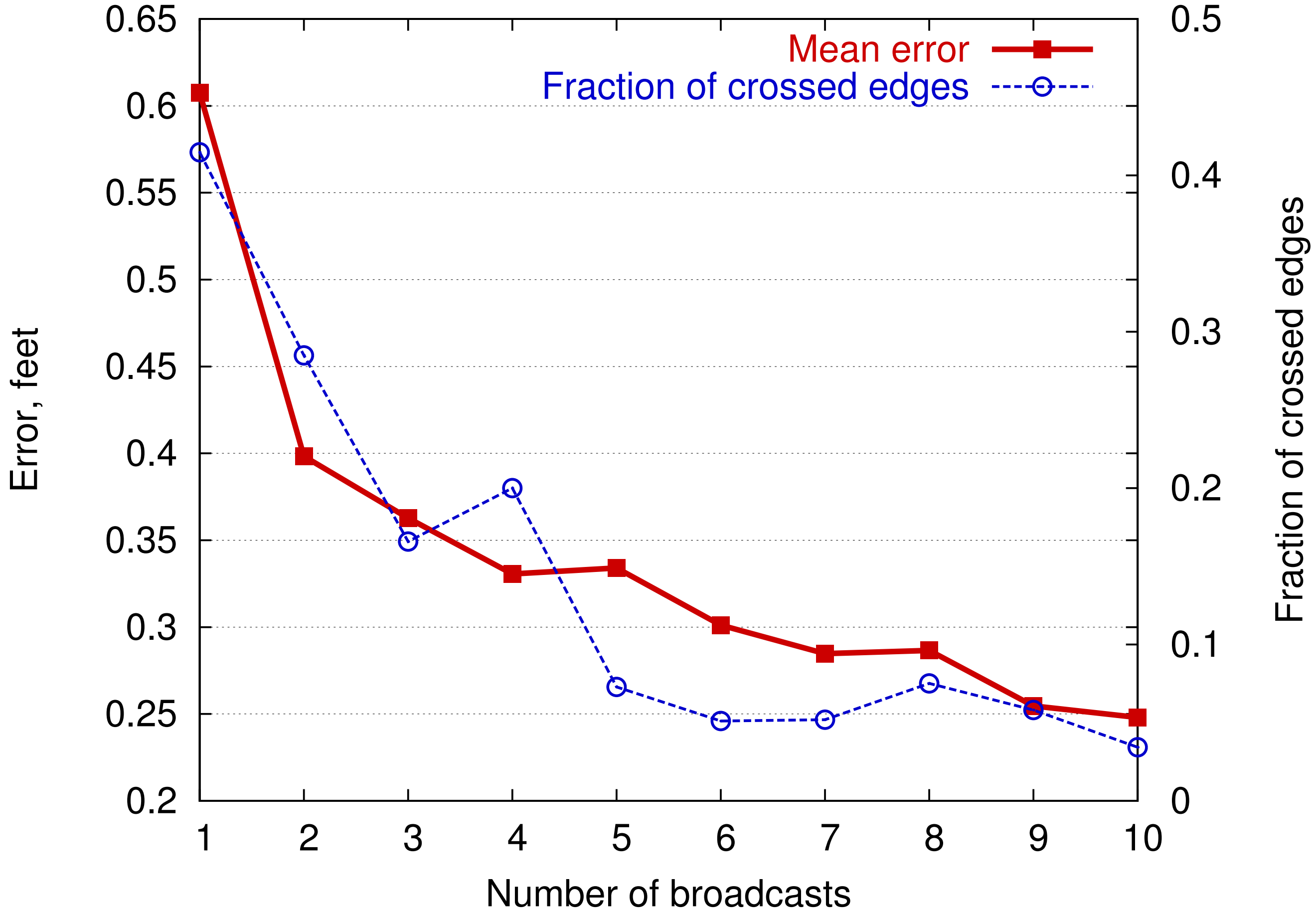
Contour finding



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

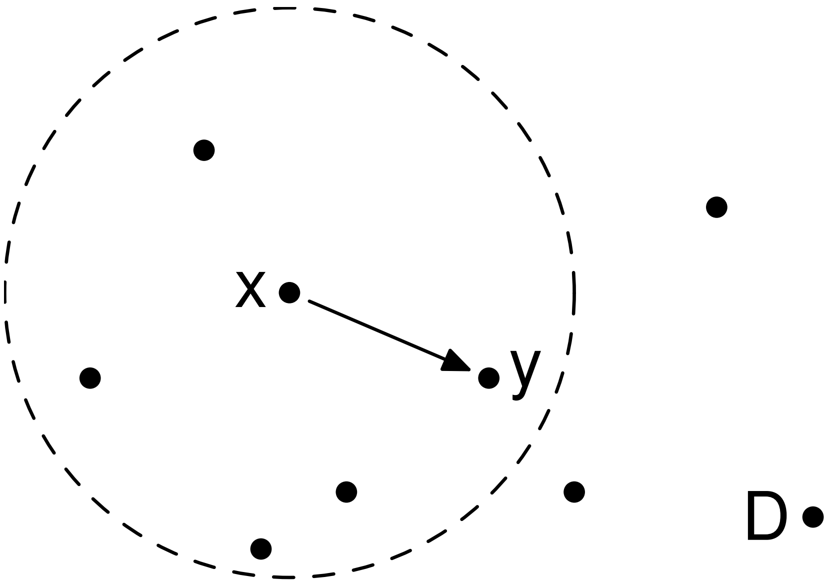
Contour detection accuracy and overhead



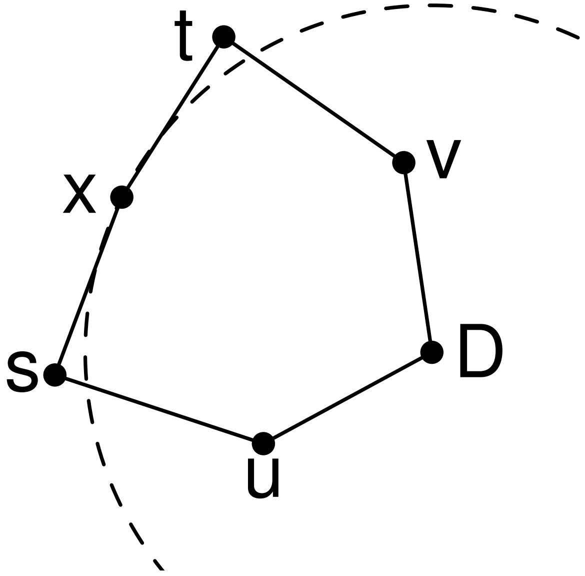
Contour finding accuracy is a function of node advertisements

