The Regiment Macroprogramming System

Ryan Newton
MIT
newton@mit.edu

Greg Morrisett
Harvard University
greg@eecs.harvard.edu

Matt Welsh
Harvard University
mdw@eecs.harvard.edu

Abstract

The development of high-level programming environments is essential if wireless sensor networks are to be accessible to non-experts. In this paper, we present the Regiment system, which consists of a high-level language for *spatiotemporal macroprogramming*, along with a compiler that translates global programs into node-level code. In Regiment, the programmer views the network as a set of spatially-distributed data streams. The programmer can manipulate sets of these streams that may be defined by topological or geographic relationships between nodes. Regiment provides a rich set of primitives for processing data on individual streams, manipulating regions, performing aggregation over a region, and triggering new computation within the network.

In this paper, we describe the design and implementation of the Regiment language and compiler. We describe the *deglobalization* process that compiles a network-wide representation of the program into a node-level, event-driven program. Deglobalization maps region operations onto associated spanning trees that establish region membership and permit efficient in-network aggregation. We evaluate Regiment in the context of a complex distributed application involving rapid detection of spatially-distributed events, such as wildfires or chemical plumes. Our results show that Regiment makes it possible to develop complex sensor network applications at a global level.

Categories and Subject Descriptors:
D.3.2 Concurrent, distributed, and parallel languages; Applicative (functional) languages; Data-flow languages
C.2.4 Distributed Systems

General Terms: Design, Languages

Keywords: functional macroprogramming, sensor networks

1. Introduction

Programming complex coordinated behaviors in sensor networks is a difficult task. When programming at the sensor node level, developers must concern themselves with low-level details of radio communication, sensing, buffer management, and concurrency, typically under severe constraints on resource usage. One approach that has been proposed to address these problems is *macroprogramming*, which involves programming the sensor network at the global level, rather than individual sensor nodes. In a macroprogramming system, the global network program is automatically translated into a node-local program.

This paper presents *Regiment*, a macroprogramming language and runtime environment based on the concept of functional reactive programming (FRP) [5]. In Regiment, the programmer views the network as a set of spatially-distributed, time-varying signals, each representing either the state of an individual sensor node (e.g., sensor readings or the results of local computation) or an aggregate taken across *regions* of sensor nodes. Regions may be defined in terms of geographic area, network topology, or functional capability (e.g., all of the climate sensors within a given area). Regiment provides a rich set of primitives for processing data on individual signals, manipulating regions, performing aggregation over signals in a region, and triggering new computation within the network.

A Regiment macroprogram compiles down to a node-level intermediate language called *token machines* (TMs). A token machine program provides basic facilities for local computation, sampling, and radio communication, as well as the ability to construct *gradients* within the network for region formation and aggregation. In contrast to declarative query languages such as TinyDB [10], Regiment offers a more complete, flexible programming environment that supports complex in-network processing, triggered execution, and coordination within local regions of the network.

In previous work, we presented an early design of the Regiment language [14] and the TM interface [13]. In this paper, we significantly extend this previous work by describing the complete Regiment macroprogramming environment, a compiler that translates Regiment code to TM programs, and a runtime system that supports the global coordination primitives in the Regiment language. In addition, we present a performance evaluation of Regiment in the context of a complex distributed application involving rapid detection of spatially-distributed events such as wildfires or chemical plumes. Our results show that sophisticated macroprograms written in Regiment can be compiled to efficient node-level code.

The rest of this paper is organized as follows. Section 2 presents the background and motivation for this work, with an emphasis on existing approaches to sensor network macroprogramming. Section 3 gives a brief overview of the Regiment language and Section 4 presents an example application. Section 5 describes the Regiment compiler and deglobalization techniques in detail. Section 6 presents a detailed performance evaluation of several variants of the event-detection program in simulation, focusing on detection latency and communication overheads. Finally, Section 7 describes future work and concludes.

2. Background and Motivation

The goal of *macroprogramming* systems is to provide a high-level programming model for sensor networks that abstracts away the details of individual sensor nodes. Rather, an application developer writes code that describes the operation of the network as a whole, which is then compiled down to a node-level program. Macroprogramming can greatly simplify application design, making it possible for non-sensor-networking experts to directly develop complex distributed programs.

2.1 Spatiotemporal Macroprogramming

A range of macroprogramming models have been proposed in the literature [16, 17, 6, 18, 10], which are each designed to support a particular style of network-wide programming. For example, TinyDB [10] is focused on data collection with a limited form of in-network aggregation, while EnviroSuite [9] has special support for tracking applications.

Regiment is designed to support an important class of sensor node applications that we call *spatiotemporal macroprograms* (STMPs). STMPs are distinguished by exhibiting significant spatial and temporal structure. That is, STMPs are typically concerned with nodes' locations and their topological relationships, with geographical localities in real space, as well as with time-varying sensor data and computational state. Common idioms in STMPs include local communication amongst neighborhoods in the network; the use of sensor data and location to estimate properties of a field; and exploiting redundancies between sensor nodes to mitigate the effects of failure. Good examples of STMPs include estimation of a gradient or contour in the sensor field [7], or the use of local neighborhood communication to estimate the location of a target vehicle [17].

STMPs describe a wide range of sensor network applications, but it is important to point out that this model is by no means universal. In contrast, non-spatiotemporal applications include those without regard for specific sensor locations or which consider the capabilities of nodes, rather than their positions, as primary. Examples of non-STMPs also include agent-based models in which code migrates between nodes.

2.2 Macroprogramming languages

Among the first macroprogramming environments were database-inspired declarative query languages in which the user describes sensor data of interest using an SQL or XML query. TinyDB [10], Cougar [19], and IrisNet [11] are examples of this approach. Query languages offer an extremely high-level interface to the sensor network, and provide exactly the right solution for applications that need to retrieve data from the whole network, or compute an aggregate over the whole network. They are, however, limited in that they do not offer much flexibility for introducing application-specific logic. In TinyDB, for example, a developer could implement new query operators, but doing so requires extensive modifications to the parser and query engine.

Kairos [6] provides an SPMD-style programming model in which parallel computation over a set of sensor nodes is represented as a loop that iterates over members of the set. EnviroSuite [9] is a language and runtime environment with special support for tracking applications. Abstract Task Graphs [1] offers a dataflow programming model with a graphical composition language, while Semantic Streams [18] use a logic program to compose services hosted on different sensor nodes into a dataflow.

One of the key challenges in defining a macroprogramming environment is resolving the tension between ease-of-use, efficiency, and providing adequate flexibility for expressing powerful distributed computations. Indeed, the wide range of proposals for macroprogramming systems suggests that there is no universal model for macroprogramming, and different application domains will involve very different tradeoffs along these axes. Regiment aims to significantly extend the functionality of declarative query interfaces by supporting a rich set of primitives for computation and communication at the node, region, and global levels.

Of course, this degree of abstraction will always come with some overhead as compared to highly-tuned, hand-coded systems. But we believe that these overheads can be minimized to the point that macroprogramming is viable for the majority of applications. Further, we will demonstrate that rapidly prototyping applications—embodying drastically different communication strategies—can be accomplished *within the macroprogramming environment*.

