The Design and Implementation of A Declarative Sensor Network System

David Chiyuan Chu
Lucian Popa
Arsalan Tavakoli
Joseph M. Hellerstein
Philip Levis
Scott Shenker
Ion Stoica

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2006-132
http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-132.html

October 16, 2006

The Design and Implementation of A Declarative Sensor Network System

David Chu*, Lucian Popa*, Arsalan Tavakoli*, Joseph M. Hellerstein*, Philip Levis†, Scott Shenker*, Ion Stoica*
*EECS Computer Science Division, University of California, Berkeley, CA 94720
Email: {davidchu,popa,arsalan,istoica,hellerstein}@cs.berkeley.edu, shenker@icsi.berkeley.edu
†Department of Electrical Engineering and Department of Computer Science, Stanford University, Stanford, CA
Email: pal@cs.stanford.edu

***Abstract*—**
Sensor networks are notoriously difficult to program, given that they encompass the complexities of both distributed and embedded systems. To address this problem, we present the design and implementation of a declarative sensor network platform, *DSN*: a declarative language, compiler and runtime suitable for programming a broad range of sensornet applications. We demonstrate that our approach is a natural fit for sensor networks by specifying several very different classes of traditional sensor network protocols, services and applications entirely declaratively – these include tree and geographic routing, link estimation, data collection, event tracking, version coherency, and localization. To our knowledge, this is the first time these disparate sensornet tasks have been addressed by a single high-level programming environment. We address a number of systems challenges that arise when building a generic compiler and runtime environment for the sensornet context; these include not only issues of limited resources, but also the management of asynchrony and requirements of predictable execution. Our results suggest that the declarative approach is well-suited to sensor networks, and that it can significantly improve software productivity and quality while still producing efficient, resource-constrained code.

I. Introduction

Wireless sensor networks have received significant research attention in the last decade, spawning multiple conferences, a variety of hardware and software implementations, and a nascent industry. Despite years of research, however, sensornet programming is still very hard. Most sensornet protocols and applications continue to be written in low-level embedded languages like NesC [11], and wrestle explicitly with issues of wireless communication, limited resources, and asynchronous event processing. This kind of low-level programming is challenging even for experienced programmers, and hopelessly complex for typical end users. The design of a general-purpose, easy-to-use, efficient programming model remains a major open challenge in the sensor network community.

In this paper we present the design and implementation of a *declarative sensor network (DSN)* platform: a programming language, compiler and runtime system to support declarative specification of wireless sensor network applications. Declarative languages are known to encourage programmers to focus on program outcomes (*what* a program should achieve) rather than implementation (*how* the program works). Until recently, however, their practical impact was limited to core data management applications like relational databases and spreadsheets [19]. This picture has changed significantly in recent years: declarative approaches have proven useful in a number of new domains, including program analysis [40], trust management [4], and distributed system diagnosis and debugging [3], [35]. Of particular interest in the sensornet context is recent work on *declarative networking*, which presents declarative approaches for protocol specification [27] and overlay network implementation [26]. In these settings, declarative logic languages have been promoted for their clean and compact specifications, which can lead to code that is significantly easier to specify, adapt, debug, and analyze than traditional procedural code.

Our work on declarative sensor networks originally began with a simple observation: by definition, sensor network programmers must reason about both data management and network design. Since declarative languages have been successfully applied to both these challenges, we expected them to be a good fit for the sensornet context. To evaluate this hypothesis, we developed a declarative language that is appropriate to the sensornet context, and then developed fully-functional declarative specifications of a broad range of sensornet applications. In Section IV, we present some examples of these declarative specifications: a data collection application akin to TinyDB [29], a software-based link estimator, several multi-hop routing protocols including spanning-tree and geographic routing, the version coherency protocol Trickle [24], the localization scheme NoGeo [34] and an event tracking application faithful to a recently deployed tracking application [30]. The results of this exercise provide compelling evidence for our hypothesis: the declarative code naturally and faithfully captures the logic of even sophisticated sensornet protocols. In one case the implementation is almost a *line-by-line translation* of the protocol inventors' pseudocode, directly mapping the high-level reasoning into an executable language.

Establishing the suitability of the language is of course only half the challenge: to be useful, the high-level specifications have to be compiled into code that runs efficiently on resource-constrained embedded nodes in a wireless network. We chose to tackle this issue in the context of Berkeley Motes and TinyOS, with their limited set of hardware resources and lean system infrastructure. This context presents a number of challenges that were not addressed in the prior work on declarative networking for the Internet. These include:

- *Predictable Execution:* Sensor networks are often intended to run unattended for long durations. Thus, operational predictability is a key concern.
- *Memory Constraints:* Memory is limited in many current sensor network platforms. While these constraints may be eased in some upcoming platforms, other platforms may keep memory constant in favor of increased miniaturization and decreased cost.
- *Asynchrony:* Because sensor networks are tightly coupled with their physical environment, asynchronous software execution is very important. At the same time, asynchrony can make programming difficult, and it is not traditionally captured in declarative languages.

We highlight these particular challenges because they had broad impact on our design decisions in DSN. We will return to these issues during the course of this paper, in discussions of our declarative language as well as the system runtime.

Our contributions in this work are threefold:

- First, we present a declarative language called Snlog, demonstrating its suitability to sensor networks via a broad variety of example programs that faithfully replicate prior work in a compact and concise manner. We show that these implementations can be composed into whole sensor network system stacks comparable to both proposed and actually deployed sensor network systems.
- Second, we present the architecture and implementation of our declarative system's compiler and runtime, which are targeted specifically to the wireless sensor network domain.
- Finally, we present an evaluation of the feasibility and faithfulness of our system, demonstrating that the resulting code can run on resource-constrained nodes, and that the code does indeed perform according to specification.

The rest of the paper is organized as follows. The next section sets our work in context. Sections III and IV outline the Snlog language, and provide examples of a variety of protocols and applications. Sections V and VI present an architectural overview of the system, along with implementation concerns. Section VII discusses evaluation methodology, measurements and results. Section VIII concludes with a discussion.

II. Related Work

Numerous deployment experiences have demonstrated that developing low-level software for sensor nodes is prohibitively difficult [36], [39]. This challenge has led to a large body of work exploring high-level programming models that capture application semantics in a simple fashion. By borrowing languages from other domains, these models have demonstrated that powerful subsets of requisite functionality can be easily expressed. TinyDB showed that the task of requesting data from a network can be written via declarative, SQL-like queries [29] and that a powerful query runtime has significant flexibility and opportunity for automatic optimization. Abstract regions [38] and Kairos [14] showed that data-parallel reductions can capture aggregation over collections of nodes, and that such programs have a natural trade off between energy efficiency and precision. SNACK [13] and Tenet [12] demonstrated that a dataflow model allows multiple data queries to be optimized into a single efficient program. Following the same multi-tier system architecture of Tenet, semantic streams [41] showed that a coordinating base station can use its knowledge of the network to optimize a declarative request into sensing tasks for individual sensors.

From these efforts it appears that declarative, data-centric languages are a natural fit for many sensor network applications. But this work typically focuses on data-centric matters, leaving many core networking issues to built-in library functions. Tenet even takes the stance that applications should not be introducing new protocols, delegating complex communication to resource-rich higher-level devices. Our goal in DSN is to more aggressively apply the power of high-level declarative languages in sensornets, both for data acquisition and for a wide range of networking logic involved in communicating that data.

In the Internet domain, the P2 project [25]–[27] demonstrated that declarative logic languages can concisely describe many Internet routing protocols and overlay networks. Furthermore, the flexibility the language gives to the runtime for optimization means that these high-level programs can execute efficiently.

DSN takes these efforts and brings them together, defining a declarative, data-centric language for describing data management and communication in a wireless sensor network. From P2, DSN borrows the idea of designing a protocol specification language based on the recursive query language Datalog. Sensornets have very different communication abstractions and requirements than the Internet, however, so from systems such as ASVMs [22], TinyDB [29], and VM* [18], DSN borrows techniques and abstractions for higher-level programming in the sensornet domain. Unlike these prior efforts, DSN pushes declarative specification through an entire system stack, touching applications above and single-hop communication below, and achieves this in the kilobytes of RAM and program memory typical to sensor nodes today.

III. Snlog Language

In this section we give an overview of our declarative language, Snlog. The typical DSN user, whether an end-user or an algorithms/service designer, writes only a short declarative specification using the language described below.

A. Snlog Language

Snlog is a dialect of Datalog [33]. Snlog's main language constructs are variables, constants, predicates, facts and rules. An Snlog program consists of statements, each terminated with a period. As a quick start, the following is a typical, but simplified, example of these elements:

```
measurement(Celsius,Time) :- timestamp(Time),
    temperature(RawReading), samplingOn(true), Celsius
     = f_raw2celsius(RawReading).
samplingOn(false).
```

Measurement, timestamp, temperature, and samplingOn are *predicates*. By instantiating a predicate with variable assignments, *tuples*

are created for that predicate. A *fact* is a tuple that is instantiated at the beginning of execution, such as samplingOn(false). A *rule* specifies tuples whose instantiations are based on the truth of a logical expression. The *body* of a rule defines a set of preconditions, which if all true, make the *head* true. For example, measurement(Celsius,Time) is the head of the rule on line 1, while the timestamp, temperature, and samplingOn predicates form its body. This rule can be read as follows: "if the timestamp is Time, and if the °C conversion of a temperature reading is Celsius, and if sampling is on, then we have a measurement of Celsius at Time." As the second line sets samplingOn to be false, the measurement rule produces no tuples: its body is never true. f_raw2celsius is a function implemented by a lower level, native code module. Following Datalog convention, predicates and constants start with lowercase while letters and variables start with upper case letters: Celsius is a variable, while true is a constant.

