Flask: A Language for Data-driven Sensor Network Programs

Geoffrey Mainland, Matt Welsh, and Greg Morrisett
Division of Engineering and Applied Sciences
Harvard University
{mainland,mdw,greg}@eecs.harvard.edu

Abstract

In this paper, we present *Flask*, a new programming language for sensor networks that is focused on providing an easy-to-use dataflow programming model. In Flask, programmers build applications by composing chains of *operators* into a dataflow graph that may reside on individual nodes or span multiple nodes in the network. To compose dataflow graphs across sensor nodes, Flask supports a lean, general-purpose communication abstraction, called *Flows*, that provides publish/subscribe semantics over efficient routing trees. At the heart of Flask is a *programmatic wiring language*, based on the functional language OCaml [13]. Flask's wiring language allows dataflow graphs to be synthesized programmatically. The Flask wiring program is interpreted at compile time to generate a sensor node program in NesC, which is then compiled to a binary.

Our design of Flask makes three main contributions. First, Flask allows the programmer to specify distributed dataflow applications in a high-level language while retaining the efficiency of compiled binaries and full access to TinyOS components. Second, Flask provides a unified framework for distributing dataflow applications across the network, allowing programmers to focus on application logic rather than details of routing code. Finally, Flask's programmatic wiring language enables rich composition of dataflow operators, making it possible to develop higher-level programming models or languages directly in Flask.

In this paper, we describe the design and implementation of the Flask language, its runtime system, the Flows communication interface, and a compiler that produces NesC code. We evaluate Flask through two motivating applications: a distributed detector of seismic activity (e.g., for studying earthquakes), and an implementation of the TinyDB query language built using Flask, showing that using Flask considerably reduces code complexity and memory size while achieving high runtime efficiency.

1 Introduction

Sensor network application design is extremely challenging due to the limited resources of sensor nodes, communication and node failures, and the need for complex in-network processing to reduce energy and radio bandwidth usage. As a result, application developers typically invest much time writing code to manage low-level details of sensor node operation, rather than focusing on application-specific logic. Part of the problem is that sensor node programming is already so complex that the developer cannot afford the extra effort required to build more general, reusable subsystems.

In this paper we present *Flask*, a new programming language for sensor network devices, that combines the power of low-level languages such as NesC [5] with a concise, high-level abstraction for constructing distributed dataflow graphs in a sensor network. In Flask, programmers build applications by composing chains of *event handlers* into a dataflow graph that may reside on individual nodes or span multiple nodes in the network. This style of dataflow programming greatly simplifies application design by presenting a unified model for sensor data acquisition, timers and interrupts, local computation, and communication. To compose dataflow graphs across sensor nodes, Flask supports a lean, general-purpose communication abstraction, called *Flows*, that provides publish/subscribe semantics over efficient routing trees.

At the heart of Flask is a *programmatic wiring language*, based on the functional language OCaml [13]. Flask's wiring language allows dataflow graphs to be synthesized programmatically, rather than using static "wiring diagrams" such as NesC configurations. The wiring program is interpreted at compile time to generate the resulting sensor node program in NesC, which is then compiled to a binary. Flask integrates well with NesC, permitting existing TinyOS components to be weaved into Flask dataflow graphs seamlessly. This approach makes it possible to write complex applications in a few lines of code while retaining the efficiency of NesC and TinyOS.

Flask makes three main contributions to the field of sensor network programming. First, Flask allows the programmer to specify distributed dataflow applications in a high-level language while retaining the efficiency of compiled binaries and full access to TinyOS components. Second, Flask provides a unified framework for distributing dataflow applications across the network, allowing programmers to focus on application logic rather than details of routing code. Finally, Flask's programmatic wiring language enables rich composition of dataflow operators, making it possible to develop higher-level programming models or languages directly in Flask.

In this paper, we describe the design and implementation of the Flask language, its runtime system, the Flows communication interface, and a compiler that produces NesC code. We evaluate Flask through two motivating applications: a distributed detector of seismic activity (e.g., for studying earthquakes), and an implementation of the TinyDB query language built using Flask. We demonstrate that Flask makes it easy to write complex sensor network applications in a few lines of code while producing efficient node-level programs. We compare the Flask implementation of TinyDB to the original NesC version [18] and a version implemented on top of the Maté virtual machine [15], and we show that using Flask considerably reduces code complexity while achieving high runtime efficiency.

The rest of this paper is organized as follows. In Section 2 we describe Flask's relationship to existing programming tools and the specific goals of our work. Section 3 presents Flask in detail through several small code examples. In Section 4 we present two complete applications implemented in Flask — a seismic event detection system, and an implementation of TinyDB. We evaluate Flask in Section 5 by comparing it to existing systems and showing it reduces code complexity and increases expressivity without sacrificing efficiency. Our evaluation includes programs running both in simulation and on a 75-node sensor network testbed. We describe future work and conclude in Section 6.

2 Background

Several recent projects have focused attention on the challenges of sensor network programming [14, 18, 23, 26, 7, 6]. Developing sophisticated sensor network applications is hampered by the inherent limitations of resource-limited sensor nodes, unreliable radio links, sensor node failures, and the need to carefully control radio bandwidth and energy usage. As a result, sensor node programs are notoriously difficult to debug, especially when the programmer is faced with subtle concurrency, timing, or memory-management problems.

Two broad approaches to these problems have emerged in the literature. The first is the development of libraries and middleware environments that provide higher-level abstractions for sensor node programming. The second is a range of so-called *macroprogramming* environments that allow an application designer to program the network as a whole, rather than individual sensor nodes. We discuss each of these in turn below.

Flask is targeted as a hybrid approach that simplifies the task of programming individual nodes using dataflow graphs, but provides facilities for composing these graphs across the network. As such, Flask is not a macroprogramming environment *per se*, but provides functionality that can be used to develop either node-level or network-level programs. Flask can also be used to build higher-level macroprogramming environments, as we show in Section 4.2 when describing a Flask implementation of TinyDB.

2.1 Node-level programming abstractions

The most popular programming language for resource-limited sensor nodes, such as the Berkeley motes, is NesC [5], used to implement the TinyOS operating system [9]. NesC provides a sophisticated component-based programming model in which individual components export one or more interfaces and multiple components are linked together at compile time using a *configuration*. A configuration describes a set of static bindings between a component that *uses* an interface and a component that *provides* that interface. NesC configurations can be nested, providing a form of encapsulation.

NesC supports the TinyOS concurrency model, which is based on the concepts of *tasks* and *events*. A task is a function that runs to completion and may not block or be preempted by another task, while an event represents a low-level signal (e.g., hardware interrupt) that may preempt another event or task. The most significant limitation of the TinyOS concurrency model is the absence of threads, requiring the use of asynchronous ("split-phase") calls for potentially long-running operations such as sampling sensors or transmitting radio messages.

