Programming Wireless Sensor Networks: Fundamental Concepts and State of the Art

Luca Mottola
University of Trento, Italy and Swedish Institute of Computer Science, Sweden
and
Gian Pietro Picco
University of Trento, Italy

Wireless sensor networks (WSNs) are attracting great interest in a number of application domains concerned with monitoring and control of physical phenomena, as they enable dense and untethered deployments at low cost and with unprecedented flexibility.

However, application development is still one of the main hurdles to a wide adoption of WSN technology. In current real-world WSN deployments, programming is typically carried out very close to the operating system, therefore requiring the programmer to focus on low-level system issues. This not only distracts the programmer from the application logic, but also requires a technical background rarely found among application domain experts. The need for appropriate high-level programming abstractions, capable of simplifying the programming chore without sacrificing efficiency, has been long recognized and several solutions have been hitherto proposed, which differ along many dimensions.

In this paper, we survey the state of the art in programming approaches for WSNs. We begin by presenting a taxonomy of WSN applications, to identify the fundamental requirements programming platforms must deal with. Then, we introduce a taxonomy of WSN programming approaches that captures the fundamental differences among existing solutions, and constitutes the core contribution of this paper. Our presentation style relies on concrete examples and code snippets taken from programming platforms representative of the taxonomy dimensions being discussed. We use the taxonomy to provide an exhaustive classification of existing approaches. Moreover, we also map existing approaches back to the application requirements, therefore providing not only a complete view of the state of the art, but also useful insights for selecting the programming abstraction most appropriate to the application at hand.

Categories and Subject Descriptors: D.3 [**Programming Languages**]: ; D.3.2 [**Language Classification**]: ; D.1 [**Programming Techniques**]: ; C.2.4 [**Distributed Systems**]:

Additional Key Words and Phrases: Wireless Sensor Networks, Networked Embedded Systems, Programming Abstractions, Middleware.

1. INTRODUCTION

Wireless sensor networks (WSNs) are distributed systems typically composed of embedded devices, each equipped with a processing unit, a wireless communication interface, as well as sensors and/or actuators. Many applications have been proposed to date that show the versatility of this technology, and some are already finding their way into the mainstream. Most often, in these scenarios tiny battery-powered devices are used for ease of deployment and increased flexibility [Akyildiz et al. 2002]. This enables embedding processing and communication within the physical world, providing low-cost, fine-grained interaction with the environment.

Although hardware advances play an important role in WSNs, the power of this technology can be fully harnessed only if proper software platforms are made

available to application developers [OnWorld ; CONET]. However, of the several experiences reported in the literature where WSN applications have been *deployed* in the real-world, only a few exceptions rely on some high-level programming support [Ceriotti et al. 2009; Buonadonna et al. 2005; Whitehouse et al. 2004]. In the majority of deployments, programming is instead carried out very close to the operating system, forcing programmers to deal with low-level system issues as well as with the design of distributed protocols. This not only shifts the programmer's focus away from the application logic, but also requires a technical background rarely found among application domain experts.

There is a growing awareness about this problem in the research community, and an increasing number of approaches are being proposed. However, on one hand existing approaches provide a wide and diverse set of functionality and, on the other hand, WSN applications have widely different characteristics and requirements. Choosing the best platform for a given application demands a clear understanding of the application needs and of the basic differences among programming approaches. Thus far, the research community has investigated these aspects only to a limited extent. Therefore, we begin by presenting a taxonomy of WSN applications in Section 2. Many applications have been proposed to date, which differ greatly along many dimensions. Therefore, it is useful to identify their fundamental differences, in that these ultimately determine the applicability of a given programming approach to the problem at hand.

The main contribution of this paper is an extensive survey and classification of the state of the art in WSN programming approaches. However, the term "programming abstraction" is widely used in WSNs, with different meanings. For instance, OS-level concurrency mechanisms [Nitta et al. 2006] as well as service-oriented interfaces are sometimes termed as "programming abstractions for WSNs". In this work, we place the emphasis on the *distributed processing* occurring *inside* the WSN, focusing on solutions that allow programmers to express communication and coordination among the WSN nodes. These aspects are of utmost importance in WSN programming and no well-established solution exists yet.

Nonetheless, clearly defining the conceptual boundaries between the subject of this paper and the overall state of the art is a particularly tricky issue in WSNs, where abstraction layers often blend for optimizing resources. Section 3 describes a reference architecture whose purpose is to define clearly what belongs to our survey and what does not. In addition, it provides the reader with a background about WSNs by concisely covering issues that bear an influence on programming abstractions.

The rest of the paper focuses on a taxonomy of WSN programming approaches. Our work captures the fundamental differences among existing solutions and is exhaustive in covering the current state of the art. Section 4 contains a brief overview of the goals and structure of our taxonomy, whose presentation is split in two complementary parts. Section 5 focuses on the characteristics of the *language* constructs provided to the programmer, therefore analyzing the different approaches for expressing communication and computation, the model used for accessing data, and the programming paradigm adopted. Section 6 focuses on *architectural* issues, by classifying approaches according to whether they replace or complement others,

to whether they can be used only for building end-user applications or lower-level mechanisms as well, to the extent they can be configured in their low-level aspects, and to their execution environment.

We illustrate each dimension in our taxonomy by analyzing the features of existing systems representative of such dimension. The presentation of each system always includes some code fragments or small applications, to provide the reader with a *concrete* grasp of the differences among approaches. Given the number of dimensions in our taxonomy, the set of systems we use as examples allows us to cover in detail a significant fraction of the existing approaches. The overall picture is completed in Section 7 by a brief description of the remaining systems, therefore covering the entire state of the art.

This work would not be complete without a mapping of the programming approaches being surveyed onto the taxonomy proposed. This is presented in Section 8. Moreover, in the same section we also map existing programming approaches onto the application taxonomy we described in Section 2. As a result, the reader gains not only a complete classification of the systems in the current state of the art, but also a tool to understand which approach is best suited for a given application. We believe that these two perspectives—features and applicability of programming approaches—together constitute an asset for both researchers and practitioners. The global view on the state of the art is also the opportunity to draw general observations about the field, and identify themes worth addressing by the research community. These aspects are discussed in Section 9, which also ends the paper with brief concluding remarks.

We are not the first to undertake a survey of programming approaches for WSNs [Sugihara and Gupta 2008; Hadim and Mohamed 2006; Römer 2004; Rubio et al. 2007; Chatzigiannakis et al. 2007; Henricksen and Robinson 2006]. However, most of the existing surveys are based on a taxonomy with only few dimensions, mostly revolving around the well-known duality between *node-centric* programming and *macroprogramming* noted by many authors [Newton et al. 2007; Gummadi et al. 2005; Bakshi et al. 2005]. Here, instead, we present a taxonomy that subsumes such distinction, and provides a more in-depth analysis through a richer set of dimensions. Other distinctive traits of our survey are the concrete illustration through code examples, the distinction between language and architectural issues, the complementary view on application requirements, and the exhaustive coverage and mapping of the state of the art.

2. WIRELESS SENSOR NETWORK APPLICATIONS

WSNs are being employed in a variety of scenarios. Such diversity translates into different requirements and, in turn, different programming constructs supporting them. In this section we identify some common traits of WSN applications that strongly affect the design of programming approaches, and cast these aspects in a dedicated taxonomy. Figure 1 graphically illustrates the dimensions we identified.

Goal. In the applications that made WSNs popular (e.g., [Mainwaring et al. 2002]), the goal is to gather environmental data for later, off-line analysis. Figure 2(a) illustrates the network architecture traditionally employed to accomplish this functionality. A network of sensor-equipped nodes funnels their readings, pos-

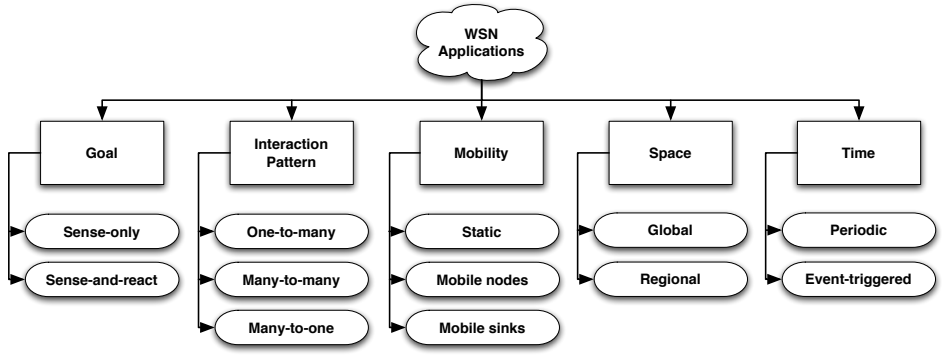
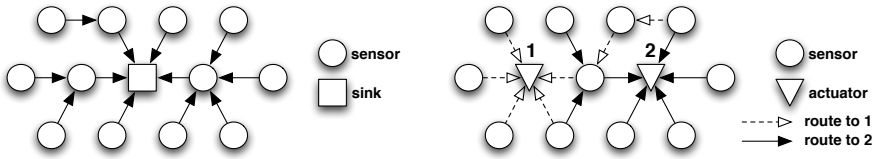


Fig. 1. A taxonomy of WSN applications.



(a) Sense-only. (b) Sense-and-react.

Fig. 2. Network architecture in sense-only and sense-and-react applications.

sibly along multiple hops, to a single base station—typically much more powerful than a WSN node—that acts as data sink by centrally collecting the data.

Along with *sense-only* scenarios, a new breed of applications emerged where WSN nodes are equipped with actuators. In wireless sensor and actuator networks (WSANs) [Akyildiz and Kasimoglu 2004], nodes can react to sensed data, therefore closing the control loop. The resulting *sense-and-react* pattern drastically affects the application scenario. Indeed, in principle the data sensed can still be reported to a single sink that hosts also the control logic and issues the appropriate commands to the actuators. However, to reduce latency and energy consumption, and to increase reliability by removing the single point of failure, it is advisable to move the application and control logic inside the network [Akyildiz and Kasimoglu 2004]. This results in a radically different network architecture, illustrated in Figure 2(b), where sensor nodes need to report to multiple receivers. The system becomes heterogeneous, in contrast with the mostly homogeneous architectures employed in sense-only scenarios. Moreover, the application behavior also changes. Applications tends to be stateful, i.e., determined by the current conditions and past evolution of the system, in contrast with the mostly stateless behavior of sense-only applications. Also, multiple activities must be carried out simultaneously, e.g., to control actuators installed in different parts of the system as in Heating, Ventilation, and Air-Conditioning (HVAC) systems in buildings [Deshpande et al. 2005].

Interaction pattern. Another fundamental distinction is in how the network nodes interact with each other, which is somehow affected also by the application goal they are to accomplish. To date, sense-only WSNs mostly feature a *many-to-one* interaction pattern, where data is funneled from all nodes in the network to a central collection point. Nevertheless, *one-to-many* and *many-to-many* interactions can also be found. The former are important when it is necessary to send configuration commands (e.g., a change in the sampling frequency or in the set of sensors active) to the nodes in the network. The latter is typical of scenarios where multiple data sinks are present, a situation commonly found in sense-and-react scenarios.

Mobility. Wireless sensor networks are characterized by highly dynamic topologies, induced by fluctuations in connectivity typical of wireless propagation and by duty-cycle patterns necessary to extend the network lifetime. However, some applications introduce an even greater degree of dynamism, due to the need to support physically *mobile* devices.

Mobility may (or may not) manifest itself in different ways:

—In *static* applications, neither nodes nor sinks move once deployed. This is by far the most common case in current deployments.

