Programming Models for Sensor Networks: A Survey

RYO SUGIHARA and RAJESH K. GUPTA
University of California, San Diego

Sensor networks have a significant potential in diverse applications some of which are already beginning to be deployed in areas such as environmental monitoring. As the application logic becomes more complex, programming difficulties are becoming a barrier to adoption of these networks. The difficulty in programming sensor networks is not only due to their inherently distributed nature but also the need for mechanisms to address their harsh operating conditions such as unreliable communications, faulty nodes and extremely constrained resources. Researchers have proposed different programming models to overcome these difficulties with the ultimate goal of making programming easy while making full use of available resources. In this paper, we first explore the requirements for programming models for sensor networks. Then we present a taxonomy of the programming models, classified according to the level of abstractions they provide. We present an evaluation of various programming models for their responsiveness to the requirements. Our results point to promising efforts in the area and a discussion of the future directions of research in this area.

Categories and Subject Descriptors: D.1.3 [**Programming Techniques**]: Concurrent Programming—*Distributed programming*; D.3.m [**Programming Languages**]: Miscellaneous

General Terms: Design, Languages

Additional Key Words and Phrases: Programming models and languages, survey, taxonomy

1. INTRODUCTION

Sensor networks and wireless sensor networks are a rapidly emerging research field because of their enormous application potential. Advances in MEMS and microelectronic circuits have made it possible to build extremely small nodes with wireless communications, decent computational power, and various sensing capabilities.

A network of sensor nodes can be viewed as a distributed system. Compared to conventional distributed systems, however, there are important differences. The nodes and network are intrinsically less reliable and/or less available than in conventional distributed systems. Even under a normal operation, links may become degraded or unavailable depending upon the state of the wireless network or the

R. Sugihara was supported by IBM Tokyo Research Laboratory. The authors acknowledge support from a DMEA subcontract, UCSD/LANL Engineering Institute and from RUNES project under EU Framework Six programme.
Authors' address: Department of Computer Science and Engineering, University of California, San Diego, 9500 Gilman Drive, La Jolla, CA 92093-0404; email: {ryo, rgupta}@ucsd.edu.

dynamically changing radio environment in which a node is embedded. Resource constraints are an important part of the sensor networks. These networks frequently use batteries, making energy a scarce resource. Similarly, the availability of network bandwidth can also be an issue depending on how much data needs to be communicated and the operating environment for the network.

All these facts have forced the application programmers of sensor networks to deal with too many implementation-level details besides the application logic that they normally focus on. Out of necessity and to maximize efficiency, the application developers design software in a rather unstructured way where the application logic is woven deep into the underlying fabric of sensor network. However, this approach is not feasible for more complex applications because of the growing interdependencies of performance, functionality aspects and inherent non-scalability of such programming. Furthermore, equipping subsidiary (but often required) features such as failure-resilience and reprogrammability incurs additional complexity. Poor debugging environment (e.g., relying on LEDs on a sensor node) makes matters worse. As a result, programming sensor network applications are exceedingly difficult even for experienced programmers, let alone end users of such networks. This motivates us to explore sophisticated models and methods for designing and programming sensor network applications. For now, we just refer to these as "programming models."

In this paper, we survey prominent programming models for sensor networks. This survey builds upon earlier reviews [Römer 2004; Hadim and Mohamed 2006], taking a broader view of the sensor networks in an attempt to build a useful taxonomy and comparison of different approaches. We first identify the requirements for sensor network programming by determining the problem space and analyzing the actual applications. Then we categorize the programming models according to the level of abstraction and evaluate each approach in terms of the requirements. Finally, we discuss trends in applications, sensor networks and their implications for programming models for future.

2. SENSOR NETWORK PROGRAMMING

Before going into the details of each programming model, let us first examine the challenges associated with the sensor network as a computing and programming environment.

2.1 The Problem Space

As a computing environment, a sensor network is different in various aspects from traditional computing and it is the primary reason for diversity in the programming models. To help structure and navigate the myriad combinations of devices and their programming methods, as shown in Figure 1, we divide these along three characteristics that attempt to group devices and networks according to their capabilities presented to the end application:

(1) Node class: Grouping by power consumption which provides a first order indication of their capabilities.
(2) Node observables: Data+Address (DA), DA+Time (DAT), DAT+Space (DATS). These provide an approximation of the reasoning capabilities and validation re-

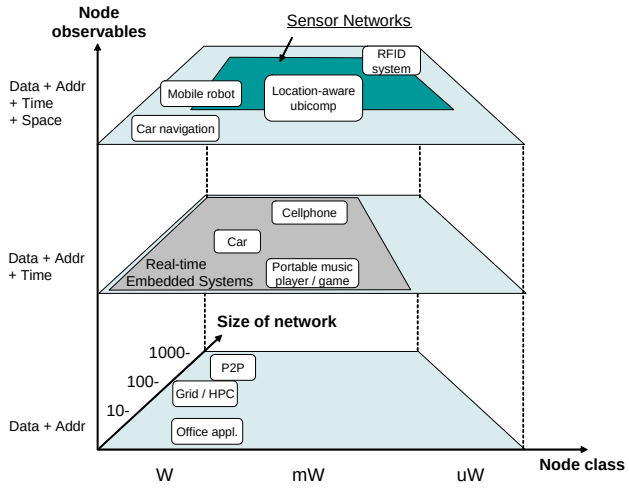

Fig. 1. Problem space of sensor networks

quirements that the end application can use.

(3) Size of network: An order of magnitude indication of the number of nodes deployed, from a few nodes to hundreds and thousands of nodes.

"Node class" identifies the hardware platforms that each computing environment assumes. PCs, workstations and servers are classified as "W-order", portable devices mostly driven by battery are "mW-order", and ultrasmall devices that usually harvest energy from external energy sources (e.g., sunlight, electromagnetic field) are classified as "μW-order." Along this axis, sensor network spans broadly, ranging from W-order nodes such as line-powered surveillance cameras to μW-order nodes like RFID. While W-order nodes have been subject of extensive research in desktop computing, and whereas μW-order nodes tend to be dominated by issues related to the design of sensors, mW-order nodes are the most interesting for the diversity of sensing/actuation, computing, and communications used. An example of a mW-order node is the "mote" from Crossbow Technology Inc.[1] and Moteiv Corporation[2]. Such nodes feature computing capabilities that are modest enough to support nontrivial programming and runtime environments, and ability to communicate to form ad-hoc networks.

"Node observables" are the concepts that a computing environment is aware of and uses in building application programs. General purpose programming builds upon manipulation of memory contents, thus Data and Address are the traditional node observables. In some distributed programming models, communication is exposed to improve performance. Increased interaction of sensor network programs with the physical world is likely to lead to additional observables that the programming model may be aware of. Chief among these is the concept of "time", which is already used in most embedded systems with specified accuracy and precision

[1] http://www.xbow.com/
[2] http://www.moteiv.com/

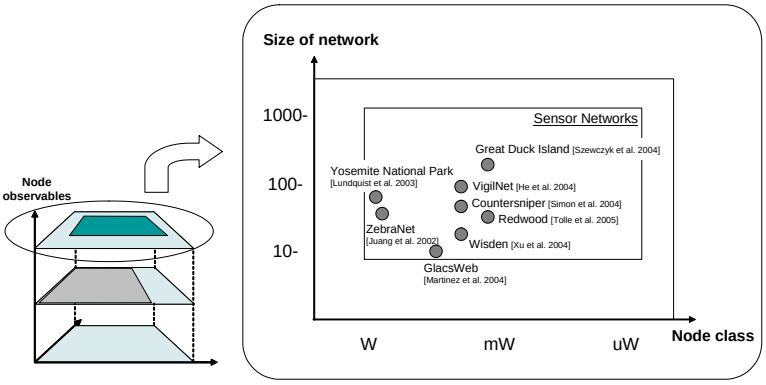

Fig. 2. Mapping of sensor network applications

to meet timeliness requirements in such programs. Going further, in the emerging sensor network applications, space is beginning to be recognized as an important observable for building new applications. In these applications, program behavior often depends on where the executing node is and computations at different places often have different meanings.

"Size of network" axis shows the number of computing units each computing environment typically assumes. There is a great diversity in the size of sensor network that a typical sensor network application assumes depending upon application needs, costs and other considerations. However, sensor network applications tend to cluster around similar magnitude of node scalability concerns that has a direct impact on how these are programmed.

Given the importance of physical constraints and an expanded set of node observables, sensor networks are different from traditional computing and their programming environments. At the same time, however, the problem space of sensor networks is not completely isolated and there are some overlapping regions with several computing environments such as mobile robotics and location-aware ubiquitous computing.

2.2 Examples of Application

Next we look into typical sensor network applications to see how the unique problem space identified above is populated, and to identify their requirements. Figure 2 is a closer view of the diagram shown in Figure 1, with the applications mapped on it.

Habitat monitoring is one of the earliest applications of sensor networks. In Great Duck Island [Szewczyk et al. 2004], researchers monitored the behavior of petrels, especially about how they use burrows both in short-term and long-term periods. They also monitored the environmental parameters inside and outside of burrows. ZebraNet project [Juang et al. 2002] monitored the behavior of zebras including long-range migration, inter-species interactions, and nocturnal behavior using tracking collars. This type of applications typically has very low duty cycle operation of the sensor network to maximize battery life.

Environmental monitoring is another frequent application of sensor networks.

