Region Streams: Functional Macroprogramming for Sensor Networks

Ryan Newton Matt Welsh

MIT & Harvard
Cambridge, MA
U.S.A.
newton@mit.edu & mdw@eecs.harvard.edu

Abstract

Sensor networks present a number of novel programming challenges for application developers. Their inherent limitations of computational power, communication bandwidth, and energy demand new approaches to programming that shield the developer from low-level details of resource management, concurrency, and in-network processing. We argue that sensor networks should be programmed at the global level, allowing the compiler to automatically generate nodal behaviors from a high-level specification of the network's global behavior.

This paper presents the design of a functional macroprogramming language for sensor networks, called Regiment. The essential data model in Regiment is based on region streams, which represent spatially distributed, time-varying collections of node state. A region stream might represent the set of sensor values across all nodes in an area or the aggregation of sensor values within that area. Regiment is a purely functional language, which gives the compiler considerable leeway in terms of realizing region stream operations across sensor nodes and exploiting redundancy within the network.

We describe the initial design and implementation of Regiment, including a compiler that transforms a macroprogram into an efficient nodal program based on a token machine. We present a progression of simple programs that illustrate the power of Regiment to succinctly represent robust, adaptive sensor network applications.

Proceedings of the First Workshop on Data Management for Sensor Networks (DMSN 2004),
Toronto, Canada, August 30th, 2004.
`http://db.cs.pitt.edu/dmsn04/`

1 Introduction

A sensor network represents a complex, volatile, resource-constrained cloud of sensors capable of collaborative sensing and computing. Programming such an entity requires new approaches to managing energy usage, performing distributed computation, and realizing robust behavior despite message and node loss.

One approach is to program the sensor network as a whole, rather than writing low-level software to drive individual nodes. Not only does such an approach raise the level of abstraction for developing novel programs, we argue that the only way to address the complexity of the underlying substrate is through automatic compilation from a high-level language. Today, few computer scientists would doubt the value of high-level languages for programming individual computers, or even groups of machines connected in a traditional network. We wish to take this approach to the next level and provide a *macroprogramming* environment for a network of sensors that automates the process of decomposing global programs into complex local behaviors.

This paper presents a functional macroprogramming language for sensor networks, called *Regiment*. The essential data model in Regiment is based on *region streams*, which represent spatially distributed, time-varying collections of node state. The programmer uses these to express interest in a group of nodes with some geographic, logical, or topological relationship, such as all nodes within k radio hops of some *anchor* node. The corresponding region stream represents the set of sensor values across the nodes in question. The operations permitted on region streams include *fold*, which aggregates values across nodes in the region to a particular anchor, and *map*, which applies a function over all values within a single region stream. Operationally, map requires no communication between elements, whereas fold requires the collapse of data to a single physical point.

Regiment is a *purely functional language* that does not permit input, output, or direct manipulation of program state. Regiment uses monads [19] to indirectly deal time-

varying values. As in other functional language designs, this approach gives the compiler considerable leeway in terms of realizing region stream operations across sensor nodes and exploiting redundancy within the network. The Regiment compiler transforms a network-wide macroprogram into an efficient nodal program based on a *token machine*. A token machine is a simple distributed state machine model in which nodes perform local sensing and computation in response to the arrival of named tokens, which may be received as radio messages or generated internally.

2 Related Work

We use the term macroprogramming to refer to programming the sensor network as a whole, rather than at the level of individual nodes. We argue that programming at this level leads to more concise and robust programs, since global behavior is specified directly. As an intuition, consider that matrix multiply algorithms are far simpler to state in terms of matrices and vectors than as parallel programs implemented in MPI.

For sensor networks, progress in macroprogramming has largely been domain specific. We have seen: languages for global-to-local compilation of spatial pattern formation [21, 17, 8]; Envirotrack [1], which exposes tracked objects as language *objects* (analogous to the way we expose regions); and, of course, database systems for querying sensor data [32, 18].

2.1 Middleware

There have been many attempts to design programming paradigms or run-time services to make application programming for sensor networks easier. These need not necessarily take a "macro" approach. In fact, many of these middleware developments are complementary to macroprogramming, and perhaps usable by a macroprogramming compiler backend. Spatial Programming [7] uses Smart Messages to provide content-based spatial references to embedded resources. For example, the programmer may refer to the first available camera in a given (predefined) spatial region. Other communication abstractions include GHT [25], DIFS [12], SPIN [13], DIMENSIONS [11], and HOOD [31]. Regiment draws on the Abstract Regions [30] model, which provides efficient communication primitives within *local regions* of the network.

2.2 Amorphous Computing

The Amorphous Computing research effort has pursued the broad goal of engineering aggregate behaviors for dense ad-hoc networks (paintable computers, Turing substrates). Their work focuses on pattern formation, taking inspiration from developmental biology. They demonstrate how to form coordinate systems [20], arbitrary two and three dimensional shapes [17], arbitrary graphs of "wires" [8], and origami-like folding patterns [21]. Yet the Amorphous Computing effort has not to date provided a model for *programming* rather than *pattern formation*. In addition, the target platforms envisioned by the Amorphous Computing effort differ significantly from existing wireless sensor networks.