3. Overview of Regiment

In this section, we provide an overview of the Regiment language and programming model. In a previous workshop paper [14], we presented an earlier form of the Regiment language, which has since evolved substantially. Thus, we offer the following summary.

Regiment is based on the concept of *functional reactive programming* (FRP) [5]. Sensor network state is represented as time-varying *signals*. Signals might represent sensor readings on an individual node, the state of a node's local computation, or aggregate values computed from multiple source signals. Regiment also supports the notion of *regions*, which are spatially distributed signals. An example of a region is the set of sensor readings from nodes in a given geographic area. Regiment abstracts away the details of sensor data acquisition, storage, and communication from the programmer, instead permitting the compiler to map global operations on signals and regions onto local structures within the network.

3.1 Regiment Language Basics

Signals: In Regiment, the principal objects that the programmer manipulates are *signals*. For example, a temperature sensor on a given node, which returns a floating-point value, has type `Signal(float)`. Conceptually, a signal is a function that maps a time t to a value v, but in practice the time t will always be "now" and the signal must be sampled to produce a discrete stream. A signal can carry primitive values (such as integers, floats, etc.) or tuples, such as records containing both the light and temperature sensor readings on a given node.

Regiment provides a number of operations for manipulating and building new signals out of existing signals. For example, `smap` applies a function to each element of a signal, returning a new signal. Thus, the code fragment:

```
smap(fun(t) {t * 10.0}, tempvals)
```

converts a sensor's floating-point signal `tempvals` to a new signal with each temperature value multiplied by ten.

Regions: Central to Regiment is the concept of a *region*, which represents a collection of signals. Part of the job of the Regiment compiler is to enable the user to treat the region as containing a "snapshot" of the values of its constituent signals, while implementing operations on the region in a distributed fashion.

It is important to note that membership in a region may vary with time; for example, the region defined as "temperature readings from nodes where temperature is above a threshold" will consist of values from a varying set of nodes over time. The collection of signals that participate in a region can also vary due to node or communication failures or the addition of nodes in the network. One of the advantages of programming at the region level is that the application is insulated from the low-level details of network topology and membership.

There are three key operations on regions: `rmap`, `rfilter`, and `rfold`. `Rmap` is the region-level version of `smap` and applies a function to each value in the region. For instance,

```
outreg = rmap(fun(x) {x / SCALEFACTOR}, inreg)
```

divides each value in the region `inreg` by a scaling factor.

`Rfilter` applies a predicate function to each sample in an input region to determine whether it should remain in the output region. For example, one can use `rfilter` to filter a region of numbers down to only those numbers above a threshold:

```
newreg = rfilter(fun(n) {n > THRESH}, oldreg)
```

Finally, `rfold` is used to aggregate the values in a region into a single signal using an associative and commutative combining function. For example,

```
sumsignal = rfold((+), 0, inputregion)
```

produces an output signal, `sumsignal`, that consists of the sum of the values in the input region. `Rfold` also takes as arguments a combining function and an initial value; here, the combining function is $+$ while the initial value is 0. The system reserves the right to fold in the initial value as many times as it likes, so it should generally be a neutral element for the combining function.

Regiment hides the details of how the aggregation is performed at runtime. As we will describe in Section 5, `rfold` is implemented using a spanning tree rooted at the node consuming the output signal of the `rfold` operation; however, it would be semantically equivalent (though less efficient) to implement `rfold` by shipping all of the input data items to a single node and then performing the aggregation there.

3.2 Node objects

A `Node` is an object in Regiment that provides access to the state of an individual sensor node. A `Node` consists of both *static* information about the node, such as its unique ID and location, as well as *dynamic* information, such as sensor readings. Regiment provides primitive operations for accessing the elements of a `Node` object; for example, `nodeid(n)` returns the ID of node `n`, while `sense("temp",n)` samples the temperature value of node `n`.

To avoid high communication overheads, Regiment enforces the rule that the dynamic components of `Node` objects are only accessible on the physical sensor node that corresponds to the `Node`. This implies that computations that operate on dynamic `Node` state must run locally on that node; for example, `sense("temp",node)` can only be executed locally on the sensor corresponding to `node`. The Regiment compiler ensures this by forbidding values of type `Node` from occurring in places that will require them to be sent across the network, such as the input to a `gossip` primitive, or in the aggregate accumulator parameter of an `rfold`. The user must first project the sensor readings of interest from the `Node` before attempting network communication.

In future versions of Regiment, we will either remove this restriction (for example, by implementing just-in-time routing of dynamic node state), or introduce a refinement in the type system to make clear to the user the distinction between *portable* network code and *pinned* code.

3.3 Region formation primitives

Thus far we have described primitives for transforming existing regions, such as `rmap` and `rfilter`. Regiment also provides operators for forming new regions based on spatial and topological relationships between nodes. For example, the code fragment below forms a `Region(Node)` within two radio hops of the given `node`.

```
nbrhood = khood(2, node)
```

There are two major categories of region formation primitives in Regiment. The first category encompasses operators that grow a region starting from a single anchor node. This includes `khood`, as well as `circle`, which creates a region of nodes within a given physical distance from a node, and `knearest`, which creates a region containing the k geographically nearest nodes to a given node. All of these primitives are implemented with a bounded flood that sets up a spanning tree within the network. These region formation operators are similar to those in Hoods [17] and abstract regions [16].

The second category consists of *gossip* based primitives. These primitives do not depend on spanning trees. Instead, they provide the Regiment programmer with access to simple one-hop radio broadcast. The most basic, `gossip`, takes a region of scalar values and shares this data with all nodes in single-hop radio range of any node in the input region. This is accomplished through a simple radio broadcast of all samples in the region. The new region contains signals of all overheard broadcasts on all nodes within the one-hop closure of the input region.

`Table_gossip` is a more user-friendly gossip primitive that aggregates the data overheard from all neighbors into a single table.[1] All nodes overhearing at least one broadcast cache the data received from their neighbors in a *table* indexed by node id. Unlike basic `gossip`, `table_gossip` imposes an epoch discipline; it periodically snapshots, flushes, and returns the tables' state. These table snapshots, taken together, make up the output region returned by `table_gossip`.

To summarize the language thus far, we have two key types, `Signal` and `Region`, and a small set of key operations on values of these types.

smap ::	$(\alpha \to \beta)$, Signal(α) $\to$ Signal(β)
rmap ::	$(\alpha \to \beta)$, Region(α) $\to$ Region(β)
rfilter ::	$(\alpha \to$ bool), Region(α) $\to$ Region(α)
rfold ::	$(\alpha, \beta \to \beta)$, β, Region(α) $\to$ Signal(β)
khood ::	Int, Node $\to$ Region(())
gossip ::	Region(α) $\to$ Region(α)
table_gossip ::	Region(α) $\to$ Region(List(Int, α))

3.4 Regiment program structure

A Regiment program consists of a set of function and variable bindings followed by a single *wiring* statement of the form "*BASE* $\leftarrow$ *expression*". The signal or region returned to *BASE* determines what will be sent from the network to the sensor network's base station. Conversely, the Regiment program acquires its *input* through the special region, `world`, which represents the entire network and has type `Region(Node)`. `Node` is an abstract type that provides access to an individual node's state (as described below).

