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,hellerstein,istoica}@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. Moreover, the declarative approach accommodates the desire for architectural flexibility and simple management of limited resources. Our results suggest that the declarative approach is well-suited to sensor networks, and that it can produce concise and flexible code by focusing on what the code is doing, and not on how it is doing it.

Categories and Subject Descriptors

H.2.4 [**Database management**]: Systems; D.1.6 [**Programming Techniques**]: Logic Programming

General Terms

Languages, Design, Performance

Keywords

Sensor Networks, Declarative Programming

SenSys'07, November 6–9, 2007, Sydney, Australia.

1 Introduction

Despite years of research, sensornet programming is still very hard. Most sensornet protocols and applications continue to be written in low-level embedded languages, and must explicitly contend 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 [22]. This picture has changed significantly in recent years: declarative approaches have proven useful in a number of new domains, including program analysis [45], trust management [5], and distributed system diagnosis and debugging [4, 40]. Of particular interest in the sensornet context is recent work on *declarative networking*, which presents declarative approaches for protocol specification [29] and overlay network implementation [28]. 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 sen-

sornet applications. In this work, we present some examples of these declarative specifications: a data collection application akin to TinyDB [31], a software-based link estimator, several multi-hop routing protocols including spanning-tree and geographic routing, the version coherency protocol Trickle [26], the localization scheme NoGeo [39] and an event tracking application faithful to a recently deployed tracking application [34]. 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 declarative approach 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. Our evaluation demonstrates both the feasibility and faithfulness of DSN for a variety of programs, showing that the resulting code can run on resource-constrained nodes, and that the code does indeed perform according to specification in both testbed and simulation environments.

1.1 Declarative Sensornets: A Natural Fit?

While declarative languages are famous for hiding details from the programmer, they are correspondingly infamous for preventing control over those details. Our experience confirms this, and suggests DSN is not well-suited to all sensornet tasks. Like most database-style languages, the language in DSN is not adept at natively manipulating opaque data objects such as timeseries, matrices and bitmasks; nor is it fit for providing real-time guarantees. In addition, as a variant of Datalog, the core of DSN's language is limited to expressing the class of programs that are computable in polynomial time. As a result of these shortcomings, we have taken the pragmatic approach and provided flexible mechanisms to interface to external code, as discussed in Section 2.

That said, we have been pleasantly surprised at the breadth of tasks that we have been able to program concisely within DSN. In Section 4.1 we describe a fully-functioning data collection implementation expressed entirely declaratively, save for natively implemented device drivers. Also, in Section 4.2 we discuss features of our language that allow for simple declarative management of resources, a vital concern for sensornets. A goal of our implementation is to allow programmers the flexibility to choose their own ratio of declarative to imperative code, while attempting in our own research to push the boundaries of the declarative language approach as far as is natural.

DSN's declarative approach does not fit exclusively into any one of the existing sensornet programming paradigm clusters. In its family of expressible network protocols, DSN can model spatial processing inherent to region-based group communication [43, 15, 33]. While DSN's execution runtime consists of a chain of database operations resembling the operator-based data processing common to dataflow models [14,32], DSN users write in a higher-level language. DSN also provides the runtime safeguards inherent to database systems and virtual machines [31, 24, 20]. Section 9 discusses this work's relationship to other high level sensornet languages in the literature in detail.

The rest of the paper is organized as follows. Sections 2 and 3 outline the declarative language, and provide examples of a variety of services and applications. Section 4 discusses additional features of DSN that suit sensor networks. Sections 5 and 6 present an architectural overview of the system, along with implementation concerns. Section 7 discusses evaluation methodology, measurements and results. Sections 8 and 9 outlines limitations of our system and related work.

2 A Quick Introduction to Snlog

In this section we give an overview of our declarative language Snlog. Snlog is a dialect of Datalog [38], a well known deductive database query language. The typical DSN user, whether an end-user, service implementor or system builder, writes only a short declarative specification using Snlog.

The main language constructs are *predicates*, *tuples*, *facts* and *rules*. Snlog programs consist of period-terminated statements. As a quick start, the following is a representative, but simplified, example of these elements:

```
% rule
temperatureLog(Time, TemperatureVal) :-
      thermometer(TemperatureVal), TemperatureVal > 15,
      timestamp(Time).

% facts
thermometer(24).
timestamp(day1).
```

The *predicates* above are temperatureLog, thermometer and timestamp; these are analogous to the tables of a database. *Tuples*, similar to rows in a table, are predicates with all parameters assigned. A *fact*, such as thermometer(24), instantiates a tuple at the beginning of execution.

A *rule* instantiates tuples based on the truth of a logical expression. Each rule consists of a *head* and *body* that appear on the left and right respectively of the rule's deduction symbol (":-"). The body defines a set of preconditions, which if true, instantiates tuple(s) in the head. For example, temperatureLog(TemperatureVal,Time) is the head of the rule above, while the thermometer and timestamp predicates form its body. This rule creates a temperatureLog tuple when there exist thermometer and timestamp tuples and the temperatureVal of the thermometer tuple is greater than 15.

The two facts establish the time and thermometer reading. The tuples given by these facts make the rule body true, so the rule creates a new tuple temperatureLog(24, day1). Following Datalog convention, predicates and constants start with lowercase while variables start with upper case letters: TemperatureVal is a variable, while day1 is a constant.

Unification further limits valid head tuples by detecting repeated variables among predicate parameters in the body. For example the following program:

```
evidence(TemperatureVal, PressureVal) :-
      temperatureLog(Time, TemperatureVal),
      pressureLog(Time, PressureVal).
```

```
pressureLog(day1,1017).
pressureLog(day2,930).
```

indicates that evidence is true if temperatureLog and pressureLog each have tuples whose first parameters, both named Time, match. Combined with the listed pressureLog facts and the temperatureLog(day1,24) tuple yielded from the previous example, the rule results in evidence(24,1017). In relational database terminology, unification is an equality join.