2.3 Database approaches

The database community has long taken the view that declarative programming through a query language provides the right level of abstraction for accessing, filtering, and processing relational data. Recently, query languages have been applied to sensor networks, including TinyDB [18], Cougar [32], and IrisNet [22]. While these systems provide a valuable interface for efficient data collection, they do not focus on providing general-purpose distributed computation within a sensor network. For example, it is cumbersome to implement arbitrary aggregation and filtering operators and arbitrary communication patterns using such a query language. We argue that a more general language is required to fully realize the potential for global network programming.

There has also been a body of work on extending programming languages to deal with database access: database programming languages or DBPLs. Many types of languages have been used in this work, including functional ones. Functional DBPLs include FAD [5] and IPL [2]. Regiment differs from these languages in being explicitely concerned with: distributed processing, spatial processing, streaming data, and with the volatility of its substrate—sensor networks.

2.4 Stream processing languages

Stream processing is an old subject in the study of programming languages. Functional Reactive Programming (FRP) is a recent formulation which uses modern programming language technology (including monads [19] and type classes [28]) to allow purely functional languages to be able to deal comfortably with real time events and time-varying streams. FRP is the inspiration for Regiment's basic type system.

Regiment's problem domain also overlaps with recent work in extending databases to deal with continuous queries over streaming data, such as STREAM [3], Aurora [33], and Medusa [33]. Regiment aims to utilize many optimization techniques developed in this body of work, but at the same time Regiment occupies a slightly different niche—it is not only intrinsically *distributed* (on a volatile substrate) but explicitely *spatial*.

3 The functional macroprogramming approach

The traditional method of programming sensor networks is to write a low-level program that is compiled and installed in each individual sensor. This amounts to a programming model consisting of access to sensor data on the

local node, coupled with a message-passing interface to radio neighbors. In contrast, our macroprogramming model captures the entirety of the sensor network state as a global data structure. The changing state of each sensor originates a stream of data at some point in space. Collectively they form a global data structure.

To express sensing and communication within local groups of nodes, region streams encapsulate subsets of the global network state that can be manipulated by the programmer as single units. They represent the time-varying state of a time-varying group of nodes with some geographic or topological relationship. Communication patterns for data sharing and aggregation can be efficiently implemented within such local regions [30, 31].

3.1 Why a functional language?

We propose that functional languages are intrinsically more compatible with distributed implementation over volatile substrates than are imperative languages. Prominent (call-by-value) functional languages include Lisp, Scheme and OCaml. Functional languages have been used to explore high-level programming for parallel machines—such as NESL [6] and *LISP [26]—and for distributed machines [24]. In our system, we get the most benefit from restricting ourselves to a *purely* functional (effect free), call-by-need language similar to Haskell [16].

Purely functional languages essentially hide the direct manipulation of program state from the programmer. In particular, the program cannot directly modify the value of variables; rather, all operations must be represented as *functions*. Monads [19] allow mutable state to be represented in a purely functional form. For sensor network applications, abstracting away the manipulation of state allows the compiler to determine how and where program state is stored on the volatile mesh of sensor nodes. For example, to store a piece of data reliably, it may be necessary to replicate it across multiple nodes in some consistent fashion. Using a functional language makes consistency moot; immutable values can be freely replicated and cached.

Because functions are deterministic and produce no output, computation can be readily migrated or replicated without affecting program semantics. Another way to state this is that functional programs support *equational reasoning*. Program optimization in such a framework can be cast as semantics-preserving application of general program transformations [23].

Regiment has a host of algebraic properties which can be used together with a static cost model or dynamic profiling information to optimize performance and resource usage.

Another advantage of the functional programs is that it is straightforward to extract parallelism from their manipulation of data. For example, a function that combines data streams from multiple sensors can be compiled into a form that efficiently aggregates each data stream within the network. In addition to such *data parallel* operations, functional programs are implicitly parallel in their evaluation of function arguments [4]. The compiler can automatically extract this parallelism and implement it in a variety of ways, distributing operations across different sensor nodes.

4 The Regiment language

The goal of Regiment is to write complex sensor network applications with just a few lines of code. In this section we describe the Regiment language through several examples. A common application driver for complex coordination within sensor networks is that of tracking moving vehicles through a field of sensors each equipped with a proximity sensor of some kind (e.g., a magnetometer). We start by showing a simple Regiment program that returns a series of locations tracking a single vehicle moving through such a network.

```
let aboveThresh (p,x) = p > threshold
    read node =
       (read_sensor PROXIMITY node,
        get_location node)
in centroid (afilter aboveThresh
                     (amap read world))
```

We use a syntax similar to Haskell. Function applications are written as *f x y*; for example, `amap read world` represents the application of the *amap* function with two arguments: *read* and *world*. One important characteristic of functional languages is that they allow functions to be passed as arguments. Here, *amap* takes the function *read* as argument, and applies it to every value of the region stream *world*; we will discuss the details shortly. `afilter` filters out elements from a region stream that do not match a given predicate, in this case the *aboveThresh* function. And `centroid` is a function that computes the spatial center of mass of a set of sensor readings (where each reading is a scalar value coupled with the *(x,y)* location of the sensor that generated the reading). We assume that every node has access to an approximation of its Euclidean location in real space, though this assumption is not essential to the Regiment language.