Some examples include meteorological and hydrologic processes at high altitudes [Lundquist et al. 2003], long-term glacial movement [Martinez et al. 2004], and temperature and humidity in the forest [Batalin et al. 2004; Tolle et al. 2005].

Structural health monitoring [Xu et al. 2004] seeks to collect and analyze structural response to ambient or forced excitation by using accelerometer and strain gauges. Compared to other applications, it usually requires better network performance. Data rate is more than 1 kbps per each sensor, latency must be low in order to allow real-time analysis, and also time synchronization is necessary in the level of milliseconds.

There is a class of "tracking-type" applications. The basic idea is to localize a target by trilateration and other techniques using multiple sensors capable of measuring distance or bearing angle of the target. Tracking-type applications are unique because they intrinsically necessitate collaborative information processing among sensors. There has been a number of theoretical work to improve the performance and efficiency in tracking such as [Chu et al. 2002; Pattem et al. 2003]. VigilNet [He et al. 2004] is a surveillance application, which can be seen as an instance of the tracking-type applications. The objective is to acquire and verify information about enemy capabilities and positions of hostile targets, and magnetic sensors are used for sensing proximity to the target. The Countersniper System [Simon et al. 2004] is another example of this type. It locates shooters by estimating the source of muzzle blast. Each sensor node has acoustic sensor and measures time of arrival.

These emerging classes of applications of sensor networks provide an early indication of their maturity and allow us to reason about their common requirements.

2.3 Requirements for Sensor Network Programming

Based on the problem space and applications focus, we identify four important requirements for sensor network applications as follows:

—Energy-efficiency

—Scalability

—Failure-resilience

—Collaboration

where the first two requirements are mostly due to the problem space chosen and the latter two are due to the application needs.

Energy-efficiency is the common requirement for almost all wireless sensor network applications. For example, in habitat and environmental monitoring applications, each sensor node needs to be alive for months. Even for W-order nodes, energy-efficiency is important because of increasing scalability of such nodes. The power consumption of mW-order nodes often displays a wide dynamic range of power consumption levels depending upon node activities, sometimes by 3-4 orders of magnitude variation. Since wireless communication is a major energy consumer in mW-order nodes, it is important to reduce unnecessary data transmission as much as possible. Programming models should help programmers in a way that they can implement applications that achieve decent level of energy-efficiency without spending too much effort. At the same time, it is preferable that programming

models allow fine-grained control of energy-efficiency to satisfy the required conditions of each specific application.

Scalability is important because many of the sensor network applications run on tens or hundreds of sensor nodes. The scalability concerns are exacerbated by the bandwidth constraints, compared say to the Internet-scale distributed systems. For example, IEEE 802.15.4, a popular wireless PHY/MAC protocol for sensor nodes, provides a maximum bandwidth of 250 kbps. Therefore, the primary concern on the scalability of sensor networks is about reducing the amount of communication to efficiently use the scarce bandwidth. Programming models should help programmers to write scalable (i.e., bandwidth-efficient) programs and should be accompanied by runtime mechanisms that achieve bandwidth-efficiency whenever possible.

Failure-resilience is also important for applications to run over a long period. Applications should be able to remain functional even in the face of unreliable communications, dead nodes and other unexpected failures. Making applications resilient to failure and adaptive to the change of environments necessitates supports from programming models, since it is too complicated to write error handling logic for every failure and is still not enough to deal with unexpected failures.

Lastly, collaboration is an increasingly important characteristic of sensor network applications. Based on the analysis of applications, we can categorize them into *data collection* type and *collaborative information processing* type. In data collection applications, the main task is to send data to a central server. Many monitoring applications (habitat, environmental, structural) fall within this type. On the other hand, tracking-type applications are regarded as collaborative information processing applications that try to extract higher-level information by processing data from multiple sensors.

Collaboration can be a very challenging requirement for programming sensor network applications by needing attention to other requirements. The energy-efficiency requirement implies that collaboration needs to happen inside the network instead of at the central server to reduce data transmission. It promotes the use of *localized algorithms* [Estrin et al. 1999] where "sensors only interact with other sensors in a restricted vicinity, but nevertheless collectively achieve a desired global objective." However, in general it is difficult to devise such algorithms. Furthermore, the failure-resilience requirement forces the sensor network to keep working even after communication and node failures. Programming models need to provide an easy way of describing the application logic with least additional complexity that originates from such distributed and unreliable nature of the problem.

Note that data collection applications also become collaborative information processing by employing in-network processing. In fact, in-network processing is often necessary in these applications. To achieve energy-efficiency, they need to reduce the amount of data transmission by way of summarizing and compressing the data inside the network. Applications that collect high-bandwidth data (e.g., structural health monitoring) would also require in-network processing due to bandwidth limitations.

There are certainly other application requirements that we have not covered so far. Some of the examples are computational-efficiency, low-latency, non-intrusiveness to the monitored environments, autonomy, time synchronization, and sensor local-

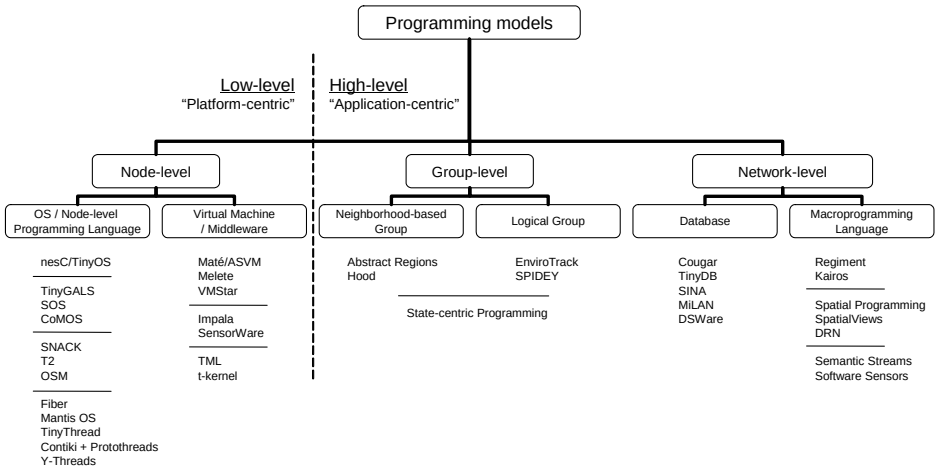

Fig. 3. A taxonomy of programming models for sensor networks

ization. On the programming aspect, ease of programming and ease of (re)configuration are important requirements. While we touch on these briefly, the focus of this paper is on the four requirements presented earlier.

3. TAXONOMY OF PROGRAMMING MODELS

We classify the approaches to programming sensor networks into low-level programming models and high-level programming models. Low-level programming models are focused on abstracting hardware and allowing flexible control of nodes. TinyOS [Hill et al. 2000] with nesC [Gay et al. 2003] is one of the earliest examples in this class and has been *de facto* standard software platform for sensor network programming. An interesting approach in this class is to run a virtual machine on each node. A virtual machine provides an execution environment for scripts that are much smaller than binary codes for TinyOS. Thus it is appropriate for the situations where the code on each node needs to be dynamically reprogrammed after deployment via a wireless channel.

High-level programming models take an application-centric view instead of the platform-centric view and address how easily application logics can be programmed, as opposed to, say, providing flexibility in optimizing the system's performance. More specifically, they mainly focus on facilitating collaboration among sensors, a major category of sensor network applications and also one of the most difficult challenges for sensor network programming. One of the typical approaches is to provide a set of operations for a group defined by several criteria. These operations include data sharing and aggregation so that programmers can describe collaborative data processing using them. Another approach is to provide communication abstraction by using a simple addressing scheme like variable access (e.g., "d1@n1" to access data d1 at remote node n1 in Kairos [Gummadi et al. 2005]).

High-level programming models are further divided into two types: "group-level abstraction" and "network-level abstraction." Group-level abstractions provide

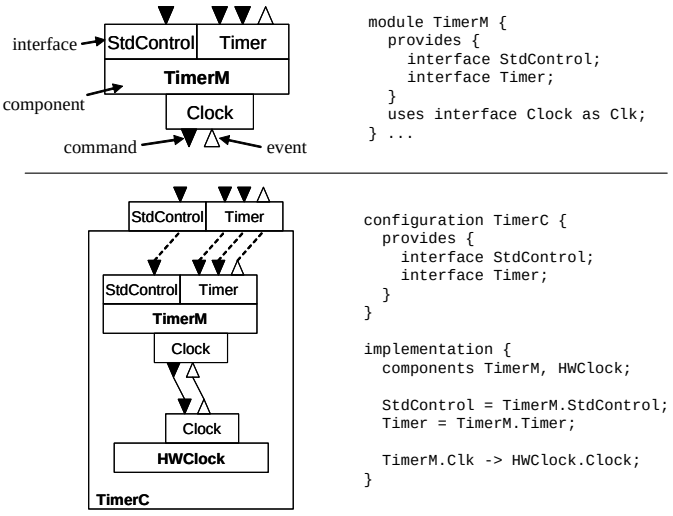

Fig. 4. Specification and graphical depiction of module (top) and configuration (bottom) in nesC (from [Gay et al. 2003], modified).

a set of programming primitives to handle a group of nodes as a single entity. These define APIs for intra-group communications and thus make it easier for the programmers to describe collaborative algorithms. Network-level abstractions, or equivalently "macroprogramming", share the approach with group-level abstractions, but go further by treating the whole network as a single abstract machine. The sensor database approach, which allows users to query sensor data by declarative SQL-like languages, falls within this category.