While NesC and TinyOS have had much success as the *de facto* programming environment for mote-class sensor networks, they are very low level and as such demand a great deal of expertise to build applications. A number of recent projects have proposed node-level abstractions to simplify program development. Directed diffusion [11, 8] is a framework for distributed event detection, filtering, and aggregation. Hoods [26] and abstract regions [23] provide a set of abstractions for communication within local neighborhoods of the network. GHT [22] and DIMENSIONS [4] are examples of distributed data storage primitives.

While these systems provide a range of primitives for building sensor network applications, they do not offer a language-based approach. More closely related to our efforts is Maté [15, 16], a virtual machine designed to permit dynamic reprogramming of sensor nodes over the air. Maté provides a simplified concurrency model based on threads and code *capsules* that can be installed at runtime. Maté allows application-specific code to be represented as opcodes in the VM instruction set, making it possible to customize the VM's functionality at compile time. Maté can also be used as target for higher-level language compilers, and the authors describe two simple scripting languages and a TinyDB implementation based on Maté.

The chief disadvantage of the Maté approach is that programs must be interpreted at runtime, incurring high computational costs and memory overheads. For example, [15] reports an average computational cost of 400 CPU cycles for interpreting each VM bytecode. As a result, the significant compile-time optimizations performed by NesC are unavailable to Maté applications. In addition, the system designer must still decide in advance which features to build into the VM, providing only limited flexibility after initial VM deployment.

2.2 Macroprogramming environments

An alternative to node-local programming is *macroprogramming* [23], which attempts to abstract away the details of individual sensor nodes in favor of programming the network as a whole. The most prominent macroprogramming system is TinyDB [18], which allows users to express a declarative query for data of interest. The query is then realized as a series of sampling, filtering, radio transmission, and aggregation operations within the network. Cougar [29] and IrisNet [19] provide similar query interfaces to sensor network data.

Other macroprogramming environments include EnviroSuite [17], Semantic Streams [25], Kairos [7], Regiment [20], and Abstract Task Graphs [1]. These systems provide a range of programming models at different levels of abstraction. For example, EnviroSuite [17] is targeted at tracking applications, while Semantic Streams [25] provides a logic-based language for composing distributed data-processing services.

The implementation complexity of these systems is considerable, involving significant runtime code as well as compiler logic to transform global programs into node-level binaries. A core goal of Flask is to support the development of macroprogramming languages and toolkits by raising the level of abstraction enough to simplify their design, while retaining the efficiency of a compiled, optimized node binary. In Section 4.2 we describe an implementation of TinyDB in Flask that directly generates a NesC binary from a TinyDB query without requiring the full complexity of a query interpreter on every node.

2.3 Flask Goals

Our design of Flask is motivated by several goals:

- **Simplified sensor node programming using a dataflow abstraction:** Rather than requiring application developers to deal with low-level details of sensing, communication, and concurrency, Flask adopts a simplified dataflow programming model. Data generated by received radio messages, sensors, or timers flows through the application where it is processed, filtered, stored, or transmitted.

- **Programmatic composition:** An important aspect of simplifying program design is giving developers enough power to rapidly build up sophisticated compositions of software modules. In Flask, this is achieved using a programmatic, rather than static, wiring language, offering a great deal of flexibility for building new abstractions.

- **Unified communication API:** Much of the complexity of sensor network programming derives from the low-level nature of radio communication. Flask provides a simple, high-level communication interface that addresses both local broadcast and multihop routing.

- **Maintain high runtime efficiency:** On resource-limited sensor nodes we must be mindful of CPU and memory overheads when implementing higher-level programming models. A core goal of Flask is to permit compilation directly to an optimized node binary, and support direct integration with existing sensor network operating systems, such as TinyOS.

3 Flask Design

In this section we describe the design of the Flask language and runtime system. Flask's basic programming model is that of a *dataflow graph*, in which values generated by external sources (sensors, timer interrupts, or radio messages) are passed through a series of *operators* that process and filter the data. A dataflow graph is generated by a *wiring program* in the Flask language that describes, programmatically, how dataflow operators are composed. Flask's runtime system dictates how dataflow graphs are executed on individual sensor nodes.

We begin by describing Flask's data, concurrency, and communication models as embodied in the runtime system, and then describe the wiring language in detail. In this section, we draw on an example application that detects and characterizes seismic waves, which can be used in a system for monitoring structures such as buildings or bridges [28, 2, 21], or monitoring earthquakes and volcanic eruptions [24].

An example of a Flask dataflow graph is shown in Figure 1. This is a simple program that attempts to detect the onset of an earthquake or other interesting seismic event using a simple "ratio of two low-pass filters" approach on a seismometer signal. The program samples the seismic sensor every 10 ms and passes the resulting data through two exponentially-weighted moving average (EWMA) filters with different gain settings. If the ratio of these filters exceeds a threshold, this indicates that the seismic signal is significantly larger than the background noise, and a "detected" message is transmitted to the base station.

3.1 Flask runtime model

A Flask program consists of a set of *dataflow operators* composed into a directed, acyclic *dataflow graph*. We use the term *value* to mean a single typed data item that is passed as input or produced as output by a dataflow operator. The type of a value may be a primitive type as `int` or `float`, or may be a tuple consisting of multiple primitive values. A *wire* connects the output of one operator to the input of another operator. Operators can have multiple distinct inputs, but only one output. However, output wires may be freely split (i.e., fan-in and fan-out for each operator may be greater than one).

Execution and concurrency model

Flask supports a simple execution and concurrency model that is intended to avoid the high overheads of supporting multiple blocking threads at runtime, and avoid common bugs that arise when operators share state. In Flask, an operator "fires" each time a new value appears on any of its input wires. The operator runs to completion, processing

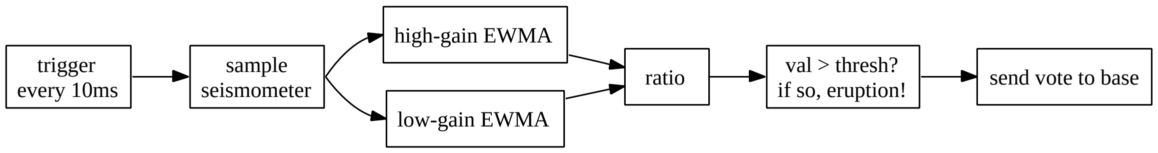

Figure 1: **Conceptual depiction of a Flask dataflow graph.** *This figure shows the dataflow graph for a Flask program implementing a basic seismic event detection algorithm.*

all of its inputs, and (optionally) producing an output value. An operator may not block during execution.