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

Geographic routing using GPSR



Greedy routing



Perimeter routing

Route messages based on *location* of destination

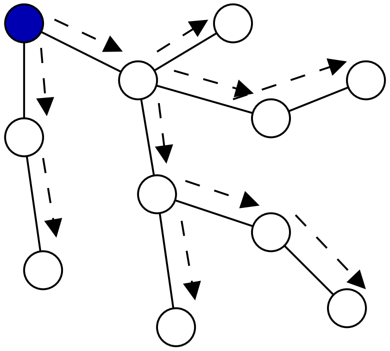
- Nodes only maintain location of immediate neighbors
- Initially route message to neighbor closest to destination (“greedy routing”)
- Requires planar graph when stuck in local minima (“perimeter routing”)

Easy to implement using radio and planar mesh regions

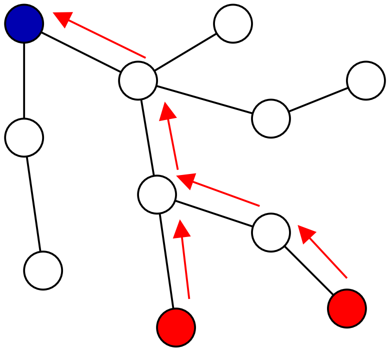
Directed diffusion

Mechanism for distributed event detection and reporting

- Sink floods interests to nodes in spanning tree region



- Nodes with matching data send results up tree to sink



- Relies on semantics of shared variable *get()* and *put()* in spanning tree

Implementation based on spanning tree region

- Only 188 lines of code for directed diffusion layer
- But ... 937 lines in the spanning tree region!

Conclusion

How do you program a entire network of distributed, volatile, resource-limited sensors?

- Program “the network” rather than individual nodes
- Requires appropriate programming models and communication primitives

Spatial programming and communication using abstract regions

- Communication and aggregation within local regions
- **Region formation** maintains neighborhood set
- **Shared variables** provide simple data sharing
- **Reductions** provide data aggregation

Exposing the resource-accuracy tradeoff to applications is crucial

- Sensor network communication is inherently statistical
- Applications must adapt to changing network conditions
- Abstract region operations provide accuracy feedback and tuning knobs

For more information:

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

Backup Slides Follow

Typical Applications

Moving vehicle tracking and pursuit

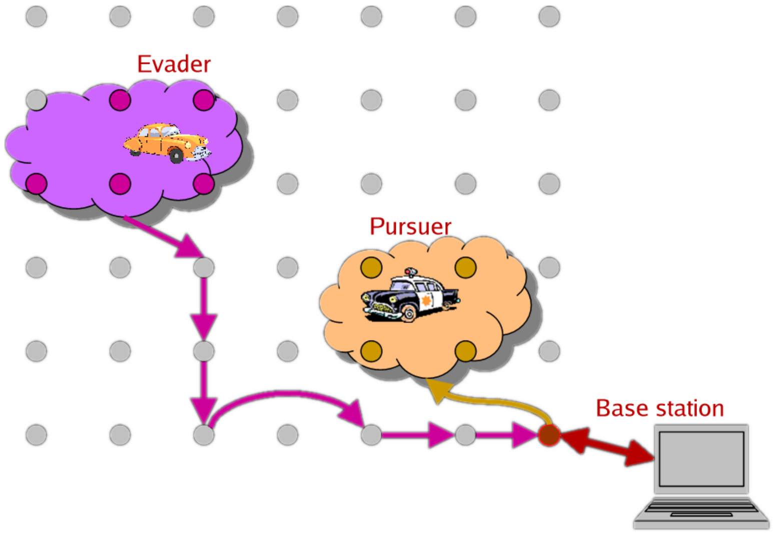
- Sensors take magentometer readings, locate object using centroid of readings
- Base station informs automated pursuer of object location

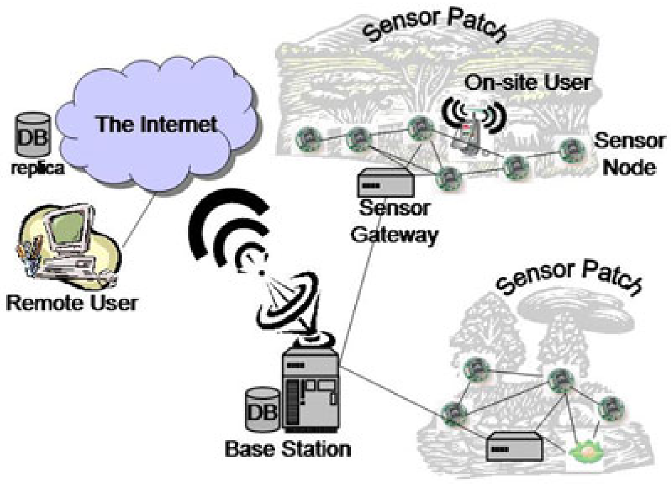
Great Duck Island - habitat monitoring

- Gather temp, humidity, IR readings from petrel nests
- Determine occupancy of nests to understand breeding/migration behavior

Spatial contour/region detection

- Detect frontier of phenomenon of interest (e.g., contaminant flow in groundwater)
- Sensors communicate locally to detect contour