For classifying programming models into the types described above, we consider the highest level of abstraction they provide to their programmers or upper components. This approach of classification is motivated by the observation that high-level programming models are often proposed with lower-level components serving as a runtime environment that allows them to run on actual hardware platforms.

Figure 3 shows the entire taxonomy of programming models for sensor networks. "Node-level programming models" are discussed in Section 4. High-level programming models, specifically "group-level abstractions" and "network-level abstractions" are discussed in Section 5 and 6, respectively. In Section 7, we evaluate each programming model in terms of the requirements for sensor network programming, identified in the previous section.

4. NODE-LEVEL PROGRAMMING MODEL

4.1 Operating System / Node-level Programming Language

TinyOS [Hill et al. 2000] is one of the most widely used operating systems for wireless sensor networks. TinyOS is written in nesC [Gay et al. 2003], a programming language based on C, and is a component-based OS, allowing modular programming. Interactions between components are realized by "command" (to lower level) and "event" (to higher level). There are two types of components: "module"

and "configuration", where the latter is used to wire other components together, connecting interfaces used by components to interfaces provided by others. An application is also described as a configuration.

Figure 4 shows examples of nesC code. The top figure is the specification of the `TimerM` module, a part of the TinyOS timer service. The `TimerM` module provides `StdControl` and `Timer` interfaces and uses the `Clock` interface. The bottom figure shows an example of configuration. `TimerC` configuration is built by wiring the `TimerM` and `HWClock` modules, and provides two interfaces to upper modules.

nesC has many advantages in programming sensor nodes. Its flexibility allows to tune every parameter for special application needs such as energy-efficiency. Encapsulation by modules provides a unified interface that frees programmers from being conscious about whether some functionality is implemented by hardware or software. On the other hand, as the level of abstraction is very low, it is often difficult to implement even simple programs. In addition, rigorously event-based style and exclusion of blocking operations are often the sources of complexity.

There are roughly three ways to tackle the complexity. The first group of work uses asynchronous message passing to increase the independence between each functional component and thus simplifies the development process. TinyGALS [Cheong et al. 2003] applies GALS (globally asynchronous, locally synchronous) model to sensor network programming. It is a hybrid of the synchronous event-driven model and asynchronous message passing model. "Module" in TinyGALS is a language construct that contains multiple TinyOS components inside. A module works as a constituent of "system", which corresponds to "configuration" in TinyOS. Synchronous method calls are used inside modules so that the sequentiality of basic components is maintained. Modules are connected via an asynchronous FIFO queue. SOS [Han et al. 2005] is another operating system that uses message passing, but implements a priority queue instead of FIFO queue. CoMOS [Han et al. 2006] also uses message passing, but message handling is preemptive.

The second approach is more software engineering oriented and revises the way of abstraction either by adding more layers or extending the event model. SNACK [Greenstein et al. 2004] is a system designed for building a set of reusable service libraries, thus allowing programmers to make applications by combining services. Services in SNACK are analogous to "configurations" in nesC, but they are parameterizable and thus more reusable, whereas nesC configurations can only specify connections between the modules. T2 [Levis et al. 2005], a new version of TinyOS, follows the same philosophy as SNACK and provides a new abstraction boundary for application programming. Other improvements of T2 over TinyOS is "telescoping abstraction." It is a hybrid of horizontal decomposition (for the lower level to support different kinds of hardware devices) and vertical decomposition (for the higher level to support platform-independent functionality), and makes it easier to support new hardware platforms. Object State Model (OSM) [Kasten and Römer 2005] is an extension of event-driven model. The authors claim event-driven programming has difficulties due to static association of events to actions and implicit handling of program state, resulting in a clumsy programming style and inefficiency. In OSM, they extend the event paradigm with state and transitions, making actions a function of both the event and the program state. They also support parallel

composition to make use of concurrency, and hierarchical composition to allow refinement of a program.

Lastly, the third approach to deal with the complexity of event-driven model is to use thread abstraction. In the operating systems community, there is a longstanding argument on event-driven model and thread model[3]. One major constraint for thread abstraction in sensor network programming is due to limited hardware resources. Statically allocating per-thread stack is often too expensive in terms of memory space. In addition, event-driven model is more suitable for letting the microcontroller sleep as much as possible and thereby achieving energy-efficiency. On the other hand, by having blocking execution contexts, thread abstraction often simplifies the programs significantly. Fiber [Welsh and Mainland 2004] is an early work in the threading approach. Fiber is a lightweight concurrency model for TinyOS and allows single blocking execution context along with TinyOS's event-driven concurrency model. Mantis OS [Bhatti et al. 2005] is at the other end of spectrum: it provides preemptive, time-sliced multithreading on MICA2 motes. TinyThread [McCartney and Sridhar 2006], Protothreads [Dunkels et al. 2006] and Y-Threads [Nitta et al. 2006] are in the intermediate level between Fiber and Mantis OS. TinyThread is a multithread library for TinyOS. It implements "cooperative multithreading", in which an execution context is yielded explicitly by calling "yield()" or implicitly by waiting on a blocking I/O routine. TinyThread has a stack estimation tool for efficient per-thread stack allocation. Protothreads are a multithreading library for Contiki [Dunkels et al. 2004], which is an event-based operating system. Protothreads are similar to TinyThread in that they also adopt the cooperative multithreading. However, for the sake of efficiency, Protothreads take "stackless" approach and do not store execution contexts. Y-Threads are conceptually similar to Fiber, but realize preemptive multithreading. They capture applications as a combination of computation and control behaviors, and have a shared stack for the nonblocking computations and multiple separate stacks for the control behaviors that are blocking. The underlying observation is that the control behaviors require only small stack space, and they achieve better memory utilization compared to pure preemptive multithreading.

4.2 Virtual Machine / Middleware

Virtual machines are widely used in high-end servers as well as consumer PCs for various purposes such as platform independence and isolation. In sensor network programming, however, the focus is on "reprogrammability", i.e., the capability to inject new codes into each node on site dynamically.

First class of work is interpreter-based virtual machines. Maté [Levis and Culler 2002] and ASVM [Levis et al. 2005] are stack-oriented virtual machines implemented on top of TinyOS. Their idea is to provide an application specific virtual machine, which is designed for a particular application domain and provides the needed flexibility, so that it can support a safe and efficient programming environment. The motivation for being application-specific is based on the observation that sensor networks are usually deployed for a certain application purpose, and thus do not

[3]For example, see [Ousterhout 1996] and [von Behren et al. 2003]. [Bhatti et al. 2005] has an extensive discussion on this issue in the context of sensor networks.

need to be general. By providing a limited number of instructions necessary for a specific application, Maté/ASVM can reduce the size of the assembly code to be transmitted to each node, and thus reduce the amount of communication. Melete [Yu et al. 2006] extends Maté and supports multiple concurrent applications. VMStar [Koshy and Pandey 2005] is another framework for building application-specific virtual machines. VMStar allows dynamic update of the system software such as VM itself, as well as the application code.

Due to the rising necessity, some operating systems have started to feature the dynamic reprogramming capability. Deluge [Hui and Culler 2004], a dissemination protocol, with TOSBoot bootloader enables in-situ code update for TinyOS. Mantis OS [Bhatti et al. 2005], Contiki [Dunkels et al. 2004; Dunkels et al. 2006], and SOS [Han et al. 2005] also support dynamic update in finer resolution such as module and thread for more efficiency.

Reprogrammability feature is sometimes realized in middleware. Some examples are Impala [Liu and Martonosi 2003] and SensorWare [Boulis et al. 2003]. Impala is a middleware designed for the ZebraNet project [Juang et al. 2002] and its goal is to enable application modularity, adaptability to dynamic environments, and repairability. Its modular design allows easy and efficient on-the-fly reprogramming via wireless channel. SensorWare supports Tcl-based control scripts for the language used for reprogramming. Compared to Maté/ASVM, SensorWare is designed for richer hardware platform such as iPAQ. SensorWare also supports multiple concurrent users.

Finally, we note use of virtual machines in sensor networks with different objectives. Instead of reprogrammability, they provide platform-independent execution models on which developers can write their programs. Newton et al. propose an intermediate language called TML (Token Machine Language) [Newton et al. 2005], which assumes distributed token machine (DTM) as the execution model. In DTM model, each node sends and receives tokens to/from other nodes and the associated token handler is executed upon receiving a token. TML provides some basic operators that programmers can use for describing token handlers. A TML program is compiled into nesC code that is distributed to and run on each node. By employing DTM execution model, TML makes it easier to implement higher-level programming constructs that are semantically far from low-level programming languages such as nesC. t-kernel [Gu and Stankovic 2006][4] is an operating system kernel focused on improving reliability. It provides features such as OS protection and virtual memory, which are not directly supported by hardware, by code modification ("naturalization") at load time. We classify t-kernel as a virtual machine it is close to this type of virtual machines in terms of the objective and approach.

5. GROUP-LEVEL ABSTRACTION

The basic idea of group-level abstraction is to provide a language construct that handles multiple nodes collectively and a set of operations on it so that people can program the behavior of a group. We can classify this approach into the following two types depending on what type of groups are allowed:

[4]This is separate work from T-Kernel, a TRON-based realtime operating system kernel for embedded systems (http://www.t-engine.org/).