—Some applications use *mobile nodes* attached to mobile entities (e.g., robots or animals) or able to move autonomously (e.g., the XYZ nodes [Lymberopoulos and Savvides 2005]). A typical case is wildlife monitoring where sensors are attached to animals, as in the ZebraNet project [Liu and Martonosi 2003].

—Some applications exploit *mobile sinks*. The nodes may be indifferently static or mobile: the key aspect is that data collection is performed opportunistically when the sink moves in proximity of the sensors [Shah et al. 2003].

Space and time. The distributed processing required by a given application may span different portions of the physical *space*, and be triggered at different instants in *time*. These aspects are typically determined by the phenomena being monitored.

The extent of distributed processing in space can be:

—*Global*, in applications where the processing in principle involves the whole network, most likely because the phenomena of interest span the entire geographical area where the WSN is deployed.

—*Regional*, in applications where the majority of the processing occurs only within some limited area of interest.

For what concerns time, distributed processing can be:

—*Periodic*, in applications designed to continuously process sensed data. The application performs periodic tasks to gather sensor readings, coordinates with other parts of the system, and possibly performs actuation as needed.

—*Event-triggered*, in applications characterized by two phases: *i)* during event detection, the system is largely quiescent, with each node monitoring the values it samples from the environment with little or no communication involved; *ii)* if and when the event condition is met (e.g., a sensor value raises above a threshold), the WSN begins its distributed processing.

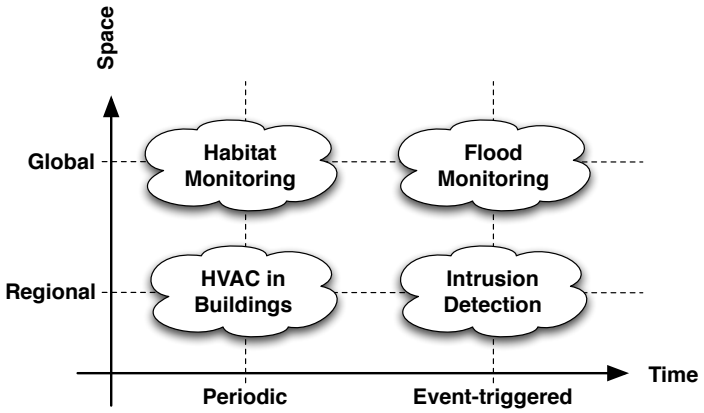


Fig. 3. Space and time characteristics of the distributed processing in example WSN applications.

Note that, in accordance with the goal of the paper, our focus here is on the *distributed processing* required to enable a functionality, not on the functionality itself. Consider an application required to trigger an alarm whenever a condition is met. If the condition is checked at the sink by periodically collecting data, such application would fall in the Periodic class, not in the Event-triggered one.

Interestingly, space and time are orthogonal, and existing WSN applications cover all combinations of these two dimensions. Figure 3 illustrates the concept using paradigmatic examples drawn from the literature. For instance, habitat monitoring [Mainwaring et al. 2002] is an application where the distributed processing is typically global and periodic. Building automation (HVAC) [Deshpande et al. 2005], instead, exemplifies applications with periodic processing that, when implemented in a decentralized fashion, limit their operation to a specific portion of space (e.g., an air conditioner in a room that operates based on the readings of nearby temperature sensors). Likewise, in applications with event-triggered processing, the triggered functionality may be either global or regional. Flood monitoring [Hughes et al. 2007; IST CRUISE Project] falls in the first class, as the processing occurring after a flood is detected still spans the entire WSN. In this scenario, application domain experts are indeed interested in understanding how the flood may affect areas where it has not reached yet. In intrusion detection [Arora et al. 2004], instead, after a potential breach is detected, the system operates only within its surroundings, as data coming from global observations are no longer relevant.

A more extensive classification is shown in Table I, which maps a representative set of applications found in the literature to the taxonomy illustrated in Figure 1. Although the mapping is not exhaustive, several observation can be drawn:

—Sense-only applications are mostly characterized by many-to-one interactions. In the few ones requiring many-to-many interactions, this is due to the need to support data access from multiple users at different locations.

—The space and time characteristics of the processing in sense-only applications covers all combinations. Applications periodically gathering data on a global

Application	Goal	Interaction	Mobility	Space	Time
Habitat Monitoring [Mainwaring et al. 2002; Buonadonna et al. 2005]	SO	Many-to-one	Static	Global	Periodic
Zebra Monitoring [Juang et al. 2002]	SO	Many-to-one	Mobile nodes	Global	Periodic
Glacier Monitoring [Martinez et al. 2004; Padhy et al. 2006]	SO	Many-to-one	Static	Global	Periodic
Grape Monitoring [Burrell et al. 2004]	SO	Many-to-one	Static	Global	Periodic
Landslide Detection [Sheth et al. 2005]	SO	Many-to-one	Static	Global	Periodic
Volcano Monitoring [Werner-Allen et al. 2006]	SO	Many-to-one	Static	Global	Periodic
Passive Structural Monitoring [Lynch and Loh 2006; Ceriotti et al. 2009]	SO	Many-to-one	Static	Global	Periodic
Fence Monitoring [Wittenburg et al. 2007]	SO	Many-to-one	Static	Regional	Event-triggered
Industrial Plant Monitoring [Krishnamurthy et al. 2005]	SO	Many-to-one	Static	Global	Periodic
Sniper Localization [Simon et al. 2004]	SO	Many-to-one	Static	Regional	Event-triggered
Intrusion Detection [Arora et al. 2004]	SO	Many-to-one	Static	Regional	Event-triggered
Forest Fire Detection [Hartung et al. 2006]	SO	Many-to-one	Static	Global	Event-triggered
Flood Detection [IST CRUISE Project ; Hughes et al. 2007]	SO	Many-to-one	Static	Global	Event-triggered
Health Emergency Response [Lorincz et al. 2004]	SO	Many-to-one	Static	Regional	Periodic
Avalanche Victims Rescue [Michahelles et al. 2003]	SO	Many-to-many	Static	Regional	Periodic
Smart Tool Box [Lampe and Strassner 2003]	SO	Many-to-many	Static	Global	Event-triggered
Vital Sign Monitoring [Baldus et al. 2004]	SO	Many-to-many	Static	Global	Event-triggered
Robot Navigation [Batalin et al. 2004]	SO	Many-to-one	Mobile sinks	Regional	Event-triggered
Badger Monitoring [WildSensing Project]	SO	Many-to-one	Mobile nodes	Global	Periodic
Sheep Monitoring [WASP Project]	SO	Many-to-many	Mobile nodes	Global	Periodic
Electronic Shepherd [Thorstensen et al. 2004]	SO	Many-to-many	Mobile nodes	Global	Periodic
Vehicular Traffic Control [Manzie et al. 2005]	SR	Many-to-many	Static	Regional	Periodic
Smart Homes [Petriu et al. 2000]	SR	Many-to-many	Static	Regional	Periodic
Assisted Living [Stankovic et al. 2005]	SR	Many-to-one/ One-to-many	Static	Regional	Periodic
Building Control and Monitoring [Dermibas 2005]	SR	Many-to-one/ One-to-many	Static	Regional	Periodic
Active Structural Monitoring [Lynch and Loh 2006]	SR	Many-to-many	Static	Regional	Periodic
Heating Ventilation and Air Conditioning Control [Deshpande et al. 2005]	SR	Many-to-many/ One-to-many	Static	Regional	Periodic
Tunnel Control and Monitoring [Costa et al. 2007]	SR	Many-to-many/ One-to-many	Static	Regional	Periodic

Table I. Mapping example WSN applications onto the taxonomy of Figure 1.

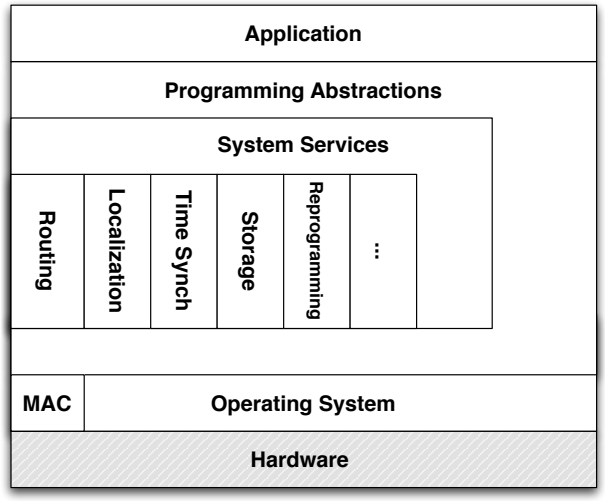


Fig. 4. Reference architecture.

scale are the most frequent.

—In contrast, sense-and-react applications are typically characterized by periodic and regional processing. The enforcement of control laws requires continuous monitoring of the environment, approximated through periodic sampling. Moreover, actuators are limited in the extent to which they can influence the environment, and therefore they usually do not require to gather sensor readings outside their range of actuation [Akyildiz and Kasimoglu 2004].

Before moving to the main contribution of this paper, the taxonomy of WSN programming approaches, we must clearly define its scope. We do so by relying on a reference architecture, described next.

3. REFERENCE ARCHITECTURE

The boundaries between programming abstractions and the rest of the software executing on a WSN node is often blurred. The scarce computing and communication resources available in WSNs, along with their application-specific nature, foster a cross-layer design where the application is often intertwined with system-level services. In addition, programming abstractions are intimately related with a number of other issues in WSNs. These include application and services (e.g., routing) built on top of the abstractions, down to the hardware and operating system the abstractions are built upon.

To help delimit clearly what is—and especially what is *not*—in the scope of our work, we introduce here a reference architecture, shown[1] in Figure 4. In the following we describe each of its constituents, thus establishing a context for our taxonomy of WSN programming approaches.

[1]The layering shown is purely conceptual, and does not necessarily reflect the code structure of actual systems, which often break layers to achieve better resource utilization.

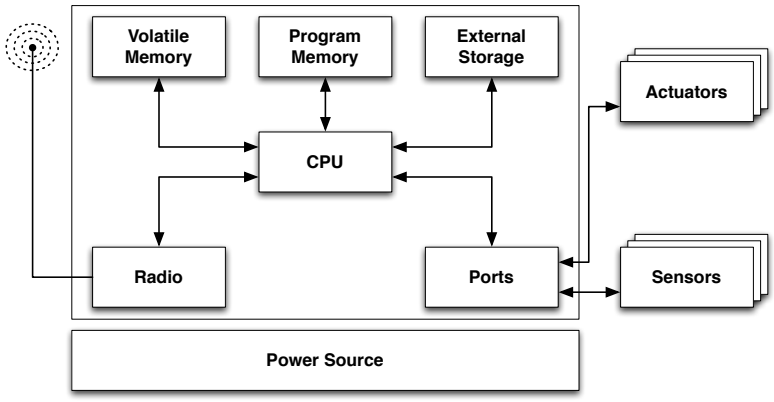


Fig. 5. A high-level schematic representation of a WSN node hardware.

Hardware. Figure 5 illustrates a very abstract view of the hardware in a typical WSN node. A plethora of WSN platforms exist both as commercial products and research prototypes [Crossbow Tech. ; MoteIV ; Body Sensor Network Nodes ; BTNode ; Eyes WSN Nodes ; Project SunSPOT ; MeshNetics Tech. ; ScatterWeb Inc. ; Aduino Sensor Node Platform]. However, the individual components used do not differ drastically. Many platforms use a 16-bit Texas Instruments MSP430 micro-controller or a 8/16-bit chip of the Atmel ATMega family. Notable exceptions are the IMote2 and SunSPOT platforms, based on the more powerful Intel PXA and ARM920T chips, respectively. Typical amounts of volatile memory range from 2 KB to 512 KB. This is used to store run-time data during program execution. The binary program code is stored in a dedicated memory whose size is typically between 32 KB and 128 KB. In addition, nodes are often equipped with separate, external storage devices (e.g., flash memory) whose size may vary from 128 KB to several gigabytes. Their use depends on the specific application. As for radio hardware, most platforms work in the 2.4 GHz ISM band, and feature IEEE 802.15.4-compliant [Baronti et al. 2007] radio chips, e.g., the ChipCon 2420. Alternative solutions operate in the 868/916 MHz band, e.g., using the ChipCon 1000 transceiver, or rely on Bluetooth interfaces. The specific type of sensing and actuating device is largely application-specific, and often custom-integrated.