As in TinyOS [9], long-running, asynchronous operations in Flask make use of "split-phase" semantics. That is, rather than an operator blocking until some condition is met, an operator may asynchronously *post* an output that will be passed as input to a "continuation" operator. An example is acquiring data from a sensor. A sampling operator would invoke a "get data" routine that requests the ADC hardware to acquire a sample. When the data is ready, the hardware interrupt routine posts the sample data as input to the next operator in the dataflow graph, thereby continuing execution. We call such a connection between two operators a *posting wire*.

The semantics of execution in Flask are defined as follows. Execution is initiated by the arrival of data from an external source, such as a timer interrupt, or an input value arriving on a posting wire. Operators are executed as a depth-first traversal of the dataflow graph, with each operator processing its inputs and optionally producing an output. The ordering of different branches in the depth-first traversal is not defined. Traversal of a given branch terminates when it reaches an operator that does not produce an output (e.g., an event "sink") or a posting wire.

The set of operators visited by such a traversal constitutes an *atomic subgraph* of the whole-program dataflow graph. An atomic subgraph runs to completion and cannot be preempted during execution. In this sense, the Flask execution model is akin to a TinyOS task, although preemption (e.g., by interrupts) is disallowed. Flask ensures that data arriving from an interrupt context is injected into the appropriate atomic subgraph via a posting wire; so interrupt handlers are not exposed directly to Flask programmers.

Example

Figure 2 shows the dataflow graph from Figure 1 with the atomic subgraphs elaborated. Each shaded region is guaranteed to run to completion without preemption. Posting wires are shown as dotted arrows connecting the *sample seismometer* and *get sample* operator, since this is a split-phase operation.

The Flask concurrency model is intended to simplify sensor node programming by avoiding common concurrency bugs. However, not all race conditions are prevented, since dataflow operators in Flask may access shared (global) state. However, operators that only access *private* internal state cannot be involved in a data race. Likewise, if two operators share state and both exist only in the same atomic subgraph, no data races between them can occur. It is important to keep in mind that an operator may exist in more than one atomic subgraph.

3.2 Communication model: Flows

Flask supports a communication model called *Flows* permitting dataflow graphs to be distributed across the sensor network. Flows provide a flexible publish/subscribe communication API that give the programmer a unified interface for communication using a wide range of topologies: point-to-point, multipoint-to-point, point-to-multipoint, or multipoint-to-multipoint.

A flow is identified by a 16-bit *flow ID*. A node may send data to a given flow by invoking the *publish* operation with the flow ID as an argument, and subsequently sending values to the flow. Likewise, a node may *subscribe* to a flow, allowing it to receive values. Flows provide best-effort communication semantics in keeping with many existing sensor network protocols [27, 11]. Flows abstract away the details of route discovery and maintenance, link-level ARQ, and buffering. From the perspective of a Flask program, a flow is abstracted as a pair of operators: one that produces values when messages arrive on the flow, and another permitting data to be sent to the flow. Publishing or subscribing to a flow is accomplished by wiring the appropriate operator into the program's dataflow graph.

Flows are intended to simplify distributed data-flow programming, and as such are not meant to provide the most efficient routing protocol for all cases. However, our implementation of Flows (described in Section 3.5) is relatively efficient and captures a wide range of potential use cases well. A program that wishes to implement its own routing protocol can fall back to using a limited form of flows that uses only single-hop radio communication. Moreover, the Flows API can be readily backed by alternative protocol implementations.

3.3 Flask wiring language

We turn now to the Flask wiring language, which provides a mechanism for constructing Flask dataflow graphs. Flask is unique in that it uses a *programmatic* wiring language in which the dataflow graph is described by a program, rather than by a static representation such as that used by NesC configurations. The Flask wiring program is executed at compile time, generating the resulting runtime binary.

There are many advantages to using a programmatic (rather than diagrammatic) wiring language. Flask allows

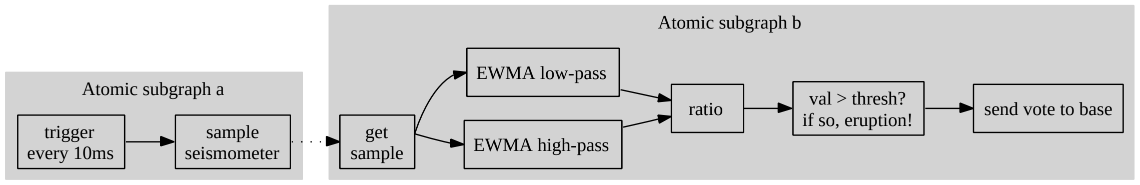

Figure 2: **Dataflow graph with atomic subgraphs labeled.** *This figure shows an updated dataflow graph for a program implementing a basic seismic event detection algorithm. The* posting wire *between the sample request and sample acquisition is shown along with the atomic subgraphs.*

dataflow graphs to be constructed algorithmically, making it easy to build up complex structures in a few lines of code, without resorting to macro expansion or "configuration templates" as found in Hoods [26] and recent versions of NesC. For example, a code "template" can be written as a Flask program fragment that is parametrized by a type descriptor. The template can be instantiated in a Flask application by invoking the program fragment as part of the wiring program.

The Flask wiring language is based on Objective Caml [13], a functional programming language in the ML family. As we will show, using a functional language allows us to concisely define new first-class operators. It is important to keep in mind that a Flask wiring program only runs at compile time and is used to generate a runtime program, in NesC; the use of OCaml does not imply that Flask requires runtime support on the sensor node for garbage collection or other features of OCaml.

Streams and stream combinators

A Flask wiring program is based on the concept of *streams* and *stream combinators*. A stream is an abstract source of typed values and represents a dataflow subgraph with a single output "wire." An example of a stream in Flask is the stream of clock ticks generated by the Flask expression `clock 10`. This expression creates a stream that generates a `void` value every 10 ms. The resulting `void` values can be tied to the inputs of other dataflow operators to initiate computation.

A *stream combinator* is a Flask expression that generates or manipulates streams. `clock 10` is an example combinator, parametrized by the constant value `10`, that generates a clock-tick stream. Another combinator is `adc`, which generates a stream of sensor readings. `adc` takes two arguments: the ID of a sensor to sample, and an input stream of `void` values used to trigger the sampling actions. The Flask expression

```
adc "Seismometer" (clock 10)
```

creates a stream of seismometer samples. Figure 3 illustrates this simple Flask wiring program and resulting dataflow subgraph.

Stream combinators are extremely powerful and allow Flask programs to generate customized runtime code

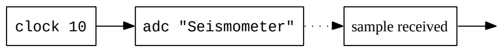

Figure 3: **Dataflow graph corresponding to simple Flask program** `adc "Seismometer" (clock 10)`.

parametrized by types or constant parameters. Consider a stream combinator `ewma` that generates a stream whose output values are an exponentially-weighted moving average of the input values. The EWMA filter is parametrized by a gain setting and an input stream. A Flask function that generates an EWMA filter with a gain setting of 0.9 could read:

```
let genfilter samples = ewma 0.9 samples
```

Using `genfilter input_stream` anywhere in the Flask program would then instantiate an EWMA filter with the appropriate gain setting. This form of encapsulation permits rapid construction of dataflow subgraphs through functional composition in the wiring program.