Unification is the process of matching variables across different predicates in a rule. For example, this rule:

```
head(A, B, C) :- bodyOne(A, B), bodyTwo(B, C, _).
```

says that head is true if bodyOne and bodyTwo each have a tuple such that the second field of the bodyOne tuple is the same value as the first field of the bodyTwo tuple. This process is checked for all tuples generated for the two predicates. In relational database terminology, unification is an equality join. The "_" appearing in the bodyTwo predicate means that in this rule we do not care what the value of this variable is.

Distributed Execution: In a fashion similar to [26], one argument of each predicate, the *location specifier*, is additionally marked with an "at" symbol (@) representing the host node id of the predicate. A single rule may involve predicates hosted at different nodes. For example, the rule:

```
toSendPredicate(@Z,C) :- localPredicate(@S,Z,C).
```

makes the predicate toSendPredicate(@Z,C) true at host Z if the predicate localPredicate(@S,Z,C) is true at host S. Practically, this can be accomplished by each node S checking for predicates of type localPredicate and sending the corresponding toSendPredicate to Z. This send is just a simple local neighborhood unicast.

As in [26], there is no explicit syntax for specifying data transmission in Snlog. Instead, the location specifiers describe the intended sources and destinations of data. It is up to the compiler and runtime to decide on a communication pattern to achieve the declarative specification of each rule's head given its body.

In particular, the host variables for two or more body predicates may differ. For example:

```
toCheckPredicate(@S,C) :- localPredicate(@S,Z,C),
    remotePredicate(@Z,C).
```

In such a scenario, internal rule rewriting in DSN automatically rewrites this to:

```
remotePredicateMsg(@*,Z,C) :- remotePredicate(@Z,C).
toCheckPredicate(@S,C) :- localPredicate(@S,Z,C),
    remotePredicateMsg(@S,Z,C).
```

where remotePredicateMsg is sent to all neighbors and unifed locally at each neighbor by the second rule. Snlog introduces a new @* construction to specify a broadcast message. For any receiver, the predicate is simply rewritten to replace @* with the receiver's host.

Built-ins: In order to link the declarative user program to specific sensor network hardware such as sensors, LEDs, radios, and timers, we allow the user to specify *built-in predicates* or to use them from our library. For example, we use the following Snlog construction to define a predicate that reads values from the light sensor:

```
builtin(light, LightImplementingModule).
```

where LightImplementingModule is the low level module (written in a language like nesC [11]) implementing the light predicate. Currently, there are four built-in predicates in DSN's library: the neighbor table link, timers timer, LEDs and temperature sensors. All of the user programs presented in the remainder of this paper are built on top of these predicates. These represent the only interfaces to the sensor network hardware. At the opposite extreme, one can in principle implement significant complexity behind built-ins, and use Snlog quite sparingly to script together the complex built-ins. However, our goal in this work is to avoid this approach, and show that it is natural to bring almost all sensornet logic into the high-level declarative framework.

Specifying Types: To efficiently support Snlog on motes we gather type information for each of the predicate's parameters with a special language construction. For example:

```
typedef(pred1, ^uint16_t, ^uint32_t, uint8_t).
```

Static knowledge of types allows for efficient memory allocation at compile time. We perform whole-system type-inference on predicate parameters such that all parameters' types are fixed yet only minimal type specifications are necessary. Our current implementation supports primitive types such as char and int. Also, Snlog allows programmers to define primary keys on predicates by using the hat symbol ("^") in front of the attribute type. In the example above, the first two attributes are the primary key. This means that only one tuple with a unique combination of those attributes can exist in the system at one time. We support simple "replace or deny" resolution policies when multiple primary keys of the same value exist.

Materialization and memory Since we deal with very strict memory constraints, we give substantial control to the user to specify all of the following per predicate type: a timeout for tuple existence, a maximum number of allowed tuples per predicate, and an eviction policy when the maximum size is achieved:

```
materialize(predicateName, timeout, maxEntries,
    evictionPolicy).
```

Unlike most logic programming systems, DSN must confront the realities of limited memory. There are no infinite or even generous storage spaces, so choosing what to remember is a real problem. In general, our experience suggests that the symptoms, *i.e.,* eviction or denied insertion, are not easy to reason about. Snlog attempts to alleviate this problem for

users in two ways. First, memory for each distinct predicate is allocated statically and not shared with other predicates. This provides a significant degree of predictability, since assumptions on available memory can be checked before execution and possible contention is only among tuples of the same predicate. Second, the primary key resolution policy mentioned previously further assists in localizing contention when primary keys match.

Priorities: Datalog systems typically provide formal semantics that define the derived set of tuples for each predicate at the end of program execution [33]. In centralized Datalog engines this semantics is traditionally achieved by an algorithm known as "semi-naive" evaluation [33]. This simple algorithm – and its recent distributed variants [25] – are easy to provide given generous memory for buffering and reliable communication. Unfortunately, these are not assets at our disposal in the sensor network context. In order to achieve predictable results when memory is exhausted, we use somewhat more operational semantics, but we give the programmer more control over rule execution to determine the outcome of the program.

In particular, we allow the programmer to assign priorities to predicates when firing rules, in a manner reminiscent of production rule systems like OPS5 [10]. This means that if we have multiple new tuples from different predicates that can potentially fire different rule bodies, these tuples will be processed in order of their predicate's priority. Though we make no guarantees over the order in which multiple rules that are fired by the same predicate are processed, this mechanism is powerful enough for all applications that we encountered. The priority specification is done by the following construct:

```
priority(predicateName, priorityValue).
```

Priorities take integer values, the default being zero, and priority increases with numerical values.

Beyond the fact that correct execution of traditional Datalog is not possible given limited memory, traditional semi-naive evaluation also provides no control over the order of execution of predicates. While from a declarative logic perspective, such order is immaterial, from a systems perspective, it can be helpful to control that order. For example, when multiple services exist in the same application, some of which might be critical (*e.g.,* new code update), priorities can help establish processing orders.

Suppressing rule execution: For program correctness, we may desire that tuples of one predicate do not fire a rule. We achieve this by adding a ~ after the non firing predicate:

```
head(A, B, C) :- triggeringPredicate(A, B),
     nonTriggeringPredicate(B, C, _)~.
```

In the above rule, a new tuples of type nonTriggeringPredicate will not fire the rule but will be available for joins when new triggeringPredicate tuples fire.

Getting Data Out of Snlog Programs: Users can generate *queries* by appending the question-mark ("?") to predicates, to specify that the predicates should be output from the Snlog runtime. For example, the following query outputs tuples from a particular predicate whenever the second field matches a particular value:

```
1 typedef(produce, uint16_t, uint8_t, uint32_t).
2 builtin(produce, SomeProducer).
3
4 store(@Y, Oid, Object):- produce(@X, Oid, Object),
      consume(@Y, Oid).
5 consume(@base, someSensorTypeId).
```

Listing 1. Single-Hop Collection

```
interestingPredicate(@AllHosts, InterestingValue)?
```

When a new tuple (for one of these predicates) is generated, it is also transmitted to an interface to the outside world, the current interface being the UART (Universal Asynchronous Receiver-Transmitter) interface. If no query is specified in a program, all the predicates are considered of interest and are sent to the UART interface.

IV. Example Applications

In this section, we investigate Snlog's potential for expressing core sensor network protocols, services and applications. Through a series of examples, we tackle different sensor network problems, at multiple, traditionally distinct levels of the system stack.

Single-Hop Collection and Dissemination: Our first example, Listing 1, is a data collection application that gathers data at a node from all the other nodes in its network range. It is simple, but appropriate for many simple sensor network deployments and is easily expressed with a single rule.

Line 4 is read as: if a node X produces a data object identified by Oid, and another (neighboring) node Y consumes Oid, then this Object should be stored with store at Y. As noted in the listing, the produce predicate is built-in. In the collection context, it can be thought of as generating a tuple per sensor reading. Combined with line 5's fact and line 4's rule, the behavior is to send produced readings to the base station.

With a different fact such as consume(@Node,someCtrlTypeId), Node != base., the same rule of line 4 that was used for collection can be reused for dissemination as well. Here, produce can be thought of as emitting new control tuples; the base station broadcasts these messages to all nodes in its local neighborhood.

Tree Routing: In-network spanning-tree routing is a well-studied sensor network routing protocol. Tree construction is a special case of the Internet's Distance Vector Routing (DVR) protocol: nodes simply construct a spanning tree rooted at the base by choosing their next hop neighbor that advertises the shortest cost to the base. This tree construction in Snlog is presented in Listing 2.