Medium Access Control (MAC). MAC protocols for WSNs must guarantee efficient access to the communication media while carefully managing the energy budget allotted to the node. The latter goal is typically achieved by switching the radio to a low-power mode based on the current transmission schedule. In contrast to other wireless platforms where the MAC functionality is realized in hardware, a WSN MAC protocol is typically implemented mostly in software, using the low-level language associated with the operating system.

Most of the existing protocols fall in two categories. *Contention-based* protocols [Ye et al. 2002; Polastre et al. 2004; van Dam and Langendoen 2003] regulate the access to the physical layer opportunistically, based on the current transmission requests. Conversely, *time-slotted* protocols assign the nodes with predefined time-

slots to schedule their transmissions over time [Rajendran et al. 2003; 2006]. The former class of protocols is easier to implement and better tolerates nodes joining or leaving. Instead, the latter enables higher reliability and greater energy savings, but with the additional requirement of tight time synchronization among the nodes in some k-hop neighborhood.

A survey of the many MAC protocols available can be found in [Demirkol et al. 2006; Naik and Sivalingam 2004].

Operating system. In contrast to mainstream computing, in WSNs the operating system is essentially a library, linked with the application code to produce a binary for execution. The operating system usually supports a companion programming language, which is typically C or a WSN-specific dialect (e.g., nesC [Gay et al. 2003] for the TinyOS [Hill et al. 2000] operating system). A low-level communication facility is also commonly provided, e.g., the Active Message [Culler et al. 2001] interface of TinyOS. Such companion language and communication primitives define the lowest level abstraction available to programmers. In a sense, they resemble the use of the C language and sockets in mainstream computing as the core programming abstractions provided by the operating system.

Several operating systems for WSNs have been proposed so far, the most common being the aforementioned TinyOS. Alternatives include Contiki [Dunkels et al. 2004], SOS [Han et al. 2005], Mantis [Abrach et al. 2003], RETOS [Cha et al. 2007], LiteOS [Cao et al. 2008], t-Kernel [Gu and Stankovic 2006], and NANO-rk [Eswaran et al. 2005]. The concurrency model employed varies from event-driven approaches [Hill et al. 2000] to preemptive, time-sliced multi-threading [Abrach et al. 2003; Cha et al. 2007; Cao et al. 2008], cooperative multi-threading [Dunkels et al. 2006], and asynchronous message passing [Han et al. 2005]. Some of the above operating systems (e.g., SOS, LiteOS, and Contiki) also provide dynamic linking capabilities, i.e., new code modules can be added at run-time to the application running on a node. Dynamic linking is particularly important in supporting wireless reprogramming of the WSN, one of the system services described next.

System services. While applications deliver useful data directly to the end user, system services are typically useful in support of applications. Examples are *localization* mechanisms [Langendoen and Reijers 2003], *time synchronization* protocols [Elson and Roemer 2003; Sundararaman et al. 2005], distributed *storage* services [Ratnasamy et al. 2002; Luo et al. 2007], *code deployment* and *reprogramming* functionality [Wang et al. 2006], and *routing* protocols [Al-Karaki and Kamal 2004]. Notably, some approaches in routing play at border between system services and programming abstractions. For instance, in Directed Diffusion [Intanagonwiwat et al. 2003] programmers specify the characteristics of the data required using attribute-value pairs. The emphasis of these approaches, however, is mostly on routing and communication issues. As a consequence, unlike the systems surveyed in the rest of the paper, they feature only very limited expressiveness as they do not provide a well-defined, structured programming abstraction.

System services are built atop the core functionality provided by the operating system, by using either the operating system language (e.g., nesC) or some of the programming abstractions we discuss in this paper. For instance, localization and

routing have been implemented successfully in Hood [Whitehouse et al. 2004]. In our survey, we distinguish between programming approaches suitable also to the development of system services, and those geared only towards applications.

4. TAXONOMY OVERVIEW

The focus of our work is on high-level language constructs allowing programmers to express various forms of distributed processing among the WSN nodes.

In this field, the only characterizing dimension that hitherto received some attention is the one of *node-centric* programming vs. *macroprogramming* [Gummadi et al. 2005]. The former generally refers to programming abstractions used to express the application processing from the point of view of the individual nodes. The overall system behavior must therefore be described in terms of pairwise interactions between nodes within radio range. Macroprogramming solutions, instead, are usually characterized by higher-level abstractions that focus mainly on the behavior of the entire network, rather than on the individual nodes.

Nonetheless, under many respects the above distinction falls short of expectation in capturing the essence of currently available programming approaches. As a result, solutions offering radically different abstraction levels are considered under the same umbrella, ultimately rendering the distinction ineffective. For instance, both TinyDB [Madden et al. 2005] and Kairos [Gummadi et al. 2005] are commonly regarded as macroprogramming solutions. However, the former provides an SQL-like interface where the entire network is abstracted as a relational table. Therefore, inter-node interactions are completely hidden from the programmer. The latter, on the other hand, is an imperative programming language where constructs are provided to iterate through the neighbors of a given node and communication occurs by reading or writing shared variables at specific nodes. Therefore, unlike TinyDB, in Kairos the application processing is still mostly expressed as pairwise interactions between neighboring nodes, and yet the level of abstraction is very different from node-centric programming approaches.

These considerations have been our motivation for defining a taxonomy of programming approaches that goes beyond the traditional dichotomy between node-centric and macroprogramming, and examines a wider set of concepts. Our taxonomy is structured along two main dimensions, each contained in a separate section of this paper:

—In Section 5, we study the *language aspects* of available WSN programming approaches. These are analyzed to understand the primitives provided to programmers for expressing communication and computation, and the peculiarities of the programming model.

—In Section 6, we consider the *architectural aspects* related with existing WSN programming solutions, by analyzing features such as their intended use, their reach into the low-level layers of the architecture, and their execution environment.

Our objective is to provide the reader with an understanding of the expressive power of the various approaches in the first part, while in the second part we intend to explore how these approaches can be used in application development, and what is their relationship with the rest of the architecture depicted in Figure 4.

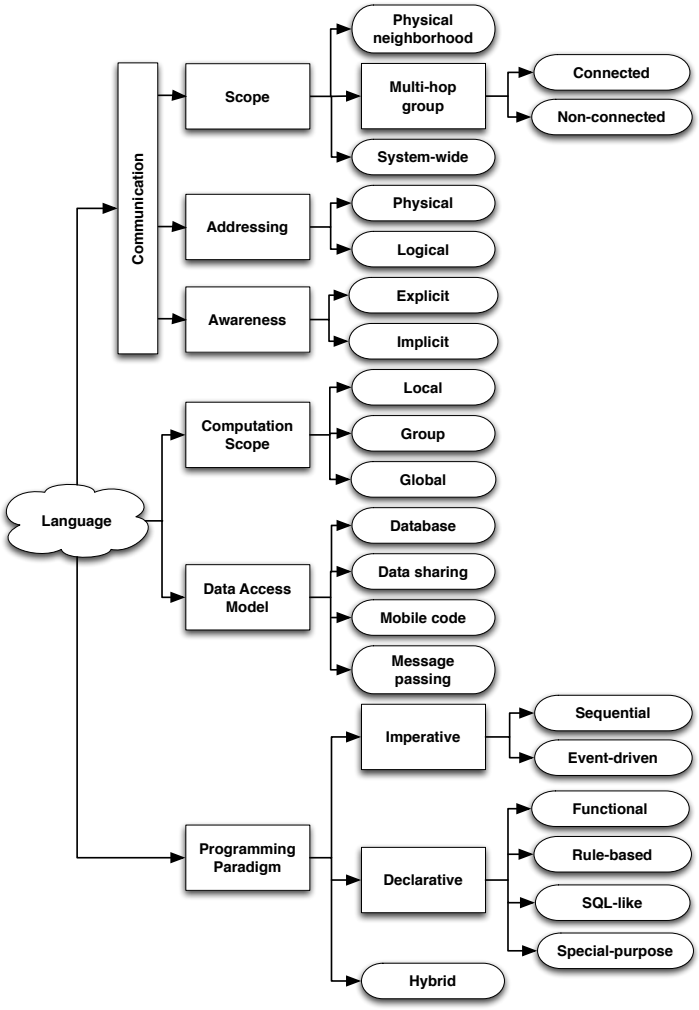


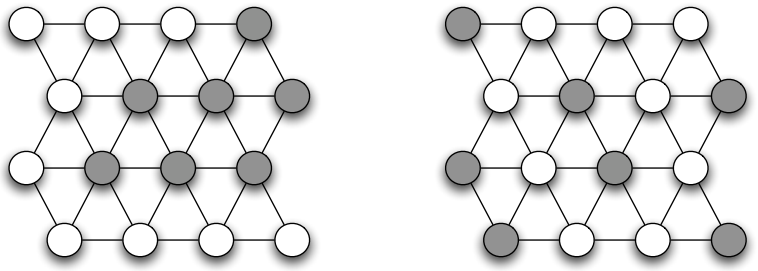
Fig. 6. A taxonomy of language aspects in WSN programming abstractions.

For each dimension of classification, we illustrate its meaning first in abstract terms, and then by focusing on a representative approach taken from the state of the art. The style of presentation is made concrete by relying on code fragments and by concisely reporting key implementation details.

5. PROGRAMMING WIRELESS SENSOR NETWORKS: LANGUAGE ASPECTS

Figure 6 provides an overview of the language dimensions in our taxonomy. We classify the various approaches based on the constructs that allow to express *communication* and *computation*, on how these are framed into a *data access model*, and on the more traditional dimension related to the *programming paradigm* adopted.

The communication dimension is particularly important. In most applications,



(a) Multi-hop, connected group. (b) Multi-hop, non-connected group.

Fig. 7. Topological characteristics of group based communication. Grey nodes are group members.

WSN nodes can hardly perform any useful task if left alone; it is the overall collaboration and coordination of numerous devices that allows the system to accomplish a higher-level goal. As shown in Figure 6, we distinguish further among aspects related to the scope of communication, the type of addressing used, and the extent to which the programmer is aware of communication.

5.1 Communication → Scope

We define the *scope* of communication as *the set of nodes that exchange data to accomplish a given application processing.*

Classification. Three approaches emerge in the current state of the art:

—**Physical neighborhood**: programmers are provided with constructs that allow data exchange only among nodes within direct radio range.

—**Multi-hop group**: data exchange is enabled among a subset of nodes across multiple hops. Two sub-cases can be identified based on the connectivity among the nodes in the group:
  —**Connected**: the nodes exchanging data may be multiple hops away from each other, yet any two nodes in the group are connected via nodes that are also part of the group. An example is depicted in Figure 7(a).
  —**Non-connected**: no assumption is made on the location of nodes belonging to the group, as in Figure 7(b).

—**System-wide**: all the nodes in the WSN are possibly involved in some data exchange.

As an example of a system where communication is restricted to the physical neighborhood, we illustrate Active Messages [Culler et al. 2001] and the companion language nesC [Gay et al. 2003]. As for communication within a multi-hop group, we study EnviroSuite [Luo et al. 2006] for the connected case, and Logical Neighborhoods [Mottola and Picco 2006a; 2006b] for the non-connected one. Finally, we illustrate TinyDB [Madden et al. 2005] as an example of system-wide communication.