So, this program can be interpreted as follows: a region stream is created that represents the value of the proximity sensor on every node in the network; each value is also annotated with the location of the corresponding sensor. Data items that fall below a certain `threshold` are filtered out. Finally, the spatial centroid of the remaining collection of sensor values is computed to determine the approximate location of the object that generated the readings.

4.1 Fundamentals: space and time

Regiment is founded on three abstract polymorphic data types. Polymorphic types are also called *generics*, and are similar in use to C++ templates; they enable generic data structures to be specialized for use with any particular type of data element. Below, the α argument to each type constructor signifies the particular type that it is specialized to hold.

- Stream α — represents a value of type α that changes continuously over time
- Space α — represents a physical space with values of type α suspended in it
- Event α — represents a discrete event that occurs at a particular point in time and that carries a value α when it occurs

The notion of Streams and Events is based on Functional Reactive Programming [10]. In this model, programs operate on a set of time-varying signals. A signal can change its behavior on the arrival of an event. In Regiment, signals become Streams and are used to represent changing sensor state or network status, Spaces represent the physical distribution of information across a network, and Events notify the program of meaningful changes to Streams, allowing triggers.

Because Regiment is a purely functional language, the Stream, Space, and Event types all describe first-class immutable values. This means that values of these types can themselves be passed as arguments, returned from functions, and combined in various ways. Semantically, we can think of each of the three types as having the following meanings:

- Stream $\alpha \approx$ Time $\rightarrow \alpha$
- Space $\alpha \approx$ Location $\rightarrow$ MultiSet α
- Event $\alpha \approx$ (Time , α)

That is, Streams may be formalized as abstract functions that map a time to the value at that time. This is not to say that we would ever *implement* a Stream object as such. Similarly, Spaces may be formalized as functions mapping a location to a set of values existing at that location. Events simply become tuples containing values paired with the associated time of their occurrence.

4.2 Areas, Regions, and Anchors

Until now, we have used "region stream" as an umbrella concept for a changing, distributed chunk of network state. Now we formalize this notion by introducing Regiment's Area and Region types. An Area is a generic data structure for representing volatile, distributed collections of data. A Region is a specific kind of Area used to represent the state of the real, physical network.

We saw before that a Space represents a "snapshot" of values distributed in space at a particular point in time. But we would like for those values—as well as the membership of values in that space—to change over time. To accomplish this we introduce the concept of an *Area*. If we visualize a `Space Int` as a volume with integers suspended throughout, then an `Area Int` would be an animated version of the same thing. The Area data type is built by using Stream and Space together:

Area α = Stream (Space α)

Note that, with this type, an Area's membership and physical extent may change over time. In fact, this type would allow the Area to become an entirely different Space at each point in time. (But the instability would cripple our implementation.) On the other hand, if Area were defined as a Space of Streams rather than Stream of Spaces, then its membership and spatial extent would be fixed but its values varying. Instead, both vary.

Areas are useful constructs, but they don't by themselves provide an initial foothold into the real world. How do we make that first Area? In order to refer to the state of specific sets of nodes in the real world, we define a *Region*, which is an *Area* of *Nodes*. A Node, in turn, is a datatype representing the state of a sensor node in the network at some point in time. It allows access to the node's state, such as its sensor readings, real world location, and the set of other nodes that are part of its communication neighborhood. The precise definition of the Node type, along with its basic operations, are shown in figure 1.

A Region is created as a group of nodes with some relationship to one another such as"all nodes within k radio hops of node N," or "all nodes within a circle of radius r around position X." Regions may be formed in arbitrarily complex ways: using spatial coordinates, network topology, or by arbitrary predicates applied to individual nodes. Hence, Regions may be non-contiguous in space, and their membership may vary over time. The goal of a Region is to get a handle on a group of sensor nodes of interest for the purpose of localizing sensing, computation, and communication within the network. The special region `world` represents all nodes in the network.

One can form a Region by identifying a particular node that acts as the reference point for determining membership in the region: an *Anchor*. The Anchor also acts as the "leader" for aggregate operations in a Region, such as combining values from multiple sensors. Note that the specific node that fulfills the role of Anchor may change over time, for example, if a node fails or loses connectivity to others in the Region. Regiment guarantees that the Anchor object persists across node failures, which may require periodic leader elections.

Examples of Regiment code for forming various Regions:

- **radio_neighborhood hops anch**:
  Forms a Region consisting of all nodes within **hops** radio hops of the given anchor.
- **circle radius anch**:
  Forms a Region consisting of all nodes whose geographical coordinates are within **radius** of **anch**.
- **knearest k anch**:
  Forms a Region consisting of the **k** nodes that are nearest **anch**.