In this program link(@Host,Neighbor,Cost) is a built-in that contains one tuple for each neighbor (line 2). The dest fact of line 16 defines a tree rooted at node 0. Line 10 and 11 are the base case and inductive case respectively for recursively building network paths to the root; nodes further and further from the root progressively learn of candidate paths. Line 12 uses an additional Snlog construct, MIN aggregation, to compute the shortest cost path seen so far from all paths known. Lastly, line 13 creates a traditional network forwarding table *i.e.,* predicates of the form nextHop(Src,Dst,NextHop,Cost).

```
1 typedef(link,uint16_t,uint16_t,uint8_t).
2 builtin(link,LinkTblC).
3 builtin(f_add,FuncAdd).
4 typedef(#nextHop,^uint16_t,^uint16_t,uint16_t,uint8_t).
5 typedef(#shortestCost,^uint16_t,^uint16_t,uint8_t).
6 materialize(dest,infinite,1,evict_random).
7 priority(shortestCost,1).
8
9 %——— Tree construction
10 path(@S,D,D,C) :- dest(@S,D)~, link(@S,D,C).
11 path(@S,D,Z,C) :- dest(@S,D)~, link(@S,Z,C1),
       nextHop(@Z,D,W,C2), C=f_add(C1,C2), S != W.
12 shortestCost(@S,D,<MIN,C>) :- path(@S,D,Z,C),
       shortestCost(@S,D,C2)~, C < C2.
13 nextHop(@S,D,Z,C) :- shortestCost(@S,D,C), path(@S,D,Z,C)~.
14
15 %——— Tree root
16 dest(@*,0).        % base id is zero
17 shortestCost(@*,0,255). % bootstrap cost to base
```

Listing 2. Tree Routing

```
1 import('tree.snl').
2
3 %——— Periodic Temperature Transmissions
4 toTransmit(@S,Reading) :- temperature(@S,Reading),
       timer(@S,1,_).
5 timer(@S,1,Tval) :- timer(@S,1,Tval).
6
7 %——— Message Forwarding
8 % package message for transmission
9 message(@Next,Src,Dst,Obj) :- toTransmit(@Src,Obj),
       nextHop(@Src,Dst,Next,Cost)~.
10 % forward to nexthop
11 message(@Next,Src,Dst,Obj) :- message(@Crt,Src,Dst,Obj),
       nextHop(@Crt,Dst,Next,Cost)~.
12 % store when at destination
13 store(@S,Src,Obj) :- message(@S,Src,Dst,Obj), S == Dst.
```

Listing 3. Multi-hop Collection/Dissemination

```
1 priority(computedDistances,2).
2 priority(shortestCost,1).
3
4 % broadcast own location to neighbors
5 location(@*,S,Xs,Ys) :- location(@S,S,Xs,Ys).
6
7 %——— geographic routing and forwarding
8 % forward message along nextHop
9 message(@Next,Src,Dst,Xd,Yd,Data) :-
       message(@Crt,Src,Dst,Xd,Yd,Data),
       nextHop(@Crt,Next,Xd,Yd), Dst != Crt.
10
11 % dynamically choose neighbor with min dist to destination
       as nextHop
12 computedDistances(@Crt,N,Xd,Yd,Dist) :-
       message(@Crt,Src,Dst,Xd,Yd,Data), link(@S,N,Cost),
       location(@Crt,N,Xn,Yn), Dist =
       f_distance(Xd,Yd,Xn,Yn), Dst != S.
13 shortestCost(@S,Xd,Yd,<MIN,Dist>) :-
       computedDistances(@S,_,Xd,Yd,Dist).
14 nextHop(@Crt,Next,Xd,Yd) :- shortestCost(@Crt,Xd,Yd,Dist),
       computedDistances(@S,Next,Xd,Yd,Dist), Dist <
       f_distance(Xd,Yd,MyX,MyY), location(@Crt,Crt,MyX,MyY).
15
16 % fallback routing e.g. right-hand rule or convex hull
17 fallback(@Crt,Xd,Yd,Dist) :- shortestCost(@Crt,Xd,Yd,Dist),
        computedDistances(@S,Next,Xd,Yd,Dist), Dist >=
       f_distance(Xd,Yd,MyX,MyY), location(@Crt,Crt,MyX,MyY).
18
19 %——— example messages sent
20 message(@Src,Src,Dst,Xd,Yd,Value) :-
       temperature(@Src,Value),temperatureCollector(@Src,Dst),
        location(@Src,Dst,Xd,Yd).
```

Listing 4. Geographic Routing

Lines 1-7 define types, primary key and table policies, and built-ins. From this point onward, we shall elide initialization statements unless they are material to our discussion.

This tree construction differs from one that might be possible with abundant memory. Note that Snlog rules are unordered, so rules that share tuple producer-consumer relationships may not necessarily execute in the expected sequential order due to asynchrony of tuple arrivals contending for limited tuple storage. For example, the rules of line 10 and 11 may first generate path tuples that cause eviction. This manifests two problems. First, path tuples generated by line 13 and needed by line 12 may be evicted; proper next hops are not formed even though the paths to them were known at some point. Priority on shortestCost solves this problem. Second, in line 12, we need to check path against shortestCost such that only a shortestCost strictly smaller than the current one results.

For the sake of simplicity, Listing 2 does not deal with failing links. This can easily be solved by minor rule modifications that periodically reconstruct the tree.

Besides serving as data collection sinks, trees are the routing primitives of several proposed schemes [8], [9], [12]. Construction of multiple trees for Listing 2 is very easy; a second tree only requires the addition of two other facts like line 16 and line 17. We are not aware of any sensornet tree implementations that support multiple instances, and certainly not so easily.

Multi-hop Collection Application: To perform multi-hop collection, we forward packets on top of tree routing at periodic intervals. This is very similar to the predominant use-case of TinyDB [29]. The program is shown in Listing 3.

Line 1 imports our previous tree routing such that we can use its nextHop interface. Lines 4-5 periodically prepare temperature for transmission.

The timer(Src,TimerId,TimerVal) built-in clears any pending timer with identifier TimerId and sets a single shot timer TimerVal milliseconds into the future with this identifier (lines 4). Line 5 makes these timers periodic.

Geographic Routing: Geographic routing is often important for sensor networks embedded in the real world. Greedy geographic routing sends packets toward the neighbor with the minimum distance to the destination [16]. Listing 4 presents this protocol.

In line 9 neighboring nodes exchange location information. When a message tuple is received, we compute the distances from all neighbors to the destination location and then we chose as the next hop the node with the smallest distance (line 12-14). The euclidean distance between two coordinates is computed by f_distance. In line 9, the forwarding is straightforward and is similar to Listing 1. An example initial application-level message is shown in line 20.

Departing from tree routing, geographic routing determines the next hop dynamically on a per-message basis. However, a user is still free to interchange these two routing protocols with only a minimal amount of work since they both ultimately export similar nextHop predicates.

One well-studied component of geographic routing is not included in our example above: "fallback routing" *i.e.,* when the current node responsible for a message is the local mini-

```
1 % send current round location to neighbors
2 neighbor(@*,S,MyX,MyY,Round) :- timer(@S,2,_),
      estimatedLoc(@S,MyX,MyY,Round), round(@S,Round).
3
4 % relaxation step for interior nodes
5 % by averaging neighbor's locations
6 estimatedX(@S,<AVG,Xs>,Round) :-
      neighbor(@S,S,Xs,Ys,Round),
      round(@S,Round),timer(@S,1,_), notPerimeter(@S).
7 estimatedY(@S,<AVG,Ys>,Round) :-
      neighbor(@S,S,Xs,Ys,Round),
      round(@S,Round),timer(@S,1,_), notPerimeter(@S).
8 estimatedLoc(@S,Xs,Yx,Round) :- estimatedX(@S,Xs,Round),
      estimatedY(@S,Ys,Round), round(Round),
      notPerimeter(@S).
9
10 % perimeter nodes just refresh their fixed location
11 estimatedLoc(@S,Xs,Ys,Round) :- fixedLoc(@S,Xs,Ys),
      round(Round),timer(@S,1,_), perimeter(@S).
12
13 % increase the round after we have computed the new
      estimate
14 round(@S,Round2) :- estimatedLoc(@S,Xs,Ys,Round1), Round2 =
      f_incr(Round1).
15 location(@S,Xs,Ys) :- estimatedLoc(@S,Xs,Ys,_).
16
17 % estimation window and broadcast timers
18 timer(@S,1,1000) :- round(@S,Round), Round < 1000.
19 timer(@S,2,200) :- round(@S,Round), Round < 1000.
```