Stream combinators can also be used to combine multiple streams into a single stream; Flask's built-in `zip` combinator is an example. `zip` takes n input streams as arguments and generates a new output stream where each value is a tuple, each element of which is drawn from the n input streams. An output tuple is produced when at least one new value has arrived on each input stream. In our running example, `zip` is used to pair the values from the high-gain and low-gain EWMA filters before passing them to the ratio combinator, as shown in Figure 4.

```
let ewma_pair samples =
  let high = ewma HIGH_GAIN samples in
  let low = ewma LOW_GAIN samples in
    zip "ewma_pair" [low; high]
```

Figure 4: **Generating a stream of paired high-gain and low-gain EWMA-filtered values from a stream of samples.**

In the example, the first argument to `zip` is the name of a C struct type that `zip` will create to hold the combined values from the input streams `high` and `low`. By convention, the struct produced by `zip` contains fields with names `item1` ... `item`N, each of which contains a value from the corresponding input stream.

3.4 Interfacing to NesC

The runtime code for each Flask dataflow operator is implemented in NesC, giving application programmers wide latitude in terms of data processing logic and the ability to interface to existing NesC and TinyOS components. In fact, a Flask wiring program compiles into a NesC application that is subsequently compiled to a sensor node binary. Flask provides a simple mechanism for inlining NesC functions, making it easy to write dataflow operators directly within the wiring program. Flask typechecks inputs and outputs to each operator, ensuring that the entire dataflow graph is type safe.

```
let inc = smap <:cfunc<
int inc(int x)
{
  return x + 1;
}
>>
```

Figure 5: **Flask code for the `inc` stream combinator.**

Figure 6 shows the Flask code for a simple dataflow operator, `inc`, that simply increments each integer value that it receives on its input stream. Flask uses the `smap` combinator to lift an inlined NesC function into a Flask dataflow operator; in effect we are mapping a NesC function onto a stream (hence, "smap"). The NesC code is contained within `<:cfunc< ... >>` and is processed by the camlp4 preprocessor [3] to parse the quoted string as NesC syntax. The `smap` combinator takes two arguments, a NesC function and an input stream. In the example, note that we only supply the first argument (the NesC function) to `smap`. The result is that `inc` is a curried function that only requires a single argument, the input stream, which can be instantiated as part of a Flask wiring program.

```
let ewma alpha stream =
    sintegrate
      <:cfunc<
void init(double* state)
{
  *state = 0.0;
}
>>
      <:cfunc<
bool ewma(double* state, double in,
          double* out)
{
  *out = *state = $float:alpha$ * in
    + (1.0 - $float:alpha$) * (*state);
  return TRUE;
}
>>
      stream
```

Figure 6: **Flask code for the `ewma` operator.**

Flask also permits the construction of stateful dataflow operators. While the NesC code within an operator can access global variables (as well as invoke other NesC functions and interfaces), it is often useful to provide an operator with its own private state that is maintained across invocations. Multiple instances of a stateful operator can be used in a Flask program, each with its own private state. As an example, Figure 6 shows the code for the `ewma` operator introduced earlier. This version takes both a gain parameter `alpha` and an input stream `stream` as arguments. The Flask value for `alpha` is injected into the NesC program at compile time, permitting the dataflow operators to be constructed from NesC "templates."

The `sintegrate` combinator is used to construct a stateful operator. A stateful operator must provide an `init()` function that is used to initialize its internal state. The operator's state is passed as an additional argument to the operator's NesC function. Flask determines the type of the internal state by inspecting the type of the `state` argument and ensures that the `init` function uses the same type. The return value of the operator's processing function is a `bool` indicating whether or not the operator produced an output value, allowing operators to optionally consume values without producing new ones.

Programmers are not limited to creating and manipulating dataflow operators when using Flask. In addition to the structured extension mechanisms that allow NesC code to be lifted to a dataflow operator, Flask allows the generated module and configuration to be extended with arbitrary NesC code. Arbitrary declarations and function definitions can be added to the module as well as `uses` and `provides` declarations. Through these mechanisms, dataflow operators can call into pre-existing NesC modules or export additional functionality not provided directly by Flask.

While Flask's NesC interfaces are intended to be very general-purpose, we do not anticipate that most Flask programmers will need most of this generality. As mentioned earlier, Flask provides a suite of built-in operators that provide commonly-used functionality such as timers, sensor data acquisition, and radio communication (described below). In this sense, Flask is a *toolkit* for building up higher-level programming interfaces and does not arbitrarily limit programmers to a single programming abstraction.

3.5 Flows

Flask provides an easy-to-use communication layer, called *Flows*, allowing dataflow graphs on different sensor nodes to be seamlessly integrated. As described in Section 3.2, Flows provide a publish/subscribe model, and can be used to provide a number of commonly-used communication topologies in sensor networks, such as tree-based routing for data aggregation, local communication within neighborhoods, or simple point-to-point routing between arbitrary nodes.

Our prototype implementation of Flows draws on much active research into appropriate routing protocols for sensor networks [27, 11]. Our goal is not to provide the most efficient routing protocol for all applications, but rather an easy-to-use interface that could be backed by different implementations for different domains. The basic protocol design is similar to spanning-tree-based protocols, such as MintRoute [27], but has been generalized to support multiple sinks, as well as a special case for local communication within one radio hop. Our prototype makes use of the link quality indicator (LQI) metric provided by the Chipcon CC2420 radio (used by the MicaZ and TMote Sky mote platforms) to estimate link quality. Multihop routes are chosen by maximizing the estimated probability of packet delivery along the entire path, determined using an empirically-derived function that maps LQI values to delivery ratios.

A Flask program subscribes to a flow of interest using the `recv` combinator, which takes a type specifier and a 16-bit flow identifier as arguments. `recv` returns a stream that emits values of the specified type when messages are received on the requested channel. Publishing to a flow is accomplished using the `send` combinator, which takes a flow ID and a stream as arguments. Values provided by the given stream are transmitted over the specified flow to any subscribers. Flask typechecks the streams passed to `send` and `recv` calls with the same flow ID.

Note that Flask requires the flow IDs supplied to `send` and `recv` to be constant values that can be determined at compile time. This is because a Flask wiring program always generates a static dataflow graph. There is currently no mechanism for dynamically publishing or subscribing to a new flow ID at runtime. We leave to future work the exploration of runtime flow creation. Of course, a Flask application can eschew the use of Flows altogether and implement its own routing protocol directly using the primitive TinyOS communication facilities.