4.3 Basic operations

Regiment defines a number of basic operations on Streams and Areas.

smap *f stream*
amap *f area*

smap applies a function f to every data sample within a Stream (across time), returning a new Stream. Similarly, *amap* applies a function f across every datum in the Area (across space and time).

afold *f init area*

An Area fold, or *afold*, is used to aggregate the samples from each location in the Area to a single value. The function f is used to combine the values in the Area, with an initial value of *init* used to seed the aggregation. *afold* returns a new Stream representing the aggregated values of the Area over time. For example, `afold (+) 0 area` generates a Stream of the time-varying sum of all values in *area*.

afilter *p area*

An Area filter, or *afilter*, pares down the elements of *area* to only those satisfying the predicate function p. This filtration must be updates dynamically as the values in *area* change over time.

Regiment also has operations for defining and handling events:

when *p stream*
whenAny *p area*
whenPercent *per p area*

when constructs an Event which fires when the current value of a stream satisfies the predicate p. *whenAny*, on the other hand, constructs an Event that fires whenever any single node in an Area matches a predicate p. *whenPercent* is similar to *whenAny* but the Event returned only fires when above a certain percentage of elements in the area meet the criteria—potentially an expensive (and difficult to implement) operation.

Using Events, two Streams can be sequenced into a single Stream using the *until* function:

until *event startstream handler*

until switches between Streams. The above call to *until* will produce values from *startstream* until such a time as *event* occurs. At that point, the *handler* (a function) is called on the value attached to the Event occurrence. This handler function must return a new Stream, that takes over producing values where *startstream* left off.

```
type Area a = Stream (Space a)
type Region = Area Node
type Anchor = Stream Node
  – Node: represents a physical mote in the context of a
  – communication network. Provides access to the node
  – state as well as the states of "neighbors".
type Node = (NodeState, [NodeState])

  – NodeState: all the relevent information for a
  – node: id, location, and a set of sensor values
  – (one for each sensor type supported by the node).
type NodeState = (Id, Location, [Sensor])
  – Sensor: force all sensor readings to be floats:
type Sensor = (SensorType, Float)
  – SensorType: predefined enumeration of sensor kinds.
type SensorType =
   PROXIMITY | LIGHT | TEMPERATURE ...

  – Function that returns the NodeState of a Node
get_nstate ::  Node -> NodeState
  – Returns the reading for a given SensorType. For
  – now we assume all nodes support all SensorTypes.
read_nstate ::
   SensorType -> NodeState -> Float

  – And here are two convenient short-hands:
  – Sensing function for Nodes
read_sensor typ nd =
   read_nstate typ (get_nstate nd)
  – Shorthand for reading location (via GPS, etc)
get_location nd =
   read_sensor LOCATION node
```

Figure 1: **Regiment's basic data types** (along with some helpful functions.)

4.4 Spatial operations

Along with these basic operators, Regiment provides several explicitly spatial operations on Areas. For example:

- **sparsify** *percent area*:
  Make *area* more sparse. Each value in the Area flips a biased coin, and stays in the Area with the given probability. This randomization is only done the first time a value enters the Area. The sparse Area is not chaotically recomputed at every time step. **sparsify** can be used, for example, to "weed out" nodes from an overly dense Region.

- **cluster** *area*:
  Cluster a fragmented Area into multiple Areas, each of which is guaranteed to be spatially contiguous. The return type is an Area of Areas.

- **flatten** *area*:
  Flatten takes an Area of Areas and returns a single combined Area. This is the inverse of **cluster**.

- **border** *area*:
  Return a Region representing the set of nodes that form a boundary around the given *area*.

4.5 Example programs

Now we will return to our original example program and examine it in greater detail. Let us start by defining `centroid` using basic Regiment constructs.

```
  – This calcs a weighted avg of vectors.
  – Used to find center of sensor readings.
centroid area =
   divide (afold accum (0,0) area)

  – 'accum' produces a weighted sum.
  – 'wsum' - sum of weights.
  – 'xsum' - sum of scaled locations.
accum (wsum, xsum) (w,x) =
   (w + wsum, x*w + xsum)

  – 'divide' the stream of scaled location
  – values by the sum of the weights.
  – Backslash defines a function.
divide stream =
   smap (\(w,x) -> x/w) stream
```

The `centroid` function takes an area as an input and uses the `accum` function to fold that area down to a stream of sums of sensor readings paired with the scaled locations of each sensor in the region. The `divide` function divides the sum of scaled locations by the sum of the sensor readings. This effectively calculates the center of mass of the locations of those sensors, in a way that recomputes automatically over time.

4.5.1 Tracking multiple targets

Using the **cluster** operation, we can track the location of multiple targets, assuming that the set of nodes near a given target do not overlap:

```
  let aboveThresh (p,x) = p > threshold
      read node =
         (read_sensor PROXIMITY node,
          get_location node)
      selected = afilter aboveThresh
                         (amap read world)
      globs = cluster selected
  in amap centroid globs
```

This program returns an Area providing approximate target locations for each target being tracked. Note that the number of targets in the Area will vary over time.