Distributed Execution: In a fashion similar to [28], one argument of each tuple, the *location specifier*, is additionally marked with an "at" symbol (@) representing the host of the tuple. A single rule may involve tuples hosted at different nodes. For example, the tuples (facts):

```
consume(@node1,base).
produce(@node1,data1).
```

are hosted at the node whose identifier is node1. Rules can also specify predicates with different location specifiers:

```
store(@Y,Object) :- produce(@X,Object), consume(@X,Y).
```

With the two tuples above, this instantiates store(@base,data1). The different nodes that appear in a rule such as base and node1 have to be within local communication range for the tuples in the head predicate to be instantiated. This is done by sending messages addressed to the tuple host node.

For broadcast, a special "*" (asterisk) location specifier is used:

```
store(@*,Object) :- produce(@X,Object).
process(@X,Object) :- store(@X,Object).
```

The first rule broadcasts the store tuple. In the second rule, any neighboring node x that receives this broadcast rewrites the location specifier of the tuple with its local id.

Interfacing to the Physical World: In order to link declarative programs to hardware such as sensors and actuators, users may specify *built-in predicates*. For example, the prior example's thermometer predicate may read values from the underlying temperature sensor:

```
builtin(temperature, 'TemperatureImplementor.c').
```

where ThermometerImplementor.c is a external module (written in a language like nesC [12]) implementing the thermometer predicate. This method of exposing sensors as tables is similar to TinyDB and is a generalization of a mechanism found in other deductive databases [28]. Actuation is exposed similarly:

```
builtin(sounder, 'SounderImplementor.c').
sounder(@Node,frequency) :- process(@Node,Object).
```

Here, sounder tuples result in sound as implemented by the SounderImplementor.c.

Querying for Data: Users pose *queries* to specify that certain predicates are of interest and should be output from the DSN runtime. For example:

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

When a new tuple of this type is generated, it is also transmitted to the user. We currently use the Universal Asynchronous Receiver-Transmitter (UART) interface for this purpose. If no query is specified in a program, all the predicates are considered of interest and delivered.

```
1 % Initial facts for a tree rooted at ''root''
2 dest(@AnyNode,root).
3 shortestCost(@AnyNode,root,infinity).
4
5 % 1-hop neighbors to root (base case)
6 path(@Source,Dest,Dest,Cost) :- dest(@Source,Dest)~,
      link(@Source,Dest,Cost).
7
8 % N-hop neighbors to root (recursive case)
9 path(@Source,Dest,Neighbor,Cost) :- dest(@Source,Dest)~,
      link(@Source,Neighbor,Cost1),
      nextHop(@Neighbor,Dest,NeighborsParent,Cost2),
      Cost=Cost1+Cost2, Source!=NeighborsParent.
10
11 % Consider only path with minimum cost
12 shortestCost(@Source,Dest,<MIN,Cost>) :-
      path(@Source,Dest,Neighbor,Cost),
      shortestCost(@Source,Dest,Cost2)~, Cost < Cost2.
13
14 % Select next hop parent in tree
15 nextHop(@Source,Dest,Parent,Cost) :-
      shortestCost(@Source,Dest,Cost),
      path(@Source,Dest,Parent,Cost)~.
16
17 % Use a natively implemented link table manager
18 builtin(link,'LinkTableImplementor.c').
```

Listing 1. Tree Routing

Additional Snlog constructs will be presented in the following sections as needed. A comprehensive DSN tutorial is also available for interested programmers [1].

3 A Tour of Declarative Sensornet Programs

In this section, we investigate Snlog's potential for expressing core sensor network protocols, services and applications. Through a series of sample programs, we tackle different sensor network problems, at multiple, traditionally distinct levels of the system stack. For the sake of exposition, we will tend to explain Snlog programs in rule-by-rule detail, though auxiliary statements like type definitions are elided from the listings. Complete program listings are availabe at [1]

3.1 Tree Routing: A Common Network Service

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 the node that advertises the shortest cost to the base as their next hop neighbor. This tree construction in Snlog is presented in Listing 1.

Each node starts with only information about link qualities of neighboring nodes given by the predicate link(@Host,Neighbor,Cost). For all nodes, the root of the tree is explicitly specified with a fact (line 2), and a bootstrap value for the shortest cost to the root is also set (line 3).

To establish network paths to the root, first nodes that are one hop neighbors from the root use local links to the destination as network paths to the root. This corresponds to the rule in line 6, which reads: "if a node Source wishes to reach a destination Dest and has a local link to this destination with cost Cost, then establish a network path path from Source to Dest with next hop of Dest and cost Cost."

The tilde ("∼"), such as in this rule's dest predicate, indicates that the arrival of new tuples from the associated body predicate do not trigger the reevaluation of the rule which is

```
1 import('tree.snl').
2 builtin(timer, 'TimerImplementor.c').
3 builtin(thermometer, 'ThermometerImplementor.c').
4
5 % Schedule periodic data collection
6 timer(@AnyNode, collectionTimer, collectionPeriod).
7 timer(@Src, collectionTimer, Period) :-
      timer(@Src, collectionTimer, Period).
8
9 % Sample temperature and initiate multihop send
10 transmit(@Src, Temperature) :- thermometer(@Src, Temperature),
      timer(@Src, collectionTimer, Period).
11
12 % Prepare message for multihop transmission
13 message(@Src, Src, Dst, Data) :- transmit(@Src, Data),
      nextHop(@Src, Dst, Next, Cost)~.
14
15 % Forward message to next hop parent
16 message(@Next, Src, Dst, Data) :- message(@Crt, Src, Dst, Data),
      nextHop(@Crt, Dst, Next, Cost)~, Crt != Dst.
17
18 % Receive when at destination
19 receive(@Crt, Src, Data) :- message(@Crt, Src, Dst, Data),
      Crt==Dst.
```

Listing 2. Multi-hop Collection

useful in the cases that reevaluation is unwanted or unnecessary.