5.1.1 *Physical neighborhood: Active Messages and nesC.*

```
1 interface AMSend {
2   command error_t send(am_addr_t addr, message_t* msg, uint8_t len);
3   command error_t cancel(message_t* msg);
4   event void sendDone(message_t* msg, error_t error);
5   command uint8_t maxPayloadLength();
6   command void* getPayload(message_t* msg, uint8_t len);
7 }
```

Fig. 8. nesC Active Message interface.

Overview. Active Messages is a set of interfaces providing basic communication primitives in the nesC programming language. This is an event-driven programming language for WSNs derived from C, whose goal is to provide programming support for the TinyOS operating system. Applications are built in nesC by interconnecting *components* that interact by *providing* or *using* interfaces. An interface lists one or more functions, tagged as *commands* or *events*. Commands are used to start operations, while events are used to collect the results asynchronously. A component providing an interface implements the commands it declares, whereas the one using the interface implements its events. Therefore, data may flow both ways between components connected through the same interface.

In Active Messages, messages are tagged with an identifier that specifies which component must process them upon reception. Components use Active Messages through nesC interfaces. An example is shown in Figure 8. Additional interfaces are provided for low-level configuration (e.g., to set the transmission power level). Although higher-level communication abstractions are available atop nesC [Levis et al. 2004], they all rely on Active Messages. In a sense, Active Messages play a role similar to sockets in mainstream distributed computing, by providing a basic building block enabling the development of higher-level functionality.

Example. Figure 9 shows a code fragment implementing a component that queries the sensing device and sends the reading in broadcast. The `booted` event in the `Boot` interface is signalled at system start-up. Inside the event handler (line 11-13), the component calls the `read` command (line 12) in the `TemperatureSensor` interface, whose providing component is bound to the sensing device. This is a typical *split-phase* operation [Gay et al. 2003]: the command returns immediately and the caller is asynchronously notified when the device completes its operation, in our case using the `readDone` event (line 15). In the corresponding event handler, the sensed value is packed in a message and the component calls the `AMSend.send` command. To make sure the component does not try to send another message while an earlier transmission is in progress, a `transmitLock` flag is set just before calling the `AMSend.send()` command (line 19). The flag is unset inside `AMSend.sendDone`, which is asynchronously called when the transmission completes (line 26-32). As this example shows, the level of abstraction provided is quite low. Programmers are forced to deal directly with message parsing and serialization as well as scheduling transmissions. In addition, although the nesC Sensor APIs [TinyOS Community Forum a] provide support for sensing, no dedicated abstractions are offered to control externally attached devices, e.g., actuators.

```
1  module Sampler {
2    uses interface Boot;
3    uses interface TemperatureSensor;
4    uses interface AMSend;
5  }
6
7  implementation {
8    bool transmitLock;
9    message_t msgBuffer;
10
11   event void Boot.booted {
12     call TemperatureSensor.read();
13   }
14
15   event void TemperatureSensor.readDone(uint16_t v){
16     uint16_t* msg_payload = (uint16_t*) call AMSend.getPayload(msgBuffer);
17     *msg_payload = v;
18     if (!transmitLock) {
19       transmitLock = TRUE;
20       if (!call AMSend.send(TOS_BCAST_ADDR, &msgBuffer, sizeof(message_t))) {
21         transmitLock = FALSE;
22       }
23     }
24   }
25
26   event void AMSend.sendDone(message_t* msg, result_t success) {
27     if(transmitLock && msg == msgBuffer ) {
28       transmitLock = FALSE;
29     } else {
30       // Error...
31     }
32   }
33 }
```

Fig. 9. Sense and broadcast component in nesC using Active Messages.

Implementation highlights. The mechanisms implementing the Active Message interfaces are normally bound to the specific MAC-level mechanisms employed, or directly to the radio hardware. As a result, most of them are platform-specific. Generally, the implementations provide (unreliable) 1-hop unicast or broadcast transmissions. Specific solutions, nonetheless, can offer some form of reliability when coupled with specific radio chips [Polastre et al. 2004; TinyOS Community Forum d]. Moreover, there is essentially no support for packet buffering, and the application must provide its own storage for sending and receiving messages. To overcome these limitations, multi-hop protocols for data collection and dissemination have been developed atop the Active Message interface [TinyOS Community Forum b; c].

5.1.2 *Multi-hop group → Connected: EnviroSuite.*

Overview. EnviroSuite is an object-based programming framework aimed at monitoring and tracking applications. In EnviroSuite, objects represent physical entities in the environment. Object instances are dynamically created when the corresponding physical entities are detected, and automatically destroyed when the same entities move out of sensing range. A one-to-one mapping between objects and physical entities is maintained as the latter move in the environment. The framework provides constructs to specify the conditions for object creation, the object attributes describing the state of the corresponding physical entity, and the logic to update

```
1 object VEHICLE {
2   object condition = ferrous_object() && vehicle_sound();
3   object_attribute location {
4     attribute_value = AVERAGE(position());
5     attribute_degree = 2;
6     attribute_freshness = 500 ms;
7   }
8   object_main_function = Vehicle.getLocation;
9 }
```

Fig. 10. Vehicle tracking in EnviroSuite.

these attributes based on sensor data. The set of nodes maintaining an object instance is assumed to be a connected region around the environmental phenomena at hand. A remote procedure call mechanism allowing for inter-object interactions is also included [Blum et al. 2003].

Example. Consider an application to track moving vehicles using magnetometers and acoustic sensors. The exact vehicle position is computed by averaging the position estimates reported by a minimum number of sensor nodes.

Figure 10 reports a fragment of the corresponding implementation in EnviroSuite, adapted from [Luo et al. 2006]. The program defines a `VEHICLE` object whose creation occurs when sensors detect a ferrous object coupled with the sound signature of a vehicle (line 2). The object exports a single attribute named `location`. Its value is derived by aggregating the position estimates of at least 2 nodes, updated every 500 ms (line 3-7). The `object_main_function` (line 8) indicates where to find the nesC code implementing the main object method. In this case, the statement points to a command `getLocation` in interface `Vehicle`, where the programmer can specify dedicated macros to send data to the base station or invoke methods on other, possibly remote, objects.

Implementation highlights. EnviroSuite object definitions are fed as input to a dedicated pre-processor that generates plain nesC code. The framework provides a library of sensor data processing algorithms to define conditions for object creation. Based on the object definition at hand, the pre-processor identifies the most appropriate protocol to manage object creation and destruction. Available choices include a protocol, based on routing trees, to maintain objects bound to a fixed set of nodes, and a scheme to deal with objects associated to moving entities. The latter features mechanisms to maintain the mapping between the objects and the environmental phenomena as these move in space. A leader is elected in the connected region of nodes sensing the moving target, which collects data from other nodes in the same region and performs the necessary computation.

5.1.3 *Multi-hop group → Non-connected: Logical Neighborhoods.*

Overview. Logical Neighborhoods is a programming abstraction that allows programmers to redefine a node's neighborhood based on the logical properties of the nodes in the network, regardless of their physical position. Neighborhoods are defined using a declarative programming language called Spidey, conceived as an extension of existing WSN languages. Programmers interact with the nodes in a logical neighborhood using an API that mimics the traditional broadcast-based

```
1  node template Device
2    static Function
3    static Type
4    static Location
5    dynamic BatteryPower
6
7  create node tl from Device
8    Function as "actuator"
9    Type as "traffic_light"
10   Location as "entrance_east"
11   BatteryPower as getBatteryPower()
```

Fig. 11. Logical Neighborhoods: node definition and instantiation for an actuator node.

```
1  neighborhood template TrafficLights(loc)
2    with Function = "actuator" and
3         Type = "traffic_light" and
4         Location = loc
5
6  create neighborhood tl_east
7    from TrafficLights(loc: "entrance_east")
8    max hops 2 credits 30
```

Fig. 12. Logical Neighborhoods: neighborhood definition and instantiation in road tunnel monitoring.

communication. Instead of the nodes within radio range, however, the message recipients are the nodes matching a given neighborhood definition. Therefore, programmers still reason in terms of neighboring relations, but retain control over how these are established. Logical Neighborhoods is suited to the highly heterogeneous and decentralized scenarios typical of sense-and-react applications, where the processing often revolves around programmer-defined subsets of nodes.

Example. The definition of logical neighborhoods is based on two concepts: *nodes* and *neighborhoods*. Nodes represent the portion of a real node's features made available to the definition of any logical neighborhood. Their definition is encoded in a *node template*, which specifies a node's exported attributes. This is used to derive instances of logical nodes, by specifying the actual source of data. Figure 11 reports a fragment of Spidey code that defines a template for a generic actuator (line 1-5), and instantiates a logical node controlling a traffic light (line 7-11).

A logical neighborhood is defined using predicates over node templates. Analogously to nodes, a neighborhood is first defined in a template, which essentially represents the membership function for the node subset targeted by the neighborhood. The neighborhood template is then instantiated by specifying where and how it is evaluated. For instance, Figure 12 illustrates the definition of a neighborhood that includes the nodes controlling the traffic lights on a specific tunnel entrance (line 1-4). The template is instantiated so that it evaluates only on nodes at most 2 (physical) hops away from the one node defining the neighborhood, and by spending a maximum of 30 "credits" (line 6-8). The latter is an application-defined notion of communication costs, which allows programmers to affect the trade-off between accuracy and resource consumption [Mottola and Picco 2006b].

```
1  SELECT AVG(light), AVG(temp), location
2  FROM sensors
3  SAMPLE PERIOD 2 s FOR 30 s
```

Fig. 13. Monitoring bird nests using TinyDB.

Implementation highlights. Logical Neighborhoods is available for both TinyOS and Contiki. A Java version is also available [Mottola et al. 2007]. Spidey definitions are input to a dedicated pre-processor generating custom code for the platform at hand. An efficient routing mechanism enables communication in a logical neighborhood. Nodes periodically disseminate their profile, i.e., the list of current attribute-value pairs. To avoid flooding the entire system, the protocol exploits the redundancy among similar profiles to limit the spreading of information. Application messages contain an encoding of the target logical neighborhood. Based on the attributes it contains, a message follows the routes established by the disseminated profiles back to the target nodes.

5.1.4 *System-wide: TinyDB.*

Overview. TinyDB, similarly to its predecessor TAG [Madden et al. 2003], is a query processing system for WSNs whose focus is to optimize energy consumption by controlling where, when, and how often data is sampled. In TinyDB, the user submits SQL-like queries at the base station. These are parsed, optimized depending on the data requested, and injected into the network. Upon reception of a query, a node processes the corresponding requests, gathers some readings if needed, and funnels the results back to the base station. The data model revolves around a single `sensors` table that logically contains one row per node per instant in time, and one column for each possible data type the node can produce (e.g., temperature or light). The data in this table is materialized only on request. Alternatively, *materialization points* can be created in the network to proactively gather and process the data. Data collection applications are easily expressed using TinyDB, as the declarative nature of the database abstraction helps programmers in focusing on the data to retrieve without specifying how to do so.

Example. Consider an application to monitor the presence of birds in nests, where the average light and temperature close to a nest must be gathered every 2 seconds for a total of 30 seconds. This processing can be encoded in a TinyDB query as illustrated in Figure 13, adapted from [Madden et al. 2005]. The `SELECT`, `FROM` and `WHERE` clauses have the same semantics as in standard SQL. The `location` attribute is assumed to be obtained from some external localization mechanism. The `SAMPLE PERIOD` construct is used for specifying the rate and lifetime of the query. The example shows how TinyDB, for this specific kind of application/functionality, enables a very compact encoding of the desired behavior.