4.5.2 Resource efficiency with sentries

As a further refinement, consider a program that only initiates target tracking within the network if any of the nodes on the periphery of the network initially detect the presence of a target. This technique can be used to save energy on the interior nodes of the network, which only need to be activated once a target enters the boundary.

```
  let aboveThresh (p,x) = p > threshold
      read node = (read_sensor PROXIMITY node,
                   get_coords node)
      selected  = afilter aboveThresh
                          (amap read world)
      targets   = amap centroid (cluster selected)
      sentries  = amap read (border world)
      event     = whenAny aboveThresh sentries
      handler ev = targets
  in until event nullArea handler
```

The last line of the program initiates computation using the **until** primitive. Until `event` fires, the program returns an empty Area (**nullArea**). Once a target is detected by any of the `sentries`, the **nullArea** is supplanted by `targets`, the evaluation of which yields a stream of approximate target locations.

The reader might reasonably be worried that the above program produces a fragile implementation. If even one node in the sentry-border dies, might that let a target through? This depends on the quality of the implementation of the `border` operator. A high quality implementation will respond to failures and have the border sealed again in a bounded amount of time. Also, the programmer may self-insure by making a two layer border as follows:

```
  let sent1 = border world
      sent2 = border (subtract world sent1)
      thickborder = union sent1 sent2
      ...
```

4.5.3 Contour finding

The following program computes the *contour* between adjacent areas of the network. Sensor readings on one side of the contour are above a certain threshold, and readings on the other side are below. The contour is returned as a list of points lying along the contour.

```
  let mesh = planarize world
      nodesAbove =
         afilter ((>= threshold) .
                  (read_sensor SENSTYP))
                 mesh
      midpoint nst1 nst2 =
         (read_nstate LOCATION nst1 +
          read_nstate LOCATION nst2) / 2
      contourpoints node =
         let neighborsBelow =
           filter ((< threshold) .
                   (read_nstate SENSTYP))
                  (get_neighbors node)
         in map (midpoint (get_nstate node))
              neighborsBelow
      all_contourpoints =
         amap contourpoints nodesAbove
  in
    afold append all_contourpoints
```

This program works by pruning the communication graph of the network into an approximately planar form. It then filters out a region of nodes—`abovethresh`—with `SENSTYP` reading above the threshold; this would be all the nodes to one side of the contour. The `contourpoints` function takes a node above the threshold and returns a list of midpoints between that node and each of its neighbors *below* the threshold (on the other side of the contour). Finally, *all_countourpoints* is aggregated by appending together all the individual lists of midpoints, thus yeilding the final countour-line—a Stream of lists of coordinates.

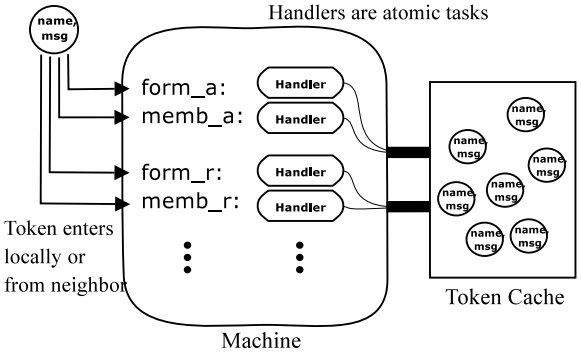

Figure 2: **The Regiment Token Machine model.**

4.6 Feedback and exception handling

Because behavior of the sensor network is stochastic, the response from a region during any time period will involve only a subset of all the nodes that "should" be in that region. The programmer needs feedback on the quality of communication with the region in question. Thus the **fidelity** operator.

fidelity *area*

This operator returns a Stream representing the fidelity of an area as a number between zero and one (an approximation based on the number of nodes responding, spatial density, and estimated message loss).

The programmer will also want feedback about (and eventually control over) the *frequency* of a Stream.

get_frequency *stream*

allows the programmer to monitor the actual frequency of a Stream of values.

Thus, by using these two diagnostic streams, the programmer may set up "exception handlers". This is accomplished by constructing events which fire when fidelity or frequency falls out of the acceptable range. For example, if fidelity drops below a certain level, one may want to switch to a different algorithm.

5 Token Machines

Compiling a global program into node-level code requires an abstract machine model for the compiler to target. The goal of this model is to capture only the essential operations supported by sensor nodes. For this purpose we provide the *Token Machine* (depicted in figure 5). It can be thought of as an intermediate language (IL) between Regiment and the native language and runtime environment supported by individual sensor nodes.