Listing 5. Virtual Coordinates Localization

```
1 builtin(f_randInUpperHalf,FuncRandInUpperHalf).
2 priority(tauVal,10).
3
4 % tau expires:
5 % Double tau up to TauHi. Reset c, pick a new t.
6 tauVal(@X,T2) :- timer(@X,1,T1), T1 < 30000,
      T2=f_mult2(T1).
7 tauVal(@X,60000) :- timer(@X,1,T1), T1 >= 30000.
8 timer(@X,2,T3) :- tauVal(@X,T2), T3 =
      f_randInUpperHalf(T2).
9 timer(@X,1,T2) :- tauVal(@X,T2).
10 msgCnt(@X,0) :- tauVal(@X,T2).
11
12 % t expires: If c < k, transmit.
13 msg_ver(@*,Y,Oid,Ver) :- ver(@Y,Oid,Ver), timer(@Y,2,_),
      msgCnt(@Y,C), C < 1.
14
15 % receive same metadata: Increment c.
16 msgCnt(@X,C2) :- msg_ver(@X,Y,Oid,CurVer),
      ver(@X,Oid,CurVer), msgCnt(@X,C), C2 = f_incr(C).
17
18 % receive newer metadata:
19 % Set tau to TauLow. Reset c, pick a new t.
20 tauVal(@X,1000) :- msg_ver(@X,Y,Oid,NewVer),
      ver(@X,Oid,OldVer), NewVer > OldVer.
21
22 % receive newer data:
23 % Set tau to TauLow. Reset c, pick a new t.
24 tauVal(@X,1000) :- msg_store(@X,Y,Oid,NewVer,Obj),
      ver(@X,Oid,OldVer), NewVer > OldVer.
25
26 % receive older metadata: Send updates.
27 msg_store(@*,X,Oid,NewVer,Obj) :- msg_ver(@X,Y,Oid,OldVer),
      ver(@X,Oid,NewVer), NewVer > OldVer,
      store(@X,Oid,NewVer,Obj).
28
29 % update version upon successfully receiving store
30 store(@X,Oid,NewVer,Obj) :- msg_store(@X,_,Oid,NewVer,Obj).
      store(@X,Oid,OldVer,Obj), NewVer > OldVer.
31 ver(@X,Oid,NewVer,Obj) :- store(@X,Oid,NewVer,Obj).
```

Listing 6. Version Coherency

mum but not the end destination. Line 17 invokes the fallback mechanism. The simplest fallback schemes *i.e.,* planarization [16], are also easy to implement declaratively. However, more advanced schemes such as [20] require several dozens of rules, due to the algorithm's inherent complexities. We discuss details about how we might ease such tasks at the language level in Section VIII.

Note that priorities are used in this program as well. For similar reasons to tree routing discussed above, we give higher priority to the computedDistances predicate because we do not want another message to fill the computedDistances predicate space before the current message has identified its next hop. This is also a reason why we give shortestCost a higher priority than normal.

Localization: The previous example left unanswered how location information is initially established. One option is to provide location as a built-in *e.g.,* as an interface to GPS. Due to cost, most individual nodes are typically not equipped with direct location sensors. A second reasonable option is to manually specify locations with location facts; this is the common case in some deployments *e.g.,* geological surveys. The third option, *localization*, computes node coordinates. Among the many algorithms in this space, NoGeo is is noteworthy for its ability to do without bootstrap location information [34]. This service is shown in Listing 5.

The NoGeo algorithm has three levels of complexity. For brevity and ease of exposition, we only present the first which assumes a region's perimeter nodes know their locations. The algorithm proceeds as follows: At short beaconing intervals, neighbors periodically exchange their current locations (line 2). At long estimation intervals, interior nodes compute their own new location estimates by averaging together locations heard from all neighbors (lines 6-8). Perimeter nodes always send the same fixed location estimation and never update their own estimate (line 11). To save space in our discussion, we omit the simple one-line facts that specify the fixed locations of perimeter nodes and initial randomly-estimated locations of interior nodes. After several rounds, nodes' locations converge to points in network connectivity-based coordinate space. The result is a location predicate which can be exported for use by other services such as geographic routing in listing 4.

Version Coherency: Listing 1 showed a data dissemination example. In such situations, it is often desirable to provide coherency as well *e.g.,* all nodes run the same disseminated SQL query. Various sensor network protocols for eventual consistency provide such version coherency. Listing 6 illustrates a declarative implementation of a leading approach: version coherency with the Trickle dissemination policy [24].

The Trickle algorithm provides conservative exponential-wait gossip of metadata when there is nothing new (line 13), aggressive gossip when there is new metadata or new data present (lines 20 and 24), both counter-balanced with polite gossip when there are competing announcers (line 16).

The algorithm is inherently timer intensive. The *t* timer, corresponding to timer(_,1,_) in the listing, performs exponential-increase of each advertisement epoch. Timer τ, corresponding to timer(_,2,_), performs jittered sending in the latter half of each epoch in order to avoid send synchronization. Lines 30 and 31 store and update to the new version once the new data

```
1 builtin(trackingSignal,TrackingSensorModule).
2 import('tree.snl').
3
4 % on detection, send message towards cluster head
5 message(@Next,Src,Dst,MyX,MyY,Val) :-
      trackingSignal(@Src,Val), detectorNode(@Src),
      location(@Src,MyX,MyY), clusterHead(@Src,Dst),
      nextHop(@Src,Dst,Next,Cost).
6 message(@Next,Src,Dst,X,Y,Val) :-
      message(@Crt,Src,Dst,X,Y,Val),
      nextHop(@Crt,Dst,Next,Cost).
7
8 % Epoch-based position estimation
9 storedTrackData(@Dst,Epoch,X,Y,Val) :-
      message(@Dst,Src,Dst,X,Y,Val),crtEpoch(@Dst,Epoch).
10 crtXEstimation(@S,Epoch,<AVG,X>) :-
      crtEpoch(@S,Epoch),storedTrackData(@S,Epoch,X,Y,V).
11 crtYEstimation(@S,Epoch,<AVG,Y>) :-
      crtEpoch(@S,Epoch),storedTrackData(@S,Epoch,X,Y,V).
12
13 % epoch management
14 crtEpoch(@S,Epoch2) :- timer(@S,1,_),
      crtEpoch(@S,Epoch1),Epoch2 = f_incr(Epoch1).
15 timer(@S,Tid,Tval) :- timer(@S,Tid,Tval).
```

Listing 7. Tracking

```
1 typedef(#init,^uint16_t,^uint16_t,uint32_t).
2
3 % ---- Periodic beacons
4 % beacon sequence number
5 seq(@S,Seq2) :- timer(@S,1,_), seq(@S,Seq1), Seq2 =
      f_incr(Seq1).
6 % broadcast beacon
7 beacon(@*,S,Seq) :- seq(@S,Seq).
8 % periodic timers
9 timer(@*,1,5000). % beacon frequency
10 timer(@S,Tid,Tval) :- timer(@S,Tid,Tval).
11
12 % ---- Link estimation
13 % incorporate new beacon into link cost
14 link(@S,N,NewPrr) :- beacon(@S,N,NewSeq),
      lastSeq(@S,N,OldSeq)~, link(@S,N,PrrOld)~, NewPrr =
      f_ewma(NewSeq,OldSeq,OldPrr).
15 % record the last seq received
16 lastSeq(@S,N,Seq) :- link(@S,N,PrrNew), beacon(@S,N,Seq)~.
17
18 % ---- Initialization
19 init(@S,N,Seq) :- beacon(@S,N,Seq).
20 link(@S,N,255) :- init(@S,N,Seq).
```

Listing 8. EWMA Link Estimation

is received.

Despite the algorithm's apparent complexity, we were very pleasantly surprised by how easy it was to implement in Snlog. In fact, the comments in Listing 6 are *directly* from the original Trickle paper pseudocode [24]. Save for setting timers in lines 6-10, each line of pseudocode translates directly into one rule. This example in particular lends evidence to our claim that Snlog is at an appropriate level of abstraction for sensor network programming.

Tracking Application: Listing 7 shows a multi-hop entity tracking application implemented in Snlog. This is built on top of a routing mechanism (in this example we use tree routing). The specification is faithful to what has been presented in recently deployed tracking applications [30].

The algorithm works as follows: the trackingSignal built-in predicate is fired when a node detects the entity, at which point a message is sent to the cluster head node (lines 5 and 6). The cluster head node periodically averages the positions of the nodes that sent messages, and in this way computes an estimate of the tracked object's position. To correctly compute the destination for each epoch, the storedTrackData predicate labels received messages with the estimation epoch in which they were received (lines 9-11). Epoch management occurs in the usual way (lines 14-15).

Link Estimation: Good link estimators are important in wireless environments [42]. Several of the previous examples assumed a built-in link table for managing local neighbors and their costs. This is a reasonable choice when the link-estimator is radio hardware-assisted. Hardware-independent link estimators typically use average beacon reception rates to calculate packet reception rate (PRR) per neighbor as an indicator of link cost. Listing 8 shows that it is even possible to implement the commonly-used beaconing EWMA (exponentially weighted moving average) link estimator [42] in Snlog.

Each neighbor sends beacons every 5 seconds (lines 5-10). f_ewma computes the EWMA PRR based on the current beacon's sequence number, the last recorded beacon's sequence number, and the previously computed PRR (line 14). It is trivial to change this set of rules to update the prr using a time window instead of for each beacon.

This example demonstrates several uses of recursion in Snlog: incrementing the sequence number, recomputing the estimated link PRR, and periodic timer setting. However, it also illustrates a bootstrapping issue: link estimation recursion requires a base case link and lastSeq tuples, but it is not possible to specify these as facts since the identities of neighbors are not known a priori. To solve this problem, we initialize link and lastSeq the first time we hear a beacon with lines 19 and 20, making critical use of the no-replace primary key policy (indicated by a hash symbol) in line 1 in order not to overwrite link on every beacon. Section VIII discssues other possible language mechanisms to address this problem.