4 Flask Applications

To demonstrate the Flask programming model, in this section we present two concrete examples of Flask applications: a seismic event detection program and an implementation of the TinyDB [18] query language.

4.1 Seismic event detection

We return to the seismic event detection program described earlier. Appendix A shows the complete Flask code for this program, which periodically samples seismic sensor data, passes the data through two EWMA filters, takes the ratio, and sends a "vote" message to the base station if a significant event is detected. The corresponding dataflow graph for this program is shown in Figure 1. For sake of brevity we do not repeat the code for the EWMA operator already shown in Figure 6.

The program is parametrized by several constants. `HIGH_GAIN` and `LOW_GAIN` determine the gain settings for the two EWMA filters. `THRESH` is the ratio threshold that triggers an event report, `PERIOD` is the sampling period of the sensor, and `FLOW` is the flow ID on which results are sent.

Translating the dataflow diagram in Figure 1 to a Flask wiring program is straightforward: wires become streams and dataflow operators become stream combinators. Some additional glue code is necessary, but the majority of the programmer's effort goes towards writing stream combinators to implement dataflow operators. This provides two key advantages to the Flask programmer. First, Flask's simplified concurrency model and de-emphasis of shared state allows the programmer to concentrate on small functional units instead of having to keep the details of large portions of the program "in her head" at any given time. Second, the move from program *specification* (a dataflow graph) to program *implementation* (Flask code) is straightforward.

The Flask wiring program (Appendix A) begins with glue code that adds the definition for a `struct vote_t` which must appear in the generated NesC module. The `vote` combinator pairs the node's ID with a boolean vote and outputs a `struct vote_t`. The rest of the implementation naturally follows from the dataflow graph. We define the `ratio` and `thresh` combinators, both of which correspond to dataflow operators in Figure 1.

Following the combinator definitions, the main wiring program starts on line 32 of the code. This code is only 38 lines long and essentially constructs a the dataflow graph by chaining each combinator together. The final line of code uses the `send` combinator to cause values flowing out of the `vote` combinator to be transmitted to the given flow.

Given the individual operators, building up a dataflow graph is extremely easy. The use of currying for creating stream combinators `thresh` and `ewma` demonstrates how to use Flask functions as "templates" that can be instantiated in a program with different parameters, and even multiple times in the same program. The lack of a complex concurrency model also makes it straightforward to chain operators together.

4.2 FlaskDB

As discussed earlier, Flask is a useful toolkit for building up higher-level sensor network abstractions while providing the efficiency of compiling to NesC. In this section, we describe *FlaskDB*, an implementation of the TinyDB [18] query system in Flask. Unlike TinyDB, which uses a runtime query processing engine, FlaskDB compiles a TinyDB query into a static, fully-optimized sensor node binary. This implies that injecting new queries into the network would require binary-level reprogramming of sensor nodes, although several solutions to this problem have been demonstrated [10, 12]. Our primary goal is to demonstrate the use of Flask for constructing a complex application.

FlaskDB is implemented as a Flask wiring program, called `queryc`, that compiles a TinySQL query to a NesC application. An important aspect of `queryc` is that it constructs the dataflow graph representing the query programmatically, leveraging the full power of the OCaml language and associated tools. In particular, we make use of `ocamllex` and `ocamlyacc`, two existing tools for building parsers in OCaml, for parsing the SQL query into an abstract syntax tree. `queryc` then operates on this AST to generate the dataflow graph.

As in TinyDB, a FlaskDB query is implemented using a spanning tree rooted at the base station. Data aggregation is performed on a hop-by-hop basis with each node collecting local data and that of its children in the routing tree, applying the aggregate function, and passing the result to its parent. A Flask flow is used to construct the routing tree; all nodes in the network publish to a single flow to which the base station subscribes. In-network aggregation is supported using an `intercept` combinator on this flow, which allows each node to intercept messages flowing up the spanning tree.

FlaskDB supports most of the features of TinyDB, including attributes, aggregates, simple arithmetic expressions, and `WHERE` clauses. We do not yet support triggers, logging to flash, `GROUP BY`, or `HAVING` clauses, although these would be straightforward to add. Unlike TinyDB, each selected item in FlaskDB may be an expression containing an arbitrary number of attributes, whereas in TinyDB only one attribute is allowed. Also, `WHERE` clauses may contain both `AND` and `OR` while TinyDB requires `WHERE` clauses to be simple conjunctions. These features would be more difficult to implement in the NesC version of TinyDB, but are easy to build in Flask's compositional wiring language.

We illustrate `queryc`'s operation by walking through the compilation of the query shown in Figure 7. For ease of exposition, we describe a simplified version of `queryc` here; the full version supports a number of optimizations that produce more streamlined NesC code. A TinySQL query is first parsed into an abstract syntax tree using a generated lexer and parser. The AST is processed by a Flask wiring program that builds the associated dataflow graph. Each subterm in the TinySQL expression is compiled to a stream, and these subterm streams are then combined to form a stream for the whole expression. Compiling an expression therefore requires little more than a depth-first traversal of the AST.

First, the `INTERVAL` clause is compiled into a stream of clock impulses at the associated rate. Attributes (such as sensor values, node ID, etc.) are represented internally using a hash table that maps attribute names to stream combinators. Each attribute combinator takes a triggering stream as input and returns a stream of values representing the attribute's value. In Figure 7, the expression `get_attr "id"` returns a combinator defined by the following code:

```
let id =
  smap <:cfunc<
int id(void)
{
  return TOS_LOCAL_ADDRESS;
}>>
```

SQL aggregates (e.g., `count`, `min`, `max`, etc.) require a somewhat more involved structure, since they must maintain state. As in TinyDB, values contributing to an aggregate may be generated internally by the sensor node, or received from child nodes in the spanning tree. We adopt a structure similar to TinyDB's *partial aggregates* for representing the intermediate state of an aggregate. Partial aggregates have four components: a *default value*, an *initialization routine* that creates a partial aggregate from local attributes, a *merge routine* that merges two partial aggregates, and an *evaluation routine* that realizes the final aggregate value represented by the partial aggregate's state.

In Figure 7, several dataflow operators collaborate to implement the `count` aggregate. `count_init` initializes the aggregate from the local `id` attribute. `count_merge` takes in two partial aggregates and produces a merged partial aggregate. When a local sample is triggered, the state representing the `count` aggregate's current partial aggregate is duplicated. One copy is saved in case the `WHERE` clause fails and the other copy is passed through the branch of the dataflow graph that incorporates the node's local readings.

