Computer Science and Artificial Intelligence Laboratory

Technical Report

MIT-CSAIL-TR-2006-069 October 3, 2006

Programming a Sensor Network as an Amorphous Medium

Jonathan Bachrach and Jacob Beal

massachusetts institute of technology, cambridge, ma 02139 usa — www.csail.mit.edu

Programming a Sensor Network as an Amorphous Medium

Jonathan Bachrach and Jacob Beal

June 8, 2006

1 Introduction

In many sensor network applications, the network is deployed to approximate a physical space. The network itself is not of interest: rather, we are interested in measuring the properties of the space it fills, and of establishing control over the behavior of that space.

Consider, for example, deploying a network of devices to manage a large farm. The tasks to be carried out by the devices—irrigation, pest management, and fertilization, for example—are naturally specified in terms of regions of the farm (e.g. "a potato field is watered every so-many hours during hot weather"). An applications programmer for farms should be able to write code at this level, rather than having to specify in depth how the sensor network will be deployed in the fields or how the devices will coordinate to carry out the programs.

The spatial nature of sensor network applications means that many can be expressed naturally and succinctly in terms of the global behavior of an *amorphous medium*—a continuous computational material filling the space of interest. Although we cannot construct such a material, we can approximate it using a sensor network.

Using this amorphous medium abstraction separates sensor network problems into two largely independent domains. Above the abstraction barrier we are concerned with long-range coordination and concise description of applications, while below the barrier we are concerned with fast, efficient, and robust communication between neighboring devices.

We apply the amorphous medium abstraction with Proto, a high-level language for programming sensor/actuator networks. Existing applications, such as target tracking and threat avoidance, can be expressed in only a few lines of Proto code. The applications are then compiled for execution on a kernel that approximates an amorphous medium. Programs written using our Proto implementation have been verified in simulation on over ten thousand nodes, as well as on a network of Berkeley Motes.

2 Sensor Network Model

As our goals include robustness and scalability, we use a challenging model for the underlying sensor network. The number of devices may range from dozens to billions. Devices are distributed arbitrarily through space and collaborate via unreliable broadcast to neighbors no more than r distance away. Devices move much more slowly than communication, if at all. Memory and processing are not limiting resources,[1] and execution is partially synchronous—each device has a clock which ticks regularly, but frequency may vary within some small ϵ and clocks have an arbitrary initial time and phase. Naming, routing, and coordinate services are not provided.[2] Finally, arbitrary point and region stopping failures and joins may occur, possibly changing network connectedness.

Energy management has long been a key concern of sensor-networks. Thus, although the amorphous medium abstraction does not address energy, we must consider its impact on energy management.

In fact, many energy management techniques for sensor networks can be expressed without violating the amorphous medium abstraction barrier. Space-centric techniques can be expressed simply in Proto: for example, Directed Diffusion[14] restricts operation to paths connecting regions of interest to a sink region, and Energy Aware Routing[21] descends along a potential field generated by its cost metric. Essentially local techniques, such as the S-MAC energy efficient wireless protocol[24] can be confined below the barrier.

While there will be some techniques which cannot be cleanly factored using the amorphous medium abstraction, our expectation is that there will be related techniques which can be cleanly factored and are only marginally less efficient. Our current implementation is relatively inefficient, but there is no bar to any number of techniques.

3 Related Work

In sensor networks research, a number of other high-level programming abstractions have been proposed to enable programming of large networks. For example, GHT[20] provides a hash table abstraction for storing data in the network, and TinyDB[16] focuses on gathering information via query

[1] Profligate expenditure of either is still bad, and memory is an important constraint for the Mote implementation.

[2] They may be made available as sensor values, with appropriate characterization of reliability and error.

processing. Both of these approaches, however, are data-centric rather than computation-centric, and do not provide guidance on how to do distributed manipulation of data, once gathered. TinyOS[11] and the Hood abstraction[23] provide useful general programming tools—indeed, our implementation of Proto on Motes uses TinyOS—but the abstractions are less powerful and lead to bulkier and less reusable code.