Second, nodes that are more than one hop from the root establish paths by deduction: a node Source that has a neighbor Neighbor that already has a established a path to the root can construct a path that goes through this neighbor with a cost that is the sum of the link cost Cost1 to neighbor and the neighbor's cost to the root Cost2 (line 9).

Here, path tuples are possible paths to the root, whereas nextHop tuples are only the shortest paths to the root. The reduction of possible paths to the shortest path occurs in the two rules of line 12 and 15. We employ a MIN database aggregation construct over the set of possible paths to find the minimum cost.

After successful tree construction, each node has selected a parent with the least cost to get to the destination. This information is captured in the nextHop(@Source,Dest,Parent,Cost) predicate and represents the network-level forwarding table for all nodes.

So far we have glossed over how the local link table link is populated and maintained. For now, let us assume a link table manager provided by an external component (line 18). In Section 4, we discuss several reasonable alternatives to constructing this link table, including a link estimator constructed declaratively.

This program does not downgrade tree paths. We can additionally add this mechanism with three more rules, which we have omitted here for brevity.

Besides serving as data collection sinks, trees often serve as routing primitives [11, 10, 13]. Construction of multiple trees based on this program is very easy; a second tree only requires the addition of two facts for the new root such as dest(@AnyNode,root2) and shortestCost(@AnyNode,root2,infinity).

3.2 Multi-hop Collection: An Initial User Application

To perform periodic multi-hop collection, we forward packets on top of tree routing at epoch intervals. This is very similar to a popular use-case of TinyDB [31]. The program is shown in Listing 2.

We first import our previous tree routing such that we can use its nextHop forwarding table (line 1). The two built-ins used are timer for interacting with the hardware timer, and thermometer for reading the temperature (line 2 and 3).

A fact sets the initial timer (line 6). A timer predicate on the right side (body) is true when the timer fires, and a timer predicate on the left side (head) evaluating to true indicates the timer is being set. Therefore, having the same timer predicate in the body and head creates a reoccurring timer (line 7). This timer's main purpose is to periodically sample temperature and initiate a multi-hop send (line 10).

Conceptually, multi-hop routing on a tree involves recursively matching a transported message with the appropriate forwarding tables along the path to the destination. This recursion is expressed succinctly in an initialization rule (line 13), recursive case (line 16) and base case (line 19) above. The initialization rule prepares the application level send request into a generic message suitable for forwarding. The recursive case forwards the message (at either an originating or intermediate node) according to each recipient's nextHop entry for the final destination. Finally, upon receipt at the final destination, the message is passed upward to the application layer which expresses interest in it (line 19).

The message predicate takes the place of the standard network queue. As such, we are able to design any queue admission policy through our operations on predicates, such as unification and database-style aggregation. On the other hand, queue eviction policies are limited by the system provided table eviction mechanisms. We discuss provisions for table eviction in Section 4.

3.3 Trickle Dissemination: Translating From Pseudocode

Various sensor network protocols utilize a version coherency dissemination algorithm to achieve eventual consistency. Listing 3 illustrates a declarative implementation of a leading approach: version coherency with the Trickle dissemination algorithm. [26].

Despite the algorithm's complexity, we were very pleasantly surprised by how easy it was to implement in Snlog. In fact, the comments in Listing 3 are *directly* from the original Trickle paper pseudocode [26]. Save for setting timers in lines 3-7, 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.

The Trickle algorithm provides conservative exponential-wait gossip of metadata when there is nothing new (line 10), aggressive gossip when there is new metadata or new data present (lines 17 and 21), both counter-balanced with polite gossip when there are competing announcers (line 13). Underscores in a predicate's arguments, such as in timer of line 10, represent "don't care" unnamed variables.

The algorithm is inherently timer intensive. The T timer, corresponding to tTimer in the listing, performs exponential-increase of each advertisement epoch. Timer τ, corresponding to tauTimer, performs jittered sending in the latter half of each epoch in order to avoid send synchronization. Lines 27 and 28 store and update to the new version once the new data

```
1 % Tau expires:
2 % Double Tau up to tauHi. Reset C, pick a new T.
3 tauVal(@X,Tau*2) :- timer(@X,tauTimer,Tau), Tau*2 < tauHi.
4 tauVal(@X,tauHi) :- timer(@X,tauTimer,Tau), Tau*2 >= tauHi.
5 timer(@X,tTimer,T) :- tauVal(@X,TauVal), T =
      rand(TauVal/2,TauVal).
6 timer(@X,tauTimer,TauVal) :- tauVal(@X,TauVal).
7 msgCnt(@X,0) :- tauVal(@X,TauVal).
8
9 % T expires: If C < k, transmit.
10 msgVer(@*,Y,Oid,Ver) :- ver(@Y,Oid,Ver), timer(@Y,tTimer,_),
      msgCnt(@Y,C), C < k.
11
12 % Receive same metadata: Increment C.
13 msgCnt(@X,C++) :- msgVer(@X,Y,Oid,CurVer), ver(@X,Oid,CurVer),
       msgCnt(@X,C).
14
15 % Receive newer metadata:
16 % Set Tau to tauLow. Reset C, pick a new T.
17 tauVal(@X,tauLow) :- msgVer(@X,Y,Oid,NewVer),
      ver(@X,Oid,OldVer), NewVer > OldVer.
18
19 % Receive newer data:
20 % Set Tau to tauLow. Reset C, pick a new T.
21 tauVal(@X,tauLow) :- msgStore(@X,Y,Oid,NewVer,Obj),
      ver(@X,Oid,OldVer), NewVer > OldVer.
22
23 % Receive older metadata: Send updates.
24 msgStore(@*,X,Oid,NewVer,Obj) :- msgVer(@X,Y,Oid,OldVer),
      ver(@X,Oid,NewVer), NewVer > OldVer,
      store(@X,Oid,NewVer,Obj).
25
26 % Update version upon successfully receiving store
27 store(@X,Oid,NewVer,Obj) :- msgStore(@X,Y,Oid,NewVer,Obj).
      store(@X,Oid,OldVer,Obj), NewVer > OldVer.
28 ver(@X,Oid,NewVer,Obj) :- store(@X,Oid,NewVer,Obj).
```

Listing 3. Trickle Version Coherency

is received.

