A Declarative Sensornet Architecture

Arsalan Tavakoli[†], David Chu[†], Joseph M. Hellerstein[†], Phillip Levis[‡], and Scott Shenker[†]

[†]UC Berkeley EECS Dept.
Berkeley, California 94720
{arsalan, davidchu, hellerstein}@cs.berkeley.edu
shenker@icsi.berkeley.edu

[‡]Stanford CS Dept.
Stanford, California 94305
pal@cs.stanford.edu

1 Introduction

Increased code reuse, independent development, and interoperability are three key missing characteristics of sensornet programming today that motivate the need for an overall sensornet architecture. Architecture traditionally implies a componentized modular framework in which narrow interfaces provide the only form of communication between layers encapsulating specific functions and services. The Internet architecture provides the classic example of the modular approach.

A modular approach is not without its limitations and issues. The rigid layering makes incorporating cross-layer services, an integral part of sensornets, difficult, often violating a host of architectural principles. Furthermore, the tight integration of the upper stack in sensornets makes separating the network, transport, and application layers along cleanly defined boundaries a difficult if not impossible task. Some even claim that a modular architecture is overly constraining and hinders growth and progress for researchers [13].

We propose a new point in the design space, DSN [2], a programming paradigm for declaratively specifying sensornet systems. DSN allows the user to specify an application using a high-level language, which is subsequently fed to a compiler which builds a runtime query processor that executes on each node. DSN provides code reuse, although at a higher level, but more importantly directly attacks the failure of previous programming paradigms to provide ease of programming. Furthermore, it provides an intuitive interface for interacting with lower portions of the system stack and other components written in systems languages like C.

We divide this paper into three major parts. First, we provide a brief overview of DSN, covering only the details relevant for an architectural discussion. The second part focuses on answering the fundamental questions that are posed if a declarative approach for building sensornet systems is taken, such as feasibility, efficiency, and extensibility. Finally, we examine the philosophical aspects in terms of comparing it to the modular approach and also evaluating how well it satisfies the requirements that motivate the need for an architecture.

DSN Implementation details and performance evaluation results can be found in [2].

2 DSN Overview

DSN is composed of three main components: a high-level declarative specification language, the DSN compiler, and a runtime query processor. In this section we describe each, and then provide a few DSN examples.

2.1 Snlog

Our high-level specification language, Snlog, is a dialect of Datalog [12], with the following primary language constructs: variables, constants, predicates, facts and rules. We begin with a simplified application example to help explain the use of these constructs:

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

Our example reads as follows: if a temperature reading is obtained and sampling has been enabled, then this data is timestamped and appears at the (one-hop away) base station. The second line enables sampling.

When a *predicate* is instantiated with *variable* and *constant* assignments, tuples are created for that predicate. A *fact* is an instance of a predicate, i.e. a tuple, that is instantiated at the beginning of execution. In our example, measurement, timestamp, temperature, and samplingOn are examples of predicates, while samplingOn(@Source, true) is a fact. By convention, variables such as Base are capitalized, and constants such as true are not. A rule instantiates a tuple if the logical premises are satisfied. A rule in Snlog has the following format:

```
headPredicate(@A, C) :- bodyPredicate1(@A, B),
        bodyPredicate2(@B, C).
```

The existence of a tuple satisfying bodyPredicate1 and another tuple satisfying bodyPredicate2 (in this case, a match on the parameter B) leads to the creation of a tuple of type headPredicate. In the previous example, our single rule states that if all the predicates in the right hand side body evaluate to true then a new measurement tuple will be instantiated. Finally, each predicate also has a location specifier, as denoted by the at ("@") symbol, which indicates which host stores the tuple. If the location specifiers of tuples in the same rule differ, the tuple rendevouz location is an optimization left to the compiler and

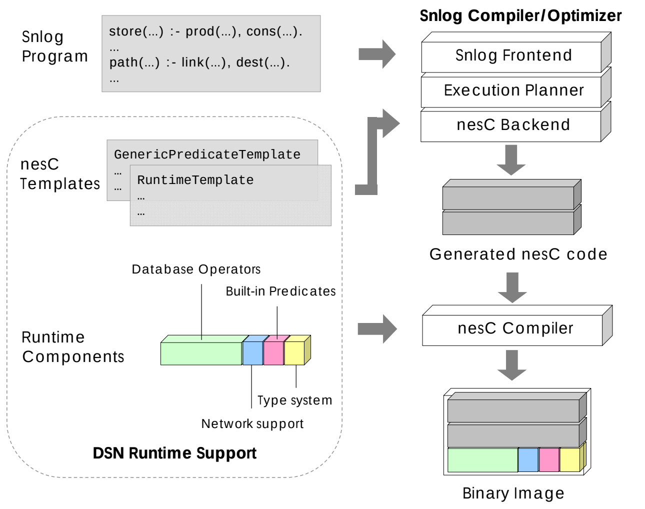

Figure 1. Overview of DSN process for converting Snlog specification into nesC binary image