More similar is the Regiment[18] language, which uses a stream-processing abstraction to distribute computation across the network. Regiment, however, is only distributed when the compiler finds optimization opportunities, and there are significant challenges remaining in adapting its programming model to sensor-networks. Kairos[12] also allows programming of a sensor network as a whole, but is addressed at its graph structure, rather than the space it inhabits.

Previous work on amorphous medium languages proposes the amorphous medium abstraction[3], general strategies for control[6], and an ancestor language of Proto[4]. Recently, we described [5] how the abstraction simplifies engineering of emergent behavior.

Other work on languages in amorphous computing [1] has shared the same general goals, but has been directed more towards problems of morphogenesis and pattern formation than general computation. A notable exception is Butera's work on paintable computing[7], which allows general computation, but lacks an abstraction barrier separating an applications programmer from low-level network details.

Finally, the structure of Proto as a dynamic network of streams is strongly influenced by Bachrach's previous work on Gooze[2], as are many of the compilation strategies used to compact Proto code for execution on Motes. There is a long tradition of stream processing in programming languages. The closest and most recent work is Functional Reactive Programming (FRP) [10] that is based on Haskell [15], which is a statically typed programming language with lazy evaluation semantics. FRP has been demonstrated on robotics [19] and graphics [10]. In these systems, less attention is spent on runtime space and time efficiency, and the type system is firmly wedded to Haskell, with all of its strengths and weaknesses.

4 Programming Amorphous Media

An *amorphous medium* is a theoretical continuous computational material which fills space. Every point in the medium is a computational device which independently executes the same code as every other device in the medium.[3] Nearby devices share state—each device has a *neighborhood* of devices nearby whose state it can access (Figure 1).

Programs written in Proto compute on an amorphous medium by manipulating fields (a field is a map assigning a value to each point in the space). A program describes the manipulation of streams of fields: when executed, the program is evaluated against the space at regular intervals to produce a stream of output fields.[4]

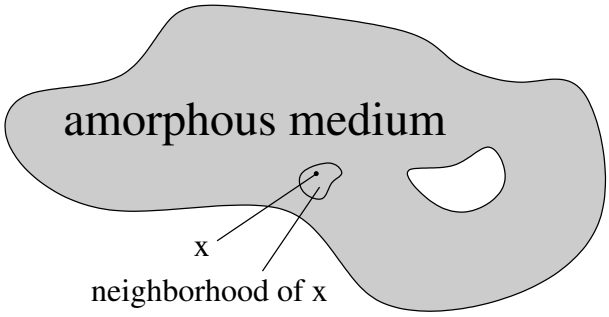


Figure 1: Every point in an *amorphous medium* is an independent device running the same program and exposing its internal state to its neighborhood.

Primitives in Proto are either terminals which produce fields, or operators which calculate an output field from input fields. For example, the expression **2** is a terminal which evaluates to a stream of fields with the value two everywhere, and **+** is an operator which adds its input fields pointwise (Figure 2). Other notable primitives are **mux**, which uses a field of booleans to select pointwise between two other inputs and **sense** and **act** which read from sensors and write to actuators, respectively, allowing the program to interact with its external environment.

Primitives are composed to form complex expressions by connecting outputs to inputs to form a directed graph of primitives connected by streams. Proto expresses this syntactically as LISP-like function application. For example, the expression **(+ 2 5)** evaluates to a graph of three nodes which produces a stream of fields valued seven everywhere (Figure 2(d)).

Abstraction is done with lambda expressions, boxing up a graph fragment to form a new primitive. For example, **(lambda (x) (* x x))** squares the value of every point in a field. Given **lambda**, it is straightforward to implement related primitives like **def**, which produces a named lambda (Figure 2(e) and 2(f)), and **let**, which syntactically allows an output stream to be connected to several inputs. Using **def**, we can define a square function **sq** as follows:

```
(def sq (a) (* a a))
```

In addition to these basic operations, Proto provides a set of operations that allow the programmer to specify behaviors that depend on the space and time relationships of fields.

