Programming Sensor Networks Using Abstract Regions

Matt Welsh and Geoff Mainland
Harvard University
Division of Engineering and Applied Sciences
{mdw,mainland}@eecs.harvard.edu
Goals

Develop a programming model for aggregate programs across an entire sensor network

- Current programming models are node centric and low level
- Scientists don’t want to think about gronky details of radios, timers, battery life, etc.

Flexible communication primitives for sensor networks

- Reduce programming effort to construct applications
- Abstract low-level details of local coordination
- Focus on spatial computation within local neighborhoods
- Inspired by MPI’s success in parallel programming
  - Abstract machine details, but still permit extensive optimizations

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 $k$

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

Expose resource consumption/accuracy tradeoff

- Expose control for resource usage
- Return feedback on accuracy and yield of collective operations
Region Operations

**Neighbor discovery** identifies nodes

- Continuous background process, can be terminated or restarted
- Node identified of changes to region membership
- e.g., Nodes moving, joining, or leaving network

**Shared variables** support 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`
Region Implementations

Radio and geographic neighborhood discovery
- $k$ nearest neighbors, all nodes within $k$ hops, etc.
- Nodes emit periodic beacons with node ID and (optionally) location
- Filter received beacons to determine neighbors (e.g., $k$ nearest nodes)
- Inspired by Cory and Kamin’s neighborhoods module

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: broadcast $put()$, $get()$ is local

Reduction implementation
- All nodes are one hop away
- Broadcast $get()$ request for all values of shared variable
- Collect replies and perform reduction after all responses received, or timeout
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

True planarity is difficult to achieve

- Requires information on location and edges from all nearby neighbors
- Delaunay triangulation difficult to compute locally
- Rather, strive for approximate planarity: allow some crossed edges
Adaptive Spanning Tree

Useful for aggregating data to a single point in the network

- Maintain adaptive tree based on Surge-like protocol
- Nodes periodically select new parent based on link quality estimate

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

- Shared variable operations may time out
- Reductions may only contact subset of nodes
- Collective operations report *yield*: fraction of nodes that responded to a request

Regions provide control over overhead-accuracy tradeoff

- Programmer can tune parameters affecting resource usage of region operations
- Examples: retransmission count, timeouts, number of neighbors in region
- Quality feedback can be used to drive adaptation to changing network conditions

Tuning examples

- Number of neighbor broadcasts affects planarity of mesh
- Number of message retransmissions affects reduction yield
- Timeouts for shared variable operations affect reliability
Adaptive reduction

Tune overhead of reduce operation to meet a target \textit{yield}

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

Very limited, lightweight, thread-like abstraction

- *Application fiber* may perform blocking ops
- *System fiber* is event-driven – default TinyOS context
- Both fibers share the same stack
- 150 instructions to context switch, 24 bytes overhead per fiber

```
interface Fiber {
    command result_t start(void *(*start)(void *arg), void *arg);
    command void yield();
    command void sleep(fiber_t **queue);
    command fiber_t *wakeup(fiber_t **queue);
    command void wakeup_one(fiber_t **queue, fiber_t *fiber);
    command fiber_t *curfiber();
    command fiber_t *getfiber(int id);
}
```

Blocking calls greatly simplify application design

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

```java
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);
}
```
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 a function of node advertisements

- Form approximate planar mesh region
- More advertisements $\rightarrow$ fewer crossed edges
- Mean error directly correlated with mesh quality
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 \texttt{get()} and \texttt{put()} in spanning tree

Abstract regions simplify implementation considerably

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

```c
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);
```
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](http://www.eecs.harvard.edu/~mdw/proj/mp)
Backup Slides Follow
Application Examples

NEST tracking demo, 29 Palms
- Nodes coordinate locally to determine location and velocity of moving object

Environmental monitoring and Great Duck Island
- Periodic sampling and routing to base station
- Response to external commands/events
- Partitioning of tasks and sensor modalities across different node types

Code Blue: Sensor nets for emergency medical response
- EMTs attach vital sign sensors to patients at disaster site
- Rescue personnel receive reports on patient status, ambulance location, hospital availability
- Coordinate activity of multiple rescue teams

In each case, lots of complex machinery and tradeoffs
- Identifying nodes to coordinate with
- Exchanging and sharing of information
- Routing data to other nodes and base station
- How often to sample? Sleep? Communicate?
### Application Line Counts

<table>
<thead>
<tr>
<th>Application</th>
<th>With fibers</th>
<th>Without fibers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tracking</td>
<td>134 lines</td>
<td>369 lines</td>
</tr>
<tr>
<td>Contour finding</td>
<td>175 lines</td>
<td>350 lines</td>
</tr>
<tr>
<td>Directed diffusion</td>
<td>–</td>
<td>313 lines</td>
</tr>
</tbody>
</table>

### Region

<table>
<thead>
<tr>
<th>Region</th>
<th>Lines</th>
</tr>
</thead>
<tbody>
<tr>
<td>Radio</td>
<td>938 lines</td>
</tr>
<tr>
<td>$k$-nearest</td>
<td>340 lines</td>
</tr>
<tr>
<td>Spantree</td>
<td>937 lines</td>
</tr>
<tr>
<td>Yao graph</td>
<td>659 lines</td>
</tr>
</tbody>
</table>

- Most of the complexity captured by region substrate
- Use of blocking fibers greatly simplifies code
Planar mesh overhead related to number of node broadcasts

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