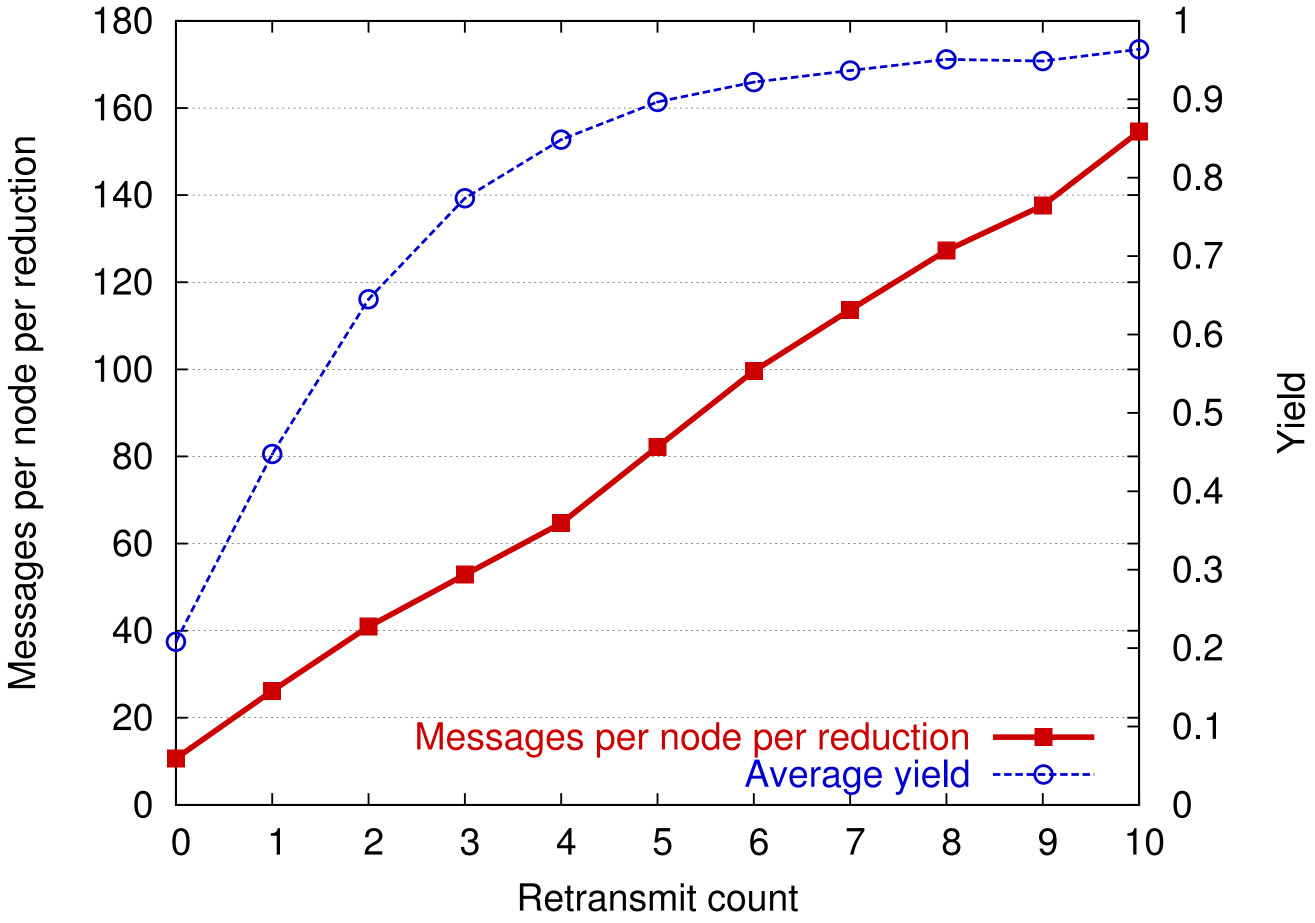


Object tracking using regions

```
location = get_location();
region = k_nearest_region.create(8);

while (true) {
  reading = get_sensor_reading();

  /* Store local data as shared variables */
  region.putvar(reading_key, reading);
  region.putvar(reg_x_key, reading * location.x);
  region.putvar(reg_y_key, reading * location.y);

  if (reading > threshold) {
    /* ID of the node with the max value */
    max_id = region.reduce(OP_MAXID, reading_key);

    /* If I am the leader node ... */
    if (max_id == my_id) {
      /* Perform reductions and compute centroid */
      sum = region.reduce(OP_SUM, reading_key);
      sum_x = region.reduce(OP_SUM, reg_x_key);
      sum_y = region.reduce(OP_SUM, reg_y_key);
      centroid.x = sum_x / sum;
      centroid.y = sum_y / sum;
      send_to_basestation(centroid);
    }
  }
  sleep(periodic_delay);
}
```

Affect of retransmission count

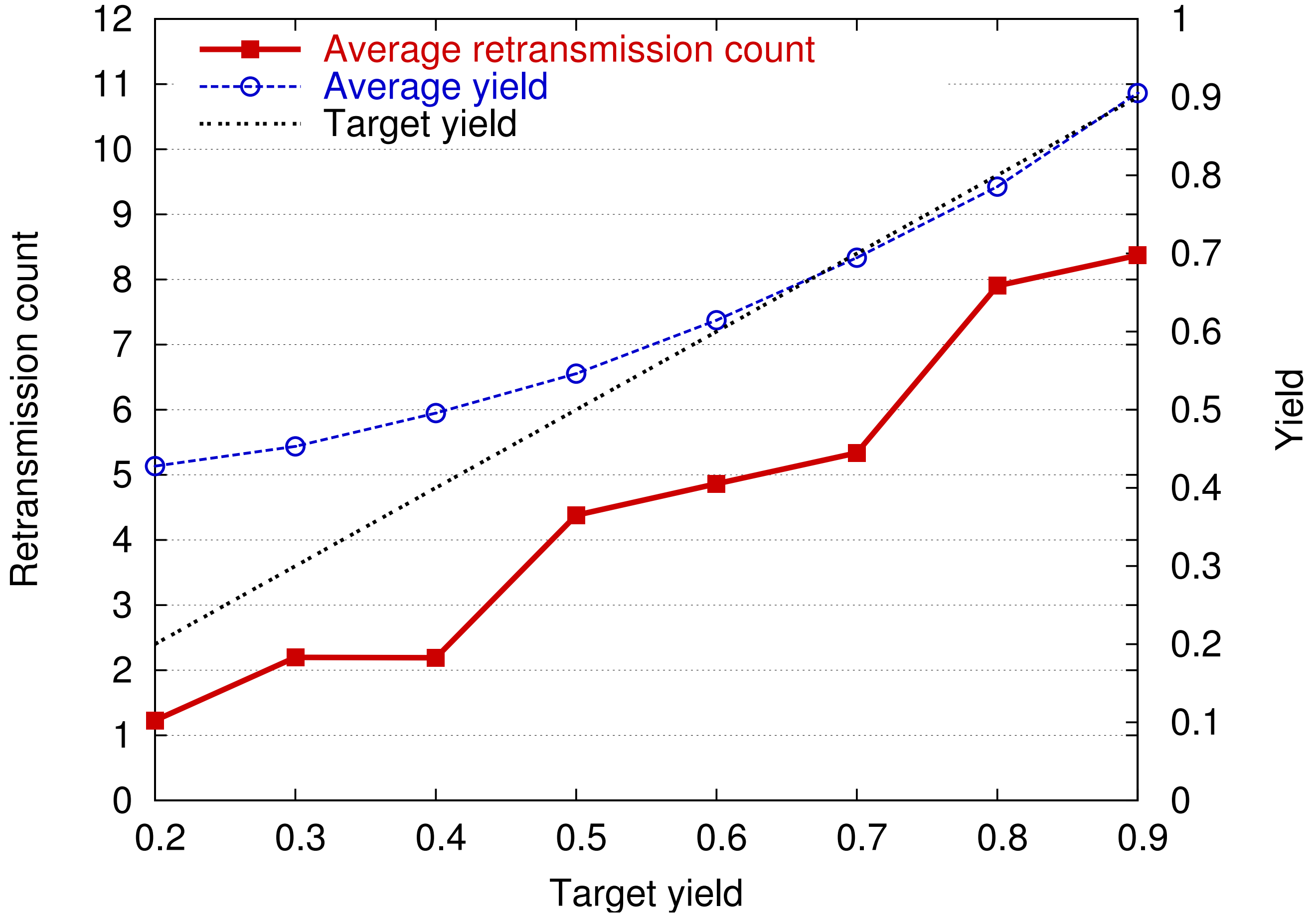
Adjusting message retransmission count affects reduction yield and message overhead:



Adaptive reduction

Tune overhead of reduce operation to meet a target *yield*

- Idea: Don't need to contact all neighbors, but some majority
- Adjust message retransmission attempts to meet target
- Additive increase/additive decrease algorithm



Tuning Parameters

Lots of knobs the programmer can turn:

- Maximum number of retransmission attempts
- Delay between retransmission attempts
- Maximum number of neighbors to consider in region formation
- Frequency of beacons for radio region formation
- Number of beacons to send during region formation
- Threshold used to remove neighbor from set
- Timeout for region formation operations
- Timeout for reduction operations
- Timeout for shared variable operations
- Timeout for retrieving location from neighbor nodes
- Timeout for waiting for edge invalidation messages

Leads to complex optimization strategy!

- Claim: Lots of knobs are better than no knobs

Fibers: Blocking operations in TinyOS

TinyOS is entirely event-driven

- Greatly complicates implementation of complex services
- Programmer must break code into multiple continuations and maintain state manually
- Difficult to check for timing errors and race conditions

Fibers: lightweight, thread-like abstraction for TinyOS

- Allows **one** blocking fiber in addition to standard event-driven TinyOS fiber
- Both fibers share the same stack!
- 150 instructions to context switch, 24 bytes overhead per fiber

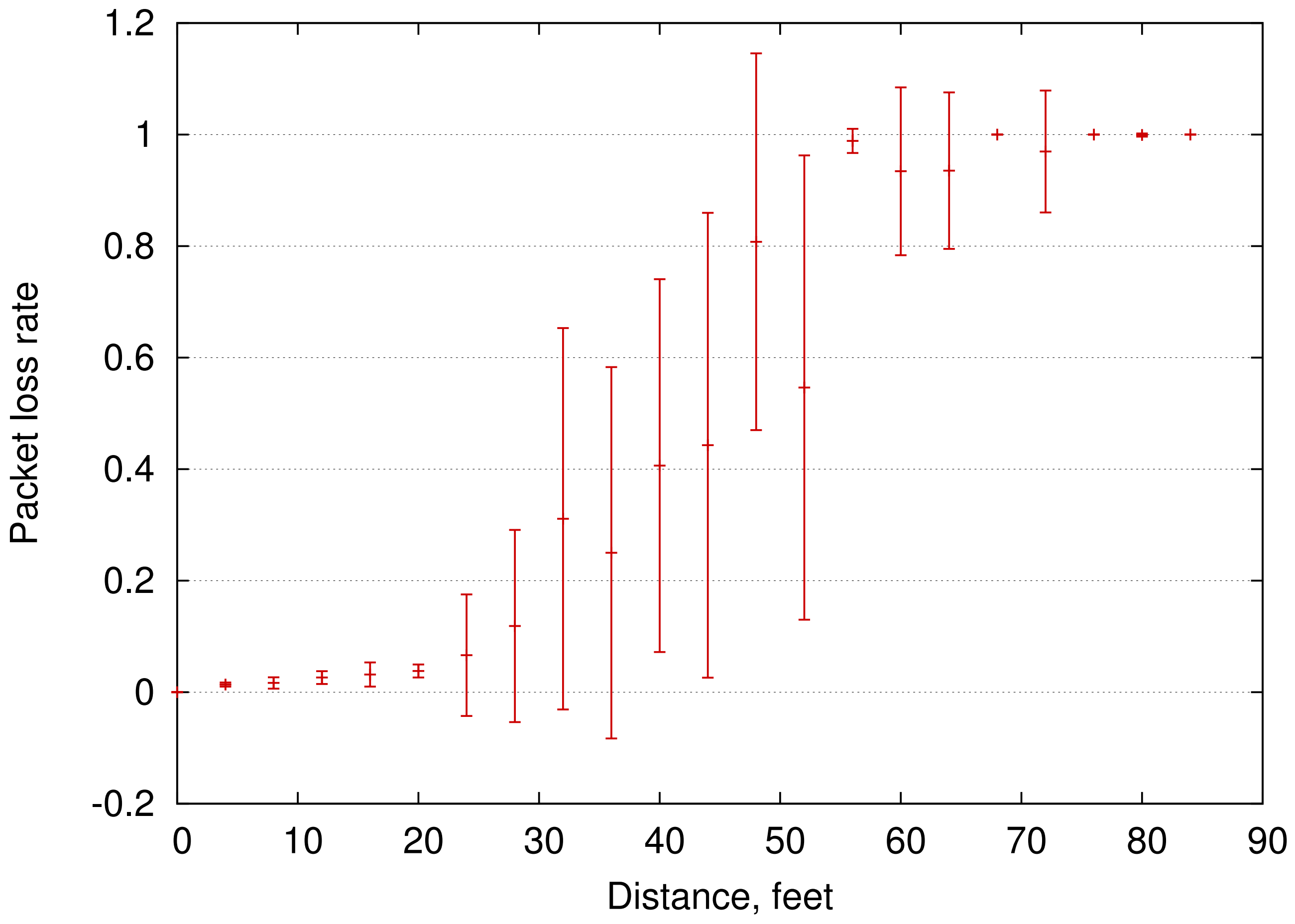
Blocking calls greatly simplify application design