5.1 Tokens and Handlers

A program in the Token Machine model consists of a collection of *token handlers* coupled with local state definitions. Each token handler is associated with a token name and is attached to an atomic task to be executed by a sensor node upon receiving a token matching that name. The Token Machine's concurrency model is similar to TinyOS [14] in that handler tasks may not be blocked or preempted, and run to completion. In many ways, the token machine model is similar to that of Active Messages [27]

Tokens are generated by a node either internally (in response to internal state changes, e.g., a timer interrupt) or by reception of a radio message containing the token identifier and parameters. The most recently received token of each name is cached by the machine. Token handlers can *emit* new tokens by broadcasting a radio message, or *call* local token handlers. Nodes can also *count* the number of times a given token has been received and *clear* the reception count for a given token. Thus Token Machines provide a simple mechanism providing local function calls, remote invocation, and data storage.

One use of tokens is to implement *gradients* [9]. A gradient emanates from a specific origin node with an associated *gradient value*, which is initialized to zero. Each node receiving a gradient token rebroadcasts the token after incrementing the gradient value; each node retains only the lowest-numbered gradient value it has received. A gradient may have an associated *time-to-live* that limits the range of its propagation. Gradients can be used to implement a range of interesting communication patterns, for example, allowing a root node to collect information from all nodes within some communication radius, or allowing nodes to estimate their distance from a set of origin points.

In practice, gradients must be refreshed continuously to maintain themselves in the presence of node and link failures. The epoch frequency for gradient-refresh drives the looping behavior of the system. Every epoch, a wave of tokens moves outward, activating the next step of computation. Gradients can be seen as a more general form of the communication model used by directed diffusion [15] and spanning trees in systems such as TinyDB [18].

5.2 Gradient example: implementing folds

As an example of the use of tokens and gradients, consider aggregating the values of a k-radio-neighborhood group of sensors to an anchor node in its center (the Regiment **afold** operator). This operation proceeds in two steps: region formation (which may be amortized over multiple **afold** operations) and data aggregation. To form the region, the anchor emits a *member* gradient with an initial hopcount of 0 and a time-to-live of k radio hops. The token handler for *member* evaluates whether the receiving node is within the region defined by the **afold** operator; in this case, if the gradient hopcount is less than k, then that node considers itself part of the region. Receiving nodes also remember the node from which they received the lowest hopcount version of the *member* token; call this the node's *parent*. Receiving nodes then increment the gradient hopcount and relay the gradient as long as the hopcount is less than the time-to-live.

Aggregating results back to the anchor is performed

with a second gradient operation, called *return*. *return* takes as arguments a local value to aggregate, as well as a token naming an *aggregation function* that combines values as they travel upwards toward the root of the *member* gradient. Each interior node attempts to keep track of the number of children it has. The handler for *return* checks whether all of the node's children have responded with their own *return* token (using the token's reception count), up to some maximum timeout period. Once this condition is met, the *return* handler combines received values from the node's children using the aggregation function and issues a *return* to its own parent with the combined value. These *return* messages include the node's parent ID so that all other nodes will ignore its reception.

5.3 Gradient example: leader election

Gradients also make it straightforward to implement distributed leader election among a group of nodes. All of the nodes participating in the election emit a gradient named *elect*, which includes its local node ID. The token handler on each node remembers the lowest-valued node ID received so far. When the token is received, if the received ID is smaller than the previously stored value, the new ID is remembered *and the gradient token is relayed*. Therefore, all nodes participating in the election initiate gradients, but only the gradient of the lowest-numbered node will continue to propagate. The root of this gradient is the leader.

6 Current status

Regiment poses implementation challenges that are both deep and broad. Presently, we are exploring the feasibility of the basic Regiment primitives through a highly restricted subset of the language. This subset eliminates general purpose function application, and forbids free variables within functions (disallowing closures). Functions may still be defined, but they may only be applied by using the Regiment primitives **afilter**, **afold**, **amap**, and **smap**. We have also postponed typing issues by making our prototype dynamically typed. We have implemented a prototype of the compiler and have demonstrated several example applications running in simulation; we intend to implement a back end compiler to generate TinyOS code from the Token Machine representation, allowing us to test the system on real sensor nodes.

6.1 Compilation strategy

A program in the restricted language is best visualized as a dataflow graph; Figure 3 depicts the graph for a simple program that computes the **smap** of a function g over the **afold** of f over a region of nodes defined by a circle around the point $(30, 40)$. Our compiler generates code for such a data-flow graph by directly translating each *edge* in the graph into some number of token handlers.

Values in the system are divided into *distributed* and *local*. Every distributed value corresponds to some phenomena happening in space. Regions and Anchors are distributed, whereas numbers and functions are local. In figure 3, edges are either solid or dotted depending on whether they carry distributed values or local ones.