3.4 Tracking: A Second End-User Application

Listing 4 shows a multi-hop entity tracking application implemented in Snlog. The specification is faithful to what has been presented in recently deployed tracking applications [34].

The algorithm works as follows: a node that registers a detection via the trackingSignal sends a message to the cluster head indicating the position of the node (lines 5 and 6). The cluster head node periodically averages the positions of the nodes that sent messages to estimate the tracked object's position (line 10). To correctly compute the destination for each epoch, the trackingLog predicate labels received messages with the estimation epoch in which they were received (line 9). Periodic timers update the current epoch (lines 14-13).

This application uses a fixed cluster head. Four additional rules can be added to augment the program to specify a cluster head that follows the tracked target.

```
1 builtin(trackingSignal,'TargetDetectorModule.c').
2 import('tree.snl').

4 % On detection, send message towards cluster head
5 message(@Src,Src,Head,SrcX,SrcY,Val) :-
      trackingSignal(@Src,Val), detectorNode(@Src),
      location(@Src,SrcX,SrcY), clusterHead(@Src,Head).
6 message(@Next,Src,Dst,X,Y,Val) :-
      message(@Crt,Src,Dst,X,Y,Val),
      nextHop(@Crt,Dst,Next,Cost).
7
8 % At cluster head, do epoch-based position estimation
9 trackingLog(@Dst,Epoch,X,Y,Val) :-
      message(@Dst,Src,Dst,X,Y,Val), epoch(@Dst,Epoch).
10 estimation(@S,Epoch,<AVG,X>,<AVG,Y>) :-
      trackingLog(@S,Epoch,X,Y,Val), epoch(@S,Epoch).
11
12 % Periodically increment epoch
13 timer(@S,epochTimer,Period) :- timer(@S,epochTimer,Period).
14 epoch(@S,Epoch++) :- timer(@S,epochTimer,_), epoch(@S,Epoch).
```

Listing 4. Tracking

3.5 Additional Examples

All of the preceding examples discussed in this section compile and run in DSN. We implemented and validated basic geographic routing [18], NoGeo localization [39] and exponentially weighted moving average link estimation [47] which appear in the companion technical report [7]. Additionally, we have sketched implementations of other sensor network services such as: in-network data aggregation [30], beacon vector coordinate and routing protocol BVR [11], data-centric storage protocol pathDCS [10], and geographic routing fallback schemes such as right hand-rules and convex hulls [23, 18]. Our conclusion is that Snlog implementations of these applications pose no fundamental challenges, being expressible in code listings of no more than several dozen rules while all running over the same minimal DSN runtime discussed in Section 5.

4 Beyond Expressing Sensornet Services

In the previous section, we showed that the declarative approach is natural for defining a wide range of sensornet services. In this section, we discuss two additional advantages. First, the declarative approach naturally accommodates flexible system architectures, an important advantage in sensornets where clear architectural boundaries are not fixed. Second, DSN facilitates resource management policies using simple declarative statements.

4.1 Architectural Flexibility

Disparate application requirements and the multitude of issues that cut across traditional abstraction boundaries, such as in-network processing, complicate the specification of a single unifying sensornet architecture: one size may not fit all. DSN strives to accommodate this need for architectural flexibility.

First, it is both possible and reasonable to declaratively specify the entire sensornet application and all supporting services, save for hardware device drivers. The previous section showed specifications of both high-level applications such as tracking and intermediate services such as Trickle. In addition, we have specified cross-layer applications such as in-network aggregation [30] and low-level protocols such as link estimation. For link estimation, we implemented a commonly-used beaconing exponentially weighted moving average (EWMA) link estimator [47] in Snlog, detailed in the companion technical report [7]. The combination of the link estimator with tree routing and multi-hop collection presented in Section 3 constitutes an end-user application written entirely in Snlog, except for the hardware-coupled built-ins thermometer and timer.

At the same time, it is straightforward to adopt packaged functionality with built-in predicates. For instance, we initially implemented link as a built-in, since it allows us to expose radio hardware-assisted link-estimations in lieu of our declarative link estimator. As a third option, we also used

SP [36], a "narrow waist" link layer abstraction for sensor networks as a built-in. In the next subsection, we outline how a substantial system service, energy management, can be incorporated into declarative programs. Similarly, higher-level functionality implemented natively such as a network transport service can also be incorporated. In this way, DSN facilitates users who want to program declaratively while retaining access to native code.

Architectural flexibility in DSN is also attractive because predicates can provide natural abstractions for layers above and below, such as link(Node,Neighbor,Cost) for the neighbor table and nextHop(Node,Destination,Parent,Cost) for the forwarding table. These predicates' tuples are accessed just like any others, without special semantics assigned to their manipulation. We can also see similar intuitive interfaces in other instances: geographic routing also provides a nextHop forwarding table like tree routing; geographic routing and localization, which are naturally interrelated, use and provide the location(Node,X,Y) predicate respectively, which is simply a table of each node's location. Both geographic routing and localization are presented in the companion technical report [7]. In these cases, the declarative approach facilitates program composition with intuitive abstraction interfaces.

Yet, the right level of declarative specification remains an open question. While a single sensornet architecture has not emerged to date, one may yet crystallize. By enabling users to freely mix and match declarative programming with existing external libraries, DSN enables the future exploration of this subject.

4.2 Resource Management

As a consequence of the physical constraints of sensornet platforms, DSN offers flexibility in the management of three fundamental resources: memory, processor and energy.