Implementation highlights. When the query is injected from the base station, a routing tree is built spanning all the nodes in the network. The routes are then decorated with meta-data to provide information on the type and nature of data sensed by nodes in a specific portion of the tree. While executing the query at each node, TinyDB performs several optimizations to reduce the amount of data

flowing towards the base station. For instance, data sampling and transmissions are interleaved to minimize power consumption without affecting the quality of the data reported. A dedicated transmission scheme is also employed to schedule the transmissions at different levels of the tree. The goal is to make data flow upward starting from the leaves, so that intermediate nodes can aggregate information coming from other devices before sending their own.

5.2 Communication → Addressing

Orthogonal to the communication scope, existing solutions differ in *the way the nodes involved are identified*, i.e., the specific *addressing* scheme employed. The nature of the constructs used to determine the target nodes bears great impact on the ease in describing the application processing.

Classification. Existing programming frameworks essentially fall in either of the two classes of addressing:

—**Physical** addressing: the target nodes are identified using statically assigned identifiers. Most often, this is used in conjunction with unicast or broadcast communication within a 1-hop neighborhood.

—**Logical** addressing: the target nodes are identified through programmer-provided, application-level properties. For instance, the target nodes may be determined based on their type or current readings.

The Active Message communication stack we described in Section 5.1.1 is an example of the former type of addressing. Both the node identifier and the Active Message identifier that binds sender and receiver components are hard-wired in the code. The communication target in the `AMSend` interface of Figure 8 is either a broadcast identifier or the identifier of a specific node.

In contrast, the Logical Neighborhoods abstraction we illustrated in Section 5.1.3 features a logical addressing scheme. The communication target is determined by defining the properties characterizing the individual nodes, and by providing the property values selecting the desired nodes. Thus, the nodes involved may even change over time without modifying the definition of the neighborhoods themselves, unlike with static node addresses.

5.3 Communication → Awareness

Another facet of how communication is made available in WSN programming is the extent to which the programmer is *aware* of communication, i.e., whether communication is explicitly exposed to developers, or instead hidden behind some higher-level construct. In the former case, the functionality necessary to prepare messages for transmission and to parse them on reception rests mostly on the programmers' shoulders, often complicating the implementation of the application processing.

Classification. Based on the above consideration, we classify available solutions as providing:

—**Explicit** communication: where this functionality is directly in the hands of programmers, who are in charge of dealing with aspects such as message buffering,

```
1  // Discover region
2  result_t Region.formRegion(<region specific args>, int timeout);
3
4  // Wait for region discovery
5  result_t Region.sync(int timeout);
6
7  // Set and get shared variables
8  result_t SharedVar.put(sv_key_t key, sv_value_t val);
9  result_t SharedVar.get(sv_key_t key, addr_t node, sv_value_t *val, int timeout);
10
11 // Wait for shared variable gets
12 result_t SharedVar.sync(int timeout);
13
14 // Reduce 'value' to 'result' with given 'operator'
15 // 'yield' returns the percentage of nodes responding
16 result_t Reduce.reduceToOne(op_t operator, sv_key_t value,
17                             sv_key_t result, float *yield, int timeout);
18
19 // Reduce and set result in all nodes
20 result_t Reduce.reduceToAll(op_t operator, sv_key_t value,
21                             sv_key_t result, float *yield, int timeout);
22
23 // Wait for reductions to complete
24 result_t Reduce.sync(int timeout);
```

Fig. 14. The API of Abstract Regions.

serialization, and parsing. In addition, programmers may be required to schedule transmissions explicitly.

—**Implicit** communication: where it occurs through higher-level language constructs with no direct intervention from programmers, who cannot precisely perceive when and how data is exchanged among nodes. For instance, this is similar to remote procedure calls in traditional distributed computing.

An exemplary solution belonging to the former class is again Active Messages, described in Section 5.1.1. In this case, programmers are in charge of serializing and parsing data by accessing the various fields of a generic `message_t` data structure. Moreover, in the absence of buffering mechanisms, programmers must schedule transmissions directly. As an example of the latter category, here we illustrate the Abstract Regions [Welsh and Mainland 2004] programming framework.

5.3.1 *Implicit communication: Abstract Regions.*

Overview. Abstract Regions is a set of general-purpose programming primitives providing addressing, data sharing, and aggregation among a subset of nodes defined as a *region*. For instance, a region may include all nodes within a given distance from each other. Data sharing is accomplished using an associative array associated to the region, while dedicated constructs are provided to aggregate information stored at different nodes within a region. Although Abstract Regions are built atop nesC/TinyOS, they also employ a lightweight thread-like concurrency model called Fibers to provide blocking operations. The Abstract Regions API is depicted in Figure 14 [Welsh and Mainland 2004]. By their nature, Abstract Regions target applications exhibiting spatial locality, e.g., tracking moving objects or identifying the contours of a physical area [Liu et al. 2002].

```
1  location = get_location();
2
3  // Region setup to include 8 nearest neighbors
4  region = k_nearest_region.create(8);
5
6  while (true) {
7    reading = get_sensor_reading();
8
9    // Store data as shared variables
10   region.put(reading_key, reading);
11   region.put(reg_x_key, reading * location.x);
12   region.put(reg_y_key, reading * location.y);
13
14   if (reading > threshold) {
15      // Retrieve the id of the node with max reading
16      max_id = region.reduceToOne(OP_MAXID, reading_key);
17
18      // If this node is leader
19      if (max_id == my_id) {
20        // Compute centroid
21        sum = region.reduceToOne(OP_SUM, reading_key);
22        sum_x = region.reduceToOne(OP_SUM, reg_x_key);
23        sum_y = region.reduceToOne(OP_SUM, reg_y_key);
24        centroid.x = sum_x / sum;
25        centroid.y = sum_y / sum;
26        send_to_basestation(centroid);
27      }
28   }
29   sleep(periodic_delay);
30 }
```

Fig. 15. Object tracking in Abstract Regions.

Example. We illustrate a simple object tracking application developed using the API in Figure 14. The application takes periodic measures from sensor devices (e.g., magnetometers), and compares them against a threshold. Nodes sensing a value above the threshold coordinate to elect the node with the highest reading as the leader. The leader computes the centroid of all readings, and transmits the result back to a base station.

Figure 15 shows the code to implement the above object tracking application using Abstract Regions, adapted from [Welsh and Mainland 2004]. Initially, each node initializes the region to include the 8 geographically closest nodes (`k_nearest_region.create()` in line 4). In the main loop (line 6-30), each node queries the sensor and makes the output available to other nodes in the region, along with its physical location. This is achieved using different shared variables and `region.put()` to set their value. If the sensor reading is above the threshold, every node first determines the highest reading in the region by using `region.reduceToOne()` with operation `OP_MAXID` (line 16). If the local node is the one with the highest reading, sum-reductions are performed over the shared variables in the region to compute the centroid, and the result is sent to the base station.

Implementation highlights. Abstract Regions leverages nesC to produce executable code. The implementation of the mechanisms behind the Abstract Regions API depends on the particular region employed. For instance, the region used in the example is implemented using geographically-limited flooding. In contrast, a planar-mesh region used in a contour-finding application can be implemented

based on Yao graphs [Li et al. 2002]. In general, different regions require different implementations, which in turn may require a considerable effort.

5.4 Computation Scope

In WSNs, the duality between communication and computation plays an important role, e.g., for minimizing communication through local aggregation. The provision of language constructs that ease the description of the very application processing is therefore key to achieve efficient implementations. To address this need, WSN programming approaches provide a variety of language constructs. Besides the particular programming paradigm employed, discussed next, available solutions mainly differ w.r.t. the computation scope, i.e., *the set of nodes directly affected by the execution of a single instruction* in the program.

Classification. In the current state of the art, the computation scope offered to programmers is one of the following:

—**Local**: the effect of an instruction is limited to the node where it is executed.

—**Group**: an instruction can alter the state of some subset of nodes at once.

—**Global**: an instruction can possibly affect the state of all nodes in the system.

A local scope characterizes the computation in nesC, where all instructions have only a local effect. This includes those concerned with message passing, which indeed do not have a *direct* effect (e.g., a state change) on neighboring nodes. At the other extreme, the TinyDB system we previously described is a natural example of a global computation scope. Indeed, the processing triggered at the sink is perceived by the programmer as directly affecting the entire system. As for group computation, we use the Regiment system [Newton et al. 2007; Newton and Welsh 2004] as a concrete example.

5.4.1 *Group computation: Regiment.*

Overview. Regiment is a functional language geared towards applications exhibiting spatial locality, e.g., object tracking or intrusion detection. In Regiment, programmers manipulate sets of data streams called *signals*. These represent readings of individual nodes, the outcome of a node's local computation, or an aggregate value obtained by processing multiple input signals. Regiment also features a notion of *region* similar to Abstract Regions, e.g., a region may include the sensor readings generated by nodes in a limited geographic area. The processing is expressed by applying programmer-provided functions to signals in a region.

Example. Consider a system for early detection of plumes. Key to the correctness of the application is to avoid false positives due to noisy readings. Programmers are thus to make sure that the overall sum of the readings gathered by nodes around the phenomena exceeds a pre-specified threshold.

Figure 16 depicts an example Regiment program to implement the above processing, adapted from [Newton et al. 2007]. The program first defines a set of functions used in the rest of the program (line 1-3) to filter sensed data (`abovethreshold`), gather the reading from the sensor (`read`), or sum all signals in a region (`sum`). In the latter, `rfold` is used to aggregate all values in region `r` into a single signal,

```
1  fun abovethreshold(t) { t > CHEM_THRESHOLD }
2  fun read(n) { sense("concentration", n) }
3  fun sum(r) { rfold((+), 0, r) }
4
5  readings = rmap(read, world);
6  detects = rfilter(abovethreshold, readings);
7
8  hoods = rmap( fun(t, nd){ khood(1,nd) }, detects);
9  sums = rmap (sum, hoods);
10 base <- rfilter( fun(t){ t > CLUSTER_THRESHOLD }, sums);
```

Fig. 16. Plume monitoring in Regiment.

using the `+` operator and `0` as initial value. Next, the program identifies a region of nodes that exceeds the local threshold value. This is accomplished by first gathering the local readings at all nodes in the system, and then performing a filtering step. The former operation is expressed as the application of `read()` to all the nodes in the system, using `rmap` (line 5). This takes as input a function and a region, and applies the function to all values in the region. The `world` region in the example represents all nodes in the system. The filtering part is accomplished using `rfilter`, which takes a boolean function and a region as inputs, and returns the region that includes values for which the input function yields true (line 6).

In the example, `hoods` is instead a *nested* region. It consists of the nodes in the one-hop neighborhood of each node in the `detects` region (line 8). This is obtained by applying a region formation function (`khood`) to all nodes in `detects`. The remaining instructions are used to sum the readings in the nested regions created earlier, and to send a notification to the base station in case any of the sums turns out to be above the safety threshold.

Implementation highlights. The Regiment system relies on multiple steps of compilation to generate the final, node-level executable. A Regiment program is first translated into an intermediate language called RQuery. Subsequently, the region streams are translated into local streams. The output of the compiler is event-driven code written in an intermediate language called Token Machine Language [Newton et al. 2005]. This language does not assume a threaded concurrency model, and is therefore suited for implementation on top of event-driven WSN operating systems, such as TinyOS. As for communication, nodes in a given region exchange data using spanning trees. These are created and maintained by the Regiment run-time support on every node.

5.5 Data Access Model

Existing solutions provide different *abstractions to provide access to the data*. The specific data access model heavily influences the way programmers deal with both communication and computation, and therefore impacts significantly the development process.

Classification. Four approaches emerge in the current WSN literature:

—**Database**: the WSN is treated as a relational database and programmers pose SQL-like queries to access the information. Data is returned as a stream of records, possibly with no reference to the specific node that output the data.