A `WHERE` clause is compiled into a stream that produces boolean values and filters the output of the `SELECT` clause based on whether the boolean value is *true* or *false*. Because nodes in the spanning tree must still route data from their children towards the root (even if local `WHERE` predicates do not match), `queryc` builds two parallel branches of the dataflow path for each case, passing a copy of the query state down each branch. If the `WHERE` clause succeeds, local data is merged with any aggregates and is sent to the destination flow. If the `WHERE` clause fails, the unmodified partial aggregate state received from children in the spanning tree is sent.

FlaskDB consists of 984 lines of Flask wiring code, of which 81 lines consist of NesC glue code (e.g., for accessing local node attributes and constructing partial aggregates). The SQL syntax description used by the lexer and parser generators is an additional 322 lines of code.

5 Evaluation

Our goal in evaluating Flask is to demonstrate that Flask makes it easy to construct sophisticated sensor network applications and achieves comparable overheads (in terms of CPU, memory, and bandwidth usage) to more conventional approaches.

5.1 Code complexity

It is difficult to evaluate the "ease of use" of Flask using quantitative measures. Instead, we discuss complexity in terms of lines of code, comparing it to a breakdown of the TinyDB and QueryVM [15] implementations. TinyDB, QueryVM, and FlaskDB are each implemented in several different languages, so a direct side-by-side comparison is problematic. Still, we report lines of code for each of the components of the three systems as a rough measure of complexity.

Figure 8 shows the source code size breakdown for all three systems. For QueryVM we only included operators that had to be written specifically to support queries. For

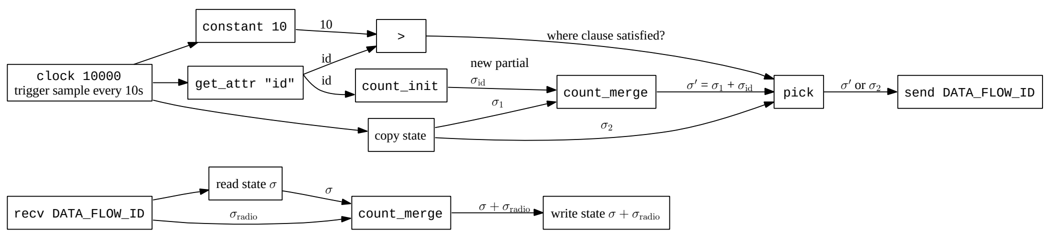

Figure 7: **Dataflow graph for the TinySQL query** `SELECT COUNT(id) WHERE id > 10 PERIOD 10s`**.**

	Lines of code
TinyDB	
Query parser (Java)	820
Query engine (NesC)	7670
Routing (NesC)	781
Resulting NesC app (C)	19446
QueryVM	
Query parser (ML)	321
VM spec (XML)	21
Operators (NesC)	1426
Routing (NesC)	524
Resulting NesC app (C)	19778
FlaskDB	
Query compiler (Flask)	1306
Glue (NesC)	78
Routing (NesC)	2943
Resulting NesC app (C)	7930

Figure 8: **Lines of code for TinyDB, QueryVM, and FlaskDB implementations.** *This is intended as a rough comparison of the code complexity for each of the three implementations of the TinyDB query model.*

Flask we did not include the base library of combinators, but only the additional code required to support query compilation.

As the figure shows, writing a query compiler in Flask requires less code than either TinyDB or QueryVM. The amount of C code output by the NesC compiler is also significantly smaller. The query compiler for QueryVM compiles to motlle, a language with a separate compiler targeting Maté. We did not include code from the motlle compiler or runtime in our line counts.

5.2 Communication overhead

The use of Flows simplifies Flask application design by providing a ready-to-use, general-purpose communications substrate. Our goal with Flows has never been to provide the most efficient protocol implementation, but rather one that can be readily plugged into a number of applications.

We compare the communication overhead of Flows with *MultiHopLQI*, an updated version of the MintRoute protocol [27] for the Chipcon CC2420 radio. MultiHopLQI forms a spanning tree rooted at a base station node. Each node uses a scaled version of the radio's link quality indicator (LQI) metric to select a parent in the tree. We test Flows and MultiHopLQI using a simple program that causes each node to route a single packet to the base station once a second. In both cases, nodes performed hop-by-hop acknowledgment and retransmission, attempting to retransmit each packet up to 5 times. The message queue size was set to 16 and the neighbor history table size to 10.

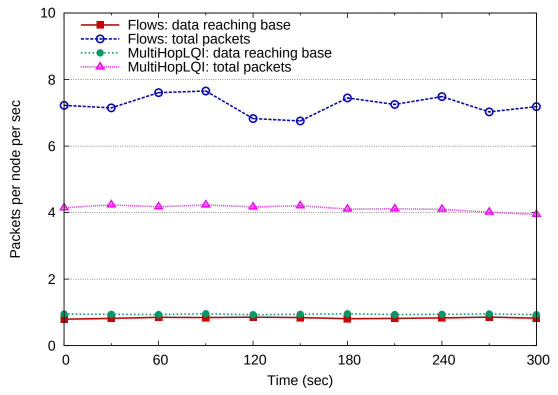

Figure 9: **Flask and MultiHopLQI overhead comparison.** *This plot shows the number of distinct data messages reaching the base station as well as the routing overhead as a function of time.*

Each experiment was executed using a testbed of 75 TMote Sky sensor nodes distributed over several floors of an office building. Each node is connected to a USB-Ethernet bridge (TMote Connect from Moteiv, Inc.) allowing nodes to be reprogrammed and debugged over a TCP/IP port; nodes are powered by the bridge. Because nodes are distributed widely throughout the building, the testbed experiences a wide range of radio link qualities and realistic packet loss. Each experiment was run for 5 minutes. The routing tree formed by the Flow is shown in Figure 10. The maximum depth of the tree is 7 hops. A similar routing topology was seen when using MultiHopLQI.

Figure 9 shows the the number of distinct data packets received at the base station node, as well as the corresponding protocol overhead. Protocol overhead includes link-estimation and route-discovery packets as well as duplicate transmissions due to packet loss. As the figure shows, both Flows and MultiHopLQI have approximately the same delivery ratio. Flows have about twice the overhead of MultiHopLQI in this configuration, although it does suffer additional overhead resulting from its increased generality. Despite the additional overhead, Flows operates reasonably well compared to state-of-the-art routing protocols. It is worth keeping in mind that the Flows implementa-

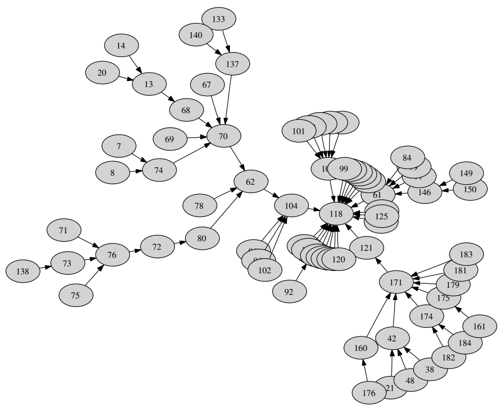

Figure 10: **Network topology for the sensor network testbed used in Figure 9.**

```
SELECT id, seqno INTERVAL 50s
```

(a) Simple query

```
SELECT COUNT(id), AVG(mag),
WEIGHTAVG(x,mag), WEIGHTAVG(y,mag)
WHERE mag > 200
```

(a) Complex query

Figure 12: **FlaskDB queries used in Figure 11.**

tion is independent of the communication API underlying Flask, meaning a new protocol can be used without changing Flask application code.