- No more need for multiple event handlers, manual continuation management
- Tracking application w/o fibers: 369 lines, 5 event handlers, 11 continuations
- Tracking application with fibers: 134 lines, one main loop

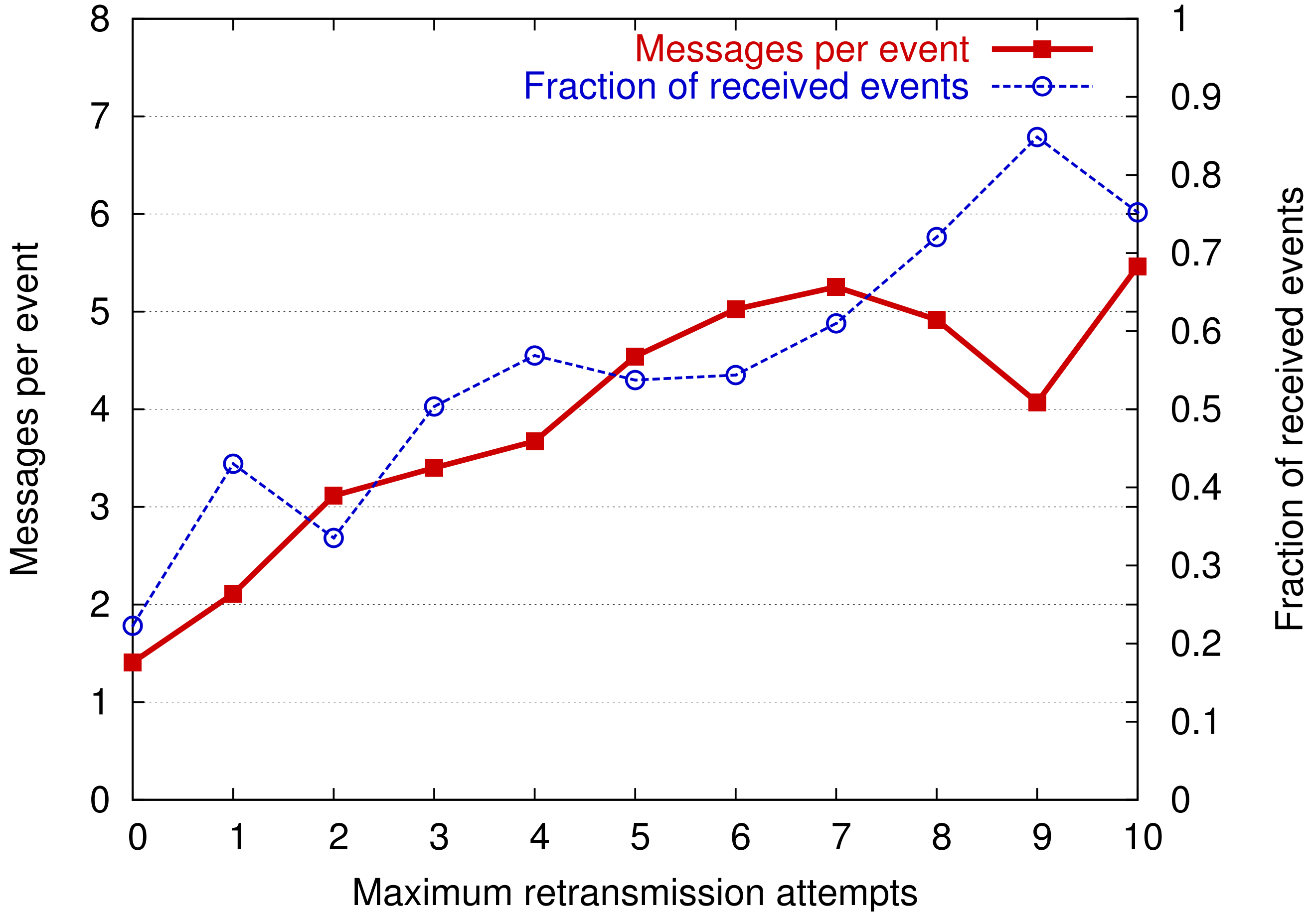
Evaluation environment

TOSSIM simulation with realistic radio model

- 100 nodes distributed as irregular grid in 20x20’ area
- Radio model derived from trace of MICA nodes in outdoor setting



Event detection accuracy using diffusion



Reliability of event detection as function of retransmission count

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

TinyDB

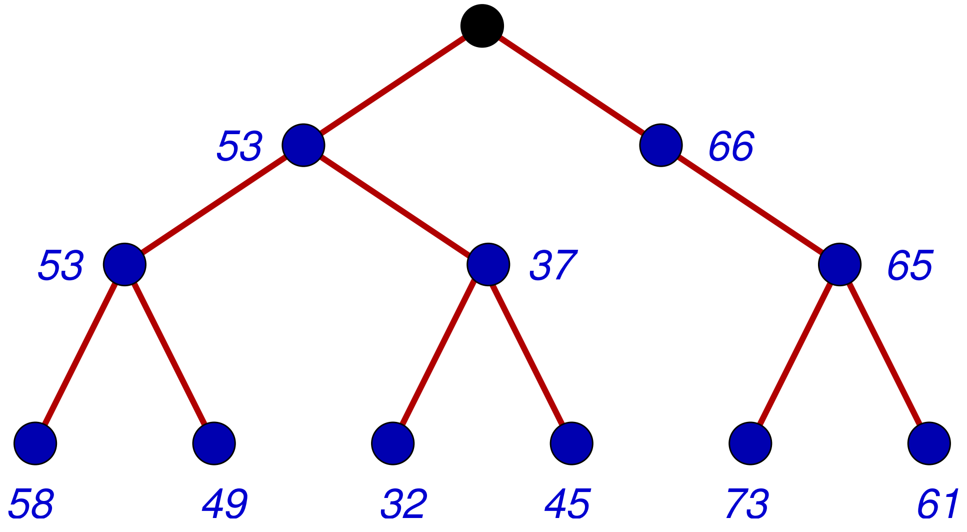
Sam Madden, Wei Hong, Joe Hellerstein, Intel Research/UCB