Memory: Since current sensornet platforms are memory constrained, DSN makes several provisions for managing memory effectively. At the programming level, the user is able to specify: the maximum number of tuples for a predicate, tuple admission and eviction strategies, and insertion conflict resolution policies. The materialize statement sets these policies for a predicate:

```
materialize(predA, entryTimeout, maxEntries, evictPolicy).
```

The above statement sets a maximum number of entries for predA and a timeout value for each tuple after which it will be removed. The eviction policy specifies how tuples should be removed when the maximum number of allocated tuples has been exceeded. Policies include random, least recently used and deny and are implemented by our runtime. This construct, borrowed from one in [28], permits the user to effectively specify static memory usage more simply than traditional sensornet programming systems.

Also, because we use a high-level language, the compiler has significant opportunity for optimization. For example, we have implemented two different memory layout schemes for generated DSN binaries, trading off between code and data memory. Since sensor network platforms separate code from data, *i.e.,* ROM from RAM, the compiler can optimize binary generation depending on the particular type of hardware platform. Section 6 discusses this more in depth. The combination of programming and compilation options enables a deductive database in a reasonable memory footprint.

Processor: Determining execution preferences among competing, possibly asynchronous, events is important, especially in embedded systems. For example, it may be desirable to prioritize event detection over background routing maintenance. DSN uses a priority mechanism to let the user specify tuple processing preference. For example, high temperature detection is prioritized over the rest of the processing below:

```
priority(highTemperature,100).

% Background rules
reportHumidity(...) :- ... .
disseminateValue(...) :- ... .

% Rule fired by prioritized predicate
reportHighTemperature(...) :- highTemperature(...),... .
```

In the above example, if multiple new tuples in the system are ready to be processed, the highTemperature tuples will be considered for deductions first, before the other regular priority tuples.

Prioritized deduction offers a simple way for users to express processing preferences, while not worrying about the underlying mechanism. It also differs from traditional deduction where execution preferences are not directly exposed to users.

Additionally, priorities can address race conditions that may arise when using intermediate temporary predicates, since DSN does not provide multi-rule atomicity. These races can be avoided by assigning high priority to temporary predicates.

Energy: Effective energy management remains a challenging task. Several systems have attempted to tackle this problem, such as Currentcy [48] for notebook computers, and a somewhat similar sensornet approach [17]. These energy management frameworks provide: 1) a user policy that allocates and prioritizes energy across tasks and 2) a runtime energy monitor and task authorizer.

Since declarative languages have been previously used for policy, we wished to assess the suitability of adopting energy management into DSN. Below, we outline how Snlog programs can naturally incorporate energy management.

For user policy, it is straightforward to specify predicates concerning desired system lifetime, energy budgets for individual sensors, and resource arbitration of energy across system services. As one example, facts of the form em_PolicyMaxFreq(@host,actionId,frequency) set up maximum frequencies allowed by the energy manager for different actions.

For task authorization, checks to the energy accounting module occur as part of a rule's body evaluation. To do this, we make authorization requests by including a em_Authorize(@host,actionId) predicate in the body of rules that relate to actionId. This means that these rules must additionally satisfy the authorization check to successfully execute.

The two new predicates mentioned map fairly naturally to the native energy management interface envisioned by the authors in [17] and [48]. Listing 5 provides an example of an Snlog program with these energy management features.

```
1 % Energy Manager-specific predicates
2 builtin(em_Authorize, EMModule).
3 builtin(em_PolicyMaxFreq, EMModule).

5 % Permit light actions at most 10 times per minute
6 em_PolicyMaxFreq(@Src, lightAction,10).
7
8 % Permit temperature actions at most 20 times per minute
9 em_PolicyMaxFreq(@Src, temperatureAction,20).
10
11 % Log light readings
12 lightLog(@Src, Reading) :- photometer(@Src, Reading),
       em_Authorize(@Src, lightAction).
13
14 % Sample temperature readings and send them to the base
15 temperatureReport(@Next, Reading) :- thermometer(@Src, Reading),
       nextHop(@Src, Dst, Next, Cost),
       em_Authorize(@Src, temperatureAction).
```

Listing 5. Specifying Energy Policy

The energy-aware program specified in Listing 5 stores light readings locally, and forwards temperature samples to a base station. Different policies are associated with each of the two main actions, lightAction and temperatureAction. (lines 6 and 9). Authorization for lightAction is requested when logging light readings, while the request for temperatureAction is processed when sampling and sending the temperature readings (line 12 and 15 respectively). If the energy budget is depleted, the underlying EMModule will evaluate these requests in accordance to the specified user policy.

In addition to addressing energy, a vital resource constraint in sensornets, this exercise also demonstrates flexibility in incorporating a new system service into DSN's existing architecture. The DSN programming tutorial provides further assistance and examples for interfacing DSN with native system services [1].

5 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 [12] native to TinyOS [3]. The generated components, along with preexisting compiler libraries, are further compiled by the nesC compiler into a runtime implementing a minimal query processor. This resulting binary image is then programmed into the 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. The runtime daemon manages the dataflow processing of tuples from and to tables and built-ins while the network daemon manages tuples arriving from and destined to the network. Figure 2 presents an overall view of this runtime activity.

5.1 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

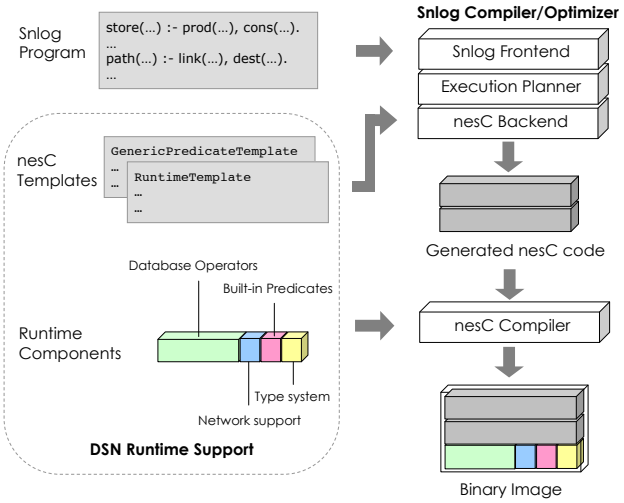


Figure 1. DSN Architecture. Snlog is compiled into binary code and distributed to the network, at which point each node executes the query processor runtime.