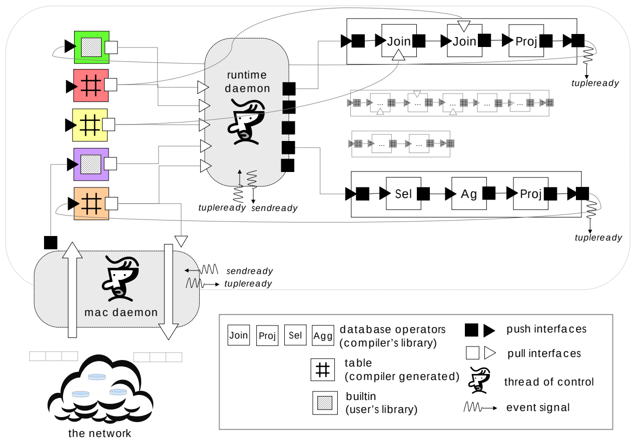

Figure 2. DSN query processor runtime.

runtime. Every rule is restricted to only reference link-local neighbors.

Two different forms of predicates exist: user-defined and built-in. The more common user-defined predicates are specified at design-time by the application developer. On the other hand, built-in predicates provide a predicate-like interface for natively-written components. Examples of currently used built-in predicates are timer exported by the hardware platform, link exported by the SP link layer [11], and other raw sensor readings like temperature.

2.2 System Architecture

Figure 1 provides an overview of the process of converting the high-level Snlog specification into a nesC binary image that is downloaded onto the motes.

One of our design decisions was to utilize a PC-intensive compilation approach, rather than a mote-oriented interpretation approach. The DSN compiler first parses the Snlog specification and performs a series of optimizations,

```
% —— Tree Construction
% base case
path(@Src,Dst,Dst,Cost) :- dest(@Src,Dst),
     link(@Src,Dst,Cost).

% deductive case
path(@Src,Dst,OneHopNei,Cost) :- dest(@Src,Dst),
     link(@Src,OneHopNei,Cost1),
     nextHop(@OneHopNei,Dst,TwoHopNei,Cost2),
     Cost=f_add(Cost1,Cost2), Src != TwoHopNei.

% use minimum cost path as next hop routing table entry
shortestCost(@Src,Dst,<MIN,Cost>) :-
     path(@Src,Dst,OneHopNei,Cost),
     shortestCost(@Src,Dst,Cost2), Cost < Cost2.
nextHop(@Src,Dst,OneHopNei,Cost) :-
     shortestCost(@Src,Dst,Cost),
     path(@Src,Dst,OneHopNei,Cost).

% —— Tree Root
dest(@*,0). %base id is 0
shortestCost(@*,0). %bootstrap cost to be there
```

Listing 1. Tree Routing

such as distributed rule rewriting [9], and then translates this into an dataflow that utilizes database operators to form execution chains (Figure 2). The compiler uses a set of generic compiler library templates to generate nesC code for the runtime query processor of each mote. Finally, during runtime, each node runs a query processor that continually incorporates new tuples that arrive wirelessly or are generated by built-ins (e.g. sensor readings) to compute a logical fixed-point.

Given the limited resources of sensornets, both from a runtime performance and code/memory footprint perspective, our runtime query processor is not full fledged. It only operates over the schema created by the program specification, meaning that it can understand new tuples inserted into the network, but not new rules or predicates. However, this has been enough to successfully implement a host of protocols and applications, as demonstrated by the examples presented next and in [2].

2.3 Application Examples

In order to more concretely demonstrate the ability of DSN to specify sensornet systems, we provide the specifications for two fundamental sensor network services: Tree Routing and Multi-hop Collection.[1].