Express global queries on entire sensor network

```
SELECT nodeid, max(light) FROM sensors
  WHERE light > 40
  EPOCH DURATION 1sec
```

Nodes perform in-network aggregation

- Data flows along *spanning tree* from nodes to root
- Nodes aggregate data with their children



TinyDB

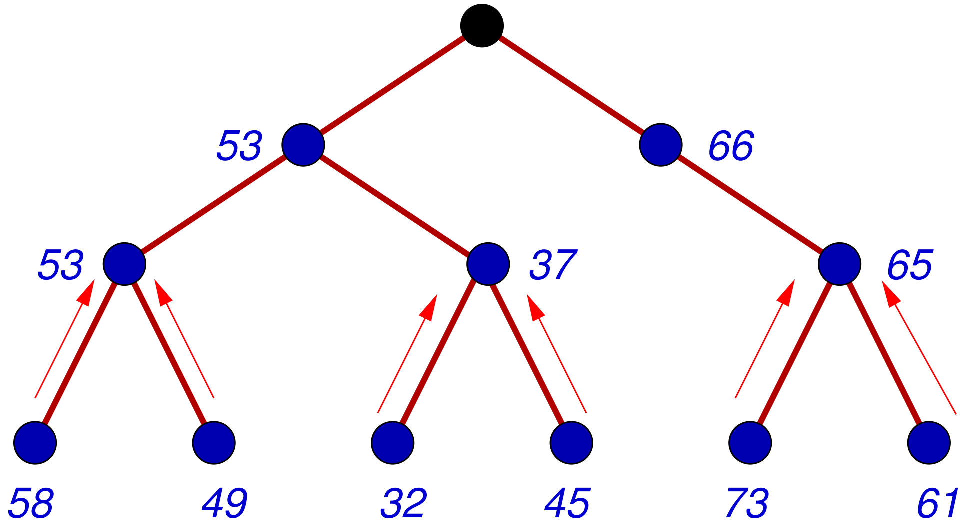
Sam Madden, Wei Hong, Joe Hellerstein, Intel Research/UCB

Express global queries on entire sensor network

```
SELECT nodeid, max(light) FROM sensors
  WHERE light > 40
  EPOCH DURATION 1sec
```

Nodes perform in-network aggregation

- Data flows along *spanning tree* from nodes to root
- Nodes aggregate data with their children



TinyDB

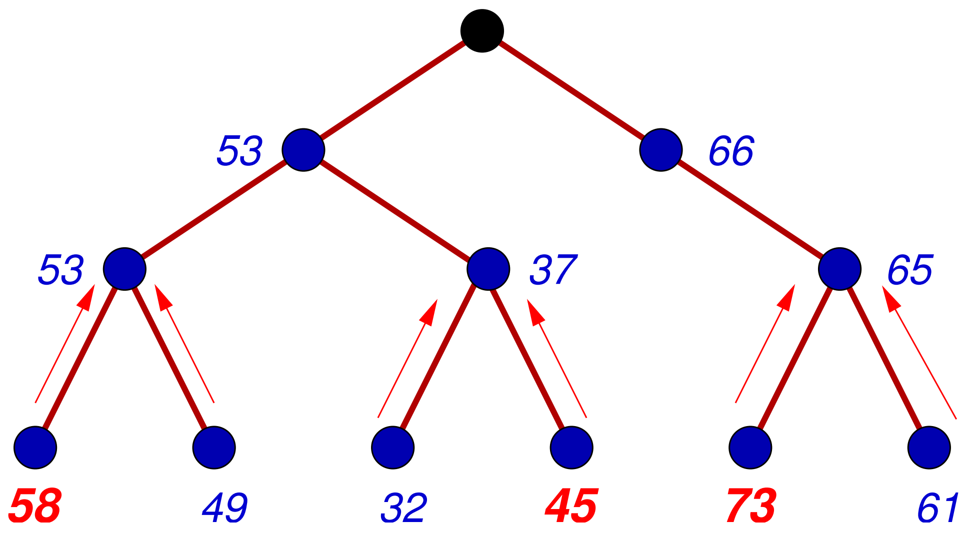
Sam Madden, Wei Hong, Joe Hellerstein, Intel Research/UCB

Express global queries on entire sensor network

```
SELECT nodeid, max(light) FROM sensors
  WHERE light > 40
  EPOCH DURATION 1sec
```

Nodes perform in-network aggregation

- Data flows along *spanning tree* from nodes to root
- Nodes aggregate data with their children



TinyDB

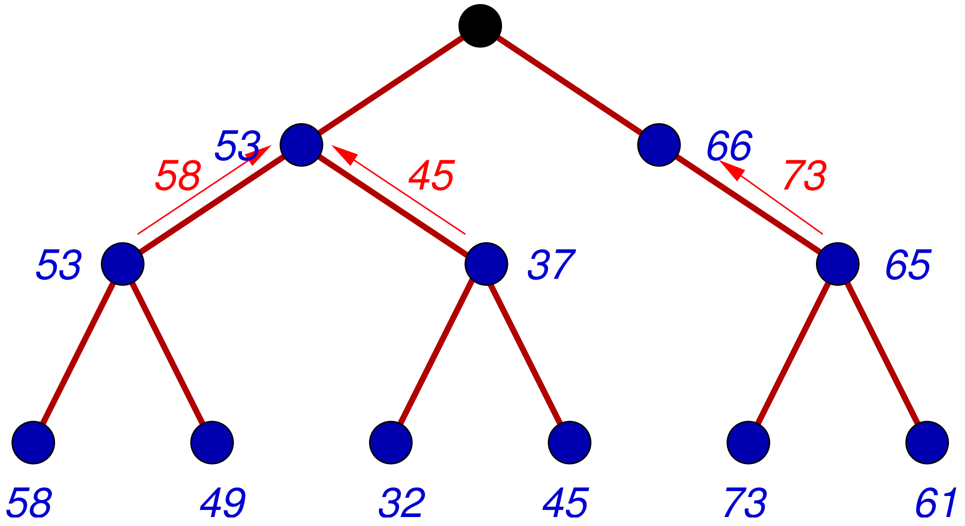
Sam Madden, Wei Hong, Joe Hellerstein, Intel Research/UCB