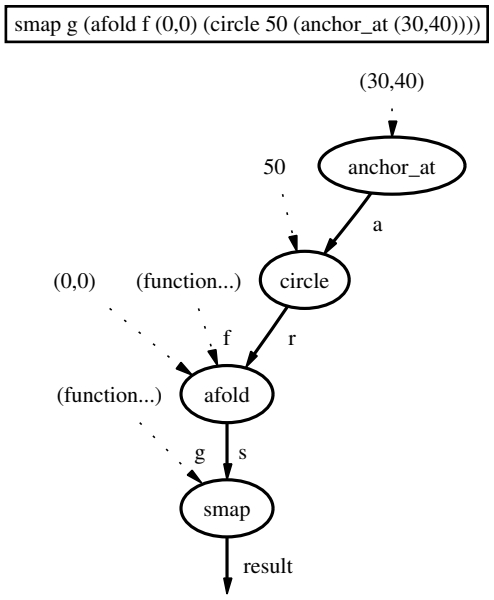

Figure 3: **A Regiment program represented as a dataflow graph.** *Network-distributed values flow along solid edges, and local constant values (including functions) flow along dotted edges.*

We standardize an interface among distributed values such that every distributed value (solid edge) produces at least a *formation* token handler and a *membership* token handler. The former represents an onus to create that Region or Anchor—form the circle, do the filtration, elect the leader—and the latter is a notice that the Area/Anchor is active and the current node is participating in it.

6.1.1 Example walk-through

The example portrayed in figure 3 has four distributed values (**a, r, s, result**) and several local values (**f** and **g** and several numeric constants). Each of the distributed values generates both a formation and membership token, for example, *form_a* and *memb_a*.

Because **a** is the only distributed value produced by a leaf node, *form_a* tokens are seeded into the network initially. They cause nodes to check their distance from the targeted Euclidean coordinate, $(30, 40)$. Nodes that are close enough to that location initiate and participate in a leader election. The token *memb_a* is fired when a node becomes leader.

Because **r** is the next step in the chain beyond **a**, the handler for the *memb_a* token immediately calls *form_r*. Forming **r** is simple; it just requires emitting a single gradient with the token *memb_r*. As nodes receive the *memb_r* token they call the *form_s* token. (Again, simply because it's next in line.) The *form_s* handler begins *return*ing values along the back-trail of the *memb_r* gradient. When they arrive

back at the root, the special return handler calls *memb_s*. The value **s** has successfully been formed. *memb_s* in turn calls *form_result*, which simply applies the function **g** locally to the stream **s**, and we have our result.

This example was simple—at no point did a primitive depend on more than one distributed value. But we hope that it conveys a feeling for the process. It is important to note also that this simple example uses only a push model for the data-flow graph. (Leaf nodes push their results down to the root.) The `until` primitive makes necessary use of the `pull` communication model because it waits for an Event before starting a Stream. The latter stream must have some kind of pull exerted on it to prompt it to begin execution.

7 Future work and Conclusion

Future work will proceed in several directions. Because of the large gap between Regiment's semantics and target architecture, compiling it is a challenge. We will explore the possibility of loosening the restrictions on our initial version of Regiment, providing more general purpose functionality. We plan to investigate both static and dynamic optimizations in terms of resource usage and communication bandwidth requirements for a range of Regiment applications. We believe that the Token Machine model and the use of gradients makes it straightforward to realize good communication locality. We intend to introduce primitives that allow the user to control tradeoffs between energy, latency, and accuracy [29], which are critical for sensor network application designers to consider.

Our vision is that sensor network applications can be expressed in a very high-level *macroprogramming* language that abstracts away the low-level details of sensing, communication, and energy management. We argue that the use of functional programming languages is essential for capturing data parallelism and enabling the compiler to make informed decisions about the scheduling and placement of computation in the sensor network. We have demonstrated some interesting first steps in this direction through the design of Regiment and its underlying runtime model, Token Machines. Regiment provides the ability to programmatically build spatial regions within the network, and use them for localized sensing, computation, and communication.

References

[1] T. Abdelzaher, B. Blum, Q. Cao, D. Evans, J. George, S. George, T. He, L. Luo, S. Son, R. Stoleru, J. Stankovic, and A. Wood. Envirotrack: Towards an environmental computing paradigm for distributed sensor networks.

[2] J. Annevelik. Database programming languages: A functional approach. In *Proc. of the ACM Conf. on Management of Data*, pages 318–327, 1991.

[3] A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, R. Motwani, U. Srivastava, and J. Widom. *STREAM: The Stanford Data Stream Management System*. 2004.

[4] Arvind and Rishiyur Nikhil. *Implicit Parallel Programming in pH*. Morgan Kaufman, 2001.

[5] F. Bancilhon, T. Briggs, S. Khoshafian, and P. Valduriez. Fad, a powerful and simple database language. In *Proc. Conf. on Very Large Data Bases (VLDB)*, 1987.

[6] Guy E. Blelloch. NESL: A Nested Data-Parallel Language. Technical Report CMU-CS-93-129, April 1993.

[7] Cristian Borcea, Chalermek Intanagonwiwat, Porlin Kang, Ulrich Kremer, and Liviu Iftode. Spatial programming using smart messages: Design and implementation. In *24th International Conference on Distributed Computing Systems (ICDCS 2004)*, March 2004.