Unlike discrete networks, each point in an amorphous medium has an infinite number of neighbors. As such, inter-

[3] Executions diverge due to differences in sensor values, randomness, and interaction with their neighborhoods.

[4] Our actual implementation does not attempt to synchronize evaluation, but merely depends on limited variation in the rate of evaluation (See Section 6)

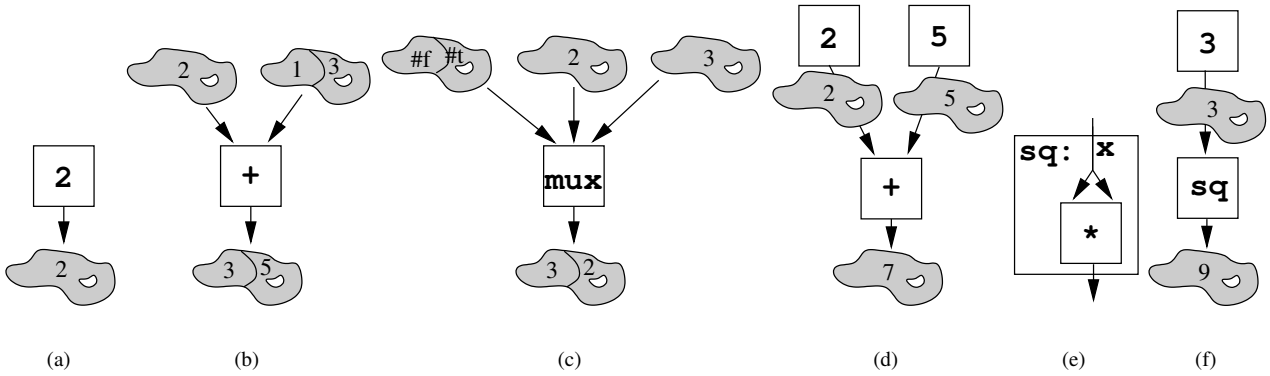


Figure 2: Proto primitives produce streams of fields. For example, **2** produces a constant stream of fields valued two everywhere (a), **+** produces a pointwise sum of its inputs (b), and **mux** uses a field of booleans to select pointwise between two other inputs (c). Primitives are composed into a directed graph of streams by connecting outputs to inputs (d). Abstractions are created by "boxing" a subgraph in a lambda expression (e), which can then be used as a new primitive (f).

action by message passing is impractical. Proto instead provides implicit communication through the **reduce-nbrs** operator, which summarizes the values in the neighborhood using a quantifier that is applicable to uncountable sets.

Proto currently implements five such quantifiers: **integral**,[5] **forall** and **exists** (AND and OR equivalents for uncountable sets), and **limsup** and **liminf** (min and max equivalents for uncountable sets).

Thus, for example, we can calculate local averages with

```
(def local-average (v)
  (/ (reduce-nbrs v integral) (reduce-nbrs 1 integral)))
```

which normalizes an integral of the value over the neighborhood against the area of the neighborhood.

In general, we do not want to tie the behavior of programs to neighborhood sizes, so Proto also provides operators for measuring distance in space and time—**nbr-range** and **nbr-lag**, respectively.[6]

Persistent state is established using delay loops, specifying an initial value and an expression for calculating the next value from current values. For example, we can define a discrete-time integrator

```
(def integrate (x) (letfed ((v 0 (+ x v))) v))
```

that creates one state variable, **v**, which starts at zero and adds the value of the input **x** at each evaluation.

A common pattern in Proto is long-range communication by using a **letfed** state variable to chain **reduce-nbrs** interactions. For example, we can create a gradient operator

```
(def gradient (src)
  (letfed ((n infinity
             (+ 1 (mux src 0 (reduce-nbrs
                               (+ n nbr-range) liminf)))))
    (- n 1)))
```

[5] a Lebesgue integral, to be precise

[6] These may be implemented coarsely or finely, depending on the hardware available: for example, our Mote implementation estimates the distance to all neighbors as its radio range, and the time lag as one round.

which measures the distance from every point to the nearest source. The addition of one drives the distance upward when it is not connected to the source, allowing the gradient to adapt to changing sources in the same way as Clement and Nagpal's active gradients[8].