—**Data sharing**: data is shared in the form of remotely accessible variables or tuples. Nodes can read or write data in the shared memory space using dedicated constructs.

—**Mobile code**: data is accessed locally to a node, by migrating the accessing code onto the node where data resides. Often, this is complemented by a data sharing scheme, although mostly for local coordination.

—**Message passing**: data is accessed through messages exchanged among the nodes involved.

The TinyDB system, described in Section 5.1.4, is an obvious representative of the first class. In TinyDB, sensed data are indeed made available as entries of a `sensors` table, and the user accesses the table using SQL-like queries. To cater with the peculiarities of WSN applications, however, further constructs are provided to express, for instance, the lifetime and period of queries.

To illustrate the remaining classes of data access models, here we present TeenyLime [Costa et al. 2007] for data sharing, Agilla [Fok et al. 2005] for mobile code, and DSWare [Li et al. 2004] for message passing.

5.5.1 *Data sharing: Teeny*Lime.

Overview. TeenyLime is based on the tuple space abstraction made popular by Linda [Gelernter 1985]. A tuple space is a shared memory space where different processes read/write data in the form of tuples. To blend with the asynchronous programming model of WSN operating systems such as TinyOS, however, in TeenyLime operations are non-blocking and return their results through a callback. Tuples are shared among nodes within radio range. In addition to Linda's operations to insert, read, and withdraw tuples, *reactions* allow for asynchronous notifications when data of interest appears in the shared tuple space. In addition, several WSN-specific features are provided. For instance, *capability tuples* enable on-demand sensing, therefore sparing the energy required to keep sensed information up to date in the shared tuple space in the absence of data consumers. TeenyLime provides constructs useful to develop stand-alone applications as well as system level mechanisms, e.g., routing protocols, as demonstrated by the real-world deployment described by Ceriotti et al. [2009].

Example. Consider an application for fire control in buildings. Sensor nodes are deployed to monitor temperature, along with actuator nodes triggering their attached devices (e.g., a water sprinkler) when temperature is above a threshold.

To implement the latter functionality, actuators install a reaction on their neighbors to watch for tuples reporting a temperature above the safety threshold. This is shown in the code fragment of Figure 17, adapted from [Costa et al. 2007]. In particular, the first parameter to the `addReaction` primitive (line 4) indicates whether reaction notifications must be reliably delivered to the requesting node. In addition, `tempTemplate` identifies the data of interest using a pattern matching mechanism that, unlike the original Linda model, allows for constraints on the value of the tuple fields. Temperature sensors periodically take a sample and pack it in a tuple stored in the local tuple space as shown in Figure 18. Insertion is accomplished using an `out` operation (line 10) by setting the `target` parameter to

```
1  command result_t StdControl.start() {
2    tuple tempTemplate = newTuple(2, actualField_uint16(TEMPERATURE),
3                                     greaterField(TEMPERATURE_SAFETY_THRESHOLD));
4    call TS.addReaction(TRUE, TL_NEIGHBORHOOD, &tempTemplate);
5    return SUCCESS;
6  }
7   event result_t TS.tupleReady(TLOpId_t operationId,
8                                tuple *tuples, uint8_t number) {
9     // Notification triggered ...
10 }
```

Fig. 17. TeenyLime code for an actuator node interested in temperature values.

```
1  command result_t StdControl.start() {
2    return call SensingTimer.start (TIMER_REPEAT, SENSING_TIMER);
3  }
4  event result_t SensingTimer.fired() {
5    return call TemperatureSensor.getData();
6  }
7  event result_t TemperatureSensor.dataReady(uint16_t reading){
8    tuple temperatureValue = newTuple(2, actualField_uint16(TEMPERATURE),
9                                         actualField_uint16(reading));
10   call TupleSpace.out(FALSE,TL_LOCAL,&temperatureValue);
11   return SUCCESS;
12 }
```

Fig. 18. TeenyLime code for a temperature node.

`TL_LOCAL`. This operation, by virtue of one-hop sharing, automatically triggers the aforementioned reaction on neighboring nodes. Actuator nodes process the tuple that caused the reaction firing in the `tupleReady` event in Figure 17 (line 7-10).

Implementation highlights. TeenyLime is built atop nesC/TinyOS and Active Messages. Remote reactions rely on a soft-state approach to deal with nodes joining or failing. Each node periodically sends messages containing control data for all remote reactions. Upon receipt of this message, a timer associated with installed reactions is refreshed. If and when the timer expires, the corresponding reaction is removed. To implement reliable operations, solutions such as [van Dam and Langendoen 2003; Rajendran et al. 2006]) can be plugged into TeenyLime with minimal effort. The current TMote Sky [MoteIV] port includes a dedicated reliability layer based on hardware-level acknowledgements.

5.5.2 *Mobile code: Agilla.*

Overview. Agilla is a mobile agent [Fuggetta et al. 1998] system for WSNs. Programs are composed of one or more software agents able to migrate across nodes. An Agilla agent is similar to a virtual machine with its own instruction set and dedicated data/instruction memory. Local coordination among agents is accomplished using a Linda-like tuple space. Agents can insert data in a local data pool to be read by different agents at later times. The use of tuple spaces allows programmers to decouple the application logic residing in the agents from their coordination and communication. Agilla therefore provides a powerful mechanism to implement applications requiring on-the-fly reconfiguration of some functionality

```
1  BEGIN  pushn fir
2         pusht LOCATION
3         pushc 2
4         pushc FIRE
5         regrxn  // Register fire alert reaction
6         wait    // Wait for reaction to fire
7  FIRE   pop
8         sclone  // Strong clone to the node detecting fire
9         ...     // Fire tracking code
```

Fig. 19. Fire tracking with Agilla.

in response to external phenomena.

Example. Consider a fire monitoring application in a forest. Fire-detection agents are deployed to monitor the temperature in various regions. When a rise in temperature is detected, fire-detection agents spawn fire-tracking agents that swarm around to collect information about the exact location of the fire.

To implement such an application, Agilla provides an API to interact with the tuple space at each node, and to clone agents. As for the former aspect, Agilla provides operations to insert, read, and remove tuples. In addition, similarly to TeenyLime, it gives programmers the ability to add reactions to the tuple space, although the matching mechanism is here limited to type-based matching and reactions are local. Migration is accomplished either by relocating the agent with `smove` and `wmove`, or by cloning it on a different node with `sclone` and `wclone`. The `w` and `s` in front of the operation name specifies whether strong or weak mobility is required [Fuggetta et al. 1998]. Strong mobility ensures that the execution state is retained across movement, enabling the agent to resume execution right after the migration instruction. Instead, weak mobility moves only the agent code, whose execution restarts from scratch.

Figure 19, adapted from [Fok et al. 2005], shows how a fire tracking agent is notified about the presence of an increase in temperature. When such an agent is injected into a node, it registers a reaction for `FireAlert` tuples and waits for it to be triggered (line 1-5). This occurs when a fire detection agent outputs the corresponding tuple in the tuple space. Upon triggering of the reaction, the agent immediately clones itself to a different node (line 7-8). Once there, it possibly keeps cloning itself to gather information in regions around the phenomena.

Implementation highlights. Agilla is implemented on top of TinyOS. The instruction set and the mechanisms enabling on-the-fly execution of code are based on the Maté [Levis and Culler 2002] virtual machine. An agent manager maintains each agent's context, allocates memory when the agent arrives, and deallocates the same memory when the agent leaves or dies. The latter aspects are dealt with using a lightweight implementation of dynamic memory, as this functionality is not available in TinyOS. A context manager determines the node location and maintains the list of reachable neighbors, whereas a tuple space manager implements the operations to read/write from/to the tuple space, and registers/triggers reactions when required. Migrating agents requires reliable transmissions. This is achieved using a hop-by-hop retransmission scheme where messages not yet acknowledged are re-sent upon expiration of a timeout.

```
1  INSERT INTO EVENT_LIST
2    (EVENT_ID, RANGE_TYPE, DETECTING_RANGE, SUBEVENT_SET, REGISTRANT_SET,
3     REPORT_DEADLINE, DETECTION_DURATION [, SPATIAL_RESOLUTION ])
```

Fig. 20. Subscription format in DSWare.

```
1  INSERT INTO EVENT_LIST
2    (explosion, AREA, [0,0;200,200],
3     SUBEVENT_SET, user_base_station, 1 sec,
4     1 hour)
5
6  SUBEVENT_SET (
7    SAFETY_TIMEOUT,
8    MIN_CONFIDENCE,
9    (temperature > 60),
10   (light > 200),
11   (compareSound(sound,explosionSignature))
12 )
```

Fig. 21. Detecting explosions with DSWare.

5.5.3 *Message passing: DSWare.*

Overview. DSWare is a message passing middleware whose focus are real-time applications for detection of sporadic events. It employs a form of publish/subscribe [Eugster et al. 2003] paradigm in which users specify subscriptions expressing the characteristics of the phenomena of interest, and are notified upon the occurrence of matching phenomena. A higher-level notion of event enables programmers to infer the occurrence of a phenomenon from raw sensor observations. For instance, an event can be defined as the composition of two physical sub-events occurring within a specific time interval from each other. Confidence levels can also be defined to fine-tune the relationships among sub-events, e.g., their relative importance or fitness to a pattern.

Subscriptions are issued at the user's base station using a dialect of SQL, according to the format in Figure 20. Besides the event identifier, `RANGE_TYPE` and `DETECTING_RANGE` specify the group of sensors responsible for detecting the event. The corresponding notification is reported before the `REPORT_DEADLINE` to every node in the `REGISTRANT_SET`. `DETECTION_DURATION` specifies the total duration of this subscription, whereas `SPATIAL_RESOLUTION` determines the geographical granularity for the event's detection. Finally, `SUBEVENT_SET` specifies a group of subevents that must occur for this event to be observed, their timing constraints and confidence levels.

Example. Consider an application to detect explosions in a given geographical area. Temperature, light, and acoustic sensors are deployed to accomplish the task. Figure 21 illustrates how to describe the required processing in DSWare. The program defines a high-level temperature sub-event occurring when the temperature is higher than a safety threshold, a light sub-event corresponding to a sharp change in the light intensity, and an acoustic sub-event representing the occurrence of a sound whose signature resembles that of an explosion (line 9-11). The higher-level explosion event is defined as the combination of the aforementioned sub-events

when occurring within a specified time interval from each other and within the same geographical region (line 2-3). In addition, upon detection of such event, the program requires a notification to be reported to the user within one second (line 3). A further option specifies that the application must monitor these types of events for one hour (line 4).

Implementation highlights. Subscriptions are propagated in the network until they reach the area of interest, or all the nodes in the system. In doing so, a routing tree is built connecting the base station to the relevant sensor nodes. Two optimizations are performed in case multiple nodes subscribe to the same information. First, in case subscriptions are for the same data yet they have different rates, DSWare places copies of the relevant information at intermediate nodes to limit the amount of information flowing in the network. Second, DSWare tries to merge paths leading to different base stations to minimize redundant transmissions [Kim et al. 2003]. To guarantee real-time delivery of event notifications, an earliest-deadline-first scheduling mechanism is employed. An alternative, energy-aware scheduling technique is also provided, although it may occasionally fail to meet the requested deadlines.

5.6 Programming Paradigm

The programming paradigm *determines the abstractions used to represent the individual elements of a program.* These include functions and variables, as well as the steps that compose a computation, e.g., assignments and iterations. The solutions hitherto described already highlight the variety of programming paradigms available. This aspect bears great influence on the learning curve for new programmers, and ultimately on their productivity.

Classification. Looking at the current state of the art in WSN programming, three major paradigms can be identified:

—**Imperative**: the intended application processing is expressed through statements that explicitly indicate how to change the program state. By far the most widespread, it can be further classified into **sequential** or **event-driven**.