—Neighborhood-based group

—Logical group

Neighborhood-based groups are defined by physical closeness, and logical groups consist of nodes sharing some logical properties such as node type and sensor reading. By hiding the detail of communication, group-level abstractions facilitate the collaboration among nodes. For example, neighborhood-based groups are useful in implementing "localized algorithms" [Estrin et al. 1999] mentioned earlier, in which a sensor node only interacts with its neighborhood sensors.

5.1 Neighborhood-based Group

A neighborhood-based group is defined locally, consisting of a node and its neighbors. This definition of group captures the nature of local collaboration, which often appears in sensor network applications. In addition, the notion of neighborhood fits well with the broadcasting nature of wireless communication and enables efficient communication within the group.

Abstract Regions [Welsh and Mainland 2004] and Hood [Whitehouse et al. 2004] provide similar programming primitives based on neighborhood. The underlying motivation is that the algorithms used in sensor network applications are often based on local data processing within a neighborhood. Figure 5 shows the programming interface of Abstract Regions. It provides neighborhood discovery, variable sharing via a Linda-like tuple space and also MPI-like reduction operations. In Abstract Regions, groups are defined either topologically (e.g., "N-radio hop"), geographically (e.g., "k-nearest neighbor"), or by their combinations[5]. Hood defines similar set of operations as Abstract Regions, but uses 1-hop neighbor as the sole option for group definition. Abstract Regions also provides a way to tune the trade-off between resource consumption and accuracy in its runtime component.

Technically speaking, using network topology for group definition (e.g., 1-hop neighbor) has little to do with pure application logics, because they are usually independent from the underlying network structure that is unknown at the time of development. It rather originates from the system's convenience for efficiently implementing communication inside groups. Nevertheless, since the definition is intuitive, it provides a good compromise between ease of programming and efficiency.

5.2 Logical Group

A logical group is a definition of group according to logical properties. Since a neighborhood-based group could also be a logical group in this definition, here we take a narrower definition and introduce group-level abstractions that use higher-level logical properties than the physical closeness. Examples of such high-level logical properties include the type of nodes and dynamic input from the environment. Volatility of membership can be another criteria to differentiate logical groups from neighborhood-based groups. While neighborhood-based groups are mostly static, logical groups are more dynamic as the group membership is often determined by dynamic properties such as sensor input.

[5]In Abstract Regions, there are also "approximate planar mesh" and "spanning tree" to define a group for the whole network.

```
/* Discover region */
result_t Region.formRegion(<region specific args>, int timeout);
result_t Region.sync(int timeout);

/* Data sharing */
result_t SharedVar.put(sv_key_t key, sv_value_t val);
result_t SharedVar.get(sv_key_t key, addr_t node, sv_value_t *val, int timeout);
result_t SharedVar.sync(int timeout);

/* Reduction */
result_t Reduce.reduceToOne(op_t operator, sv_key_t value, sv_key_t result,
                            float *yield, int timeout);
result_t Reduce.reduceToAll(op_t operator, sv_key_t value, sv_key_t result,
                            float *yield, int timeout);
result_t Reduce.sync(int timeout);
```

Fig. 5. Programming interface of Abstract Regions (from [Welsh and Mainland 2004], modified).

EnviroTrack [Abdelzaher et al. 2004] is a programming abstraction specifically for target-tracking applications. One characteristic feature in EnviroTrack is that the addresses are assigned to physical events in the environment. A group is defined as the set of sensors that detected the same event. As well as Abstract Regions and Hood, EnviroTrack provides the data sharing and aggregation facilities. However, as the situation is more dynamic, EnviroTrack has a sophisticated distributed group management protocol.

Mottola and Picco propose the notion of logical neighborhood in their SPIDEY language [Mottola and Picco 2006]. A node is represented as a logical node that has several exported attributes. These attributes include both static (e.g., node type) and dynamic properties (e.g., sensor readings). A node can define its logical neighborhood by using a predicate that conditions these attributes. SPIDEY provides communication APIs within logical neighborhood and also an efficient routing mechanism.

5.3 Other Group-level Abstractions

Liu et al. propose a state-centric programming abstraction mainly to alleviate the complexity in programming collaborative signal and information processing (CSIP) applications [Liu et al. 2003]. They employ the notion of "collaboration group": programmers can specify its "scope" to define the members and its "structure" to define the roles each member plays in the group. They provide ways to create various generic patterns of groups including both neighborhood-based group and logical group. Since their focus is more on flexibility, they choose not to provide a rich set of communication operations as in Abstract Regions and Hood. However, their framework provides sufficient expressiveness that allows it to work as a building block for higher level abstractions including other group-level abstractions.

6. NETWORK-LEVEL ABSTRACTION

In network-level abstractions, a sensor network is treated as a whole and is regarded as a single abstract machine. The term "macroprogramming" is often used to mean the same approach, and we will use both network-level abstraction and macropro-

```
SELECT AVG(volume),room FROM sensors
   WHERE floor = 6
   GROUP BY room
   HAVING AVG(volume) > threshold
   SAMPLE PERIOD 30s
```

Fig. 6. Example of SQL-like query in TinyDB (from [Madden et al. 2003]). The scenario is to monitor the occupancy of the conference rooms on the 6-th floor of a building by using microphone sensors. The query reports all rooms where the average volume is over a specified threshold. Updates are delivered every 30 seconds.

gramming in the same meaning. There are two major approaches in network-level abstractions. One is database abstraction. Since sensor networks are often for collecting sensing data, the database is an intuitive metaphor, though there are crucial differences that make the "sensor databases" approach difficult. The other approach is to provide new macroprogramming languages that have broader coverage of applications than the database approach. The goal of macroprogramming languages is to realize programming from macroscopic viewpoint that every node and data can be accessed without considering low-level communications among nodes.

6.1 Database

The database is one of the earliest examples of high-level abstractions for sensor network programming. Cougar [Bonnet et al. 2000] and TinyDB [Madden et al. 2003] fall within this category. As shown in Figure 6, they allow users to issue queries in a declarative SQL-like language. To achieve energy-efficiency, Cougar pushes selection operations to the sensor nodes so that they can reduce the amount of data to be collected. For the same objective, TinyDB focuses on acquisitional issues: where, when and how often to sample and deliver the data. TinyDB also optimizes the routing tree for disseminating a query and collecting the results. Similarly, SINA [Srisathapornphat et al. 2000] provides a database interface that users can query by SQL, but also allows more explicit tasking. In SINA, users can embed scripts written in an imperative language called SQTL (Sensor Querying and Tasking Language) [Jaikaeo et al. 2000] in an SQL query. By this hybrid approach, they can perform more complex collaborative tasks than what SQL can describe.

MiLAN [Heinzelman et al. 2004] provides a data service that features QoS support. In MiLAN, an application submits a query with a QoS requirement. QoS is defined by the level of certainty about an attribute, based on the assumption that each sensor can measure some basic attributes with predefined reliability. In response to a query, MiLAN creates an execution plan, which specifies the source nodes and the routing tree, such that it satisfies the QoS requirement while maximizing energy efficiency. DSWare [Li et al. 2003] is another QoS-aware data service specialized for event detection. Users describe their interests in SQL and register them to DSWare. Similarly to QoS support in MiLAN, DSWare has a notion of confidence about event detection. Confidence is defined on a "compound event", which has multiple atomic events as its sub-events. The presence/absence of these sub-events determines the confidence of a compound event. DSWare also supports real-time semantics where users can specify the time constraint about the latency

```
let aboveThresh (p,x) = p > threshold
    read node =
     (read sensor PROXIMITY node,
      get location node)
in centroid (afilter aboveThresh
                (amap read world))
```

Fig. 7. Example of a program for tracking application written in Regiment (from [Newton and Welsh 2004]). The program estimates the location of a target by computing centroid of the nodes whose proximity sensor measured over a certain threshold value.

```
(a) spatialview sv1
        = Camera @ CampusB % 100;
(b) Rectangle CampusB
        = new Rectangle(...);
(c) visiteach x :  sv1 {
       Picture p=x.getPicture();
       ...
    }
```

Fig. 8. SpatialViews language (from [Ni et al. 2005], modified). (a) Definition of a "spatialview." Type of sensor, space, and space granularity is specified. (b) Definition of space. (c) Iterators can be applied to a spatialview.

until getting a notification after detecting an event.

Database abstraction provides a simple and easy-to-use interface. However, it is suitable only for describing query operations to a sensor network. Although Cougar and TinyDB extend SQL so that users can express continuous sensing tasks, they are still not expressive enough to cover all sorts of sensor network applications, particularly those that require significant amount of fine-grained control flow.

6.2 Macroprogramming Language

Database abstraction is well-designed, providing an intuitive way to access the sensor information, but limits the range of applications. Macroprogramming languages are a complementary approach and are intended to provide more flexibility. There are two major research topics in this approach. One is to provide programming languages suitable for describing global behavior. Ideally, a program that describes global behavior is compiled into node-level binary code. Another major topic is resource naming. When viewing the whole network as a single entity, specifying a node or a group of nodes is essential for programming. There should be a sophisticated resource naming scheme that encapsulates low-level unreliable infrastructure and realizes other features that applications require.