A complete program: The following complete Regiment program computes the average temperature across the sensor network.[2] Note that the type-annotation for `dosum` is purely for documentation, the compiler is capable of inferring all types.

```
dosum :: float, (float, int) → (float, int)
fun dosum(temp, (sumtemp, count)) {
  (sumtemp+temp, count+1)
}

tempreg = rmap(fun(nd){sense("temp",nd)}, world);
sumsig = rfold(dosum, (0,0), tempreg);
avgsig = smap( fun((sum,cnt)) {sum / cnt},
               sumsig);
BASE ← avgsig
```

[1] `table_gossip` can be implemented from a simpler `gossip` primitive if the language includes an `rintegrate` operator (`rmap` that locally maintains state between invocations). `Rintegrate` is omitted from this paper for simplicity.

[2] No paper on macroprogramming would be complete without this classic program!

The program starts by applying `rmap` to `world` to obtain a region of temperature readings for all sensor nodes. It then uses `rfold` to aggregate this region into a signal called `sumsig`. Each value of `sumsig` is a tuple consisting of two components: a sum of the temperature readings across the network, and a count of the number of nodes contributing to the sum. Aggregation is accomplished with the help of the combining function `dosum`, which takes as input a temperature value (one element from the `tempreg` region) and an accumulator (the tuple (`sumtemp`, `count`)). `Dosum` returns an updated accumulator value that includes the new temperature with the count incremented by one. Finally, `smap` is applied to the resulting signal to divide the temperature sums by the counts, resulting in a signal of averaged temperatures (`float` values), `avgsig`. `Avgsig` is the signal returned to *BASE*.

While the above average temperature program is somewhat more involved than, say, a two-line SQL query, it is clear that Regiment provides a great deal of flexibility while abstracting away most details of sensing, communication, and computation. For instance, it is straightforward to implement a wide range of in-network aggregations by providing alternative combining functions to `rfold`. Likewise, the rich set of primitive operations on signals and regions permits complex distributed computations to be described in a few lines of code. Further examples are presented in Section 4 below.

3.5 Other Regiment features

Regiment provides the basic set of language features seen in other functional programming languages. In addition to the primitive operators over regions and signals, the programmer may use conditionals, pass functions as values, and so on. At compile time, Regiment performs aggressive reductions to simplify the global network program into a form that can be readily optimized and mapped into node-local code. The technique Regiment employs is called multi-stage programming, meta-programming, or partial evaluation. This technique enables the use of programming abstractions in the meta-language that are not supported by the target platform. However, it places restrictions on the kinds of Regiment programs one can write. For example, only bounded recursions are allowed, because they must terminate with inlining. Higher order functions cannot be stored in regions, because they have no runtime representation. The program reductions applied by Regiment are described in Section 5.

3.6 Limitations

Certain applications do no fit naturally into the Regiment framework. First, because Regiment compiles queries into stream-processing dataflow graphs, applications with highly dynamic behavior—multiple modes of operation, changing data paths—are more difficult to encode in Regiment. Second, the Regiment runtime is based on a specific spanning tree library. Applications that require other communication services (such as any-to-any routing), cannot make use of Regiment, nor can applications that require explicit control of other low level hardware features (for example, a custom duty cycling policy). Finally, Regiment is designed to compile and install long-running queries; due to its reliance on custom user code rather than predefined operators, and its focus on compilation rather than interpretation of queries, it is not well-engineered to support rapid, repeated, short-lived queries. Such queries are better supported through a database-like macroprogramming system (such as TinyDB [10]).

4. Application examples

As a motivating example of Regiment in a realistic context, we focus on a specific application domain: detection of spatially-distributed events, such as chemical plumes [3] or wildfires [4]. These applications involve a network of sensors distributed over a potentially large area (many thousands of acres), each capable of monitoring local conditions such as temperature or airborne chemical concentrations. The goal is to rapidly detect the onset of a plume or wildfire by collecting readings from the sensor network, and to avoid false positive reports which may be caused by noisy readings from individual sensors. In addition, to save power, it is desirable to avoid sending data over long multi-hop paths unless absolutely essential to reporting a plume. These applications represents just one of a class of STMP programs for which Regiment is well-suited.

4.1 Example programs for plume monitoring

In this section we develop a series of programs that all perform chemical plume monitoring, but which employ substantially different communication and filtering mechanisms. A major thrust of the Regiment system is to enable the rapid prototyping of a range of solutions for a given sensor network problem. The use of a high-level language makes it possible to quickly develop complex distributed programs in a few lines of code, which can be subsequently tailored to different environments. Indeed, we recognize that in many applications, no single strategy will work well under all network sizes, topologies, and environmental conditions.

NoFilt: The most basic plume-detection program, called *Nofilt*, periodically samples the data from each node and delivers it to the base station:

```
fun read(nd) { (sense("conc",nd), nodeid(nd)) }
BASE ← rmap(read, world)
```

This program returns a region of tuples with each node's chemical concentration sensor value and node ID. Note that the sampling period is not given explicitly; it can be specified either within the source or set as a configuration parameter. Further, notice that Regiment permits that a region value be returned as the program's output; all samples returned to the base station without aggregation. Because *NoFilt* returns all the data samples at every point in time, it clearly involves high communication overhead.

LocalFilt: A somewhat more sophisticated program filters out sensor values below a threshold before reporting to the base station:

```
fun abovethresh((c,id)) { c > CHEM_THRESHOLD }
fun read(nd) { (sense("conc", nd), nodeid(nd)) }
BASE ← rfilter(abovethresh, rmap(read, world))
```

Here, `abovethresh` takes a tuple of (conc, nodeid) and returns `true` if the corresponding concentration reading is above the threshold. This program suffers from the problem that reports from individual nodes may not reliably indicate that an event has occurred. For example, noise from individual sensor readings could cause spurious event detections. Apart from false positive detections, this approach leads to wasted transmissions to the base station.

KhoodFilt: The third variant of the program uses collaborative event detection within the network to suppress spurious detections. In this case, the sum of the sensor readings in a neighborhood of nodes must exceed a threshold before the phenomena is reported to the base station.

```
fun abovethresh(t) {t > CHEM_THRESHOLD}
fun read(n) {sense("conc", n)}
fun sum(r) {rfold((+), 0, r)}

detects = rfilter(abovethresh, rmap(read, world));
hoods = rmap( fun(t,nd){khood(1,nd)}, detects);
sums = rmap(sum, hoods);
BASE ← rfilter( fun(t){t > CLUSTER_THRESH}, sums)
```

In the above code, `detected` is a region of nodes that surpass their local thresholds. `hoods` is a *nested region* consisting of the sets of nodes in the one-hop radio neighborhood of every node in the `detected` region. Nested regions are an extremely powerful structure in Regiment that facilitate complex, coordinated operation in the sensor network.