The tree routing specification that runs in DSN is shown in Listing 1. The logic is as follows: each node discovers the paths available to reach the root, either by having a direct link to the destination (base case), or by going through a neighbor that has established a path to the base station (deductive case). Among all the paths, the specification dictates that the lowest cost path be chosen, and this forms the basis for the nextHop predicate, which is subsequently broadcasted to all neighbors. link(@Host, Neighbor, Cost) is a built-in predicate that contains a single tuple for each entry in the neighbor table. The root(s) of the tree is provided by the dest(@Host, BaseID) fact.

[1]We provide a complete listing of the specifications but elide initialization constructs. We refer the reader to [2] for more details.

```
import('tree.snl') % uses the tree routing specification

% ---- Periodic Temperature Transmissions
toTransmit(@Src,Reading) :- temperature(@Src, Reading),
      timer(@Src,1,Tval).
timer(@Src,1,Tval) :- timer(@Src,1,Tval). % ``1'' is a timer
      id

% ---- Message Forwarding
% package message for transmission
message(@Next,Src,Dst,Obj) :- toTransmit(@Src, Obj),
      nextHop(@Src,Dst,Next,Cost).

% forward to next hop
message(@Next,Src,Dst,Obj) :- message(@Crt,Src,Dst,Obj),
      nextHop(@Crt,Dst,Next,Cost).

% store when at destination
store(@Ds,Src,Obj) :- message(@Dst,Src,Dst,Obj).
```

Listing 2. Multi-Hop Collection

This specification illustrates the recursive power of Snlog: it succinctly expresses the deduction of global knowledge (nextHop routing table) from local state (link table).

Listing 2 provides the specification for a multi-hop collection protocol, running on top of the tree formed by the specification in Listing 1. The specification has two parts. First, it uses a periodic timer to collect temperature data, where both temperature and timer are built-in predicates. On the network front, each node either sends a packet it created, forwards a message received from a neighbor, or stores data if it is the intended destination. One interesting aspect to note about this program is that its only use of the underlying topology creation algorithm is the nextHop predicate. In other words, this specification would function correctly over any other routing protocol that exported the nextHop predicate.

3 Fundamental Questions

In order to establish DSN as a viable architectural approach for sensornets, a series of fundamental questions must first be posed and answered:

1. Does DSN provide the user with enough expressivity and flexibility?
2. Can DSN build efficient sensornet systems?
3. Will DSN be able to provide new features with minimal implementation pain?
4. Do DSN systems play nice with others?
5. Can DSN provide hardware portability?
6. How well does DSN integrate with the rest of the system?
7. What are the fundamental limitations of the DSN model?

This section discusses these issues based on our analysis and deployment experience to date, although we note that additional time and testing will allow for more definitive responses.

3.1 Programming expressivity

One of the most important factors in determining the success of any architectural framework is its ability to provide the required functionality by cleanly implementing a majority of applications in the domain of interest. DSN provides the ability to specify substantial portions of system stacks using a high-level declarative language, in essence decoupling the logic of the application from the actual implementation. We have demonstrated specifications for several actual applications and services, such as multihop routing, tracking, version coherent dissemination, geographic routing, exponentially weighted moving average link estimation, collection and event detection.

One of the major benefits of DSN, that directly addresses ease of programming, is the lines of code needed for implementing these applications and services. Often the differences were nearly two order of magnitudes over native implementations, as Tree Routing required 580 lines in the native implementation, but only 6 high-level rules using DSN. This reduction simplifies application development greatly, directly addressing one of the major pains of sensornet programming. It is important to note that this reduction is without loss of expressivity despite using high-level rules.

There are certain classes of programs that, at present time, can not be implemented using DSN. For example, users lack the ability to dictate the packet formats in DSN. This makes writing standards-compliant programs, such as an IP-compatible service, difficult. However, we believe such a need is not difficult to satisfy. As part of each declarative program, the user already specifies storage layout of tables as initialization statements. It is not hard to extend this idea to specifying layout of packets as well. This further narrows the distinction between stored tables and network messages in DSN.

A single system may have multiple applications, each requiring its own unique packet formats. DSN can be easily extended to provide customized packet formats for individual rules and predicates. Also, an easily added feature, drawing from the plethora of database literature on nested tuples, is the ability couple and layer certain tuples in packets. This allows DSN to replicate layered protocols, such as a TCP packet that stacks the link, IP, and TCP headers on top of each other.

In addition, extremely timing sensitive operations may be difficult to achieve. While DSN has demonstrated very faithful replication of some very timing intensive distributed algorithms such as Trickle, low level packet time-stamping that should occur immediately as the packet is received (e.g. within the radio device driver) may be better left to native radio driver implementations. Built-ins are good candidates for achieving this functionality.