6.2.1 *Description of Global Behavior.* Regiment [Newton and Welsh 2004; Newton et al. 2007] is a functional language specially designed for macroprogramming sensor networks and has a syntax similar to Haskell. Figure 7 is an example of Regiment program for a tracking application. One of the motivations for using a functional language is to hide the direct manipulation of program states from the programmers. This makes it easier for the compiler to extract parallelism, because it can freely decide how and where the program states are stored. Regiment is capable of expressing groups of nodes with geographical, logical, and topological relationships. Regiment programs are compiled into TML [Newton et al. 2005], an intermediate language, and then to nesC code. The main reason for using TML is the big semantic gap between Regiment and nesC/TinyOS.

Kairos [Gummadi et al. 2005] is another programming abstraction that allows macroprogramming. Unlike Regiment, Kairos is language-independent so that it can be implemented as an extension to existing programming languages. Another difference is that Kairos focuses on providing a small set of constructs, containing

only abstractions for nodes, one-hop neighbors, and remote data access, whereas Regiment defines much more data types and operations. Remote data access, which is realized simply by "var@node", provides shared memory abstraction across nodes. To reduce communication overhead, Kairos employs a weak consistency model called "eventual consistency."

6.2.2 *Resource Naming.* Spatial Programming [Borcea et al. 2004] is a space-aware programming model that allows resource to be referenced by physical location and other properties. For example, it allows notation like "Hill1:camera[0]" for addressing a camera node on "Hill1", which is a predefined spatial region. Spatial programming realizes transparent access to network resources in this way using Smart Messages, which are similar to mobile agents. A Smart Message consists of code, data, and execution state and migrates to nodes of interest and then executes the computation.

Similarly, SpatialViews [Ni et al. 2005] is a high-level language that allows specification of a "spatialview" with the nodes having properties of interest. A spatialview is a first-class abstraction that is defined to be a network of nodes explicitly named by the services and locations. Each spatialview is instantiated dynamically across time and thus realizes dynamic binding, whereas Spatial Programming only uses static binding. Figure 8 shows an example of SpatialViews program. In this way, programmers can define a subset of sensors as a group and perform operations on it.

Intanagonwiwat et al. propose a declarative resource naming (DRN) scheme for macroprogramming [Intanagonwiwat et al. 2005]. It is more flexible compared to both Spatial Programming and SpatialViews. In DRN, a set of resources can be specified not only in an imperative way but also in a declarative way by indicating desired properties by boolean expressions. DRN supports both dynamic and static resource binding, and allows sequential and parallel access to the specified resource for better handling of concurrency.

6.3 Other Network-level Abstractions

Semantic Streams [Whitehouse et al. 2006] is a framework that allows declarative queries over semantic interpretations of sensor data. Semantic interpretation is made possible by composing inference units, which infer semantic information (e.g., "detect a vehicle") from incoming events (e.g., "magnetometer reading exceeded the threshold"). Users can issue queries directly over semantic interpretations, instead of querying for the sensor data and interpreting the result by themselves. The Prolog-based implementation enables automatic composition of sensors and inference units.

Software Sensor [Lin 2004] is another programming model that originates from a software engineering perspective. The basic idea is to provide an abstraction of each sensor hardware by a Software Sensor, which is a service in the context of service oriented architecture. By composing multiple Software Sensors, programmers can define a variety of large-scale collaborations in a flexible way. The Software Sensor model is supported by their *SensorJini* middleware, which is implemented on top of Jini, a Java-based network architecture designed for distributed systems. *SensorJini* provides a lookup service so that an application can find and bind to

the Software Sensors of interest.

7. EVALUATION

Figure 9 summarizes how each of the programming models addresses the requirements we have identified earlier. For the rest of the section, we highlight important findings, focusing on the typical strategies to satisfy each requirement. Note that the requirements are not independent to each other, and thus a single feature of programming models can work in favor of (or against) multiple requirements. For instance, reducing the amount of communication improves energy-efficiency and also scalability, but may reduce failure-resilience.

7.1 Energy-Efficiency

These are three primary mechanisms to improve energy-efficiency through programming:

(1) Efficient runtime mechanisms: caching, routing, in-network summarization
(2) Control trade-off with "quality"
(3) (In the extended definition) Efficient code update

Energy efficiency is often realized by the runtime mechanisms that accompany the programming models. Caching reduces the communication and thus helps save the energy. Caching is employed in a number of programming models including Abstract Regions [Welsh and Mainland 2004], Hood [Whitehouse et al. 2004], EnviroTrack [Abdelzaher et al. 2004], and Kairos [Gummadi et al. 2005] on data sharing. For sharing data within logical groups, SPIDEY [Mottola and Picco 2006] has an efficient routing mechanism. In-network summarization is intensively used in most database approaches.

A complementary approach for energy efficiency is to let programmers and users control the trade-off between efficiency and various kinds of quality. An example of this trade-off is between the data sampling rate and the amount of communication: lower sampling rate induces larger latency from event occurrence and thus lower quality, but requires less amount of communication. Abstract Region takes this approach by exposing the low-level control knobs such as "number of retransmission." As a higher level approach, BBQ [Deshpande et al. 2004], an extension of TinyDB [Madden et al. 2003], allows query with quality specification and replies with an answer satisfying the specification. Semantic Streams [Whitehouse et al. 2006] allows an energy-constrained query as well, which restricts the amount of energy consumed for resolving the query. Similar idea is implemented also in TinyDB as an acquisitional query processor, which adaptively changes sampling rate so that the battery lasts until the specified lifetime.

When we extend the scope of energy efficiency to the whole lifecycle of sensor networks, code update in VM-based approach needs to be done efficiently. Maté [Levis and Culler 2002] and Melete [Yu et al. 2006] realize this by using a compact intermediate code. Reprogramming per module is another way to achieve efficiency by eliminating duplicate components to be transmitted over network. It is implemented in VMStar [Koshy and Pandey 2005], Impala [Liu and Martonosi 2003], Mantis OS [Bhatti et al. 2005], and SOS [Han et al. 2005].

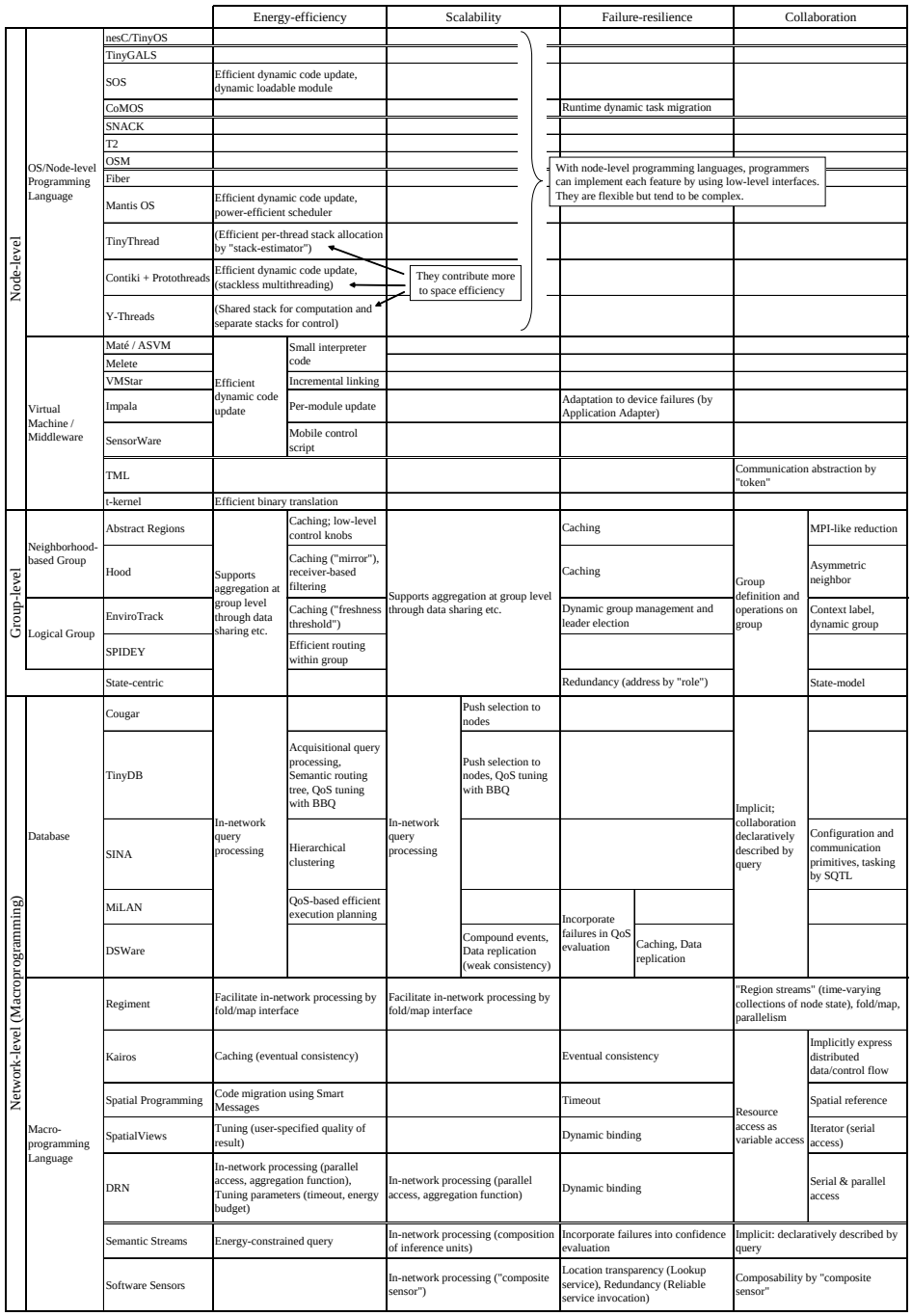

Fig. 9. Evaluation of programming models for sensor networks

7.2 Scalability

The primary approach to achieving scalability is to reduce data by in-network summarization. Especially in data collection type application, a key to scalability is to reduce the data that are sent all the way to the center. Otherwise, even if the amount of data is small at the edge of the network, it quickly fills up the scarce bandwidth as it approaches the center because of the aggregation over nodes. Thus reducing the communication is both beneficial in terms of energy and also in terms of scalability.