Although the entire amorphous medium shares a program, we do not generally want the whole program running everywhere. The **if** operator restricts the space against which an expression is evaluated. For example, we can define a dilation operator that selects everything within r units of a source

```
(def dilate (r source) (<= (gradient source) r))
```

then clip it against an arbitrary boundary,

```
(def bound (source max boundary)
  (if (not boundary) (dilate max source)))
```

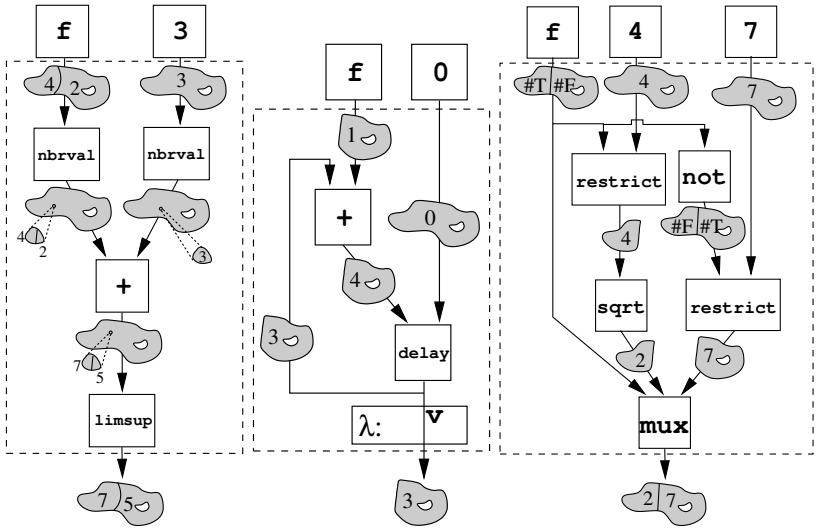
so that the dilation runs only within the boundaries containing the source.

5 Example Sensor-Network Apps

Many sensor network tools and applications can be implemented simply and efficiently using Proto.

A useful example is the coordinate system mechanism from Butera's paintable computing[7], which derives coordinates from a provided source and destination. We will need to measure the distance between these places, which we can do with a **distance** operator and a value-carrying version of **gradient**

```
(def grad-value (src v)
  (let ((d (gradient src)))
    (letfed ((x 0 (mux src v
                    (2nd (reduce-nbrs (tup d x) liminf)))))
      x)))
(def distance (p1 p2)
  (grad-value p1 (gradient p2)))
```



(a) **reduce-nbrs** (b) **letfed** (c) **if**

Figure 3: Proto provides operations that allow the programmer to specify communication (e.g. **(reduce-nbrs (+ f 3) limsup)** in (a)), state (via feedback loops like **(letfed ((v 0 (+ f v))) v)** in (b)) and restriction of space (e.g. **(if f (sqrt 4) 7)** in (c)).

The paintable computing **channel** mechanism, which finds a wide path connecting two points, uses a trail-following operator to trace a gradient back up to the source. This is fairly fragile, so we instead find the trail geometrically by triangulation against **distance**, then widen it using **dilate**.

```
(def channel (src dst width)
  (let* ((d (distance src dst))
         (trail (<= (+ (gradient src) (gradient dst)) d)))
    (dilate width trail)))
```

Implementing the **coordinates** mechanism requires one more operator, **choose-leader**, which is used to break symmetry by selecting a single location in the channel

```
(def choose-leader (selector)
  (letfed ((v (if selector (random 1.0) infinity))
           (minv v (reduce-nbrs minv liminf)))
    (and (< v infinity) (= v minv) v)))
```