This subsection raises two important design considerations for DSN. In a declarative system, there is often a tension between how much control to provide the user and simplicity. The current implementation is an initial starting point; the level of control the user has over the inner-workings of the system, such as how data is processed and when, can be altered as the system matures. Second, one of the advantages of modeling the system as a distributed database is that an extensive library of literature exists that provides invaluable guidance when designing such a framework.

3.2 System Feasibility

The resource-constrained, performance sensitive nature of sensornets necessitates efficient, correct implementations. Efficiency metrics are plentiful, and we feel the following are most pertinent here: code and memory footprint and runtime performance.

From a footprint perspective, DSN implementations are comparable to vertically integrated monolithic implementations, being slightly larger. The main source of overhead is the fixed cost of the runtime query processor. This cost is incurred regardless of the complexity of the program. However, as programs gain complexity, the code size stays stable and the memory footprint grows sublinearly, allowing complex DSN applications to fit within the limited ROM/RAM space of a mote. [2] substantiates this claim by providing ROM/RAM comparisons between DSN and native implementations for applications that span a wide range of complexity.

Except for low-level task scheduling and device drivers provided by TinyOS, and SP link services, no code from the monolithic implementations is reused. Rather, the rest of the system is a distributed deductive database. Consequently, it becomes important to demonstrate that the behavior of DSN applications is comparable, if not equivalent, to their native counterparts. [2] used collection, tree formation, and Trickle as the three benchmark applications, and DSN performed equivalently to the alternative in all categories.

3.3 System Extensibility

An important consideration for any architecture is the ease with which new features can be added, either through the addition of components, the definition of new interfaces, or the introduction of additional mechanisms. DSN was designed in an extensible manner to allow for incremental extensions without requiring an overhaul of the compiler. The majority of extensions involve the addition of new features to the high-level specification language, providing enhanced capabilities. These extensions require relatively minor changes in the compiler, but otherwise do not affect the system.

This is greatly aided by the well-defined interfaces and subcomponents inherited from databases. For example, new dataflow operators such as those shown in Figure 2 use well-understood open/getnext/close and open/sendnext/close interfaces. Similarly, the introduction of new rule-level optimizations, such as determining the rendevous location for a rule with multiple location specifiers, is itself localized to the high-level optimizer, separated cleanly from the front-end parsing, intermediate dataflow planning, and backend code generation.

3.4 Interoperability

The importance of interoperability of an architecture varies depending on the intended use; we divide interoperability into software and communication components.

Communication interoperability refers to the ability of various systems to communicate seamlessly with each other, often by specifying a standard communication exchange or protocol format. Different DSN applications are interoperable with each other in the sense that a node can receive a packet from another node, and potentially process it for link estimation purposes. However, without a shared schema, the data of other systems will be meaningless and potentially harmful if different applications can not be distinguished. In terms of communication interoperability as a whole, DSN does not take a position on a single protocol standard; rather, as discussed above it provides the mechanism for specifying such a format if developers chose to do so. In essence, protocol interoperability in DSN boils down to schema matching, a heavily studied database technique.

Software interoperability measures the degree to which interfaces are well defined so that applications and services can be developed independently. Currently, DSN utilizes a loosely coupled layering scheme in which, for example, the tree formation service exports a single interface that can then be used by different routing protocols. A strong effort was not made to provide strict layering and well-defined interfaces, in part because we feel that the cost of creating a rigid framework far outweighs the benefit of providing a greater degree of software interoperability. Certain services will export well-defined interfaces, but as we noted above, writing an entire application from scratch generally consists of less than 15 rules.

3.5 Portability

In a field as diverse as sensornets, a unifying overall sensornet architecture must be compatible with the range of hardware platforms and their software counterparts that exist. This means working across the different types of motes, some with starkly different characteristics and features. DSN is as portable as its system libraries. Currently, this means DSN runs on TinyOS and SP supported platforms. SP decouples the network and link layers, essentially providing a reasonable level of radio hardware independence. TinyOS provides hardware independence for a variety of microprocessors and sensors.