The region `sums` is not a nested region because the helper function `sum` has reduced each neighborhood back down to a single summed value. The code shown here only returns the set of summed sensor values; the complete program would also need to return the node IDs of the cluster-heads reporting each sum. This extra bookkeeping information must be threaded through the program using tuples, and is not shown here to simplify the discussion. [3]

GossipFilt: The above program provides simple neighborhood-based filtering, but has a substantial downside. By forming local neighborhoods and folding their values, *KhoodFilt* operates on a request-response model. Nodes request to form local neighborhoods, and their neighbors respond with local data through the fold operation. Thus a node may respond individually to several requests for its local data rather than leveraging the broadcast nature of the communication medium. In the case of one-hop radio neighborhoods, we can optimize this program further by eliminating the use of `khood` region construction and using as simple gossip mechanism instead. This approach is similar to the data sharing primitives explored in Hoods [17] and abstract regions [16].

```
# Reduces a table to a number:
fun sum(tbl) {
  fold( fun((id,t), acc){t + acc}, 0, r)
}
detects = rfilter(abovethresh, rmap(read, world));
tables  = table_gossip(detects);
sums = rmap(sum, tables);
 BASE ← rfilter(fun(t){t>CLUSTER_THRESH}, sums)
```

The *GossipFilt* program uses the `table_gossip` operator introduced in Section 3.3. *GossipFilt* collects the temperature data published by neighbors into tables residing on each node. Each table is then subjected to a fold operation to aggregate data for that neighborhood. Finally, only those aggregated values that are above the threshold are reported to the base station.

As these examples show, Regiment makes it easy to construct programs with increasing degrees of sophistication in a few lines of code. The *GossipFilt* program shown above involves local sensing, region formation, in-network aggregation, and distributed filtering in just eight lines of code (including the code for the `abovethresh` and `read` functions, borrowed from *LocalFilt*).

5. The Regiment Compiler

The basic job of the Regiment compiler is to map the global structure of the program into node-level code that implements that structure. The Regiment compiler performs many stages of code normalization, analysis, and optimization. However, the core goal of *deglobalization*—converting macroprograms to local, event-driven programs —is accomplished in three key steps:

- **Normalize**: reduce the Regiment code into a small sub-language called RQuery.
- **Switch-POV**: region-streams become local streams as we switch from network point-of-view (POV) to node POV.
- **Event-Convert**: translate the node-level, stream-processing program into an event-driven program.

[3]In the evaluated version of *KhoodFilt* and *GossipFilt* we sum the max three temperature readings from the neighborhood. This is a more robust aggregate than a simple sum, given that the degree of the network may vary.

The output of the compiler is low-level, event-driven code written in an intermediate language that we described in earlier work [13]. We call these programs *token machines*; they represent a computational model suitable for implementation in sensor-embedded operating systems such as TinyOS. That is, programs are represented as sets of asynchronous event-handlers—no thread support is assumed. These event-handlers execute in response to external events (e.g., arrival of radio messages), or by one event-handler posting an event for another.

Ultimately, Regiment depends only on a simple spanning-tree based communication layer. We have implemented one such layer on top of our event-driven compilation-target. Each region is affiliated with a *gradient* (spanning tree) that establishes membership in the region and permits communication with nodes in the region. Region operations such as `rmap`, `rfilter`, and `rfold` are implemented on top of the region's spanning tree. Region spanning trees are constructed and maintained by the Regiment runtime system, a runtime which consists entirely of a set of simple event handlers that are linked against the event-handlers compiled from the user query. For example, the gradient library provides handlers for *emitting* new gradients as well as sending and receiving messages along existing gradients.

5.1 Translation to RQuery and Normalization

The first step of compilation takes Regiment programs in their general form and reduces them, through a process of partial evaluation, to a sub-language which we call RQuery. The key parts of the RQuery subset are defined below: [4]

$$\begin{array}{lcl}
Q & ::= & \texttt{rmap}(\textbf{fun}(x)\,B,\ Q) \mid \texttt{rfilter}(\textbf{fun}(x)\,B,\ Q) \\
 & & \mid \texttt{table\_gossip}(Q) \mid \texttt{world} \\
B & ::= & e \mid S \\
e & ::= & x \mid c \mid (e_1, e_2) \mid \texttt{if}\,e_1\,\texttt{then}\,e_2\,\texttt{else}\,e_3 \mid \cdots \\
S & ::= & \texttt{rfold}(\textbf{fun}(x,s)\ B,\ e,\ R) \\
R & ::= & \texttt{rmap}(\textbf{fun}(x)\,B,\ R) \mid \texttt{rfilter}(\textbf{fun}(x)\,B,\ R) \\
 & & \mid \texttt{table\_gossip}(R) \mid \texttt{khood}(c, x)
\end{array}$$

A query (Q) produces a region of scalar values from the distinguished `world` region. The query consists of a chain of region-level dataflow operators, parameterized by functions and expressions. The bodies of these functions (B), can be either simple expressions or nested queries against sub-regions constructed by `khood` (S). Simple queries, that involve no nested-regions (and therefore no `rfolds`) are relatively easy to compile to node-level dataflow programs. Nested queries are handled via the Switch-POV step described below.

In order to map Regiment programs to the RQuery subset, we apply a number of reduction rules to the initial program until it is in RQuery-normal form. The reduction rules include the usual notions of β-reduction (function in-lining) and δ-reduction (constant folding) for functional languages. In addition, the reduction rules include the following transformations:

$$\begin{array}{rcl}
\texttt{rmap}(f,\ \texttt{rmap}(g,\ e)) & \rightarrow & \texttt{rmap}(f \circ g,\ e) \\
\texttt{rfilter}(f,\ \texttt{rfilter}(g,\ e)) & \rightarrow & \texttt{rfilter}(f\ \&\&\ g,\ e) \\
\texttt{rfold}(f,\ u,\ \texttt{rmap}(g,\ e)) & \rightarrow & \texttt{rfold}(\textbf{fun}(v,a) f(g(v),a), u, e) \\
\texttt{rfold}(f,\ u,\ \texttt{rfilter}(g,\ e)) & \rightarrow & \texttt{rfold}((\textbf{fun}(v,a) \\
 & & \quad \texttt{if}\ g(v)\ \texttt{then} f(v,a)\ \texttt{else}\ a), \\
 & & \quad u,\ e)
\end{array}$$

The first two rules allow us to fuse adjacent maps and adjacent filters. The third and fourth rules allow us to collapse an inner `rmap`

[4]We omit certain parts of the full-blown Regiment language here for clarity, including: `rintegrate`, `smap`, `sfilter`, and non-region typed queries.

```
timer :: Int → Signal(())
emit ::  Int, Signal(RID,()) → Signal(RID,())
aggr ::  (α→β→β), β, Signal(RID,α)
         → Signal (RID,β)
```

Figure 1: The types of node-level stream operators.

or `rfilter` into an outer `rfold`. In essence, we delay applying the map or filter function until the fold operation is invoked.