Butera's coordinates mechanism (Figure 4) can then be defined as an operator that is relatively straightforward for a programmer to create and understand.

```
(def coordinates (src dst width)
  (let* ((field (channel src dst width))
         (axis (channel src dst 1))
         (d1 (gradient src))
         (d2 (gradient dst))
         (dp (distance src dst))
         (buoy (choose-leader
                 (and field (< d1 dp) (< d2 dp))))
         (y (/ (+ (* d2 d2) (- (* d1 d1)) (* dp dp))
               (* 2 dp)))
         (x (sqrt (- (* d2 d2) (* y y))))
         (neg (bound buoy (+ width dp)
                     (or (< y 0) (> y dp) axis))))
    (tup (if neg (- x) x) y)))
```

Butera's channel operator can also be used to restrict communication. For example, in this tracking code

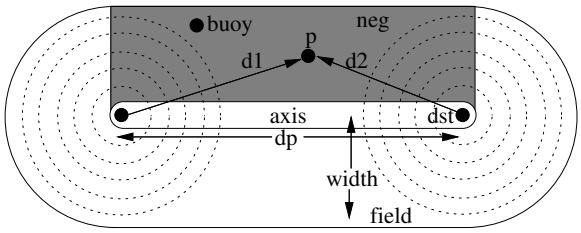
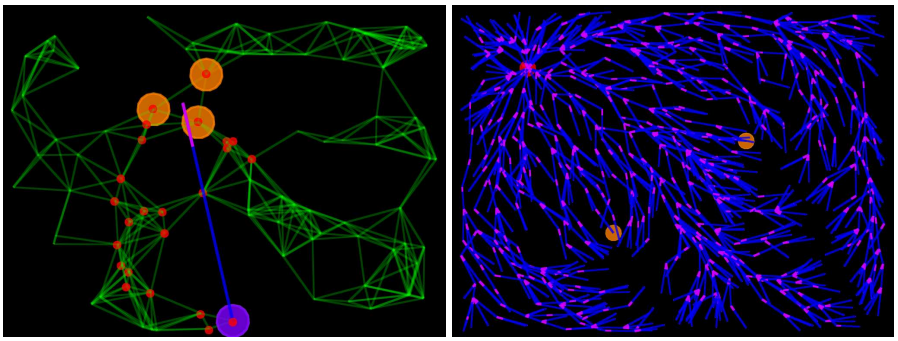


Figure 4: Calculating coordinates with a mechanism adapted from paintable computing[7]: the two anchor points of the coordinate system send out gradients, producing **d1**, **d2** and **dp** which determine the location of **p** except for the sign of its vertical coordinate. The sign is found by using leader election to break symmetry.

```
(def track (target dst coord)
  (let ((point
         (if (channel target dst 10)
           (grad-value target
             (mux target
                  (local-average-tup coord)
                  (tup 0 0)))
           (tup 0 0))))
    (mux dst (vsub point coord) (tup 0 0))))
```

a clique of nodes detecting a target estimate its location by averaging their coordinates. The location flows back to a base station (**dst**) along the channel, so that the information is not transmitted to uninvolved portions of the network. Figure 5 shows the target tracking program verified in simulation.

Another useful application is threat avoidance. Given co-



(a) Tracking (b) Threat Avoidance

Figure 5: Snapshots of the target tracking program (a) and threat avoidance program (b) being verified on 100 and 1000 simulated devices, respectively.

ordinates, a threat sensor and a model of exponentially decaying threat, we can calculate the expected safest path to a destination after the fashion of the threat avoidance program described by Eames[9]. The equivalent modules in Eames' implementation contain more than 2000 lines of nesC, while this Proto implementation is a mere 22 lines long.