3.6 Integration with Lower Architecture

We chose to build DSN on top of a low-level OS kernel. We feel that at lower levels a modular, the imperative approach is more appropriate because of the fine-grained timing requirements, and also the precision needed to interact with hardware components. Consequently, we separate an overall architecture into two distinct segments: a programming paradigm and the underlying infrastructure. We define the underlying infrastructure to be the modularized framework that exports narrow interfaces to higher-layers, focusing on the design of communication protocols. The programming paradigm sits on top of this underlying infrastructure, and interacts directly with the end-user, focusing on application, and occasionally protocol, development. We do not take a position on where in the stack the dividing line falls , but rather point out that the interface between the two components is what allows for generality and interoperability. DSN falls into the programming paradigm category. Other examples of programming paradigms are [1, 6, 8, 15]. Any programming paradigm must consequently provide seamless integration with the underlying infrastructure.

DSN's built-in predicate mechanism provides a simple way of interacting with any native nesC component, whether it is the underlying infrastructure, or a higher-level component that has just been written in nesC, such as a unique transport protocol. A built-in predicate allows a developer to export any native component using a custom predicate, which can then be used in the same manner as user-defined predicates. Furthermore, by specifying a standard interface for a certain built-in predicate, differing implementations of the underlying component can be utilized without affecting the application.

3.7 Fundamental Limitations

Throughout this section we have discussed some of the benefits and current shortcomings of DSN. However, many of them stem from limitations of the current implementation; the important question is what are the fundamental limitations of DSN. Given the list of benefits described in this section, why is it that declarative high-level languages are not more widely used today?

A declarative system faces two main hurdles to adoption. First, the problem domain must be well-understood in order to transform high-level statements into efficient implementations. We feel that sensor networks are now reasonable candidates for this consideration. Second, declarative specifications inherently forego certain operational control in favor of simplicity. Applications requiring precise timing requirements, or intense low-level computations are difficult to implement declaratively. However, TinyDB was one of the first systems to demonstrate that the data-centric nature of sensornets provides a good fit for dataflow-oriented declarative programming models. DSN expands this vision, moving beyond simple data collection toward an entire declarative system.

4 Philosophical Questions

This section discusses the architectural contributions of DSN. We compare it to a traditional modular system architecture, as well as discuss the extent to which the motivating factors for an architecture are satisfied.

4.1 DSN vs. Modular Architecture

The Sensornet Architecture Project at UC Berkeley has focused on building a modular architecture for sensornets [3] to increase code reuse and permit independent development. The main two artifacts of the project to date have been SP [11, 14], and NLA [4]. SP, a unifying link abstraction, serves as the narrow waist of the modular architecture, effectively decoupling the network and link layers with shared common components such as message pools, link estimators and neighbor tables. NLA sits directly on top of SP, and provides a framework for building network protocols and decomposing them into four basic components: a routing topology, routing engine, forwarding engine, and output queue. A general interface is then exported to higher layers.

SP and NLA provide a general framework for link and network protocol designers. However, they also expose the shortcomings of this modular approach. Incorporating cross-layer services into a modular architecture is difficult, and interfaces for each service must be explicitly embedded into each layer. This can make future additions complicated. Furthermore, NLA demonstrates that the coupling of a wide range of application requirements and vertically integrated stacks prevent even a general but rigid framework from satisfying all needs. Above the network layer, the degree of integration only intensifies, making rigid interfaces a less satisfying design approach.

DSN provides a notably different way of building sensornet systems. There is no fundamental concept of layering; the system executes a runtime query processor that handles only database-like tuples. Underneath the covers, this query processor interfaces with the operating system scheduler and device drivers. Arbitrary functionality may be made available to the DSN user via the built-in predicate mechanism, much like system libraries in a traditional operating system. For example, we used SP's neighbor table as a built-in because it abstracted radio hardware. However, we also implemented neighbor tables natively in DSN when we wished to use a different underlying link layer. It is easy for a developer to selectively sample from preexisting nesC implementations while retaining the ease of use of the declarative specifications. At one extreme, DSN permits highly stylized library-based wiring [6]. We have chosen to primarily explore the other end of the spectrum, and investigate the degree to which it is both possible and sensible to "push declarative" through the system stack.

One of the main advantages of DSN over a completely modular architecture is the natural support for extensibility and evolution. In the previous section we discussed the ability of DSN to provide loosely-specified layering of specification rules, how new interfaces, or rather predicates, can be defined by each application, and also the ability to add new features by incrementally upgrading the compiler. We feel these capabilities fit quite well with the requirements of sensornet application designers, while also addressing the original issues that motivated the need for a unifying architecture. Finally, DSN allows a user to split up an application into several versions, and then allow for easy specification of the node-version mapping. This feature facilitates the tasking of heterogeneous and multi-tier networks using DSN.