—**Declarative**: the application goal is described without specifying how it is accomplished. Relevant sub-classes of declarative approaches include **functional**, **rule-based**, **SQL-like**, and **special-purpose**.

—**Hybrid**: the programming approach is a combination of multiple programming paradigms, e.g., imperative and declarative.

The nesC language, illustrated in Section 5.1.1, features an imperative event-driven paradigm based on split-phase operations. The control flow is divided across different operations that are asynchronously executed when some events occur, e.g., upon receiving a message. Although this increases parallelism, it generally makes implementations more entangled and difficult to reason about. Next, we illustrate Pleiades [Kothari et al. 2007], which instead adopts an imperative sequential paradigm.

The Regiment system, illustrated in Section 5.4, is representative of the declarative functional paradigm. Constructs are provided to apply given functions to nodes

in a region and store the output at a single node or at all devices in a region. In the following, we describe Snlog [Chu et al. 2007] to illustrate declarative solutions that adopt a rule-based approach. The TinyDB system described in Section 5.1.4 is an example of a declarative approach based on SQL-like constructs. TinyDB programmers specify constraints on the data of interest without specifying the exact procedure to gather the data themselves. Logical Neighborhoods, described in Section 5.1.3, exemplifies a special-purpose declarative paradigm, whose custom constructs are used to identify the target nodes.

Finally, the ATaG [Bakshi et al. 2005] framework, illustrated in the following, features a hybrid approach. Communication among tasks executed on separate nodes is described in a declarative manner, whereas the local node computation is expressed using an imperative language. This choice decouples local processing from inter-node coordination.

5.6.1 *Imperative → Sequential: Pleiades.*

Overview. Pleiades is a programming language providing a centralized view on the sensor network. It extends the C language with constructs to address the nodes in the network and to access their local state. A Pleiades program normally features a single sequential thread of control, i.e., execution unfolds with only one node in the system executing any Pleiades instruction at any point in time. Nonetheless, a dedicated language construct `cfor` is provided to introduce concurrent executions at multiple nodes. Whenever required, the underlying run-time guarantees serializable execution of `cfor` statements. Because of these features, Pleiades targets concurrent applications that require guarantees on their distributed execution. An example of a similar scenario follows.

Example. Figure 22 depicts a Pleiades program implementing a street-parking application, adapted from [Kothari et al. 2007]. The goal is to identify the free spot closest to the driver's destination. To do so, sensors are deployed in parking spots to monitor their occupancy. The control flow iterates among the nodes in the network in search of the first free spot, starting from the node closest to the desired destination. The program makes use of most of the language features in Pleiades:

—The `node` data type abstracts a single WSN device, whereas `nodeset` represents a collection of nodes. Helper functions are provided to obtain such collections. For instance, `get_network_nodes()` returns a `nodeset` containing all nodes in the system, and `get_neighbors(n)` returns `n`'s one-hop neighbors.

—Variables are normally shared across all nodes in the system, unless they are tagged by programmers with the `nodelocal` attribute, e.g., `isfree` in Figure 22 (line 2). Node-local variables are accessed using the notation `var@e`, where `var` is a `nodelocal` variable and `e` is a `node`.

—The `cfor` construct works as a normal `for` loop, except the execution of its body is concurrent w.r.t. the nodes in a `nodeset`. The Pleiades run-time can ensure that the effect of a `cfor` corresponds to some sequential execution of the loop. Here, this is required to make sure that only one free node is reserved for the car arriving (line 20-39). Access to `loose` variables, on the other hand, is not synchronized inside `cfor` loops.

```
#include "Pleiades.h"
boolean nodelocal isFree=TRUE;
nodeset nodelocal neighbors;
node nodelocal neighborIter;

void reserve (pos dst) {
  boolean reserved = FALSE;
  node nodeIter, reservedNode = NULL;
  node n=closest_node(dst);
  nodeset loose nToExamine = add_node(n, empty_nodeset());
  nodeset loose nExamined = empty_nodeset();

  if (isfree@n) {
    reserved = TREE; reservedNode = n;
    isfree@n = FALSE;
    return;
  }

  while (!reserved && !empty(nToExamine)) {
    cfor (nodeIter=get_first(nToExamine);
          nodeIter!=NULL;
          nodeIter = get_next(nToExamine)) {
      neighbors@nodeIter=get:neighbors(nodeIter);
      for (neighborIter@nodeIter=get_first(enighors@nodeIter);
           neighborIter@nodeIter!=NULL;
           neighborIter@nodeIter=get_next(neighbors@nodeIter)) {
        if (!member(neighborIter@nodeIter,nExamined))
          add_node(neighborIter@nodeIter,nToExamine);
      }
      if (isfree@nodeIter) {
        if (!reserved) {
          reserved=TRUE; reservedNode=nodeIter;
          isfree@nodeIter=FALSE;
          break;
        }
      }
      remove_node(nodeIter,nToExamine);
      add_node(nodeIter,nExamined);
    }
  }
}
```

Fig. 22. A street-parking application in Pleiades.

Implementation highlights. The Pleiades compiler performs data-flow analysis to partition the program in independent execution units called *nodecut*s, each running on a single node. The compiler assigns nodecuts to nodes based on the expected communication cost for accessing variables at remote nodes. At run-time, the execution flow moves from one node to the other in case the flow transitions between nodecuts assigned to different devices. A dedicated locking mechanism is provided to implement serializable execution of `cfor`s. A coordinator is elected among the nodes involved. It manages the locks on shared variables according to the current state of execution, and monitors the execution state of the other nodes involved to determine the presence of deadlocks, e.g., caused by nested `cfor` statements.

5.6.2 *Declarative → Rule-based: Snlog.*

Overview. Snlog is a rule-oriented approach inspired by logical programming. The core language constructs are *predicates*, *tuples*, *facts*, and *rules*. Predicates specify schemas for data as ordered sequences of fields, analogously to how tables in relational databases specify the format of records. Tuples represent the actual data, similarly to instantiated records in database tables. Facts are particular tuples

```
1  builtin(trackingSignal, 'TargetDetectorModule.c').
2  import(tree.snl).
3
4  message(@Src, Src, Head, SrcX, SrcY, Val) :-
5    trackingSignal(@Src, Val), detectorNode(@Src),
6    location(@Src, SrcX, SrcY), clusterHead(@Src, Head).
7  message(@Next, Src, Dst, X, Y, Val) :-
8    message(@Crt, Src, Dst, X, Y, Val),
9    nextHop(@Crt, Dst, Next, Cost).
10
11 trackingLog(@Dst, Epoch, X, Y, Val) :-
12   message(@Dst, Src, Dst, X, Y, Val), epoch(@Dst, Epoch).
13 estimation(@S, Epoch, <AVG, X>, <AVG, Y>) :-
14   trackingLog(@S, Epoch, X, Y, Val), epoch(@S, Epoch).
15
16 timer(@S, epochTimer, Period) :- timer(@S, epochTimer, Period).
17 epoch(@S, Epoch++) :- timer(@S, epochTimer, _), epoch(@S, Epoch).
```

Fig. 23. An object tracking application in Snlog.

that are instantiated at system start-up, whereas rules express the actual processing. Similarly to Datalog-like languages, rules consists of a head and body part. Programmers express in the body the conditions for outputting the tuples specified in the head. Distributed executions are described using a location specifier, which represents the node hosting a tuple in case this is not co-located with the node executing a given rule. Only 1-hop interactions are supported. Atomicity is guaranteed at the rule level. Native C or nesC code can be linked to the rule engine to interact with low-level devices or implement efficient memory management. Snlog has been used at different levels of the stack, to implement applications such as tracking moving objects as well as routing protocols.

Example. Figure 23 reports the implementation of a simplified object tracking application in Snlog, adapted from [Chu et al. 2007]. Compared to the analogous applications we described for EnviroSuite and Abstract Regions, here the leader (cluster-head in this example) processing the sensor measurements is statically determined. In [Chu et al. 2007], the authors mention that only 4 additional rules are needed to dynamically identify the leader.

At the beginning of Figure 23, the `import` construct includes an external Snlog file implementing a tree-based collection protocol. Essentially, this makes available the `nextHop` tuple used to direct data towards the leader (line 9). When a node detects the target, it creates a tuple to report the node position to the cluster-head (line 4-9). Upon receipt, this information is timestamped with the current epoch value (line 11-12) and then used to average the position of the moving object (line 13-14). Periodic timers are used to update the epoch value (line 16-17), where a timer predicate in a rule body indicates a timer firing and the same timer predicate in a rule head represents the timer being set.

Implementation highlights. The Snlog compiler outputs executable nesC code. A generic run-time layer is provided to support rule execution, whereas the individual rules are compiled into a data-flow chain of database operators such as joins, selection, and projection. Each operator is mapped to a nesC component obtained from a generic template that the compiler customizes depending on the nature of

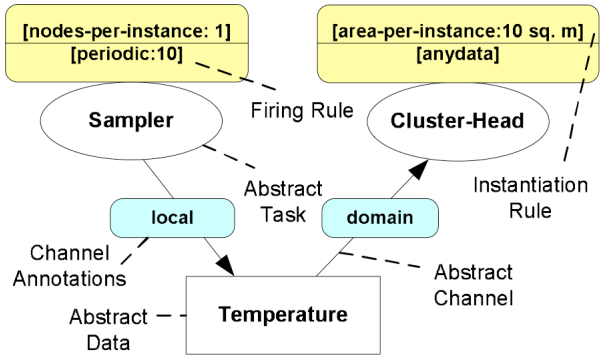


Fig. 24. A cluster-based data collection program in ATaG.

the rules involved. Optimization goals can also be set at compile-time, e.g., to minimize code size as opposed to data size. Communication is handled using the Active Message stack described in Section 5.1.1.

5.6.3 *Hybrid: Abstract Task Graph (ATaG).*

Overview. ATaG is a programming framework providing a mixed declarative-imperative approach. The notions of *abstract task* and *abstract data item* are at the core of the programming model. A task is a logical entity encapsulating the processing of one or more data items, which represent the information. Different copies of the same task may run on different nodes. The flow of information between tasks is specified declaratively with *abstract channels* connecting a data item to the tasks that produce or consume it.

The code in a task is written in an imperative language, and relies on a shared data pool for local communication, allowing tasks to output data or to be notified when some data of interest becomes available. To support the former, a `putData(DataItem)` operation is made available. As for the latter, programmers are provided with a task template that lists an empty `handleDataItem()` function for each incoming channel. ATaG helps programmers in expressing multi-stage, data-centric processing. It is therefore suited to sense-and-react applications, where the application typically requires complex operations to decide on the actions to take.

Example. Figure 24 illustrates a sample ATaG program, adapted from [Pathak et al. 2007], specifying a cluster-based data gathering application. Sensors within a cluster take periodic temperature readings collected by the corresponding cluster-head. The former behavior is encoded in the *Sampler* task, while the latter is specified in *Cluster-Head*. The *Temperature* data item is connected to both tasks using a channel originating from *Sampler*, and a channel directed to *Cluster-Head*.

Tasks are annotated with firing and instantiation rules. The former specify when the processing in a task must be triggered. In our example, the *Sampler* task is triggered every 10 seconds according to the `periodic` rule. The *Cluster-Head* fires whenever at least one data item is available on *any* of its incoming channels, according to the `any-data` firing rule. Tasks run on the indi-

```
// ...
while (TRUE) {
  sleep(10000);

  // Written by the programmer
  int temperature = getTemperature();
  TemperatureDataItem t = newTemperatureDataItem(temperature);
  putData(t);
}
// ...
```

Fig. 25. Fragment of the imperative code for the *Sampler* tasks of Figure 24.

```
// Asynchronously called when data is available on some input channel
void handleDataItem(TemperatureDataItem t) {

  // Written by the programmer
  int temperature = getDataFrom(t);
  updateAverage(temperature)
}
// ...
```

Fig. 26. Fragment of the imperative code for the *Cluster-Head* tasks of Figure 24.

vidual nodes according to the instantiation rules specified by programmers. The `nodes-per-instance:1` construct requires the task to be instantiated once on every node. The `area-per-instance` construct, instead, partitions the geographical space according to the given parameter, and determines the deployment of one task instance per partition.