When combined with Listings 2 and 1, Listing 8 constitutes an end-user application written entirely in Snlog, save for the built-in sensor temperature and the built-in timer timer.

Other Applications and Snlog Extensions: We have also considered using Snlog to implement other sensor network protocols such as in-network aggregation [28], beacon vector coordinate and routing BVR [9], data-centric storage protocols such as pathDCS [8], geographic routing fallback schemes such as right hand-rules and convex hulls [16], [20]. Our conclusion is that these applications can be implemented in Snlog with no fundamental challenges.

V. System Architecture

In this section we present a high level view of our system design and implementation. The high level architecture for transforming Snlog code into binary code that runs on motes is shown in Figure 1. At the core of the framework lies the Snlog compiler that transforms the Snlog specification into the nesC language [11] native to TinyOS [2]. The generated components, along with preexisting compiler libraries, are further compiled by the nesC compiler into a dataflow engine

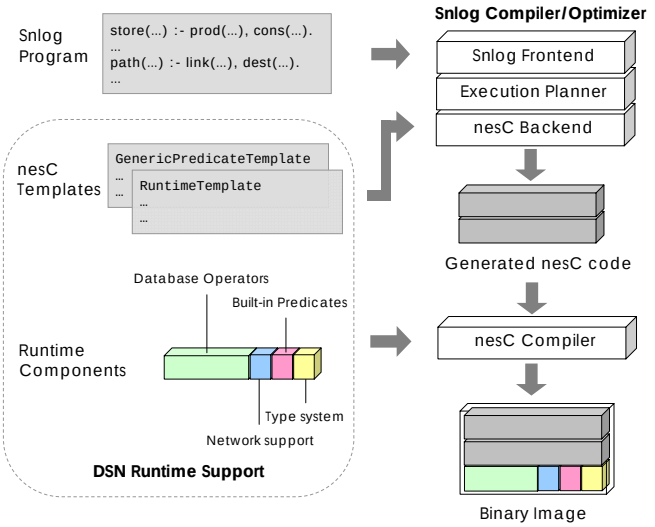


Fig. 1. DSN Architecture. Snlog is compiled into binary code and distributed to the network, at which point each node executes the query processor runtime.

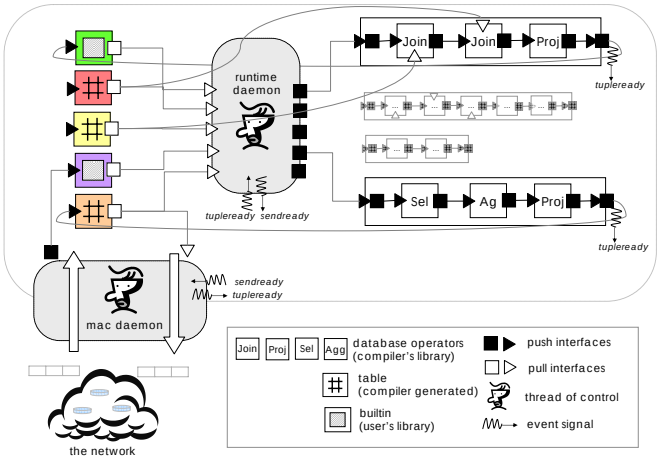


Fig. 2. DSN Implementation structure. Each rule is compiled into a dataflow chain of database operators.

implementing a minimal query processor. This resulting binary image is then programmed into all nodes in the network.

As an overview, each rule from the Snlog program gets transformed in the compiled code into a sequence of components that represent database operators like join, select, and project, which, to facilitate chaining, implement uniform push/pull interfaces. Figure 2 presents an overall view of the runtime component layout.

A. The Compiler

A fundamental choice of DSN is heavy use of PC-side program compilation as opposed to mote-side program interpretation. This relates directly to our goals of reducing runtime memory footprint and providing predictable operation.

The compiler parses the Snlog program and does a set of high level transformations and optimizations, including rewriting distributed rules as described in Section III. Next, it translates the program into an intermediary dataflow representation that uses chains of database operators (such as joins and selects) to describe the program. Then, for each chain, the compiler issues nesC code by instantiating components from a set of compiler library generic templates. Finally, the generated components, the system runtime and any necessary library runtime components are compiled together into a binary image using the nesC compiler.

B. The Runtime

We chose to implement the runtime system as a compiled dataflow of the user provided rules in the Snlog program. As is well known in the database community, declarative logic maps neatly to dataflow implementations. An example compiled runtime is shown in Figure 2.

The constrained resources and predictability concerns of sensor nodes make full fledged query processors for our purposes (*e.g.,* runtime rule interpreters) difficult to justify. While interpreters are used in several high-level sensor network languages for data acquisition [22], [29], we were wary of the performance implications of interpreting low-level services such as link estimators. In addition, we felt static compiler-assisted checks prior to deployment were worth the loss of flexibility. As a result of aggressive compilation, the resulting runtime system is equivalent to a dedicated query processor compiled for the initial set of rules, allowing new tuples (but not new rules) to be dynamically inserted into the network. Recall from our examples in Section IV that tuples are often enough to control system execution.

C. Code Installation

We rely on traditional embedded systems reprogrammers to distribute initial binary images onto each node prior to deployment. There may be cases when it is desirable to change the binary images on the nodes in the field, perhaps to modify the query processor to handle new classes of queries. Wireless code dissemination for reprogramming purposes has been relatively well studied in sensor networks [21]. In DSN, this could potentially be implemented by a built-in predicate responsible for microprocessor reprogramming though we leave this for future work.

VI. Implementation

A. Implementation Choices

In the following, we explain the most important implementation choices we made and how they affect the performance and semantics of the system.

Dynamic vs Static allocation

TinyOS does not have a default dynamic memory allocation library. On the other hand, database systems often make substantial use of dynamic memory allocation, and previous systems like TinyDB [29] have implemented dynamic memory allocation for their own use. In our implementation, we decided to use static allocation exclusively, for the following reasons. First, we believe that static allocation with a per-predicate granularity gives programmers good visibility and control over the common case when memory is fully consumed. By contrast, out-of-memory exceptions during dynamic allocation are less natural to expose at the logic level,

and would require significant exception-handling logic for even the simplest programs. Second, our previous experiences indicated that we save a nontrivial amount of code space in our binaries that would be required for the actual dynamic allocator code and its bookkeeping data structures. Finally, because tuple creation, deletion and modification of different sizes is common in DSN, the potential gains of dynamic allocation could be hard to achieve due to fragmentation. Instead, in our system all data is allocated at compile time. This is a fairly common way to make embedded systems more robust and predictable.

Memory Footprint Optimization

In general, in our implementation we chose to optimize for memory usage over computation since memory is a very limited resource in typical sensor network platforms, whereas processors are often idle.

Code vs. Data Trade off: Our dataflow construction is convenient because, at a minimum, it only requires a handful of generic database operators. This leads to an interesting choice on how to create instances of these operators. Many microprocessors common in sensor nodes present artificial boundaries between code storage (ROM) and data storage (RAM). Thus, operator instance parameters can either be stored as code or data. We permit both modes of parameter generation: *code-heavy generation* generates (efficient) code for every operator instance, whereas *data-heavy generation* generates different data parameters for use by a single generic operator.

For our current TelosB platform [32], in most cases it makes sense to use generic implementations and generate data parameters because of the hardware's relative abundance of ROM. However, for other popular platforms, the reverse is true. The choice ultimately becomes an optimization problem to minimize the total bytes generated subject to the particular hardware's constraints on ROM and RAM. Currently this decision is controlled by a compiler option that affects all dataflow operators in a program, but in principle this optimization could be made automatically based on hardware parameters.

Reduce Temporary Storage: To further improve memory footprint, we routinely favored recomputation over temporary storage. First, we do not use temporary tables in between database operators but rather feed individual tuples one at a time to each chain of operators. Second, all database operator components are implemented such that they use the minimal temporary storage necessary. For instance, even though hash joins are computationally much more efficient for evaluating unifications, our use of nested loop joins avoids any extra storage beyond what is already allocated per predicate. Our aggregation withholds use of traditional group tables by performing two table scans on inputs rather than one. Finally, when passing parameters between different components, we do not pass tuples but rather generalized tuples, *Gtuples*, containing pointers to the already materialized tuples. Gtuples themselves are caller-allocated and the number necessary is known at compile time. The use of Gtuples saves significant memory space and data copying, and is similar to approaches in traditional databases [15].

Rule Level Atomicity

In our environment, rules not requiring rewrite are guaranteed to execute atomically. We find that this permits efficient implementation as well as convenient semantics for the programmer. First, as compared to any forms of multi-rule atomicity, our implementation is simpler and more efficient since it does not have to take into account complex relationships between rules. In conjunction with rule level atomicity, priorities assist with execution control and are discretionary rather than mandatory. Second, by finishing completely the execution of a rule before starting a new rule we avoid many potential race conditions in the system due to the asynchronous nature of predicates (*e.g.,* tuples received on the network) and to the fact that we share code among components.

B. Implementation Description

Below we present more details on the DSN system implementation such as component interactions and the network interface. We call a "table" the implementation component that holds the tuples for a predicate.