By applying all of the reduction rules exhaustively, we are able to generate programs in the RQuery normal form. For example, consider the collaborative sensing (*KhoodFilt*) program from Section 4.1 which has the following high-level structure:

```
fun sum(r) {rfold(f1,u1,r)}
detects = rfilter(f2, rmap(f3, world));
hoods = rmap( fun(t,nd) {khood(1,nd)}, detects);
sums = rmap(avg, hoods);
BASE ← rfilter(f2, sums)
```

After function in-lining and constant folding, the program body has the form:

```
rfilter(f2, rmap(fun(r){rfold(f1,u1,r)},
                 rmap(fun(t,nd){khood(1,nd)},
                    rfilter(f2, rmap(f3, world)))))
```

After applying the first reduction rule to fuse `rmaps`, the compiler yields the following, which is in RQuery normal form.

```
rfilter(f2,
  rmap(fun(t,nd){rfold(f1,u1,khood(1,nd))},
    rfilter(f2, rmap(f3, world))))
```

In a technical note [12], we define the full syntax and typing rules for a larger subset of Regiment, give a complete set of reduction rules, and prove that an exhaustive application of these rules results in well-typed RQuery normal forms. We are currently working to prove that for well-typed Regiment programs, the process of normalization always terminates and produces unique (up to input/output equivalence) normal forms.

An advantage of the normalization process is that, through aggressive inlining and fusion, the overheads of many small functions can be avoided. This allows programmers to write many small, reusable fragments of code that can be conveniently stitched together to rapidly produce an application. A potential disadvantage is that the aggressive normalization can lead to excessive code duplication, though for the small programs that we have used thus far, this has not proven to be a problem.

5.2 Switch POV

Now that the program is in the restricted RQuery normal form, it is possible to switch to the node-local POV (point of view). This requires flattening the nested, region-processing RQuery into a stream-processing program that can execute at each node. In doing so, it replaces `Region` values with local streams augmented with region identifiers. Likewise, region-level communication operators, such as `rfold`, map onto node-level communication services.

5.2.1 Unraveling Nested Regions

Because regions can be *nested*, it is possible for the user function provided to as argument to `rfold` to take a `Node` and internally construct and consume a nested `Region`. The node-local stream language that we compile into does not allow nesting; each user function must take and produce only plain data. Consider this example program:

```
rfold0(fun(n,a) { ...
        rfold1(fun(n,a) { ...
                rfold2(fun(n,a) { ...
                khood2(c2,n)),
        khood1(c1,n)),
world)
```

To unravel the nested regions, the compiler traces a linear chain of control through the RQuery. This pipeline represents transfers of control from one region to another: from a parent region to its child, and back. Control originates with `world` and proceeds into successively nested regions ($\texttt{khood}_1$, $\texttt{khood}_2$). From the innermost nested region, control moves back outward, aggregating data through successive `rfolds`. Thus the flow of control for the operators in the example is as follows.

$$\texttt{world} \rightarrow \texttt{khood}_1 \rightarrow \texttt{khood}_2 \rightarrow \texttt{rfold}_2 \rightarrow \texttt{rfold}_1 \rightarrow \texttt{rfold}_0$$

Constructing this control graph is possible because the normalization process ensures that regions produced by khood are *immediately consumed*. Another way of saying this is that there remain no variables with type `Region`. Thus, while a region is a *datum* within the user's source program, the sequence of operations through which this datum travels is completely known to the compiler at this phase. Indeed, this is a critical property of Regiment. Regions, as distributed objects, are expensive to control at runtime; they must instead know what operations to apply to themselves.

5.2.2 Substituting Local Types/Operators

Now the compiler must generate a stream-processing dataflow graph that matches the structure of the control graph described above. Because of the invariants introduced by the normalization transformation, this process is relatively simple. In fact, the code for user functions that parameterize the RQuery does not change substantially. Only the following substitutions need to be applied to the control graph to yield the correct node dataflow graph.

	Operators :	
`world`	→	`timer`
`rmap`	→	`smap`
`rfilter`	→	`sfilter`
`khood`	→	`emit`
`rfold`	→	`aggr`
	Types :	
`Region`(α)	→	`Signal`$((\texttt{RID}, \alpha))$
`Node`	→	()

`RID` is a new type representing a *region identifier*. The `emit` and `aggr` operators are hooks into the gradient service running on all nodes. These communication operations are part of the node-level dataflow graph, but their incident edges represent data routed through the network, rather than between operators within a node. In particular, `emit` initiates a bounded flood (one-to-many), and `aggr` sends data up the resulting tree (many-to-one), aggregating it in the process. Types for these operators are given in Figure 1.

As a final illustration let us return to our example program from Section 5.1.

```
rfilter(f2,
  rmap(fun(t,nd){rfold(f1,u1,khood(1,nd))},
    rfilter(f2, rmap(f3, world))))
```

once traversed, becomes:

$$\texttt{world} \rightarrow \texttt{rmap(f3)} \rightarrow \texttt{rfilter(f2)} \rightarrow \texttt{khood} \rightarrow$$
$$\rightarrow \texttt{rfold(f1,u1)} \rightarrow \texttt{rfilter(f2)} \rightarrow$$

Finally, the substitutions are applied to yield the node-level dataflow, pictured in Figure 2. Let's examine how this node-level dataflow graph works. `Timer`, which was given periodicity (3000ms) by the

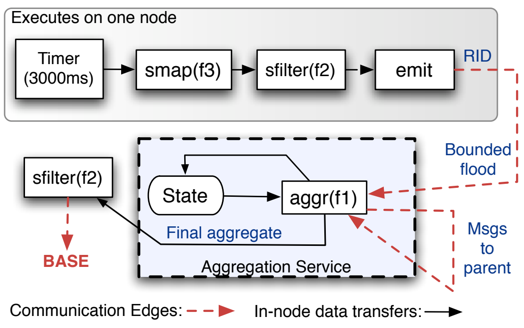

Figure 2: Node-level dataflow graph *produced from the KhoodFilt example program found in Sections 4.1, 5.1, and 5.2.2*

compiler based on configuration information, generates a stream of unit values (zero-length tuples). These unit values are received by `smap`, which applies the function `f3` locally—`f3` receives no data from the timer, but it is free to read local node values, including sensors. Next, `sfilter` simply filters events from the stream, and then we arrive at the interesting part of the pipeline.

When a data element passes the filter `f2`, it enters `emit` and triggers the emission of a local gradient. In the current implementation, this means that a flood message is sent with a time-to-live and nodes use an ETX[2] metric to choose their parents. These emitted gradients may overlap; thus the messages sent by `emit` carry an *RID* that uniquely identifies that region. `Emit` takes an RID on its input stream, and addst to it the node-ID of the current node (the root of the gradient) to make a unique RID.