The DSN approach has disadvantages as well, relative to a modular architecture. First, imperative operations are difficult to specify. For example, the execution order of rules may matter, especially those with side-effects such as from built-ins. While there is no straightforward mechanism in traditional logic to specify execution order, DSN provides an elementary priority control mechanism. Second, some algorithmic data structures may not have natural relational representations. For example, the convex hull computation of a geographic routing fallback mechanism presented in [7] uses a stack. While a table is strictly more general than a stack, we incur a mental translation overhead to convert these into the relational format. Lastly, as mentioned previously, achieving fine-grained timing operations is difficult, although DSN has successfully implemented applications with such requirements.

4.2 Code Reuse and Independent Development

The original motivating factors for a well-defined sensornet architecture were a lack of code reuse, limitations of independent development, and a need for interoperability [3].

We feel that the first two factors are simply proxies for ease of programming, which DSN addresses directly. In the end, the most important aspect of sensornet programming is that the end-user and programmer can interact with the system to achieve their goals. This implies not only an expressive interface, but also an intuitive and easy to use one as well. Code reuse and independent development also seek to simplify the task of the programmer. Declarative-level reuse can be established in DSN by simply declaring exported and imported predicates. Yet even without this feature, users are able to specify real applications, such as tracking, using a very modest number of rules and facts. In general, the declarative nature of the system allows for a decoupling of the logic of an application and the actual implementation. This separation is critical for non-technical users.

The issue of interoperability is a well-studied and active database research area. For example, we can think of communications with other network entities as simply schema matching on packets. This would permit mixed networks of traditional C implementations and DSN implementations to cooperate in building routing trees. Additionally, operating across tiered networks is straightforward. Emitted data is already in the form of tuples, easily consumable by many outside systems [10, 5]. The ability to easily assign different roles to different nodes allows the developer to differentiate between different nodes or tiers even within the same segment of code. For example, master-slave networks in which the powerful master nodes perform all computation, similar to that presented in [6] can be easily specified in DSN. For each rule in a specification, the user can add an additional predicate that limits the execution of that rule to either masters or slaves, and then use facts to assign roles to the nodes in the network.

5 Conclusion

DSN is a new way programming paradigm that has significant implications for sensornet architectures. The fundamental abstraction of a deductive database query processor is a natural fit for both sensor data and network design. We have shown several examples of programs that run in our system, and our experiences have indicated that these are often several orders of magnitude smaller than equivalent native implementations. This contributes greatly to the ultimate goals of every architecture: to simplify development while permitting innovation.

Future work includes specifying a wider variety of applications and protocols using DSN, and also further integrating it with other specific architectures, such as energy management, resource arbitration, and security.

6 References

[1] T. Abdelzaher, B.Blum, Q. Cao, Y. Chen, D. Evans, J. George, S. George, L. Gu, T. He, S. Krishnamurthy, L. Luo, S. Son, J. Stankovic, R. Stoleru, and A. Wood. Envirotrack: Towards and environmental computing paradigm for distributed sensor networks. *IEEE ICDCS*, 2004.

[2] D. Chu, L. Popa, A. Tavakoli, J. M. Hellerstein, P. Levis, S. Shenker, and I. Stoica. The design and implementation of declarative sensor network system. In *The 5th ACM Conference on Embedded Networked Sensor Systems*, 2007.

[3] D. Culler, P. Dutta, C. T. Ee, R. Fonseca, J. Hui, P. Levis, J. Polastre, S. Shenker, I. Stoica, G. Tolle, and J. Zhao. Towards a sensor network architecture: Lowering the waistline. *USENIX HotOS*, 2005.

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

[5] M. Franklin, S. Jeffrey, S. Krishnamurthy, F. Reiss, S. Rizvi, E. Wu, O. Cooper, A. Edakkuni, and W. Hong. Design considerations for high fan-in systems: The hifi approach. *CIDR*, 2005.

[6] 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.

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

[8] 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.

[9] 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.

[10] 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.

[11] 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.

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

[13] T. Roscoe. The end of internet architecture. *HotNets*, 2006.

[14] 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.

[15] K. Whitehouse, F. Zhao, and J. Liu. Semantic streams: a framework for composable semantic interpretation of sensor data. *EWSN*, 2006.