Compiler

Frontend and Intermediary The frontend is formed by the following components: the lexical analyzer; the parser; the high level transformer and optimizer (HLTO); and the execution planner (EP). The parser translates the rules and facts into a list which is then processed by the HLTO, whose most important goal is rule rewriting. The EP translates each rule into a sequence of database operators. There are four classes of operators our system uses: Join, Select, Aggregate and Project. For each rule, the execution planner generates several dataflow join plans, one for each of the of the different body predicates that can trigger the rule.

Backend nesC Generator The nesC Generator translates the list of intermediary operators into a compilable nesC program. For each major component of our system we use template nesC source files. For example, we have templates for the main runtime task and each of the operators. The generator inserts compile-time parameters in the template files, and also generates linking and initialization code. Examples of generated data are: the number of columns and their types for each predicate, the specific initialization sequences for each component, and the exact attributes that are joined and projected. Similarly, the generator constructs the appropriate mapping calls between the generated components to create the desired rule.

Runtime Interactions

Our dataflow engine requires all operators to have either a push open/send/close or pull open/get/close interface. Typically, the runtime daemon pushes tuples along the main operator path until they end up in materialized tables, as in Figure 2. This provides rule-level atomicity. To handle asynchrony, the runtime daemon and network daemon act as pull to push converters (pumps) and materialized tables act as push to pull converters (buffers). This is similar to Click [17].

A challenging task in making the runtime framework behave correctly is to achieve the right execution behavior from the generic components depending on their place in the execution chain. For instance, a specific join operator inside a rule

receiving a Gtuple has to: pull data from the appropriate secondary table and join on the expected set of attributes. A project operator has to know on which columns to project depending on the rule it is in. Furthermore, function arguments and returns must be appropriately arranged. To manage the above problem under data-heavy generation, we keep all necessary data parameters in essentially a compact parse tree such that it is accessible by all components at runtime. The component in charge of holding these parameters is called *ParamStore*. The task of ensuring the different operational components get the appropriate parameters is done by our compiler's static linking. Under code-heavy generation, we duplicate calling code multiple times inlining parameters as constants.

Built-in Predicates

The use of the well-understood, narrow operator interfaces not only makes it very easy to chain together operators, but also means writing built-in predicates is straightforward. In general, users can write arbitrary rules containing built-in predicates and can also include initial facts for them. Some built-ins only make sense to appear in the body (sensors) or only in the head (actuators) of rules, while others may be overloaded to provide meaningful functionality on both the head and body (*e.g.,* timer). We permit this by allowing built-ins to only provide their meaningful subset of interfaces.

Interfacing the Network

As discussed in Section III, we chose to let the DSN user specify the entire system stack from the link layer up. Given the desire to be portable across a variety of radio and link layers, we have built on top of SP, a unifying link layer abstraction designed to decouple the network layer from the link layer [31], [37].

VII. Evaluation

In this section we evaluate a subset of the Snlog programs described in Section IV. We measure DSN's behavior and performance in comparison with native TinyOS nesC applications using the 28 node UC Berkeley testbed [1] (Fig. 3) and TOSSIM [23], the standard TinyOS simulator.

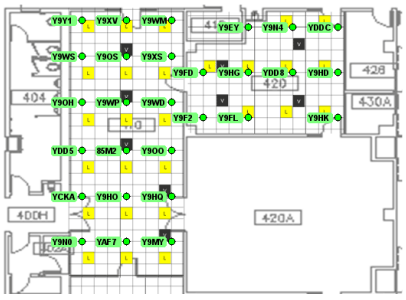


Fig. 3. 28 mote Omega Testbed at UC Berkeley

A. Applications and Metrics

We present evaluations of tree formation, collection, and the Trickle version coherency protocol relative to preexisting native implementations. Furthermore we describe our experience in deploying a DSN tracking application at a conference demo.

Three fundamental goals guide our evaluation. First, given DSN's substantially different architecture, we want to establish the correctness of the Snlog programs by demonstrating that they faithfully emulate the behavior of native implementations. Second, given the current resource-constrained nature of sensor network platforms, we must demonstrate the feasibility of running DSN on the motes. Finally, we perform a quantitative analysis of the level of effort required to program in Snlog, relative to other options.

To demonstrate the correctness of our system, we employ application-specific metrics. To evaluate tree-formation, we look at the distribution of node-to-root hop-counts. We then run collection over the tree-formed by this initial algorithm, measuring end-to-end reliability and total network traffic. For Trickle, we measure the data dissemination rate as well as the number of application-specific messages required. To demonstrate feasibility, we compare code and data sizes for Snlog applications with native implementations. Finally, we count lines of user written code to quantify the programming effort.

B. Summary of Results

The data shows that DSN successfully meets algorithmic correctness. DSN Tree forms a routing tree very similar to that formed by the TinyOS reference implementation in terms of hop-count distribution and our collection implementation achieves the same reliability as the native implementation. Finally, DSN Trickle provides near-perfect emulation of the behavior of the native Trickle implementation.

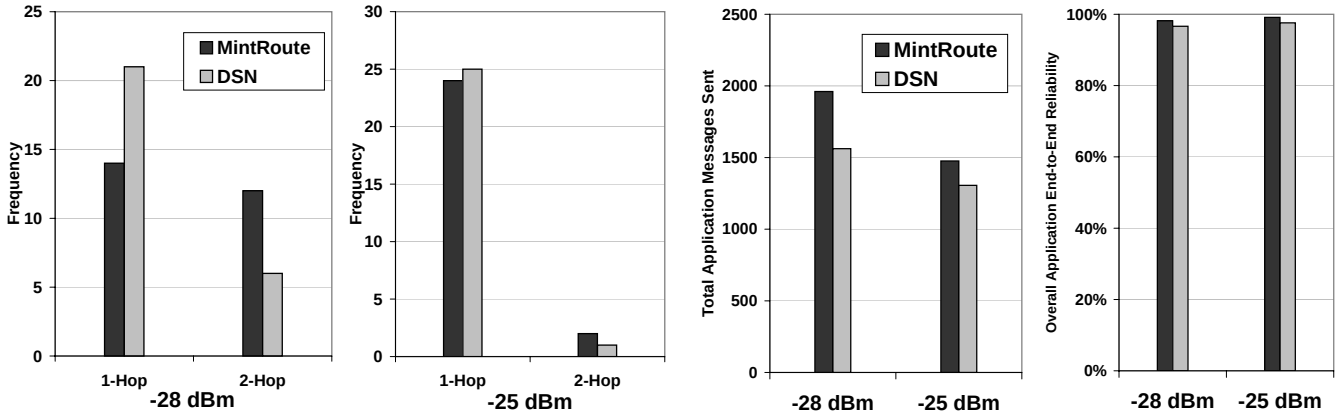
In terms of feasibility, DSN implementations are larger in code and data size than native implementations. For our profiled applications, our overall footprint (code + data) is always within a factor of three of native implementation and all our programs fit the current resource constraints. Additionally, significant compiler optimization opportunities still remain.

Concerning programming effort, the quantitative analysis is clear: the number of lines of nesC required for the native implementations are typically orders of magnitude greater than the number of rules necessary to specify the application in Snlog. For example tree construction requires only 6 rules in Snlog, as opposed to over 500 lines of nesC for the native implementation.

C. Tree/Collection Correctness Tests

For tree formation, we compared our algorithm presented in Section IV to MintRoute [43], the de facto standard for tree formation/routing in TinyOS. To vary node neighborhood density, we used two radio power levels: power level 3 (-28dBm), which is the lowest specified power level for our platform's radio, and power level 4 (-25dBm). Results higher than power level 4 were uninteresting as, given our testbed, the network was entirely single-hop.

Figure 4(a) shows a distribution of the frequency of nodes in each hop-count for each implementation. As a measure of



(a) Hop Count Distribution - The frequency distribution of number of hops to the root for nodes at two power levels

(b) Total Application Messages Transmitted and Overall Network End-to-End Reliability for Collection

Fig. 4. Results from experiments involving tree-formation and collection on the university testbed

routing behavior, we record the distance from the root, in terms of hops, for each node. Node 11, the farthest node in the bottom left corner in Figure 3 was assigned the root of the tree.

We see that both DSN and MintRoute present similar distributions, although under DSN, nodes are on average closer to the root, from a hop-count perspective. This discrepancy is due to the fact the two implementations use different link estimators. MintRoute implements its own link estimator, while DSN uses the one provided by the SP layer. If desired, a new link estimator could be specified in Snlog, as demonstrated in Section IV, though we did not feel this was warranted given the similarity of the behavior of the two implementations.

The collection algorithm for DSN, presented in Section IV, runs on top of the tree formation algorithm discussed above. For testing the pre-existing implementation, we used TinyOS's SurgeTelos application, which periodically sends a data message to the root using the tree formed by the underlying routing layer, MintRoute. Link layer retransmissions were enabled and the back-channel was again used to maintain real-time information.

Figure 4(b) shows the results of the experiments for two metrics: overall end-to-end reliability, and total message transmissions in the network. The network-wide end-to-end reliability of the network was calculated by averaging the packet reception rate from each node at the root. We see that DSN and SurgeTelos perform nearly identically, with an absolute difference of 1-2%. On the other hand, DSN transmits substantially fewer total messages in order to achieve this similar reliability. Picking poorer links would cause more link retransmissions, thereby increasing the total number of messages; the fact that DSN uses fewer messages suggests that its link estimation did not degrade its performance.