This issue of scalability has been recognized since the early days. Cougar [Bonnet et al. 2000], one of the earliest work in the database approach, pushes query to the nodes so that selections are performed at the edge and thus the data transmission is reduced. Other database approaches including TinyDB [Madden et al. 2003] also have a similar mechanism.

Other than runtime support, explicitly writing a program for reducing data within network helps increasing scalability. The MPI-like aggregation operation equipped in group-level abstractions and some macroprogramming languages is a good tool for this purpose.

7.3 Failure-resilience

There are four measures taken in programming models to improve failure-resilience:

(1) Redundancy: addressing by roles
(2) Caching
(3) Dynamic binding
(4) Incorporate "failure" in quality/confidence assessment

Redundancy is a standard way to improve the reliability of a function, and addressing by roles instead of node names fits well to redundant deployment. When duplicating a function to multiple nodes, it can be awkward to address these nodes by their unique names. Therefore the idea is to assign a role to the group of nodes and use the name of role. This allows a flexible deployment because it does not matter what degree of redundancy is chosen. State-centric programming [Liu et al. 2003] has this mechanism.

Caching is effective for energy-efficiency as we have already discussed, but also for failure-resilience, as it alleviates communication failures by substituting the data with old cached data. However in this case, the old data may or may not be useful, depending on the application's purpose.

Dynamic binding is useful to cope with node failures. Unlike statically binding to nodes, it enumerates the available set of nodes when we need to access them. SpatialViews [Ni et al. 2005] and DRN [Intanagonwiwat et al. 2005] have dynamic binding mechanisms.

A unique approach to deal with failures is to view them in a more quantitative way. In many application scenarios, the availability of each node and data affects the quality of monitoring. For instance, the error of location estimation is likely to be smaller when there are more proximity sensor nodes available. In some QoS-aware programming models such as MiLAN [Heinzelman et al. 2004] and DSWare [Li et al. 2003], failures are treated as quality degradation or added uncertainty.

Based on the quantitative evaluations, we can make appropriate actions to alleviate the failures.

7.4 Collaboration

Collaboration is a unique and important requirement in sensor networks. As represented by target localization by triangulation/trilateration, there are certain types of information that are not determined by a single sensor but only by multiple of them. There are three primary mechanisms in programming models to support collaboration among nodes:

(1) Group definition/operations for group
(2) Declarative query
(3) Resources as variables

Group definition and operations for group make it easier to describe collaborative behavior among nodes. They allow programmers to define a group by multiple criteria. They also enable data sharing within a group either explicitly or implicitly as in shared address space. Group-level abstractions are designed primarily for facilitating data sharing and some macroprogramming languages such as Spatial Programming [Borcea et al. 2004], SpatialViews [Ni et al. 2005], DRN [Intanagonwiwat et al. 2005] also have this feature. Other than data sharing, the operations on group include enumeration using iterator and MPI-like reduction.

Declarative query in database approach is a different way to realize collaboration. Instead of explicitly defining a group and operations on it, users only describe the data they need. The corresponding execution plan is devised by each query processor's runtime mechanism. The execution plan may employ localized algorithm when possible, but it is not under control of neither users nor programmers, with a few exceptions like SINA [Srisathapornphat et al. 2000], in which users can embed imperative descriptions of collaborative tasks into an SQL-like query.

Lastly, some macroprogramming languages enable programmers to describe a resource access as a variable access. A collaboration among multiple nodes is more easily programmed by manipulations of multiple variables. Kairos [Gummadi et al. 2005] implements this in a form of "data@node." Spatial Programming, SpatialViews, DRN combine this approach with group definition.

7.5 Other evaluation metrics

As discussed in the end of Section 2, sensor network applications have various other requirements and the four evaluation criteria discussed above are by no means complete. Among the criteria not discussed so far, "ease of programming" is very important one that is most directly related to programming models. Clearly there is a trade-off between ease of programming and flexibility. In general, low-level programming languages are more flexible but bring more complexity in programming, while high-level programming models accommodate more intuitive programming styles but are often difficult or unable to fine-tune the performance. In sensor network programming, a similar argument holds. Node-level programming languages such as nesC correspond to the low-level and network-level programming models such as TinyDB correspond to the high-level.

One crude way to evaluate ease of programming is by comparing the lines of code for the same program in different programming languages, based on the assumption that shorter code implies less programming effort required. Let's take an example of target tracking application that calculates the centroid of nodes that detected the target. The lines of code are 369 in nesC, 134 in Abstract Regions with Fiber (both figures from [Welsh and Mainland 2004]) and 6 in Regiment [Newton and Welsh 2004]. Of course, this is just an example. It is difficult to evaluate ease of programming in general since the criteria of easiness is inherently subjective and the complexity of code largely depends on each application. If an application requires very precise control, node-level programming languages would be the only possible choice. On the other hand, if an application is simple data collection and programmers prefer fast prototyping rather than sophisticated, energy-efficient code, database or other network-level programming models would fit the best.

8. FUTURE RESEARCH DIRECTIONS

In this section, we discuss possible future research directions in programming models for sensor networks. Specifically, we identify the problems that will be necessary and important in this area from the perspective of applications and systems, and discuss their implications for programming models.

8.1 Heterogeneity

Recent trends in sensor network applications exhibit more heterogeneity in hardware platform than early mote-based systems. Some of the deployed systems [Martinez et al. 2004; Whitehouse et al. 2006] include multiple different types of sensors, which is a basic form of heterogeneity. Hierarchical architecture is also often the case, in which the upper tier devices are more computationally powerful than the lower ones and usually with more energy and network resources. These upper tier devices may even have a continuous power supply and wired network, and it is shown that a small number of these powerful nodes have a significant impact on performance as a whole [Yarvis et al. 2005]. Sensor-actuator networks are an emerging area and another major example of heterogeneous setting. Many of the traditional problem settings in sensor networks do not apply to these cases. From the viewpoint of programming, how to use heterogeneous devices effectively and how to program the whole system including these devices remain non-trivial problems.

Some recent work addresses the heterogeneity. Tenet [Gnawali et al. 2006] is an architecture for tiered embedded networks. Tenet explicitly assumes a tiered network that consists of 32-bit platform nodes in the upper tier and motes in the lower tier. Further, there is a clear separation of work between two tiers: the motes only perform simple tasks whereas other complex tasks such as multisensor fusion only happen at the upper tier nodes. The upper tier nodes host the applications and assign tasks to the lower tier motes. Tenet simplifies application development by imposing these constraints on the network architecture. CoMOS [Han et al. 2006] is an operating system designed for mPlatform [Lymberopoulos et al. 2007], which is a sensornet platform that has multiple heterogeneous processors.

As well as the physical heterogeneity discussed so far, it is often the case that each sensor node needs to behave differently to achieve the coordinated action

specified by the applications. This can be referred to as "logical heterogeneity." One common example is clustering, where each node sends information to the clusterhead assigned to each cluster. The clusterhead summarizes the data to reduce the communication to the base station. One possible approach for programming such logically heterogeneous nodes is to decompose the whole process into role assignment and programming. Liu et al. [Liu et al. 2003] propose the notion of role, but mainly as an indirection with which node failure can be accommodated. Frank and Römer propose a generic role assignment scheme where roles and rules for their assignment are specified by using a declarative configuration language [Frank and Römer 2005]. They devise an efficient algorithm for assigning roles to sensor nodes in a manner such that the configuration specified by users is satisfied. Their scheme is focused on assigning roles to nodes and not on the subsequent programming part. A comprehensive framework that incorporates both of them would be beneficial.

8.2 Strict QoS Support

Quality of service (QoS) is prevalent in sensor networks due to the fact that it is often too inefficient to collect high quality data. There is a trade-off between energy and various kinds of quality such as accuracy, latency, and error rates. People often need to balance them so that they can efficiently get the data with sufficient quality.

QoS is sometimes considered "nice-to-have" in other computer systems, but is often an integral element in sensor network applications. This is because the ultimate objective of using sensor networks is not the sensing per se, but taking actions that are either implicitly or explicitly determined according to the result of sensing. It could hardly be valuable if the information obtained from sensor networks has low quality insufficient for the subsequent actions. For example, when detecting a wildfire, too many false positive and false negative errors would ruin the application. Also, when detecting and tracking seismic waves to issue an emergency alert, too large latency would make the application useless. In many sensor network applications including these examples, various types of QoS such as accuracy and latency are essential.