The exponentially decaying threat model is implemented with an exponential decay gradient which takes its value from the single most salient threat at any point.

```
(def exp-gradient (src d)
  (letfed ((n src (max (* d (reduce-nbrs n limsup)) src)))
    n))
```

Given a threat field, cumulative survival probabilities are calculated by

```
(def dist (p1 p2)
  (sqrt (+ (sq (- (1st p1) (1st p2)))
           (sq (- (2nd p1) (2nd p2))))))
(def l-int (p1 v1 p2 v2)
  (pow (/ (- 2 (+ v1 v2)) 2) (+ 1 (dist p1 p2))))
(def max-survival (dst v p)
  (letfed
      ((ps 0 (mux dst 1
                (reduce-nbrs
                  (* (l-int p v (local p) (local v)) ps)
                  limsup))))
    ps))
```

which relaxes out from the source to find the expected survival probability on the best path to the source (the **local** operator in the **reduce-nbrs** expression accesses the local value instead of the neighbor's value). Finally, **greedy-ascent** selects the best direction to move in at each point.

```
(def greedy-ascent (v coord)
  (- (2nd (reduce-nbrs (tup v coord) limsup)) coord))
```

Combining these three yields a threat-avoidance program:

```
(def avoid-threats (dst coords)
  (greedy-ascent
   (max-survival
    dst
    (exp-gradient (sense :threat) 0.8) coords) coords))
```

Figure 5 shows the threat avoidance program verified in simulation.

6 Approximating Amorphous Media

In order to execute Proto programs on a sensor network, we program the devices with a kernel that approximately simulates an amorphous medium. The main challenges for the kernel are approximating global evaluation with local code and continuous space with discrete devices. Despite the apparent complexity of Proto and the responsibilities of the kernel, our implementation runs on Mica2 Motes.

Global evaluation against the whole space must be approximated using code distributed through the network and executed locally on each device.

Rather than attempt to synchronize, each device executes independently once every t seconds by its own clock. So long as the program does not have any tight time dependencies, and can execute in less than t seconds per node, this provides a good rough approximation of periodic global evaluation.

Our implementation supports over the air programming. The entire amorphous medium runs a single program, so when any device is programmed, the kernel distributes the code virally, using a mechanism similar to those described in [17], [13], and [22]. To prevent conflicts during an upgrade process, each state broadcast also contains a version number, allowing devices to ignore state from different versions.

The devices of the sensor network can be viewed as a finite sample of the uncountably infinite devices in the amorphous medium. Thus, the neighborhood values are approximated by a node's own state plus a table of neighbor values similar to those used in [7] and [23].

Each operation that accesses neighbor state (e.g. **reduce-nbrs** or **nbr-range**) corresponds to a field in a state broadcast. Each device chooses a unique ID and broadcasts its state between evaluations. The broadcasts update table entries, and neighbors that haven't updated in several rounds are assumed

to have failed and are discarded.[7]

Finally, each quantifier for **reduce-nbrs** has a discrete implementation that is applied to the neighborhood values. The **limsup** and **liminf** summaries select the minimum and maximum neighborhood value, **forall** and **exists** take the AND and OR of neighborhood values, and **integral** sums the values in the neighborhood, weighted by the estimated area each represents.[8]

No formal guarantees of correctness are currently provided. It is up to the user to design programs that will tolerate noise caused by imperfections in the abstraction.

7 Conclusion

The amorphous medium abstraction is a powerful and practical tool for implementing sensor-network applications. Many sensor network problems become simpler when factored using this model, since the specified behavior of the space and the network operations to implement it are largely separated.

It is important, however, to consider the current limitations: there is much work remaining to be done, both in the theory of the amorphous medium, and in making Proto a useful platform for development.

It is an open question what types of abstractions are most intuitive for feedback control of spaces. Candidates in the form of distributed algorithms from amorphous computing and elsewhere need to be imported to Proto and analyzed within its context.

Finally, only a few applications have been ported to Proto at present. Extending the range of applications implemented will drive further Proto development and continue the process of determining the strengths and weaknesses of the amorphous medium abstraction.

References

[1] H. Abelson, D. Allen, D. Coore, C. Hanson, G. Homsy, T. Knight, R. Nagpal, E. Rauch, G. Sussman, and R. Weiss. Amorphous computing. Technical Report AIM-1665, MIT, 1999.