D. Trickle Correctness Tests

In order to demonstrate that the Snlog version of Trickle presented in Section IV is an accurate implementation of the the Trickle dissemination protocol, we compare the runtime behavior of our implementation against Drip, a widely used TinyOS implementation. To emulate networks with longer hopcounts and make a more precise comparison, we performed the tests in simulation rather than on the previous two hop testbed. Data is gathered from simulations over two grid topologies of 60 nodes: one is essentially linear, arranging the nodes in a 30x2 layout and the other is a more balanced rectangular 10x6 grid. The nodes are situated uniformly 20 feet apart and the dissemination starts from one corner of the network.

Figure 5 presents simulation results for the data dissemination rate using the two implementations. These results affirm that the behavior of the DSN and the native implementation of Trickle are practically identical.

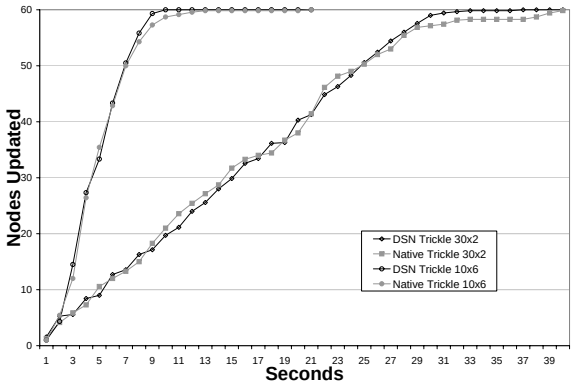


Fig. 5. Trickle dissemination rate in Tossim simulation.

In addition, we counted the total number of messages sent by the two algorithms and the number of message suppressions. Table I presents the total number of Trickle messages sent by both implementations and the total number of suppressed messages for the 30x2 topology. Again, these results demonstrate the close emulation of native Trickle by

our DSN implementation.

TABLE I
TRICKLE MESSAGES

	DSNTrickle	Drip
Total Messages Sent	299	332
Suppressed Messages	344	368

E. Tracking Demo

We recently demoed the tracking application specified in Snlog (and presented in Section IV) [5]. Our set-up consisted of nine TelosB nodes deployed in a 3x3 grid with radio set such that each node only heard from spatially adjacent neighbors. A corner-node base station was connected to a laptop, which was used for displaying real-time tracking results and up-to-date network statistics collect from the network. A tenth "intruder" node broadcasted beacon messages periodically and the stationary nodes then tracked the movement of this intruder and reported their observations to the base station. The demo was successful in the sense that it highlighted the specification, compilation, deployment, and real-time response of a tracking application similar to actually deployed tracking applications [30].

F. Lines of Code

Measuring the programmer level of effort is a difficult task, both because quantifying such effort is not well-defined and a host of factors influence this effort level. However, as a course measure of this programming difficulty, we present a side-by-side comparison of the number of lines of nesC native code against the number of lines of Snlog logic specifications necessary to achieve comparable functionality. This approach provides a quantifiable metric for comparing the level of effort necessary across different programming paradigms.

Table II provides a comparison in lines of code for multiple (functionally equivalent) implementations of tree routing, data collection, Trickle and tracking. The native version refers to the original implementation, which is currently part of the TinyOS tree [2]. NLA, or network layer architecture, is the implementation presented in [7], which decomposes network protocols into basic blocks as part of the overall sensornet architecture [6].

The reduction in lines of code when using Snlog is dramatic at roughly two orders of magnitude. TinyDB is also extremely compact, consisting of a single line query. However, as discussed in section II, TinyDB is limited to only data acquisition, rather than entire protocol and application specification. We conjecture that such a large quantitative distinction translates into a qualitatively measurable difference in programming effort level. To this we also add our subjective (and very biased) views that during the development process, we strongly preferred programming in Snlog, as opposed to nesC.

G. Feasibility

Code/Data size: The TelosB mote, the main platform on which DSN was tested, provides 48KB of ROM for code, and 10KB of RAM for data. Given these tight memory constraints, one of our initial concerns was whether we could build a declarative system that fits these capabilities.

TABLE II
CODE AND DATA SIZE COMPARISON

Program	Lines of Code			
	Mono	NLA	TinyDB	DSN
Tree Routing	580	106[a]	N/A	6 Rules (12 lines)
Collection	863	-	1	12 Rules (23 lines)
Trickle	560	-	N/A	13 Rules (25 lines)
Tracking	950[b]	-	N/A	13 Rules (34 [c] lines)

[a]NLA decomposes protocols into four basic blocks in such a way that the protocol-specific non-reusable code is generally limited to a single component. This value represents the lines of code for that specific component for tree routing.

[b]Note that this implementation may contain extra functionality beyond the DSN version, although we attempted to minimize this estimation error as best we could

[c]Includes 9 location facts.

Table III presents a comparison in code and data size for the three applications profiled in Table II. For a fair comparison, the presented memory footprints for the native applications do not include modules offering extra functionality which our implementation does not support.[1]

TABLE III
CODE AND DATA SIZE COMPARISON

Program	Total Code Size (kilobytes)			Data Size (kilobytes)		
	Mono	NLA	DSN	Mono	NLA	DSN
Tree Routing	18.2	24.8	26.2	0.8	2.8	3.1
Collection	25.8	-	29.2	1	-	4
Trickle	12.3	-	26.1	0.4	-	3.9
Tracking	26.7	-	27.9	0.9	-	7.8

The main reason for the larger DSN *code size* is the size of the database operators. As an important observation, note that this represents a *fixed* cost that has to be paid for all applications using our framework. This architectural fixed cost is around 22kB of code, which includes the code for the radio and link layer. As we can see in Table III, constructing bigger applications has only a small impact on code size.

On the other hand, the main reason for which the DSN *data size* is significantly larger than the other implementations is the amount of parameters needed for the database operators and the allocated tables. This is a variable cost that increases with the number of rules, though, for all applications we tested, it fit the hardware platform capabilities. Moreover, although not yet implemented, there is significant room for optimization and improvement in our compiler backend. Finally, if data size were to become a problem, the data memory can be transfered into code memory by generating more operator code and less operator parameters (see Section VI).

The overall *memory footprint* (measured as both code and data) of DSN implementations approaches that of the native implementations as the complexity of the program increases.

[1]Note however that the extracted modules still have a small impact on the code size due to external calls and links to/from them.

Such behavior is expected given DSN's relatively large fixed cost, contrasted with a small variable cost.

We also mention that our system is in general much more flexible than the original implementations. For instance, we are able to create multiple trees with the addition of a single Snlog initial fact, and no additional code (unlike the native implementation).

As a final note, technology trends are likely to produce two possible directions for hardware: sensor nodes with significantly more memory (for which memory overhead will be less relevant), and sensor nodes with comparably limited memory but ever-decreasing physical footprints and power consumption. For the latter case, we believe we have proved by our choice of Telos platform and TinyOS today that the overheads of declarative programming are likely to remain feasible as technology trends move forward.

Overhead: Two additional potential concerns in any system are network packet size overhead and runtime delay overhead. Our system adds only a single byte to packets sent over the network, serving as an internal predicate identifier for the tuple payload. Finally, from a runtime delay perspective, we have not experienced any delays or timer related issues when running declarative programs.

VIII. Discussion and Future Work

In this section we briefly discuss the impact of the threshold between Snlog rules and native ("built-in") program components, as well as present the current limitations of the system and our intended future work.

A. Architectural Flexibility

A user writing a Snlog declarative program has the choice of implementing any funcationality natively. The availability of this choice presents a big advantage in terms of flexibility and the space of implementable applications. At one extreme, the user could *declare aggressviely e.g.,* specifying link estimators as in Listing 8) and even link layer acknowledgements. Though this is easy to do, having access to lower level buffers, control interfaces, and timing information may be desirable and necessary as one proceeds downard towards hardware. At the other extreme, each native program can be seen as a built-in predicate which is fired by an initial fact.

There are two important takeaway points here. First, the user can choose the threshold based on the competing factors of ease and implementation and desired efficiency. Second, the space of applications that can be tackled by our system is equivalent to that implementable by any native program.

In DSN, the default threshold between native and declarative is at the device drivers for sensors and at the link layer and neighbor table for the network. Previous work has identified this as the "narrow waist" of shared funcationality for sensor networks [6], [31], [37].

B. Limitations

Perhaps the main new challenge raised in our work is the modeling and management of limited memory resources, and the way that this interacts with rule scheduling and asynchrony. One set of questions surrounds the mechanisms we have provided to deal with these issues. On the scheduling front, our current approach using priorities worked fairly nicely for the various examples we tried, but we are aware of its limited expressiveness, and are pursuing other options as well. A more expressive but heavyweight mechanism could be to enable transactional firing of subsets of rules, possibly with nested transactions for nested subsets of rules. In terms of memory allocation, our decision to use static memory allocation on a per-predicate granularity is also limiting. An alternative option we have considered is to let users define additional Snlog rules to specify eviction policies that can dynamically reallocate memory across predicates; such rules could naturally express content-based decisions (e.g. "value of information" rules, like allocating memory to sensor predicates in proportion to their variance in time).