The *output* edge from `emit`—which fires on every node that receives a message from the bounded flood—feeds directly into `aggr(f1,u1)`. The aggregation operator is responsible for maintaining state, applying the function `f1` to the input data paired with the existing state, and periodically sending aggregated state to the parent of the current node. `Aggr` uses a table (indexed by RID) to maintain separate state for each overlapping nested region in which the node participates. It further uses the RID retrieved from its input stream to determine *which* routing tree to use. When the data gets back to the root of the gradient, the RID is shortened by one, paired with the final aggregated state, and sent along the output edge of the `aggr` operator.

5.3 Event-Convert Streams

The output of Switch-POV is a node-level dataflow graph. This would be a reasonable point at which to plug into another back-end or intermediate language for a particular sensor network platform, for instance, the dataflow language Flask [15]. However, Regiment goes one step further and compiles the DFG to an event-driven program. This step is an application of well-understood compiler technology. Our DFGs are instances of a deterministic process network.

Each operator in the directed graph is compiled to a simple function: an event-handler that fires when a data-element is available on any of its input edges. One complication is that functions such as `sense`, which require a split-phase implementation, must be first separated out into their own "boxes" in the dataflow graph. However, the major source complexity in this step is not in the transformation itself, but in the libraries that are linked against the event-driven program. For instance, the gradient library, upon which Regiment relies for all its communication needs, is written in this event-handler style. Applications of communication primitives, *timer*, *emit*, *aggr*, become event postings that are consumed by these linked libraries.

6. Evaluation

In this section we compare the performance of four Regiment macroprograms for detecting spatially-distributed events, such as a chemical plume or a wildfire. Though they accomplish the same end, these programs differ greatly in terms of their communication patterns and their energy usage at different phases of the event-detection life-cycle. Our goal is to demonstrate that Regiment makes it easy to write complex sensor network applications at the global level while achieving good communication performance.

6.1 Experimental setup

We evaluate Regiment using a detailed simulation of sensor nodes that models local node sensing, computation, and message transmission. The simulator directly supports the Token Machine programming model described earlier. An implementation of the Token Machine environment in NesC for TinyOS is currently under development, and we expect to run these programs on real sensor nodes soon.

The simulated network consists of 250 nodes distributed in an area covering 5000 by 5000 m. Nodes are placed in a "perturbed grid" configuration in which each node is placed roughly on the corners of grid lines spaced 316 m apart. However, the location of each node is perturbed from the grid by up to 158 m in each of the x and y directions. The radio link quality model varies the packet delivery ratio as a function of the distance between nodes. Links shorter than 300 m have a 100% delivery ratio, while links longer than 500 m have a 0% delivery ratio. These settings create a topology with average degree 7.2, standard deviation 1.5. Yet many of these links are low quality. The average transmission success rate is 65%, with standard deviation 31%.

Node failures are not modeled. It should be noted, however, that the Regiment runtime maintains no hard state. Any corruption of the network lasts only until the next global spanning tree refresh.

Each simulation is run for 3600 simulated seconds. Chemical plumes originate at fixed locations within the coverage area. We assume an isotropic (wind-free) diffusion model in which plumes diffuse radially from the source at a rate of 1 meter per second, and eventually dissipate after 500 sec. The plume generates a uniform chemical concentration of 200 ppm over its coverage area. Chemical concentration falls off as one over the distance from the edge of the plume. Nodes detect the proximity of a plume using onboard chemical sensors with normally-distributed noise (σ =6 ppm) and maximum recorded concentration of 200 ppm.

The base station receives chemical concentration reports from nodes in the network and determines that a plume has started when at least three nodes have reported concentrations over 100 ppm. Based on the location of the plume, the placement of nodes, and the time it takes each program to process and communicate sensor data, the probability that a plume is correctly detected, and the detection latency, will vary accordingly.

6.2 Code complexity

The four programs evaluated in our experiment are shown in Section 4. We refer to *LocalFilt* as the program that reports periodic data with local filtering; *KhoodFilt* as the program that performs collaborative in-network filtering using a `khood` region to collect data from a nodes' neighbors, and *GossipFilt* as the program that uses local (one-hop neighborhood) gossip to actively share data with each nodes' neighbors. In addition, we show results for a slightly improved version of *GossipFilt*, called *GossipFiltSuppress*,

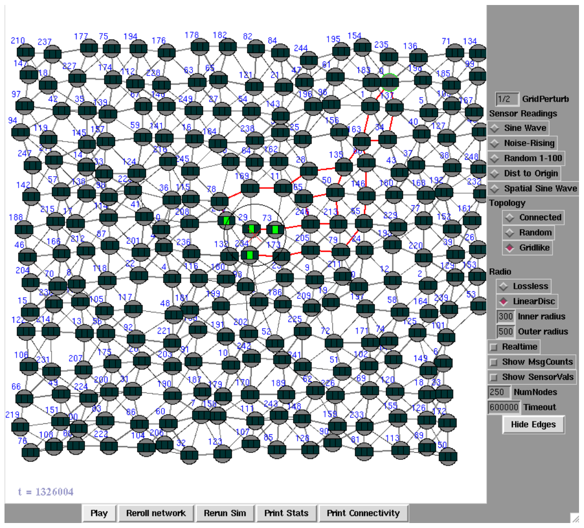

Figure 3: TM simulator GUI. ***This figure shows the simulation running a chemical plume simulation in a 250 node network. The plume diffuses at a constant rate from a starting point. Nodes in the vicinity of the plume send back messages to the base station using the global spanning tree (red).***

Program	Regiment	TM
LocalFilt	3	290
KhoodFilt	15	495
GossipFilt	40	394
GossipFiltSuppress	78	633

Figure 4: Lines of code for each program, ***showing both the Regiment code and the Token Machine code produced by the compiler.***

which causes a node to suppress sending repeated plume detection messages (gossips) for a period of 20 sec following an initial detection. The idea here is to reduce communication overhead, since a node that detects a plume once should detect it in subsequent periods as well.

Figure 4 shows the lines of code for the Regiment programs as well as the resulting Token Machine code. As described earlier, the TM representation is a simple, node-local state machine consisting of a set of event handlers with basic support for sensing, radio communication, and so forth. A TM program can be likened to a Motlle program as described in [8]. As the table shows, fairly complex Regiment programs can be implemented in a few lines of code, and increased complexity translates into larger node-level code as expected (with the exception that basic GossipFilt is a relatively simple node-level program).