Some programming models have supports for QoS. Abstract Regions [Welsh and Mainland 2004] exposes some low level tuning knobs that affect the quality of results, so that programmers can indirectly control the trade-off between quality and efficiency. However, this way of QoS control does not directly help users obtain the data that satisfies designated level of quality, though it is probably useful for evaluating the post-facto quality. MiLAN [Heinzelman et al. 2004], DSWare [Li et al. 2003] and Semantic Streams [Whitehouse et al. 2006] have explicit QoS support, but their definitions are rather subjective and/or qualitative. For example in MiLAN, QoS is determined by reliability between $0.0 - 1.0$. Each sensor has predefined reliability of measuring a certain attribute. For example, a blood pressure sensor has reliability of less than 1.0 on heart rate, since it can only indirectly measure the heart rate. However, it is difficult to determine reasonable reliability values.

When we consider the importance of QoS in the context of sensor networks, it is necessary to have support for QoS in a more strict way. In other words, users should be able to take total control over the quality-efficiency trade-off at the application-

level and also to know the quality of data they collect. Programming models should export such interface to users as well as have an underlying runtime mechanism that realizes precise control of quality.

There are some work that allow users control the accuracy and exploit the quality-efficiency trade-off. BBQ [Deshpande et al. 2004] is a model-based query processor that can be used along with TinyDB. It gives statistical guarantee about the accuracy of data described using confidence levels, but based on the assumption that the sensor data follows a multivariate Gaussian distribution. Compressive Wireless Sensing [Bajwa et al. 2006] is an efficient field estimation technique that guarantees the accuracy of data. It requires no prior knowledge about data properties but assumes Gaussian noise. Sugihara and Chien [Sugihara and Chien 2005; 2006] have proposed an energy-efficient algorithm to satisfy a user-specified accuracy requirement in the data-gathering application. This algorithm does not need any assumption on the data and noise, and works more efficiently when data has stronger spatial correlation. This strong guarantee is appropriate for detecting the outliers, which is often important in sensor network applications such as wildfire detection.

9. SUMMARY AND CONCLUSIONS

Sensor networks' capabilities open exciting possibilities for their applications. These applications often push the conceptual limits to traditional programming environments, because of the physical constraints in which they operate. This paper takes a systematic look at these growing requirements on the programming models for sensor networks. While this survey extensively covers the most prominent programming approaches for sensor networks, by no means it is complete. Indeed, to limit our scope, we defined class of nodes (scoped as W-, mW-, and μW-order nodes) and focus our attention on mW-order nodes that seek to balance computing, communications, and sensing capabilities. We built a simple but useful taxonomy that classifies the programming models into three classes based on their levels of abstraction. We evaluated each of these programming models in terms of the requirements imposed by the application.

While this paper seeks to provide a framework to reason about the evolving programming models for sensor network applications, we have left the issue of suitable programming language(s) unaddressed. Clearly, programming models have an influence upon the choice or design of a suitable programming languages, while shifts in programming languages are rather infrequent and, more importantly, not often predictable. These are often driven by pragmatic issues such as availability of diverse array of programming language tools, debuggers, libraries etc. Based on our understanding of the sensor network applications programming, dominant among these is the need to keep application logic separate and independent of the myriad application level constraints that a particular platform may impose. This largely reflects the trade-off between programming ease and implementation efficiency that several programming models, discussed in this paper, have made.

10. ACKNOWLEDGMENTS

The authors would like to thank Tarek Abdelzaher and the anonymous reviewers for their valuable comments.

REFERENCES

Abdelzaher, T., Blum, B., Cao, Q., Chen, Y., Evans, D., George, J., George, S., Gu, L., He, T., Krishnamurthy, S., Luo, L., Son, S., Stankovic, J., Stoleru, R., and Wood, A. 2004. EnviroTrack: towards an environmental computing paradigm for distributed sensor networks. In *ICDCS'04: Proceedings of the 24th International Conference on Distributed Computing Systems*. 582–589.

Bajwa, W., Haupt, J., Sayeed, A., and Nowak, R. 2006. Compressive wireless sensing. In *IPSN'06: Proceedings of the 5th international conference on Information processing in sensor networks*. 134–142.

Batalin, M. A., Rahimi, M., Yu, Y., Liu, D., Kansal, A., Sukhatme, G. S., Kaiser, W. J., Hansen, M., Pottie, G. J., Srivastava, M., and Estrin, D. 2004. Call and response: experiments in sampling the environment. In *SenSys'04: Proceedings of the 2nd international conference on Embedded networked sensor systems*. 25–38.

Bhatti, S., Carlson, J., Dai, H., Deng, J., Rose, J., Sheth, A., Shucker, B., Gruenwald, C., Torgerson, A., and Han, R. 2005. MANTIS OS: an embedded multithreaded operating system for wireless micro sensor platforms. *Mobile Networks and Applications 10,* 4, 563–579.

Bonnet, P., Gehrke, J., and Seshadri, P. 2000. Querying the physical world. *IEEE Personal Communications 7,* 5, 10–15.

Borcea, C., Intanagonwiwat, C., Kang, P., Kremer, U., and Iftode, L. 2004. Spatial programming using smart messages: design and implementation. In *ICDCS'04: Proceedings of 24th International Conference on Distributed Computing Systems*. 690–699.

Boulis, A., Han, C.-C., and Srivastava, M. B. 2003. Design and implementation of a framework for efficient and programmable sensor networks. In *MobiSys'03: Proceedings of the 1st international conference on Mobile systems, applications and services*. 187–200.

Cheong, E., Liebman, J., Liu, J., and Zhao, F. 2003. TinyGALS: a programming model for event-driven embedded systems. In *SAC'03: Proceedings of the ACM symposium on Applied computing*. 698–704.

Chu, M., Haussecker, H., and Zhao, F. 2002. Scalable information-driven sensor querying and routing for ad hoc heterogeneous sensor networks. *International Journal of High Performance Computing Applications 16,* 3, 90–110.

Deshpande, A., Guestrin, C., Madden, S., Hellerstein, J., and Hong, W. 2004. Model-driven data acquisition in sensor networks. In *VLDB'04: Proceedings of the 30th International Conference on Very Large Data Bases*.

Dunkels, A., Finne, N., Eriksson, J., and Voigt, T. 2006. Run-time dynamic linking for reprogramming wireless sensor networks. In *SenSys'06: Proceedings of the 4th international conference on Embedded networked sensor systems*. 15–28.

Dunkels, A., Grönvall, B., and Voigt, T. 2004. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *EmNetS-I: Proceedings of the 1st IEEE Workshop on Embedded Networked Sensors*.

Dunkels, A., Schmidt, O., Voigt, T., and Ali, M. 2006. Protothreads: simplifying event-driven programming of memory-constrained embedded systems. In *SenSys'06: Proceedings of the 4th international conference on Embedded networked sensor systems*. 29–42.

Estrin, D., Govindan, R., Heidemann, J., and Kumar, S. 1999. Next century challenges: scalable coordination in sensor networks. In *MobiCom'99: Proceedings of the 5th annual ACM/IEEE international conference on Mobile computing and networking*. 263–270.

Frank, C. and Römer, K. 2005. Algorithms for generic role assignment in wireless sensor networks. In *SenSys'05: Proceedings of the 3rd international conference on Embedded networked sensor systems*. 230–242.

Gay, D., Levis, P., von Behren, R., Welsh, M., Brewer, E., and Culler, D. 2003. The nesC language: a holistic approach to networked embedded systems. In *PLDI'03: Proceedings of the ACM SIGPLAN conference on Programming language design and implementation*. 1–11.

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

SenSys'06: Proceedings of the 4th international conference on Embedded networked sensor systems. 153–166.

Greenstein, B., Kohler, E., and Estrin, D. 2004. A sensor network application construction kit (SNACK). In *SenSys'04: Proceedings of the 2nd international conference on Embedded networked sensor systems*. 69–80.

Gu, L. and Stankovic, J. A. 2006. t-kernel: providing reliable os support to wireless sensor networks. In *SenSys'06: Proceedings of the 4th international conference on Embedded networked sensor systems*. 1–14.

Gummadi, R., Gnawali, O., and Govindan, R. 2005. Macro-programming wireless sensor networks using *kairos*. In *DCOSS'05: Proceedings of the 1st international conference on Distributed Computing in sensor systems*. 126–140.

Hadim, S. and Mohamed, N. 2006. Middleware challenges and approaches for wireless sensor networks. *IEEE Distributed Systems Online 7,* 3.

Han, C.-C., Goraczko, M., Helander, J., Priyantha, J. L. B., and Zhao, F. 2006. CoMOS: An operating system for heterogeneous multi-processor sensor devices. *Microsoft Research Technical Report MSR-TR-2006-177*.

Han, C.-C., Kumar, R., Shea, R., Kohler, E., and Srivastava, M. 2005. A dynamic operating system for sensor nodes. In *MobiSys'05: Proceedings of the 3rd international conference on Mobile systems, applications, and services*. 163–176.

He, T., Krishnamurthy, S., Stankovic, J. A., Abdelzaher, T., Luo, L., Stoleru, R., Yan, T., Gu, L., Hui, J., and Krogh, B. 2004. Energy-efficient surveillance system using wireless sensor networks. In *MobiSys'04: Proceedings of the 2nd international conference on Mobile systems, applications, and services*. 270–283.

Heinzelman, W., Murphy, A., Carvalho, H., and Perillo, M. 2004. Middleware to support sensor network applications. *IEEE Network 18,* 1, 6–14.