Elegant mechanisms are one avenue for handling the challenges arising from limited memory. Another approach is to come up with a new declarative semantics that lets programmers formally and intuitively cope with these limitations. It seems difficult in general to preserve traditional Datalog semantics with arbitrary tuple evictions, even if programmers are given more expressive ways to control execution and memory. Instead of giving users control over execution, it would be "more declarative" to have them specify priorities over the *contents* of predicates at the end of execution. As one example, perhaps the entire language model could be recast around a notion of "ranked" results (a la web search), and the system would always be guaranteed to produce the "top-k" tuples for each predicate. Users would then be responsible for providing ranking expressions akin to SQL's "ORDER BY" clause. This example seems quite challenging to guarantee for complex programs, but we are attracted to the idea of developing a fully declarative approach to coping with limited memory.

IX. Conclusion

Data and communication are fundamental to sensor networks. Emboldened with these two guiding principles, we have presented a declarative solution to specify entire sensor network system stacks. By example, we showed several real Snlog programs that address disparate functional needs of sensor networks, often in orders of magnitude fewer lines of code, yet with program text still matching the designer's intuition on program behavior. This lends considerable support to our hypothesis that the declarative approach may be a good match to sensor network programming.

Our DSN system implementation shows that these declarative implementations are faithful to native code implementations and are feasible to support on current sensor network hardware platforms.

References

[1] Omega testbed website: omega.cs.berkeley.edu.

[2] Tinyos website: www.tinyos.net.

[3] S. Abiteboul, Z. Abrams, S. Haar, and T. Milo. Diagnosis of Asynchronous Discrete Event Systems - Datalog to the Rescue! In *ACM PODS*, 2005.

[4] M. Y. Becker and P. Sewell. Cassandra: Distributed Access Control Policies with Tunable Expressiveness. In *5th IEEE International Workshop on Policies for Distributed Systems and Networks*, 2004.

[5] D. Chu, A. Tavakoli, L. Popa, and J. M. Hellerstein. Entirely declarative sensor network systems. In *VLDB '06: Proceedings of the Thirty Second International Conference on Very Large Data Bases*, 2006.

[6] D. Culler, P. Dutta, C. T. Ee, R. Fonseca, J. Hui, P. Levis, and J. Zhao. Towards a Sensor Network Architecture: Lowering the Waistline. In *Proceedings of HotOS*, 2005.

[7] C. Ee, R. Fonseca, S. Kim, A. Tavakoli, D. Culler, S. Shenker, and I. Stoica. A Network Layer Architecture for Sensornets. In *Under Submission*, 2006.

[8] C. Ee, S. Ratnasamay, and S. Shenker. Practical data-centric storage. In *NSDI*, 2006.

[9] R. Fonseca, S. Ratnasamy, J. Zhao, C. T. Ee, D. Culler, S. Shenker, and I. Stoica. Beacon vector routing: Scalable point-to-point routing in wireless sensornets. In *NSDI '05*, 2005.

[10] C. Forgy. OPS5 user's manual. Technical Report CMU-CS-81-135, Carnegie Mellon University, 1981.

[11] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesc language: A holistic approach to networked embedded systems. In *In ACM SIGPLAN Conference on Programming Language Design and Implementation, 2003.*, 2003.

[12] O. Gnawali, B. Greenstein, K.-Y. Jang, A. Joki, J. Paek, M. Vieira, D. Estrin, R. Govindan, and E. Kohler. The tenet architecture for tiered sensor networks. In *Sensys*, 2006.

[13] 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.

[14] R. Gummadi, N. Kothari, R. Govindan, and T. Millstein. Kairos: a macro-programming system for wireless sensor networks. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 1–2, New York, NY, USA, 2005. ACM Press.

[15] J. M. Hellerstein and M. Stonebraker. Anatomy of a database system. *Readings in Database Systems, 4th Edition*.

[16] B. Karp and H. T. Kung. GPSR: greedy perimeter stateless routing for wireless networks. In *Mobile Computing and Networking*, pages 243–254, 2000.

[17] E. Kohler, R. Morris, J. J. Benjie Chen, and M. F. Kaashoek. The click modular router. In *Proceedings of the 17th Symposium on Operating Systems Principles*, 2000.

[18] J. Koshy and R. Pandey. Vm*: Synthesizing scalable runtime environments for sensor networks. In *In Proceedings of the third international Conference on Embedded Networked Sensor Systems (Sensys)*, 2005.

[19] B. Lampson. Getting computers to understand. *J. ACM*, 50(1), 2003.

[20] B. Leong, B. Liskov, and R. Morris. Geographic routing without planarization. In *NSDI*, 2006.

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

[22] P. Levis, D. Gay, and D. Culler. Active sensor networks. In *NSDI*, 2005.

[23] P. Levis, N. Lee, M. Welsh, , and D. Culler. Tossim: Accurate and scalable simulation of entire tinyos applications. In *In Proceedings of the First ACM Conference on Embedded Networked Sensor Systems (SenSys 2003).*, 2003.

[24] P. Levis, N. Patel, D. Culler, and S. Shenker. Trickle: A self-regulating algorithm for code propagation and maintenance in wireless sensor networks. In *In First Symposium on Network Systems Design and Implementation (NSDI)*, Mar 2004.

[25] B. T. Loo, T. Condie, M. Garofalakis, D. E. Gay, J. M. Hellerstein, P. Maniatis, R. Ramakrishnan, T. Roscoe, and I. Stoica. Declarative networking with distributed recursive query processing. In *ACM SIGMOD International Conference on Management of Data*, June 2006.

[26] B. T. Loo, T. Condie, J. M. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica. Implementing declarative overlays. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 75–90, New York, NY, USA, 2005. ACM Press.

[27] B. T. Loo, J. M. Hellerstein, I. Stoica, and R. Ramakrishnan. Declarative routing: Extensible routing with declarative queries. In *ACM SIGCOMM Conference on Data Communication*, August 2005.

[28] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. Tag: A tiny aggregation service for ad-hoc sensor networks. In *OSDI*, 2002.

[29] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. Tinydb: An acquisitional query processing system for sensor networks. *Transactions on Database Systems (TODS)*, March 2005.

[30] S. Oh, P. Chen, M. Manzo, and S. Sastry. Instrumenting wireless sensor networks for real-time surveillance. In *Proc. of the International Conference on Robotics and Automation*, May 2006.

[31] J. Polastre, J. Hui, P. Levis, J. Zhao, D. Culler, S. Shenker, and I. Stoica. A unifying link abstraction for wireless sensor networks. In *SenSys '05: Proceedings of the 3rd international conference on Embedded networked sensor systems*, pages 76–89, New York, NY, USA, 2005. ACM Press.

[32] J. Polastre, R. Szewczyk, and D. Culler. Telos: enabling ultra-low power wireless research. In *Proceedings of the 4th international symposium on Information Processing in Sensor Networks*, 2005.

[33] R. Ramakrishnan and J. D. Ullman. A survey of research on deductive database systems. *Journal of Logic Programming*, 23(2):125–149, 1993.

[34] A. Rao, C. Papadimitriou, S. Shenker, and I. Stoica. Geographic routing without location information. In *MobiCom '03: Proceedings of the 9th annual international conference on Mobile computing and networking*, pages 96–108, New York, NY, USA, 2003. ACM Press.

[35] A. Singh, P. Maniatis, T. Roscoe, and P. Druschel. Distributed Monitoring and Forensics in Overlay Networks. In *Eurosys*, 2006.

[36] R. Szewczyk, J. Polastre, A. Mainwaring, and D. Culler. Lessons from a sensor network expedition. In *1st European Workshop on Wireless Sensor Networks (EWSN)*, 2004.

[37] A. Tavakoli, J. Taneja, P. Dutta, D. Culler, S. Shenker, and I. Stoica. Evaluation and Enhancement of a Unifying Link Abstraction for Sensornets. In *UC Berkeley Technical Report*, 2006.

[38] M. Welsh and G. Mainland. Programming sensor networks using abstract regions. In *NSDI*, 2004.

[39] G. Werner-Allen, K. Lorincz, J. Johnson, J. Lees, and M. Welsh. Fidelity and yield in a volcano monitoring sensor network. In *In Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2006)*, 2006.

[40] J. Whaley and M. S. Lam. Cloning-Based Context-Sensitive Pointer Alias Analysis Using Binary Decision Diagrams. In *PLDI*, 2004.

[41] K. Whitehouse, J. Liu, and F. Zhao. Semantic streams: a framework for composable inference over sensor data. In *The Third European Workshop on Wireless Sensor Networks (EWSN), Springer-Verlag Lecture Notes in Computer Science*, February 2006.

[42] A. Woo and D. Culler. Evaluation of efficient link reliability estimators for low-power wireless networks, 2003.

[43] A. Woo, T. Tong, and D. Culler. Taming the underlying challenges of reliable multihop routing in sensor networks. In *SenSys '03: Proceedings of the 1st international conference on Embedded networked sensor systems*, pages 14–27, New York, NY, USA, 2003. ACM Press.