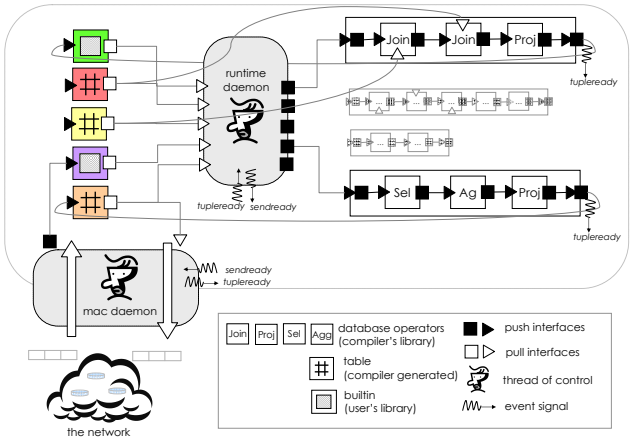


Figure 2. DSN Runtime. Each rule is compiled into a dataflow chain of database operators.

of initial rule-level level transformations on distributed rules (those whose location specifiers are not all the same). 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.

5.2 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 [24, 31], 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 any 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.

5.3 Code Installation

We rely on traditional embedded systems reprogrammers to distribute initial binary images onto each node prior to deployment. Users are free to install different rules and facts on different nodes, while retaining a common predicate set definition (database schema) across nodes. This permits basic support for different nodes requiring different functionality, as in heterogeneous sensor networks.

6 Implementation

In this section we discuss implementation design decisions and detail compiler and runtime interactions.

6.1 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. The resulting system exhibits sizeable dissimilarities from existing networked deductive databases [28].

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 [31] have implemented dynamic memory allocation for their own use. In our implementation, we decided to use static allocation exclusively. While dynamic allocation may better support the abstractions of limitless recursion and flexible table sizes, static allocation remained preferable for the following reasons. First, we believe that static allocation with a per-predicate granularity gives programmers good visibility and control over the 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 Tradeoff: 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. *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. This choice affects the sizes and ratios of code and data memory of the generated binary. Many microprocessors common in current sensor nodes present strict boundaries between code and data memory (*i.e.,* ROM vs. RAM). The choice is further influenced by the volatile/nonvolatile characteristics of the different memory modules (*e.g.,* typically only ROM is persistent, holding both code and data constants). We have implemented both modes of parameter generation. For our primary platform TelosB [37], it typically makes sense to employ data-heavy generation because of the hardware's relative abundance of RAM. However, for other popular platforms that DSN supports, the reverse is true. The choice ultimately becomes an optimization problem to minimize the total bytes generated subject to the particular hardware's memory constraints. Currently this decision is static and controls dataflow operators in a program, but in principle this optimization could be automated based on hardware parameters.

Reduce Temporary Storage: To further improve memory footprint, we routinely favored recomputation over temporary storage. First, unlike many databases, 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 [16].

Rule Level Atomicity

In our environment, local rules (those whose location specifiers are all the same) are guaranteed to execute atomically with respect to other rules. We find that this permits efficient implementation as well as convenient semantics for the programmer. In conjunction with rule level atomicity, priorities assist with execution control and are discretionary rather than mandatory. In addition, 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.

6.2 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 for distributed rules. 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 different body predicates that can trigger the rule.

Backend nesC Generator The nesC Generator translates the list of intermediary operators into a nesC program ready for compilation. 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-based open/send/close or pull-based open/get/-close interface. The runtime daemon pushes tuples along the main operator path until they end up in materialized tables before operating on new tuples, 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 [19].

A challenging task in making the runtime framework operate 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 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

Well-understood, narrow operator interfaces not only make it very easy to chain together operators, but also facilitate development of built-in predicates. 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.

7 Evaluation

In this section we evaluate a subset of the Snlog programs described in Section 3. We analyze DSN's behavior and performance in comparison with native TinyOS nesC applications using a 28 node testbed [2] shown in Figure 3 and TOSSIM [25], the standard TinyOS simulator.

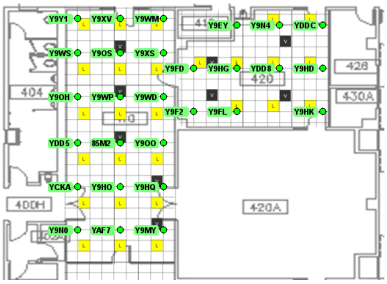


Figure 3. 28 mote Omega Testbed at UC Berkeley

7.1 Applications and Metrics

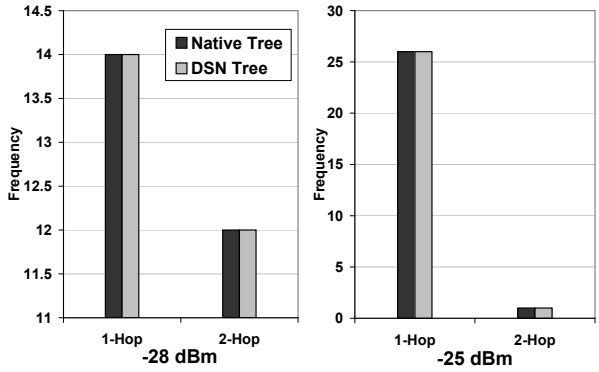
We present evaluations of tree formation, collection, and Trickle relative to preexisting native implementations. Furthermore we describe our experience in deploying a DSN tracking application at a conference demo.

Three fundamental goals guide our evaluation. First, 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.