Hill, J., Szewczyk, R., Woo, A., Hollar, S., Culler, D., and Pister, K. 2000. System architecture directions for networked sensors. In *ASPLOS-IX: Proceedings of the 9th international conference on Architectural support for programming languages and operating systems*. 93–104.

Hui, J. W. and Culler, D. 2004. The dynamic behavior of a data dissemination protocol for network programming at scale. In *SenSys'04: Proceedings of the 2nd international conference on Embedded networked sensor systems*. 81–94.

Intanagonwiwat, C., Gupta, R., and Vahdat, A. 2005. Declarative resource naming for macro-programming wireless networks of embedded systems. *UCSD Technical Report CS2005-0827*.

Jaikaeo, C., Srisathapornphat, C., and Shen, C.-C. 2000. Querying and tasking in sensor networks. In *SPIE's 14th Annual International Symposium on Aerospace/Defense Sensing, Simulation, and Control (Digitization of the Battlespace V)*. Vol. 4037. 184–197.

Juang, P., Oki, H., Wang, Y., Martonosi, M., Peh, L. S., and Rubenstein, D. 2002. Energy-efficient computing for wildlife tracking: design tradeoffs and early experiences with ZebraNet. In *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*. 96–107.

Kasten, O. and Römer, K. 2005. Beyond event handlers: programming wireless sensors with attributed state machines. In *IPSN'05: Proceedings of the 4th international symposium on Information processing in sensor networks*. 45–52.

Koshy, J. and Pandey, R. 2005. VMSTAR: synthesizing scalable runtime environments for sensor networks. In *SenSys'05: Proceedings of the 3rd international conference on Embedded networked sensor systems*. 243–254.

Levis, P. and Culler, D. 2002. Maté: a tiny virtual machine for sensor networks. In *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*. 85–95.

Levis, P., Gay, D., and Culler, D. 2005. Active sensor networks. In *NSDI'05: Proceedings of the 2nd USENIX/ACM Symposium on Networked Systems Design and Implementation*.

Levis, P., Gay, D., Handziski, V., Hauer, J.-H., Greenstein, B., Turon, M., Hui, J., Klues, K., Sharp, C., Szewczyk, R., Polastre, J., Buonadonna, P., Nachman, L., Tolle, G., Culler, D., and Wolisz, A. 2005. T2: A second generation os for embedded sensor networks.

Technical Report, Telecommunication Networks Group, Technische Universität Berlin TKN-05-007.

Li, S., Son, S. H., and Stankovic, J. A. 2003. Event detection services using data service middleware in distributed sensor networks. In *IPSN'03: Proceedings of the 2nd international symposium on Information processing in sensor networks*. 502–517.

Lin, E. 2004. Design and implementation of a programming model and middleware for sensor networks. M.S. thesis, University of California, San Diego.

Liu, J., Chu, M., Liu, J. J., Reich, J. E., and Zhao, F. 2003. State-centric programming for sensor and actuator network systems. *IEEE Pervasive Computing 2,* 4, 50–62.

Liu, T. and Martonosi, M. 2003. Impala: a middleware system for managing autonomic, parallel sensor systems. In *PPoPP'03: Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*. 107–118.

Lundquist, J. D., Cayan, D. R., and Dettinger, M. D. 2003. Meteorology and hydrology in Yosemite national park: A sensor network application. In *IPSN'03: Proceedings of the 2nd international symposium on Information processing in sensor networks*. 518–528.

Lymberopoulos, D., Priyantha, N. B., and Zhao, F. 2007. mPlatform: a reconfigurable architecture and efficient data sharing mechanism for modular sensor nodes. In *IPSN'07: Proceedings of the 6th international conference on Information processing in sensor networks*. 128–137.

Madden, S., Franklin, M. J., Hellerstein, J. M., and Hong, W. 2003. The design of an acquisitional query processor for sensor networks. In *SIGMOD'03: Proceedings of the ACM SIGMOD international conference on Management of data*. 491–502.

Martinez, K., Hart, J. K., and Ong, R. 2004. Environmental sensor networks. *IEEE Computer 37,* 8, 50–56.

McCartney, W. P. and Sridhar, N. 2006. Abstractions for safe concurrent programming in networked embedded systems. In *SenSys'06: Proceedings of the 4th international conference on Embedded networked sensor systems*. 167–180.

Mottola, L. and Picco, G. P. 2006. Logical neighborhoods: A programming abstraction for wireless sensor networks. In *DCOSS'06: Proceedings of the 2nd international conference on Distributed Computing in sensor systems*. 150–168.

Newton, R., Arvind, and Welsh, M. 2005. Building up to macroprogramming: an intermediate language for sensor networks. In *IPSN'05: Proceedings of the 4th international symposium on Information processing in sensor networks*. 37–44.

Newton, R., Morrisett, G., and Welsh, M. 2007. The regiment macroprogramming system. In *IPSN'07: Proceedings of the 6th international conference on Information processing in sensor networks*. 489–498.

Newton, R. and Welsh, M. 2004. Region streams: functional macroprogramming for sensor networks. In *DMSN'04: Proceedings of the 1st international workshop on Data management for sensor networks*. 78–87.

Ni, Y., Kremer, U., Stere, A., and Iftode, L. 2005. Programming ad-hoc networks of mobile and resource-constrained devices. In *PLDI'05: Proceedings of the ACM SIGPLAN conference on Programming language design and implementation*. 249–260.

Nitta, C., Pandey, R., and Ramin, Y. 2006. Y-threads: Supporting concurrency in wireless sensor networks. In *DCOSS'06: Proceedings of the 2nd international conference on Distributed Computing in sensor systems*. 169–184.

Ousterhout, J. K. 1996. Why threads are a bad idea (for most purposes). In USENIX Technical Conference (Invited Talk).

Pattem, S., Poduri, S., and Krishnamachari, B. 2003. Energy-quality tradeoffs for target tracking in wireless sensor networks. In *IPSN'03: Proceedings of the 2nd international symposium on Information processing in sensor networks*. 32–46.

Römer, K. 2004. Programming paradigms and middleware for sensor networks. In *GI/ITG Workshop on Sensor Networks*. 49–54.

Simon, G., Maróti, M., Lédeczi, Á., Balogh, G., Kusy, B., Nádas, A., Pap, G., Sallai, J., and Frampton, K. 2004. Sensor network-based countersniper system. In *SenSys'04: Proceedings of the 2nd international conference on Embedded networked sensor systems*. 1–12.

Srisathapornphat, C., Jaikaeo, C., and Shen, C.-C. 2000. Sensor information networking architecture. In *ICPP'00: Proceedings of the International Workshop on Parallel Processing*. 23–30.

Sugihara, R. and Chien, A. A. 2005. Accuracy-aware data modeling in sensor networks. In *SenSys'05: Proceedings of the 3rd international conference on Embedded networked sensor systems*. 282–283.

Sugihara, R. and Chien, A. A. 2006. An efficient general-purpose mechanism for data gathering with accuracy requirement in wireless sensor networks. *UCSD Technical Report CS2006-0854*.

Szewczyk, R., Osterweil, E., Polastre, J., Hamilton, M., Mainwaring, A., and Estrin, D. 2004. Habitat monitoring with sensor networks. *Communications of the ACM 47,* 6, 34–40.

Tolle, G., Polastre, J., Szewczyk, R., Culler, D., Turner, N., Tu, K., Burgess, S., Dawson, T., Buonadonna, P., Gay, D., and Hong, W. 2005. A macroscope in the redwoods. In *SenSys'05: Proceedings of the 3rd international conference on Embedded networked sensor systems*. 51–63.

von Behren, R., Condit, J., and Brewer, E. 2003. Why events are a bad idea (for high-concurrency servers). In *HotOS IX: Proceedings of the 9th workshop on Hot Topics in Operating Systems*.

Welsh, M. and Mainland, G. 2004. Programming sensor networks using abstract regions. In *NSDI'04: Proceedings of the 1st USENIX/ACM Symposium on Networked Systems Design and Implementation*.

Whitehouse, K., Sharp, C., Brewer, E., and Culler, D. 2004. Hood: a neighborhood abstraction for sensor networks. In *MobiSys'04: Proceedings of the 2nd international conference on Mobile systems, applications, and services*. 99–110.

Whitehouse, K., Zhao, F., and Liu, J. 2006. Semantic streams: A framework for composable semantic interpretation of sensor data. In *EWSN'06: Proceedings of the 3rd European Workshop on Wireless Sensor Networks*. 5–20.

Xu, N., Rangwala, S., Chintalapudi, K. K., Ganesan, D., Broad, A., Govindan, R., and Estrin, D. 2004. A wireless sensor network for structural monitoring. In *SenSys'04: Proceedings of the 2nd international conference on Embedded networked sensor systems*. 13–24.

Yarvis, M., Kushalnagar, N., Singh, H., Rangarajan, A., Liu, Y., and Singh, S. 2005. Exploiting heterogeneity in sensor networks. In *INFOCOM'05: Proceedings of the 24th Annual Joint Conference of the IEEE Computer and Communications Societies*. 878–890.

Yu, Y., Rittle, L. J., Bhandari, V., and LeBrun, J. B. 2006. Supporting concurrent applications in wireless sensor networks. In *SenSys'06: Proceedings of the 4th international conference on Embedded networked sensor systems*. 139–152.

Received Dec 2006; revised Jun 2007; accepted *** 200X