[8] Daniel Coore. *Botanical Computing: A Developmental Approach to Generating Interconnect Topologies on an Amorphous Computer*. PhD thesis, MIT Department of Electrical Engineering and Computer Science, February 1999.

[9] Daniel Coore, Radhika Nagpal, and Ron Weiss. Paradigms for structure in an amorphous computer. Technical Report AIM-1614, MIT, 6, 1997.

[10] Conal Elliott and Paul 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] Deepak Ganesan, Ben Greenstein, Denis Perelyubskiy, Deborah Estrin, and John Heidemann. An evaluation of multi-resolution search and storage in resource-constrained sensor networks. In *Proc. the First ACM Conference on Embedded Networked Sensor Systems (SenSys 2003)*, November 2003.

[12] Benjamin Greenstein, Deborah Estrin, Ramesh Govindan, Sylvia Ratnasamy, and Scott Shenker. DIFS: A distributed index for features in sensor networks. In *Proc. the First IEEE International Workshop on Sensor Network Protocols and Applications*, May 2003.

[13] Wendi Heinzelman, Joanna Kulik, and Hari Balakrishnan. Adaptive protocols for information dissemination in wireless sensor networks. In *Proc. the 5th ACM/IEEE Mobicom Conference*, August 1999.

[14] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David E. Culler, and Kristofer S. J. Pister. System architecture directions for networked sensors. In

Proc. the 9th International Conference on Architectural Support for Programming Languages and Operating Systems, pages 93–104, Boston, MA, USA, November 2000.

[15] Chalermek Intanagonwiwat, Ramesh Govindan, and Deborah Estrin. Directed diffusion: A scalable and robust communication paradigm for sensor networks. In *Proc. International Conference on Mobile Computing and Networking*, August 2000.

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

[17] Attila Kondacs. Biologically-inspired self-assembly of 2d shapes, using global-to-local compilation. In *International Joint Conference on Artificial Intelligence (IJCAI)*, 2003.

[18] Samuel Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. TAG: A Tiny AGgregation Service for Ad-Hoc Sensor Networks. In *Proc. the 5th OSDI*, December 2002.

[19] Eugenio Moggi. Computational lambda-calculus and monads. In *Proceedings 4th Annual IEEE Symp. on Logic in Computer Science, LICS'89, Pacific Grove, CA, USA, 5–8 June 1989*, pages 14–23. IEEE Computer Society Press, Washington, DC, 1989.

[20] Nagpal, Shrobe, and Bachrach. Organizing a global coordinate system from local information on an ad hoc sensor network. In *2nd International Workshop on Information Processing in Sensor Networks (IPSN '03)*, April 2003.

[21] Radhika Nagpal. *Programmable Self-Assembly: Constructing Global Shape using Biologically-inspired Local Interactions and Origami Mathematics*. PhD thesis, MIT Department of Electrical Engineering and Computer Science, June 2001.

[22] Suman Nath, Yan Ke, Phillip B. Gibbons, Brad Karp, and Srinivasan Seshan. IrisNet: An architecture for enabling sensor-enriched Internet service. Technical Report IRP-TR-03-04, Intel Research Pittsburgh, June 2003.

[23] Simon L. Peyton Jones and André L. M. Santos. A transformation-based optimiser for Haskell. volume 32, pages 3–47, 1998.

[24] Pointon, Trinder, and Loidl. The design and implementation of Glasgow Distributed Haskell. *Lecture Notes in Computer Science*, 2001.

[25] 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 sensornets. In *Proc. the First ACM International Workshop on Wireless Sensor Networks and Applications (WSNA)*, Atlanta, Georgia, September 2002.

[26] G. L. Steele and W. D. Hillis. Connection machine lisp: Fine grained parallel symbolic programming. pages 279–297.

[27] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active messages: a mechanism for integrating communication and computation. In *Proc. the 19th Annual International Symposium on Computer Architecture*, pages 256–266, May 1992.

[28] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad-hoc. In *Conference Record of the 16th Annual ACM Symposium on Principles of Programming Languages*, pages 60–76. ACM, January 1989.

[29] Matt Welsh. Exposing resource tradeoffs in region-based communication abstractions for sensor networks. In *Proc. the 2nd ACM Workshop on Hot Topics in Networks (HotNets-II)*, November 2003.

[30] Matt Welsh and Geoff Mainland. Programming sensor networks using abstract regions. In *Proc. the First USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI '04)*, March 2004.

[31] Kamin Whitehouse, Cory Sharp, Eric Brewer, and David Culler. Hood: A neighborhood abstraction for sensor networks. In *Proc. the International Conference on Mobile Systems, Applications, and Services (MOBISYS '04)*, June 2004.

[32] Yong Yao and J. E. Gehrke. The Cougar approach to in-network query processing in sensor networks. *ACM Sigmod Record*, 31(3), September 2002.

[33] S. Zdonik, M. Stonebraker, M. Cherniack, U. Cetintemel, M. Balazinska, and H. Balakrishnan. The aurora and medusa projects. *Bulletin of the Technical Committee on Data Engineering*, 2001.