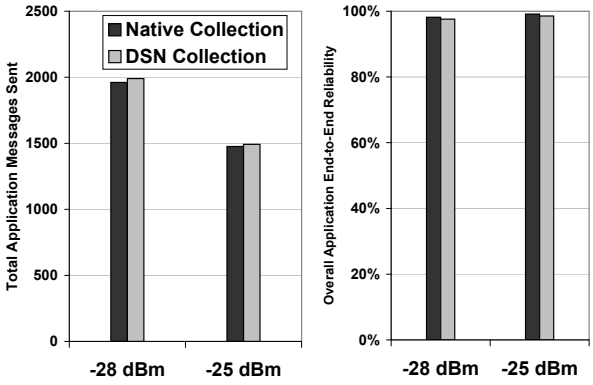
To demonstrate the correctness of our system, we employ application-specific metrics. To evaluate tree-formation, we look at the distribution of node-to-root hop-counts. We then run collection over the tree-formed by this initial algorithm, measuring end-to-end reliability and total network traffic. 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 use lines of code as a metric for evaluating ease-of-programming.

7.2 Summary of Results

The results indicate that DSN successfully meets algorithmic correctness requirements. DSN Tree forms routing trees very similar to those formed by the TinyOS reference implementation in terms of hop-count distribution and our col-



(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.

Figure 4. Results from experiments involving tree-formation and collection on the 28 node testbed

lection implementation achieves nearly identical 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. However, for our profiled applications, our overall memory footprint (code + data) is always within a factor of three of native implementation and all our programs fit within the current resource constraints. Additionally, several compiler optimizations which we expect will significantly reduce code and data size are still unimplemented.

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 7 rules in Snlog, as opposed to over 500 lines of nesC for the native implementation.

7.3 Tree/Collection Correctness Tests

For tree formation, we compared our DSN Tree presented in Section 3 to MultihopLQI, the de facto Native Tree implementation in TinyOS for the Telos platform. To compare fairly to Native Tree, we augmented DSN Tree to perform periodic tree refresh using the same beaconing frequency and link estimator. This added two additional rules to the program.

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. By the same token, at power level 2, nodes become partitioned and we experienced heavy variance in the trials, due to the unpredictability introduced by the weak signal strength at such a power level.

Figure 4(a) shows a distribution of the frequency of nodes in each hop-count for each implementation. As a measure of 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 Tree and Native Tree present identical distributions at both power levels.

The collection algorithm for DSN, presented in Section 3, runs on top of the tree formation algorithm discussed above. For testing the Native Collection, we used TinyOS's SurgeTelos application, which periodically sends a data message to the root using the tree formed by the underlying routing layer, MultihopLQI. 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 Collection and Native Collection perform nearly identically, with an absolute difference of less than 1%.

7.4 Trickle Correctness Tests

In order to demonstrate that the Snlog version of Trickle presented in Section 3 is an accurate implementation of the the Trickle dissemination protocol, we compare the runtime behavior of our implementation against a widely used native Trickle implementation, Drip [42]. 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. We used lossy links with empirical loss distributions.

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.

In addition, we counted the total number of messages sent

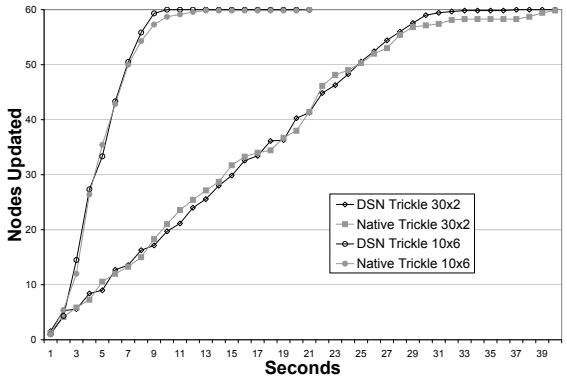


Figure 5. Trickle dissemination rate in Tossim simulation.

by the two algorithms and the number of message suppressions. Table 1 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 1. Trickle Messages

	DSN Trickle	Native Trickle
Total Messages Sent	299	332
Suppressed Messages	344	368

7.5 Tracking Demo

We demoed the tracking application specified in Snlog (and presented in Section 3) at a conference [6]. Our setup consisted of nine TelosB nodes deployed in a 3x3 grid with the communication range 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 successfully highlighted the specification, compilation, deployment, and real-time response of a tracking application similar to actually deployed tracking applications [34].

7.6 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 coarse 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 2 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 [3]. NLA, or network layer architecture, is the implementation presented in [9], which decomposes network protocols into basic blocks as part of the overall sensornet architecture [8].

Table 2. Lines of Code Comparison

Program	Lines of Code			
	Native	NLA	TinyDB	DSN
Tree Routing	580	106[a]	-	7 Rules (14 lines)
Collection	863	-	1	12 Rules (23 lines)
Trickle	560	-	-	13 Rules (25 lines)
Tracking	950[b]	-	-	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.

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 Section 9 discusses, 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.

7.7 Feasibility

In this section we evaluate the feasibility of our system to meet the hard memory constraints of the current sensor network platforms. We show that there is a significant fixed cost for our runtime system, but this is manageable even for the current platforms and comparable to existing proposals.

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.[1] Given these tight memory constraints, one of our initial concerns was whether we could build a declarative system that fits these capabilities.

Table 3 presents a comparison in code and data size for the three applications profiled in Table 2. 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. Note however that the extracted modules still have a small impact on the code size due to external calls and links to/from them.

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 21kB of code and 1.4kB of data. As we can

[1]The Mica family of platforms are also supported but compiler optimizations favorable to the Micas are not yet completed.

Table 3. Code and Data Size Comparison

Program	Code Size (KB)			Data Size (KB)		
	Native	NLA	DSN	Native	NLA	DSN
Tree Routing	20.5	24.8	24.8	0.7	2.8	3.2
Collection	20.7	-	25.2	0.8	-	3.9
Trickle	12.3	-	24.4	0.4	-	4.1
Tracking	27.9	-	32.2	0.9	-	8.5

see in Table 3, 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 6).

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

We also mention that our system is typically more flexible than the original implementations. For instance, we are able to create multiple trees with the addition of two 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.