5.3 Memory size

Flask's ability to compile directly to NesC provides a significant savings in binary size and runtime memory requirements. Figure 11 shows a breakdown of the memory requirements for TinyDB, QueryVM (implemented using Maté), and two different FlaskDB queries shown in Figure 12. These sizes represent the TinyOS binary for the MicaZ platform, compiled for the Atmega128L microcontroller.

The "simple" query issues a periodic report of each node's ID and temperature every 50 sec. The "complex" query represents a tracking system that uses magnetometers to detect the position of a vehicle moving in a sensor field. The query reports both the number of nodes reporting a nearby vehicle (over the detection threshold), and the magnetometer readings weighted by the x and y components of the node's location. This is a simple tracking algorithm featured in several previous papers [14, 23]. Although TinyDB does not support the `WEIGHTAVG` operator, it was straightforward to implement this feature in Flask by building the appropriate combinators in the wiring program.

We break the code size down by corresponding functionality into three categories: *TinyOS Base* (core TinyOS components), *Communications* (link and radio protocols), and *Application* (query-specific logic). As the table shows, the application code and memory size for FlaskDB queries

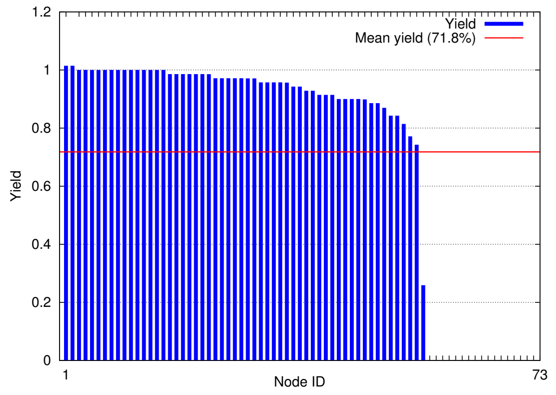

Figure 13: **Per-node yield for the FlaskDB query** `SELECT id, seqno INTERVAL 10s`.

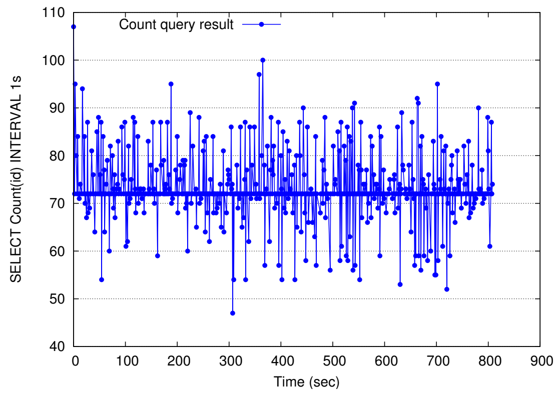

Figure 14: **Results from the FlaskDB query** `SELECT count(id) INTERVAL 1s`.

is about an order of magnitude less than in TinyDB and QueryVM, which each require a large, general-purpose query processing engine (as in TinyDB) or virtual machine (as in QueryVM). In contrast, FlaskDB queries are compiled to an optimized node-level binary. The total binary size of a FlaskDB query is only 41% of TinyDB, and 35% of QueryVM. If it is necessary to inject new queries into the network at runtime, over-the-air reprogramming systems such as Deluge [10] and MNP [12] can be readily employed.

The Flows protocol implementation is somewhat more complex than the simpler routing trees used in TinyDB and QueryVM, leading to larger code size. In addition, Flows (as configured in this setting) allocates a pool of 20 radio message buffers, explaining the higher RAM requirements. The Flows implementation can be readily tuned to reduce this size, and the greatly reduced application memory requirement for a compiled query offsets this overhead.

5.4 FlaskDB Performance

Our final evaluation of Flask demonstrates the performance of two simple FlaskDB queries: one that reports the ID of each node every 10 sec, and another that reports a count of the total number of nodes in the network every 1 sec. We have been unable to get TinyDB and QueryVM running on

	TinyOS Base		Communications		Application		**Total**	
	ROM	RAM	ROM	RAM	ROM	RAM	ROM	RAM
TinyDB	10248	402	3258	934	46572	1293	60078	2629
QueryVM	11720	345	4300	642	54710	2652	70730	3639
FlaskDB simple	8508	430	7910	2454	4996	142	21414	3026
FlaskDB complex	8450	430	8224	2454	6466	159	23140	3043

Figure 11: **MicaZ binary size (in bytes) for TinyDB, QueryVM, and FlaskDB.**

our testbed; this is partly because TinyDB does not currently support the TMote Sky node platform. Rather, we report the quality of the data obtained by these queries as an indication that FlaskDB is working in a realistic end-to-end setting.

Figure 13 reports the *yield* of a simple query that requests each node to report its local ID and a unique sequence number once every second; yield is measured as the fraction of expected results that arrive at the base station. As the figure shows, the data yield is very high, averaging 93.5% for those nodes that managed to report any data back to the base station. 19 out of the 75 nodes in this experiment did not join the routing tree and as such no data was received from them during this run.

Figure 14 shows the result of running the `SELECT count(id) INTERVAL 1s` query on the network. In this run, only 73 nodes in the testbed were active, so the expected result for each interval is 73. As the graph shows, for the first 60 sec of the query, the results are somewhat erratic, as nodes join the routing tree and the topology stabilizes. Thereafter the average result reported is 71.

In a few instances there is an aliasing effect where the query result counts too few nodes but the subsequent result counts an identical number of additional nodes. This is simply because of the timeouts used in our implementation: in some cases, the query result fires before all children have reported their values to their parent but these tuples are counted in the next period. The "corrected count" line in the figure filters out this phenomenon, showing that in a very small number of cases loss of a packet in the network results in the count being reported as too small. Cases where the query result is too large can be explained by result packet being dropped in combination with the aliasing effect just described.

6 Future Work and Conclusions

Flask combines the strengths of the high-level functional language OCaml and NesC. Using Flask, a programmer can construct reusable, parametrized stream combinators for manipulating data, free of low-level implementation details while still escaping to NesC when access to such detail is necessary. Because the Flask wiring language is run only at compile time, one can still maintain tight bounds on node resource usage, and the resulting compiled programs are efficient. Programs are expressed concisely in terms of streams and stream combinators, and the high-level, logical flow of data through a system embodied in dataflow diagrams maps directly to Flask code. Flask generates programs that run efficiently on real hardware.