Express global queries on entire sensor network

```
SELECT nodeid, max(light) FROM sensors
  WHERE light > 40
  EPOCH DURATION 1sec
```

Nodes perform in-network aggregation

- Data flows along *spanning tree* from nodes to root
- Nodes aggregate data with their children



TinyDB

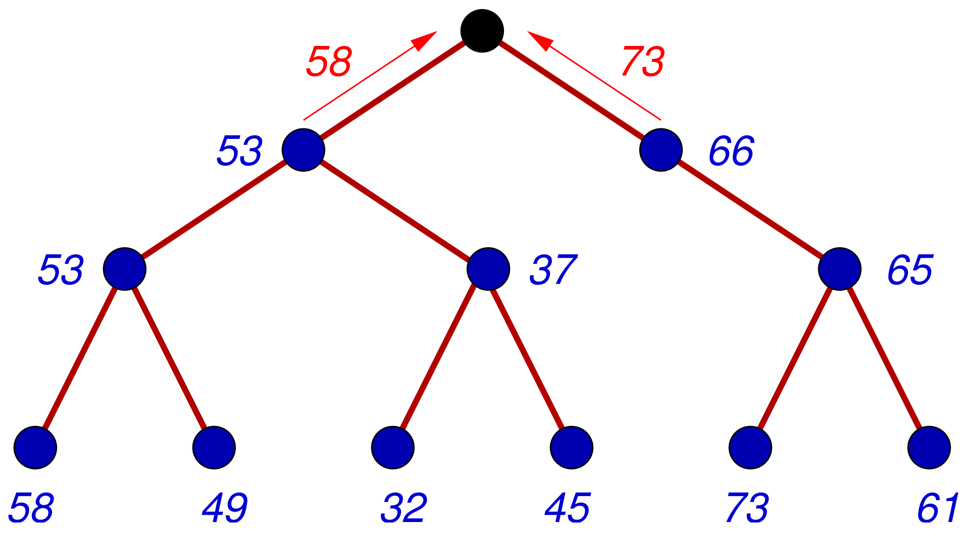
Sam Madden, Wei Hong, Joe Hellerstein, Intel Research/UCB

Express global queries on entire sensor network

```
SELECT nodeid, max(light) FROM sensors
  WHERE light > 40
  EPOCH DURATION 1sec
```

Nodes perform in-network aggregation

- Data flows along *spanning tree* from nodes to root
- Nodes aggregate data with their children



Query interface limitations

Not general enough to capture arbitrary in-network processing

- Focus on streaming results to root of tree
- More sophisticated inter-node operations difficult to express
- e.g., detecting edges of regions in network

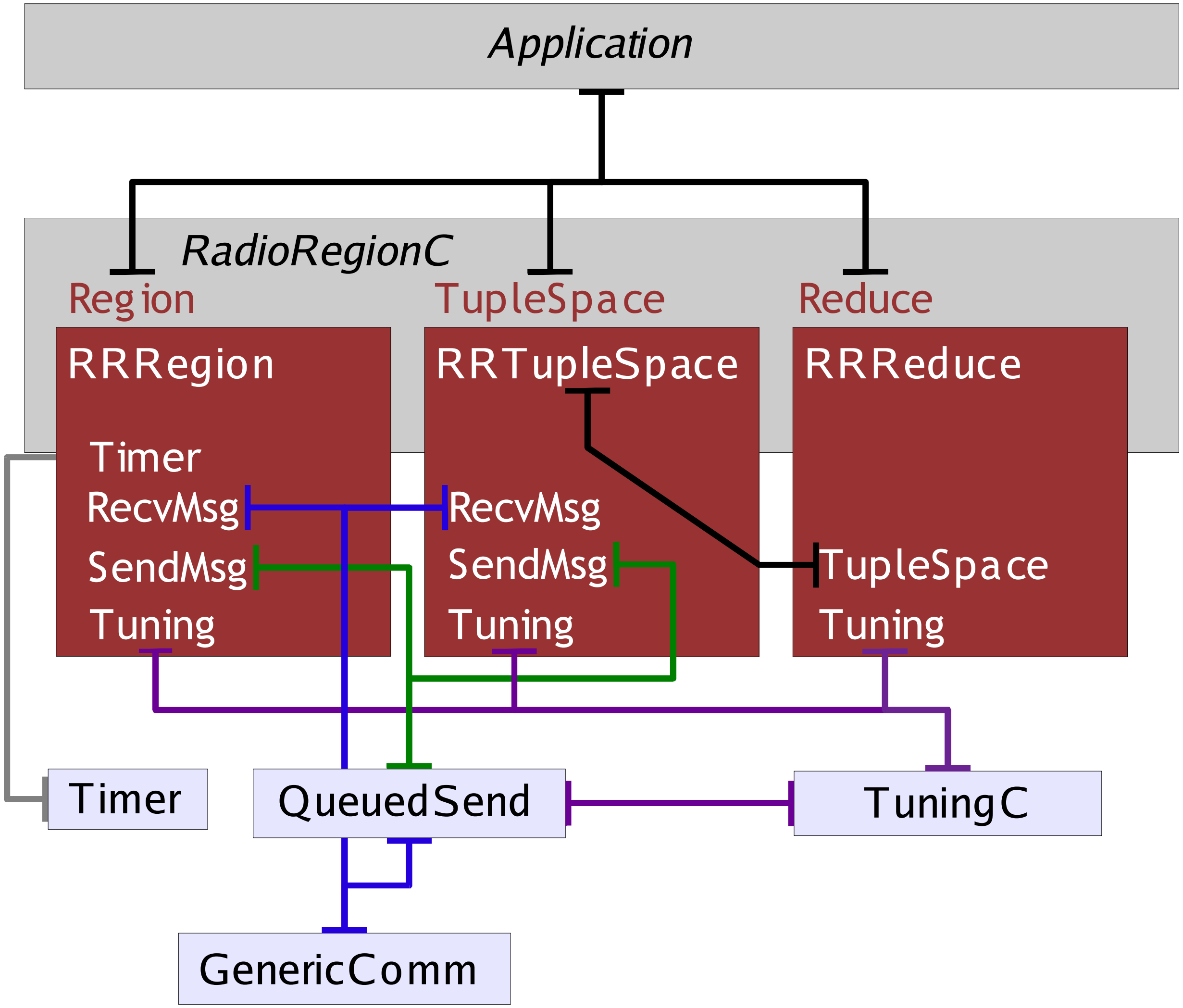
Purely declarative programming model

- Query processor drives execution and makes implicit tradeoffs
- e.g., How to adapt sampling rate when interesting events happen?
- Would like explicit control over network operation