6.3 Communication overhead

Our first experiments show the communication overhead for each of the programs described earlier. Figures 5, 6, and 7 show the behavior of the *LocalFilt*, *KhoodFilt*, and *GossipFilt* programs, including the number of messages transmitted per second *for the entire network*, as well as the corresponding chemical reports received by the base station. The plumes starts at time $t = 1000$. (The tall spike in communication at $t = 1200$ corresponds to a global spanning tree refresh initiated by the base station.) We will judge each program according to its communication behavior dur-

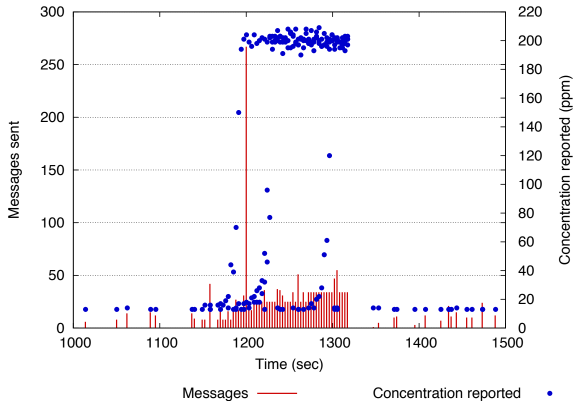

Figure 5: Message load and chemical concentration reports for the ***LocalFilt*** **data-collection program.** ***A plume starts at time $t = 1000$. Nodes detecting a chemical concentration over 12 ppm send a report to the base station. The spike in traffic at $t = 1200$ is a spanning tree refresh. Traffic load is fairly high with many chemical reports being sent to the base station.***

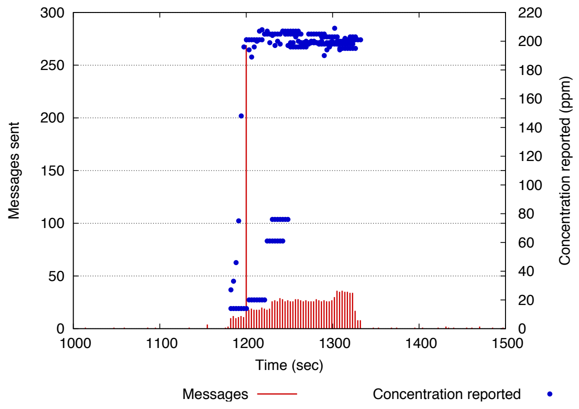

Figure 6: Detection with collaborative filtering. ***Nodes locally detecting a plume aggregate data from their one-hop neighborhoods using a*** `khood` ***region, before transmitting the data to the base. Message load is somewhat higher than simple local filtering, due to local spanning-tree formation and aggregation.***

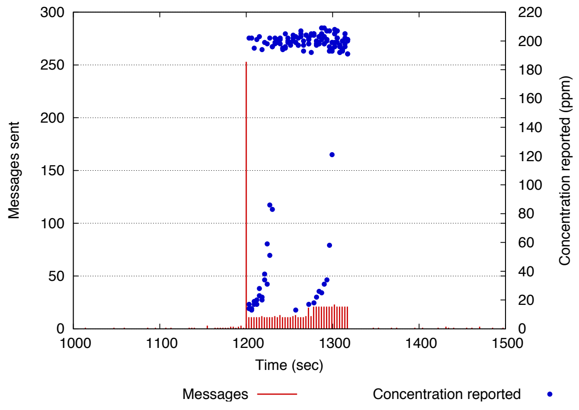

Figure 7: Detection using local gossip. ***Nodes locally detecting a plume aggregate data by gossiping within a one-hop neighborhood, which exhibits lower communication overhead than*** `khood`.

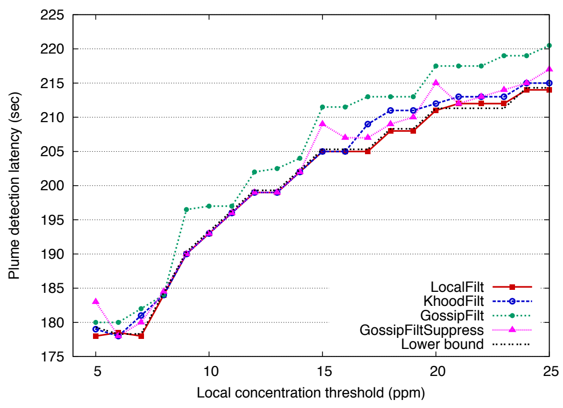

Figure 8: **Plume detection latency as a function of local concentration threshold.**

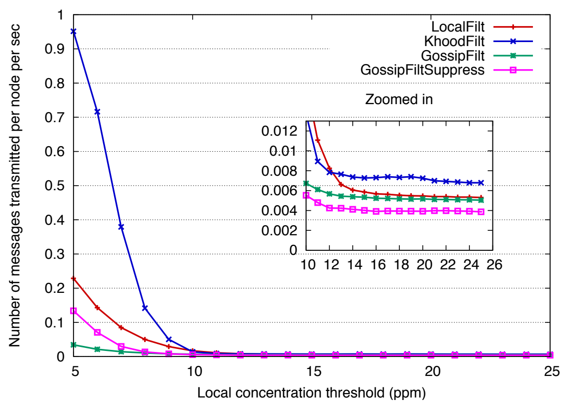

Figure 9: **Communication overhead as a function of local concentration threshold.**

ing the *dormant* phase before a chemical plume, and during active detection. In the *LocalFilt* program, every node with a chemical concentration of over 12 ppm is reporting data to the base station at a rate of once every 3000ms, resulting in high dormant-phase communication overhead. Each individual spurious detection must be transmitted to the base station along a multi-hop route.

The second program, *KhoodFilt*, causes nodes detecting a plume to collect and aggregate sensor data from nodes in a one-hop radio neighborhood. Only if the sum over the maximum three readings in the neighborhood exceeds the threshold is a report sent to the base station. The goal here is to reduce the number of false positives, at the expense of possibly increased plume detection latency. As Figure 6 shows, far fewer chemical reports are transmitted to the base station. However, overall communication load is *higher* than in the case of *LocalFilt*. This is because any node that exceeds the (fairly sensitive) local concentration threshold forms a `khood` region and performs aggregation, increasing the total number of message transmissions in the network. Keep in mind, however, that we have run our simulation for a relatively short period of time (five minutes), and simulated three chemical plumes in that short duration. Realistically, the network would be dormant the vast majority of the time, which makes the background traffic incurred by *LocalFilt's* spurious reports a serious disadvantage.

Figure 7 shows the behavior of *GossipFilt*, which uses a lightweight local gossip to share information on plume detections. Compared with *KhoodFilt*, we can see that *GossipFilt* has somewhat lower communication overhead, and the chemical sensor readings returned to the base station follow a continuous profile as the plume nears each of the sensors.

6.4 Detection accuracy vs. overhead

In our next set of experiments, we explore the effect of the local filtering threshold on plume detection latency, communication overhead, and false positive rate. We vary the local concentration threshold from 4 ppm to 50 ppm. We do not vary the network topology.[5] Results are averaged over three runs for each case. We calculate the *detection latency* as the time between the origination of the plume and the time that the base station detects the plume's presence. We also report the number of messages transmitted per node per second as a measure of overall communication load.

[5]However, varying network density (by changing radio ranges) between average degree 5.3 and average degree 19.9 does not significantly affect detection latency, though it does reduce communication overhead by shrinking the diameter of the network.

Figure 8 shows the plume detection latency as a function of the concentration threshold for all four programs. Not surprisingly, an increased detection threshold also increases plume detection latency. While the plume detection times may seem to be large (generally between 180 and 220 sec), the fact that the plumes are diffusing very slowly relative to the inter-node spacing implies that it will take some time before enough sensor readings can trigger a detection. Figure 8 also shows the *lower bound* of the detection latency, which is taken as the minimum detection time possible with an ideal network with no sensor error. Several of the measured detection latencies appear to be below the lower bound due to premature detections caused by noise in the sensor values; these are counted as false positives as discussed below. We can see that all four programs achieve very close to the lower bound, with *LocalFilt* having the lowest overall latency.