We view Flask as a firm foundation for future tools that continue to ease the burden placed on sensor network programmers. Some of the approaches we wish to explore in future work are described in summary below.

- **Core ML object language:** Flask makes extensive use of NesC. This can be awkward, particularly because NesC lacks ML-style tuples which leads to a proliferation of `struct` declarations. We have begun writing a prototype that allows a cut-down subset of ML to be used to parametrize combinators instead of NesC. Strict bounds on resource usage are maintained by disallowing closures, refs, lists, and any other language features, such as garbage collection, that would require extra runtime overhead. This language fits much more naturally with the rest of Flask and reduces the amount of code that must be written when creating combinators.

- **Graphical language for distributed operator graphs:** Although Flask focuses on programming individual nodes in a sensor network, its use of Flows to provide a unified interface for data streams transmitted over the radio and node-local data streams leads naturally to moving from node-local dataflow graphs to network-wide *distributed dataflow graphs* where the placement of dataflow operators in the network is automatically generated by the compiler. We plan to investigate writing a compiler in Flask for a graphical language that directly represents these distributed dataflow graphs.

- **Macroprogramming:** FlaskDB and the distributed dataflow graph language just described are both examples of *macroprogramming*. Our experience implementing FlaskDB leads us to believe that Flask is an excellent general substrate on which to build other macroprogramming tools. However, it is unclear how general the abstractions we have developed for Flask really are and if they provide a natural foundation for a wide-range of macroprogramming systems. Our work in exploring this wide-open space is ongoing.

References

[1] A. Bakshi, V. K. Prasanna, J. Reich, and D. Larner. The abstract task graph: A methodology for architecture-independent programming of networked sensor systems. In *Proc. Workshop on End-to-End, Sense-and-Respond Systems, Applications, and Services*, pages 19–24, 2005.

[2] K. Chintalapudi, J. Paek, N. Kothari, S. Rangwala, J. Caffrey, R. Govindan, E. Johnson, and S. Masri. Monitoring Civil Structures with a Wireless Sensor Network. IEEE Internet Computing, March/April 2006.

[3] D. de Rauglaudre. Camlp4 reference manual, version 3.07. http://caml.inria.fr/pub/docs/manual-camlp4/index.html.

[4] D. Ganesan, B. Greenstein, D. Perelyubskiy, D. Estrin, and J. 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.

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

[6] B. Greenstein, E. Kohler, and D. Estrin. A sensor network application construction kit (SNACK). In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 69–80, New York, NY, USA, 2004. ACM Press.

[7] R. Gummadi, O. Gnawali, and R. Govindan. Macro-programming wireless sensor networks using Kairos. In *Proc. DCOSS'05*, 2005.

[8] J. S. Heidemann, F. Silva, C. Intanagonwiwat, R. Govindan, D. Estrin, and D. Ganesan. Building efficient wireless sensor networks with low-level naming. In *Proc. the 18th SOSP*, Banff, Canada, October 2001.

[9] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. E. Culler, and K. 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, Nov. 2000.

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

[11] C. Intanagonwiwat, R. Govindan, and D. Estrin. Directed diffusion: A scalable and robust communication paradigm for sensor networks. In *Proc. International Conference on Mobile Computing and Networking*, Aug. 2000.

[12] S. S. Kulkarni and L. Wang. MNP: Multihop Network Reprogramming Service for Sensor Networks. In *Proc. 25th IEEE International Conference on Distributed Computing Systems (ICDCS'05)*, pages 7–16, 2005.

[13] X. Leroy. The objective caml system, release 3.09. http://caml.inria.fr/pub/docs/manual-ocaml/index.html.

[14] P. Levis and D. Culler. Maté: A tiny virtual machine for sensor networks. In *Proc. the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS X)*, 2002.

[15] 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.

[16] P. Levis, D. Gay, and D. Culler. Bridging the gap: Programming sensor networks with application specific virtual machines. Technical Report UCB//CSD-04-1343, U.C. Berkeley, August 2005.

[17] L. Lu, T. He, T. Abdelzaher, and J. Stankovic. Design and comparison of lightweight group management strategies in EnviroSuite. In *Proc. International Conference on Distributed Computing in Sensor Networks (DCOSS)*, Marina Del Rey, CA, June 2005.

[18] 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.

[19] 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.

[20] 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.

[21] S. N. Pakzad, S. Kim, G. L. Fenves, S. D. Glaser, D. E. Culler, and J. W. Demmel. Multi-purpose wireless accelerometers for civil infrastructure monitoring. In *Proc. 5th International Workshop on Structural Health Monitoring (IWSHM 2005)*, Stanford, CA, September 2005.

[22] 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.

[23] 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.

[24] G. Werner-Allen, K. Lorincz, M. Ruiz, O. Marcillo, J. Johnson, J. Lees, and M. Welsh. Deploying a wireless sensor network on an active volcano. *IEEE Internet Computing*, March/April 2006.

[25] K. Whitehouse, J. Liu, and F. Zhao. Semantic Streams: a framework for composable inference over sensor data. In *Proc. Third European Workshop on Wireless Sensor Networks (EWSN)*, Zurich, Switzerland, February 2006.

[26] 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.

[27] A. Woo, T. Tong, and D. Culler. Taming the underlying challenges of reliable multihop routing in sensor networks. In *Proc. the First ACM Conference on Embedded Networked Sensor Systems (SenSys 2003)*, November 2003.

[28] N. Xu, S. Rangwala, K. Chintalapudi, D. Ganesan, A. Broad, R. Govindan, and D. Estrin. A Wireless Sensor Network for Structural Monitoring. In *Proceedings of the ACM Conference on Embedded Networked Sensor Systems*, pages 13–24. ACM Press, November 2004.

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

A Appendix: The full seismic event detection program

```
let _ = add_decl <:cdecl<
struct vote_t {
    typename uint16_t id;
    bool vote;
};
>>

let vote =
  smap <:cfunc<
bool vote(bool v)
{
  struct vote_t result =
      {TOS_LOCAL_ADDRESS, v};

  return result;
}

let ratio =
  smap <:cfunc<
float f(struct pair p)
{
  return p.item2 / p.item1;
}

let thresh t =
  smap <:cfunc<
bool f(float x)
{
  return x > $float:t$;
}

let clk = clock PERIOD in
let mag = adc "Seismometer" clk in
let high = ewma HIGH_ALPHA mag in
let low = ewma LOW_ALPHA mag in
let r = ratio (F.zip "pair" low; high) in
let t = thresh THRESH r in
  send (vote t) FLOW
```