Difficult to express actuation

- Sensor networks may be used for complex control scenarios
- Network takes local action based on sensor readings
- e.g., Distributed control of unmanned pursuit vehicle
- Nodes joining/leaving/moving, etc. difficult to capture

Component Structure

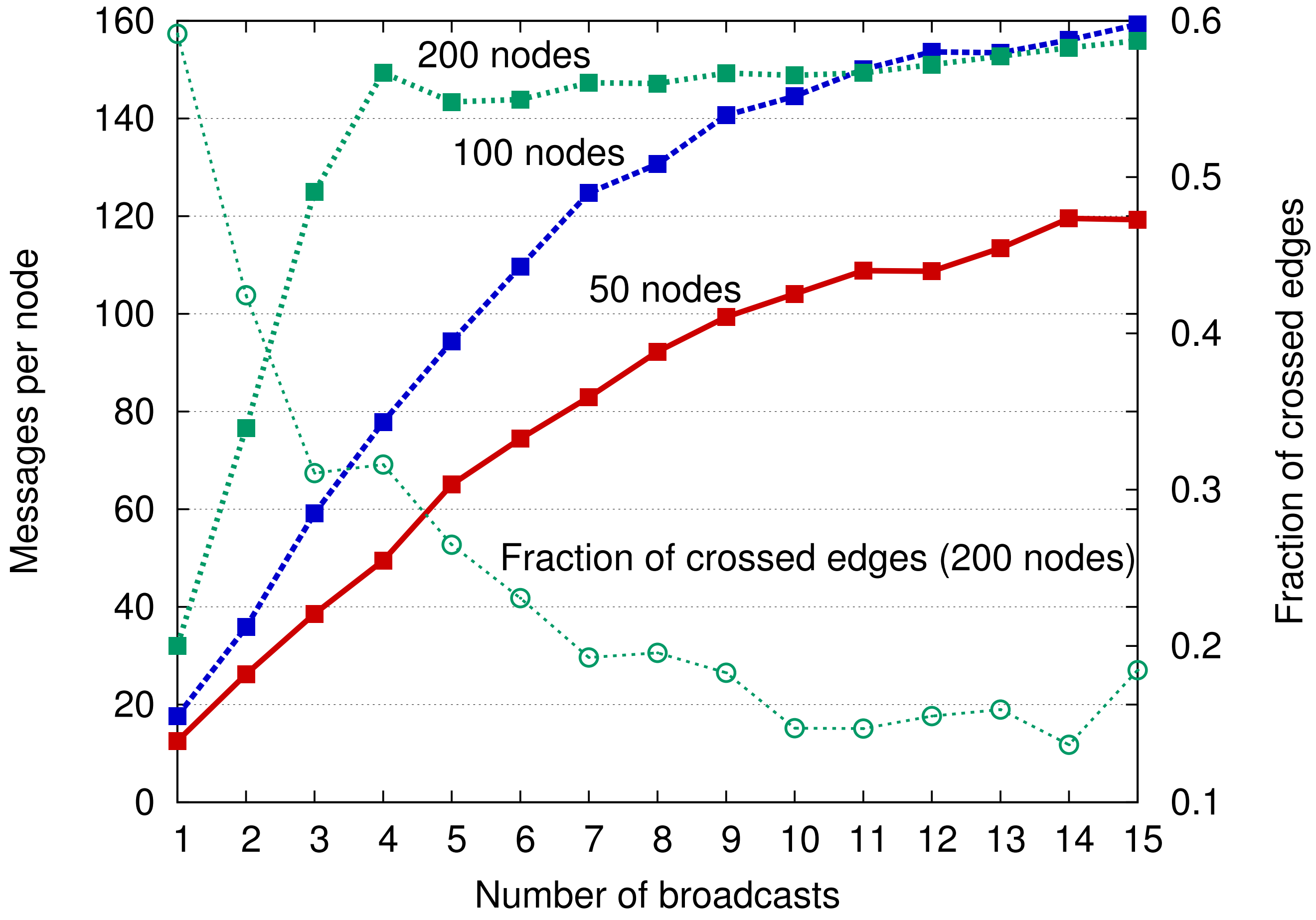


Application Line Counts

Application	*With fibers*	*Without fibers*
Tracking	134 lines	369 lines
Contour finding	175 lines	350 lines
Directed diffusion	–	313 lines
Region		
Radio	938 lines	
k**-nearest**	340 lines	
Spantree	937 lines	
Pruned Yao graph	659 lines	

- Most of the complexity captured by region substrate
- Use of blocking fibers greatly simplifies code

Approximate planar mesh construction



Planar mesh overhead related to number of node broadcasts

- Quality of mesh increases with additional advertisements
- Overhead of mesh construction increases with node density

SplitNesC Language

Linguistic support for SIMD programming using abstract regions

- Inspired by Split-C – parallel C variant with global pointers
- Support region operations as first-class operations
- Compile down to NesC components
- Generate necessary dependencies, AM handlers, etc.

```
region onehop {
  uint16_t my_reading;
  uint16_t sum_value;
} myregion;

/* Read remote values */
localvar1 = myregion.myreading[node1];
localvar2 = myregion.myreading[node2];
if (!myregion.sync(TIMEOUT)) { // Error ... }

/* Set local value */
myregion.sum_value = localvar1 + localvar2;

/* Perform reduction */
localvar3 = myregion.reduce(OP_MAX, my_reading);
```

Market-oriented Programming

Induce global behavior using market-based algorithm design

- Balancing sampling, communication, and sleeping is a complex optimization process
- Nodes act as self-interested agents with simple behaviors
- e.g., Take sensor reading, broadcast value, aggregate, or sleep
- Nodes do not have knowledge of high-level program!