Figure 9 shows the communication load for each program as the local sensor threshold is varied. The overall message load is fairly low; the highest value is 0.95 msgs/node/sec for *KhoodFilt* with a detection threshold of 5 ppm. This program has the highest load due to the formation of local spanning trees for `khood`-based aggregation. The inset figure shows the communication load for larger thresholds, in which *GossipFiltSuppress* has the lowest communication load. At lower thresholds, *GossipFiltSuppress* exhibits higher load than *GossipFilt* because of its alternative strategy for integrating the stream of gossips it receives. Because *GossipFilt*'s input does not come regularly it maintains detections from its neighbors for a period of 20 seconds. This can exacerbate the impact of frequent false detections.

Finally, we look at the *false positive rate* of each application, which we define as the number of event detections at the base station that occur during periods when no plume is active. We also consider as false positives any "premature" detections induced by the sensor noise. This can occur when a plume is active but a noisy sensor causes the base station to detect an event before the lower-bound detection time that would occur in an ideal, noise-free network. In some sense these premature detections are "lucky," but we do not count them as true positives.

The results are shown in Figure 10. The overall false positive rate is very low, less than 5% in most cases, apart from *KhoodFilt* when the local sensor threshold is close to the sensor noise floor.

As these results show, different designs for in-network aggregation exhibit differing tradeoffs in terms of communication overhead and detection latency. Regiment makes it possible to rapidly explore the design space by developing differing versions of a fairly sophisticated sensor network application in just a few lines of code.

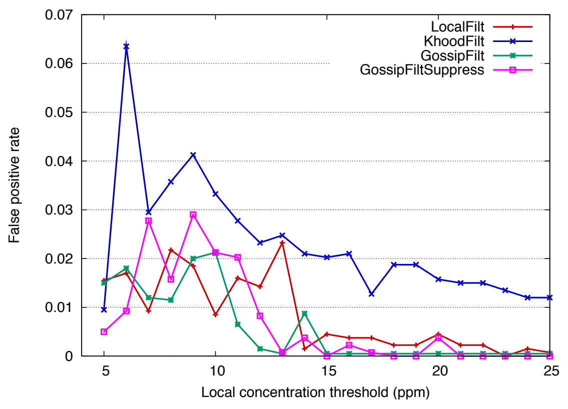

Figure 10: False positive rate as a function of local concentration threshold.

7. Conclusions and Future Work

In this paper, we have presented the Regiment macroprogramming system, which consists of a high-level language for global programming of sensor networks, as well as a compiler for Regiment macroprograms. Regiment allows the programmer to develop complex applications with in-network computation without explicitly dealing with low-level features of node programming. This approach does not, however, entirely insulate the programmer from the performance implications of their algorithms. We believe that no macroprogramming language can completely abstract away the performance details of distributed program design while remaining expressive. For this reason, the macroprogrammer must implement several versions of their program, and refine them according to the particular parameters of their network environment, before finally choosing a program for deployment.

We have demonstrated the use of Regiment through a series of chemical plume detection programs, each implemented in a few lines of code. Regiment supports this usage model through a high-level language with expressive, composable constructs based on the concepts of `Signals` and `Regions`. These programs are automatically compiled down to a simple node-level runtime based on the Token Machine model. Our results demonstrate the flexibility of macroprogramming in Regiment while retaining good communication performance.

Several ongoing threads of work are focused on improving the Regiment system. We are expanding the set of features while continuing to re-evaluate our restrictions on the source language so as to achieve a good balance between flexibility for the programmer, and restrictions for efficient compilation. Further, we currently have a prototype TinyOS compiler for our TM intermediate representation. It successfully compiles a subset of Regiment programs, which we hope to expand to the full set in the near future.

8. References

[1] A. Bakshi, V. Prasanna, J. Reich, and D. Larner. The abstract task graph: A methodology for architecture-independent programming of networked sensor systems. In *Workshop on End-to-End, Sense-and-Respond Systems, Applications, and Services (EESR)*, 2005.

[2] D. D. Couto, D. Aguayo, J. Bicket, and R. Morris. A high-throughput path metric for multi-hop wireless routing, 2003.

[3] E. A. Cowen and K. B. Ward. Chemical plume tracing. *Environmental Fluid Mechanics*, 2(1-2), June 2002.

[4] D. M. Doolin and N. Sitar. Wireless sensors for wildfire monitoring. In *Proc. SPIE Symposium on Smart Structures and Materials*, 2005.

[5] 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.

[6] R. Gummadi, O. Gnawali, and R. Govindan. Macro-programming wireless sensor networks using. In *Proc. Int. Conference on Distributed Computing in Sensor Systems (DCOSS)*, 2005.

[7] J. M. Hellerstein, W. Hong, S. Madden, and K. Stanek. Beyond average: Towards sophisticated sensing with queries. In *Proc. the 2nd International Workshop on Information Processing in Sensor Networks (IPSN '03)*, March 2003.

[8] P. Levis, D. Gay, and D. Culler. Active sensor networks. In *NSDI '05: Proceedings of the Second USENIX/ACM Symposium on Networked System Design and Implementation*, 2005.

[9] L. Luo, T. Abdelzaher, T. He, and J. Stankovic. Envirosuite: An environmentally immersive programming framework for sensor networks. *ACM Transactions on Computational Logic*, V(N), October 2005.

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

[11] S. Nath, Y. Ke, P. B. Gibbons, B. Karp, and S. Seshan. IrisNet: An architecture for enabling sensor-enriched Internet service. Technical Report IRP-TR-03-04, Intel Research Pittsburgh, June 2003.

[12] R. Newton. Normalizing regiment queries. `http://www.regiment.us/docs`, Nov 2006.

[13] R. Newton, Arvind, and M. Welsh. Building up to macroprogramming: An intermediate language for sensor networks. In *Proc. Fourth International Conference on Information Processing in Sensor Networks (IPSN'05)*, April 2005.

[14] R. Newton and M. Welsh. Region streams: Functional macroprogramming for sensor networks. In *Proc. the First International Workshop on Data Management for Sensor Networks (DMSN)*, Toronto, Canada, August 2004.

[15] G. M. M. Welsh and G. Morrisett. Flask: A language for data-driven sensor network programs. Technical Report TR-13-06, Harvard University Technical Report, May 2006.

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

[17] K. Whitehouse, C. Sharp, E. Brewer, and D. Culler. Hood: A neighborhood abstraction for sensor networks. In *Proc. the International Conference on Mobile Systems, Applications, and Services (MOBISYS '04)*, June 2004.

[18] K. Whitehouse, F. Zhao, and J. Liu. Semantic streams: a framework for composable semantic interpretation of sensor data. In *Proc. European Workshop on Wireless Sensor Networks (EWSN)*, 2006.

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