8 Limitations

We divide the current limitations of our approach into two categories. First, there are certain drawbacks that are inherent to a fully declarative programming approach, which we have only been able to ameliorate to a degree. Second, DSN has certain limitations. The restrictions that fall into the second category can typically be lifted by introducing additional mechanisms/features; we leave these for future work. Conversely, while the shortcomings of the declarative approach can potentially be mitigated, they still remain as fundamental costs of declaratively specifying systems.

As noted in Section 1.1, a declarative language hides execution details that can be of interest to the programmer. This has two implications. First, it is less natural to express some programming constructs where imperative execution order is required such as matrix multiplication. Second, the declarative approach is not appropriate for code with high efficiency requirements such as low level device driver programming. For instance, in our declarative program, the granularity of user control is the rule. Also, real time guarantees may be hard to build in the complex declarative system. Therefore, we expect the low level programming for device drivers to be done natively and incorporated through built-ins.

Going one step further, we observe that while the high level language offers more room for compiler optimizations, the overall efficiency of a system implemented declaratively will most likely not surpass a hand-tuned native one. Fundamentally, we are trading expressivity and programming ease for efficiency, and, as we have shown throughout this paper this may be the right tradeoff in a variety of scenarios.

Finally, a declarative sensor network system has to interface with the outside world, and the callouts to native code break the clean mathematical semantics of deductive logic languages; however in this case there is some potentially useful prior work on providing semantic guarantees in the face of such callouts [21].

A few of the limitations of DSN were briefly discussed in Section 1.1, namely the ability to do only polynomial-time computation and the lack of support for complex data objects. These are somewhat ameliorated by the ability of DSN to call out to native code via built-in predicates. While the computational complexity restraint will not likely affect DSN's practicality, the lack of complex data objects may. We are considering the implementation of an Abstract Data Type approach akin to that of Object-Relational databases, to enable more natural declarations over complex types and methods [35]. In addition, we recognize that many embedded programmers may be unfamiliar with Snlog and its predecessor Datalog. We actively chose to retain Snlog's close syntactical relationship to its family of deductive database query languages, though we are also looking at more familiar language notation to facilitate adoption.

Currently, users can only select among a fixed set of eviction policies. We are considering a language extension which would allow users to evict based on attribute value, a construction that we expect to fit most practical eviction policies.

Finally, in Section 4.2 we have presented several mechanisms to increase the user control over the execution, notably we use priorities to express preference for the tuple execution order. We note that these constructs take the expressive power of our language outside the boundaries of traditional deductive database semantics, and a formal modeling of these constructs remains an important piece of future work.

9 Related Work

Numerous deployment experiences have demonstrated that developing low-level software for sensor nodes is very

difficult [44, 41]. 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 [31] and that a powerful query runtime has significant flexibility and opportunity for automatic optimization. Abstract regions [43] and Kairos [15] 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 [14], Tenet [13], Regiment [33] and Flask [32] 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 [46] 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 these works typically focuses on data gathering and processing, 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 to data acquisition, the networking logic involved in communicating that data, and the management of key system resources, while retaining architectural flexibility.

In the Internet domain, the P2 project [29, 28, 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 [24], TinyDB [31], and VM* [20], 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.

10 Conclusion

Data and communication are fundamental to sensor networks. Motivated by 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. These programs' text were often orders of magnitude fewer lines of code, yet still matched the designer's intuition. In addition, DSN enables simple resource management and architectural flexibility by allowing the user to mix and match declarative and native code. This lends considerable support to our hypothesis that the declarative approach may be a good match to sensor network programming. The 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.

Acknowledgment

This work was supported by generous gifts from Intel Research, DoCoMo Capital, Foundation Capital, a Stanford Terman Fellowship, a National Science Foundation Graduate Research Fellowship, and National Science Foundation under grants #0615308, #0435454, #0454432 and #IIS-0205647.

11 References

[1] Dsn programming tutorial. http://db.cs.berkeley.edu/dsn.

[2] Omega testbed. http://omega.cs.berkeley.edu.

[3] Tinyos. http://www.tinyos.net.

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

[5] 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.

[6] 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.

[7] D. C. Chu, L. Popa, A. Tavakoli, J. M. Hellerstein, P. Levis, S. Shenker, and I. Stoica. The design and implementation of a declarative sensor network system. Number UCB/EECS-2006-132, Oct 2006.

[8] 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.

[9] 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.

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

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

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

[17] X. Jiang, J. Taneja, J. Ortiz, A. Tavakoli, P. Dutta, J. Jeong, D. Culler, P. Levis, and S. Shenker. An architecture for energy management in wireless sensor networks. In *International Workshop on Wireless Sensor Network Architecture (WSNA07)*, 2007.

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

[19] 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.

[20] 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.

[21] R. Krishnamurthy, R. Ramakrishnan, and O. Shmueli. A framework for testing safety and effective computability of extended datalog. In *SIGMOD '88: Proceedings of the 1988 ACM SIGMOD international conference on Management of data*, pages 154–163, 1988.

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

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

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

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

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

[31] 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.

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

[33] R. Newton and M. Welsh. Region streams: functional macroprogramming for sensor networks. In *DMSN '04: Proceeedings of the 1st international workshop on Data management for sensor networks*, pages 78–87, New York, NY, USA, 2004. ACM Press.

[34] 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.

[35] J. Ong, D. Fogg, and M. Stonebraker. Implementation of data abstraction in the relational database system ingres. *SIGMOD Rec.*, 14(1):1–14, 1983.

[36] 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.

[37] 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.

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

[39] 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.

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

[41] 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.

[42] G. Tolle and D. Culler. Design of an application-cooperative management system for wireless sensor networks. January 2005.

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

[44] 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.

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

[46] 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.

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

[48] H. Zeng, C. S. Ellis, A. R. Lebeck, and A. Vahdat. Currentcy: A unifying abstraction for expressing energy. In *Usenix Annual Technical Conference*, June 2003.