[2] J. Bachrach. Gooze: a stream processing language. In *Lightweight Languages 2004*, November 2004.

[3] J. Beal. Programming an amorphous computational medium. In *Unconventional Programming Paradigms International Workshop*, September 2004.

[4] J. Beal. Amorphous medium language. In *Large-Scale Multi-Agent Systems Workshop (LSMAS)*. Held in Conjunction with AAMAS-05, 2005.

[5] J. Beal and J. Bachrach. Infrastructure for engineered emergence in sensor/actuator networks. *IEEE Intelligent Systems*, pages 10–19, March/April 2006.

[6] J. Beal and G. Sussman. Biologically-inspired robust spatial programming. Technical Report AI Memo 2005-001, MIT, January 2005.

[7] W. Butera. *Programming a Paintable Computer*. PhD thesis, MIT, 2002.

[8] L. Clement and R. Nagpal. Self-assembly and self-repairing topologies. In *Workshop on Adaptability in Multi-Agent Systems, RoboCup Australian Open*, Jan. 2003.

[9] A. Eames. Enabling path planning and threat avoidance with wireless sensor networks. Master's thesis, MIT, June 2005.

[10] C. Elliott and P. Hudak. Functional reactive animation. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '97)*, volume 32(8), pages 263–273, 1997.

[11] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesc language: A holistic approach to networked embedded systems. In *Proceedings of Programming Language Design and Implementation (PLDI) 2003*, June 2003.

[12] R. Gummadi, O. Gnawali, and R. Govindan. Macro-programming wireless sensor networks using *airos*. In *DCOSS*, pages 126–140, 2005.

[13] J. W. Hui and D. Culler. The dynamic behavior of a data dissemination protocol for network programming at scale. In *Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 81–94. ACM Press, 2004.

[14] C. Intanagonwiwat, R. Govindan, and D. Estrin. Directed diffusion: a scalable and robust communication paradigm for sensor networks. In *Mobile Computing and Networking*, pages 56–67, 2000.

[15] S. P. Jones and J. Hughes. Report on the programming language haskell 98., 1999.

[16] S. R. Madden, R. Szewczyk, M. J. Franklin, and D. Culler. Supporting aggregate queries over ad-hoc wireless sensor networks. In *Workshop on Mobile Computing and Systems Applications*, 2002.

[17] J. McLurkin. Stupid robot tricks: A behavior-based distributed algorithm library for programming swarms of robots. Master's thesis, MIT, 2004.

[18] R. Newton and M. Welsh. Region streams: Functional macroprogramming for sensor networks. In *First International Workshop on Data Management for Sensor Networks (DMSN)*, Aug. 2004.

[19] J. Peterson, P. Hudak, and C. Elliott. Lambda in motion: Controlling robots with haskell. In *First International Workshop on Practical Aspects of' Declarative Languages (PADL)*, January 1999.

[20] S. Ratnasamy, B. Karp, L. Yin, F. Yu, D. Estrin, R. Govindan, and S. Shenker. GHT: a geographic hash table for data-centric storage. In *Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications*, pages 78–87. ACM Press, 2002.

[21] R. Shah and J. Rabaey. Energy aware routing for low energy ad hoc sensor networks, 2002.

[22] A. Sutherland. Towards rseam: Resilient serial execution on amorphous machines. Master's thesis, MIT, 2003.

[23] K. Whitehouse, C. Sharp, E. Brewer, and D. Culler. Hood: a neighborhood abstraction for sensor networks. In *Proceedings of the 2nd international conference on Mobile systems, applications, and services*. ACM Press, 2004.

[24] W. Ye, J. Heidemann, and D. Estrin. An energy-efficient mac protocol for wireless sensor networks. In *INFOCOM*, 2002.

[7] We have generally used a 5 round timeout.

[8] The simplest version, of course, is to weight each equally. The more information is known about the position of each node with respect to its neighbors, the better the estimate can be.