Abstract channels are annotated to express the interest of a task in a data item. In this example, the *Sampler* task generates data items of type *Temperature* which remains `local` to the node where they were generated. The *Cluster-Head* uses the `domain` annotation to gather data from the temperature sensors in its cluster, which binds to the system partitioning obtained from `area-per-instance` by connecting tasks running in the same partition. ATaG has also been extended with instantiation rules and channel annotations based on application-level properties of the nodes [Mottola et al. 2007], e.g., the sensing devices they are equipped with.

Based on the declarative part of an ATaG program, the compiler generates a set of templates for the different task types that programmers fill with the imperative code required. Figure 25 depicts a fragment of imperative code for the *Sampler* task. The ATaG compiler generates a loop containing only the `sleep` instruction whose parameter reflects the `periodic` rule used for the same task. In the case of *Cluster-Head*, as illustrated in Figure 26, the `handleDataItem` function is entirely filled by programmers to process temperature readings arriving through one of the input channels.

Implementation highlights. The ATaG compiler takes as input the description of tasks and channels, examines the corresponding flow of data, and decides on the allocation of tasks to nodes depending on information on the target environment, e.g., the location of nodes. The output of the compiler targets a dedicated node-level run-time layer designed to be highly modular [Bakshi et al. 2005]. Some of the

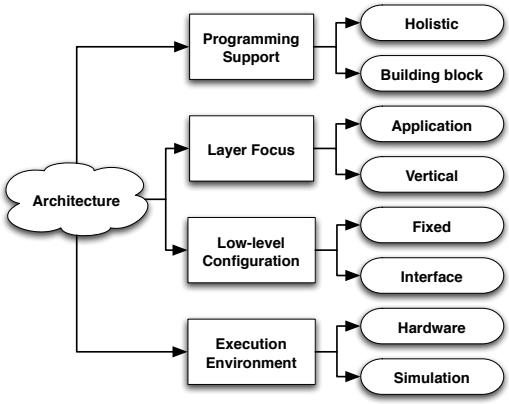


Fig. 27. A taxonomy of architectural aspects in WSN programming abstractions.

mechanisms are, however, not provided beforehand. For instance, the programmer must provide the most appropriate routing scheme depending on the specific application and target environment. Logical Neighborhoods, described in Section 5.1.3, is used as underlying communication layer when task allocation is determined based on application-level properties of the nodes.

6. PROGRAMMING WIRELESS SENSOR NETWORKS: ARCHITECTURAL ASPECTS

The objective of this section is to understand the impact of a given programming approach on the design and development of WSN applications. Specifically, we study the relationships among the various programming solutions in the context of the reference architecture we identified in Section 3. Figure 27 provides a bird's eye view on our taxonomy of architectural aspects.

6.1 Programming Support

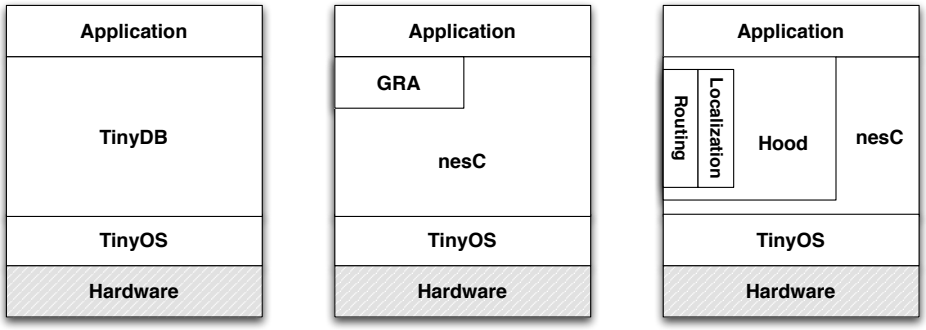
The first dimension we investigate deals with *the extent to which a given programming abstraction provides support to the programmer*. Some of the existing approaches are meant to be the only tool that programmers use in developing applications. However, recently it has been observed that smaller, composable building-blocks could be a way to tackle the complexity of WSN applications [Embedded WiSeNts Project].

Classification. Based on the observations above, we make a distinction along the following lines:

—**Holistic** solutions are intended to be used as the only programming support, and are unable to work in combination with other approaches.

—**Building blocks** are conceived to be used in conjunction with other solutions, each targeting independent issues.

The above distinction is further illustrated in Figure 28(a) and Figure 28(b). The majority of existing WSN programming approaches falls in the former class. For instance, TinyDB [Madden et al. 2005], illustrated in Section 5.1.4, addresses



(a) TinyDB: Holistic programming support, Application layer focus.

(b) GRA: Building block programming support.

(c) Hood: Vertical layer focus.

Fig. 28. Architectural issues: Programming Support vs. Layer Focus.

the needs of most data collection applications by itself, without leaving room for integration with other programming solutions, as shown in Figure 28(a). In contrast, as an example of a building-block solution, here we describe Generic Role Assignment (GRA) [Frank and Römer 2005; 2006]. As shown in Figure 28(b), GRA is not meant to provide complete support for application development, rather it is designed to focus on a specific facet of WSN programming, and to work in conjunction with other approaches.

6.1.1 *Building-block: Generic Role Assignment (GRA).*

Overview. GRA tackles the problem of dynamically self-configuring WSN nodes according to programmer-specified requirements, whereas it leaves concerns such as data collection and dissemination to other, complementary solutions. To address the configuration issue, GRA provides a declarative role specification language and distributed algorithms for dynamic role assignment. A role specification is a list of role-rule pairs. For each role, the corresponding rule describes the conditions for the role to be assigned to the local node. Rules are expressed as boolean predicates referring to the properties of the node considered (e.g., remaining energy or geographical location), or to the properties of other nodes within a given number of hops (e.g., how many temperature sensors are reachable within 3 hops). Using the constructs provided, a wide range of role assignment problems can be expressed, from cluster-head election to coverage, as described next.

Example. Consider the classic coverage problem. A certain geographical area is said to be covered if every physical point in the area lies within the observation range of at least one sensor node. If nodes are densely deployed, redundant nodes can be turned off to save energy. However, when active nodes run out of power, the redundant nodes must be switched back on.

The above application essentially requires a proper assignment of the roles `ON` and `OFF` to nodes, and an update of the assignment when nodes join or fail. Figure 29, adapted from [Frank and Römer 2005], shows the corresponding role specification.

```
1 ON :: {
2   temp-sensor == true &&
3   battery >= threshold &&
4   count (2 hops) {
5     role == ON &&
6     dist(super.pos, pos) <= sensing-range
7   } <= 1 }
8 OFF :: else
```

Fig. 29. A GRA role specification for the coverage problem.

For a given node to take the `ON` role, it must have a temperature sensor, a minimum battery level, and at most another node with role `ON` within the node's sensing range (line 1-7). The latter condition is specified using the `count` operator. This takes as input a number of hops and returns the number of nodes within such range matching the specification in curly braces. If a node does not match the conditions for the `ON` role, it defaults to `OFF` (line 8).

Implementation highlights. All nodes in the network are provided with the complete role specifications. Based on these rules, the nodes evaluate how many hops they need to push their local information to let other nodes evaluate their rules. To account for changing node properties and network dynamics, the role specifications are periodically re-evaluated. In the former case, a node re-evaluates the specification only if the property change may affect its own role or the one of some other node. As each node is aware of the complete role specification, this decision can be taken locally. As for topology changes, distributed protocols are provided to recognize when nodes join or fail and to trigger a re-evaluation of the current role assignment.

6.2 Layer Focus

In contrast to the previous dimension, here we look at *which architectural layers are the main focus* of the approach under consideration. Essentially, we are considering whether a programming approach provides support only for the application level or spans other levels of the reference architecture in Figure 4.

Classification. The above considerations suggest the following classification:

—**Application**-level solutions are intended to support development only of end-user applications, i.e., the topmost layer in Figure 4.

—**Vertical** solutions, instead, provide support potentially throughout all layers. These approaches can be used, for instance, to implement localization or routing mechanisms.

The distinction can be understood by relying again on Figure 28. TinyDB is an example of a system that focuses entirely and solely on the application layer. In contrast, Figure 28(c) shows that the Hood system [Whitehouse et al. 2004], described next, can be used to implement any of the software layers on top of the operating system.

Interestingly, Figure 28 also clarifies the relation between the two dimensions of classification just introduced, i.e., Programming Support and Layer Focus. Loosely

```
1 generate attribute LightAttribute from int;
2
3 generate neighborhood LightHood {
4   wire filter LightThrehshold;
5   set max_neighbors to 5;
6   reflection LightRefl from LightAttribute;
7 }
```

Fig. 30. Reading light values using Hood.

speaking, the former focuses on whether *multiple* approaches can coexist *horizontally* below the application layer, whereas the latter focuses on whether a *single* approach is able to extend *vertically* throughout the reference architecture.

6.2.1 *Vertical: Hood.*

Overview. Hood provides a notion of neighborhood as a programming primitive in nesC. Constructs are provided to identify a subset of a node's physical 1-hop neighbors based on application criteria, and to share state with them. A node exports information in the form of *attributes*, defined by the programmer at compile-time. Membership in a Hood neighborhood is specified by *filters*, boolean functions determining whether a node is part of a neighborhood based on the value of its attributes. If so, a *mirror* for that particular neighbor is created on the local node. The mirror contains both *reflections*, i.e., local copies of the neighbor's attributes used to access the shared data, and *scribbles*, which are local annotations about that neighbor. The complexity of node discovery and data sharing is dealt with by the underlying Hood run-time. According to Whitehouse et al. [2004], Hood can provide support in developing a wide range of functionality, including applications, time synchronization and other system services, and MAC protocols. This naturally fosters cross-layer interactions. Whitehouse et al. [2004] indeed describe an object tracking application that exploits different neighborhoods for routing, localization, and application-level processing of tracking information, where information is shared across different functionality through scribbles.

Example. Consider an application to monitor the light intensity. Figure 30 depicts a fragment of Hood code, adapted from [Whitehouse et al. 2004], that defines a neighborhood containing light sensors whose current reading is above a threshold.

The `generate` construct is used to define an attribute or to declare a new neighborhood. As for the former, programmers create a `LightAttribute` out of an integer value (line 1), while the `LightHood` neighborhood is created (line 3-7) by specifying the filter for establishing membership in this neighborhood (`LightThreshold`), the maximum number of members of this neighborhood (`5`), and the specific attribute mirrored on the local node. The actual processing to implement the filter is supposed to be provided as a nesC module implementing a standard interface.

A simple API is provided to interact with the neighborhood in the application code. nesC commands can be used to iterate through the current members of a neighborhood and access their local mirrors, while nesC events are defined that fire when the value of a locally mirrored attribute changes.

Implementation highlights. The Hood constructs to define attributes and neighborhoods are given as input to a dedicated pre-processor that outputs plain nesC code. The underlying distributed implementation is based on a periodic 1-hop broadcasting, along with filtering on the receiver side. This mechanism is also employed for neighbor discovery and maintenance. In case the application wishes to control the dissemination of local information directly (e.g., because of a sudden increase in a sensed value or to adhere to specific timing constraints) it can also force a broadcasting of the local attributes on demand.

6.3 Low-Level Configuration

Some of the existing solutions provide *interfaces to the lower layers* to give applications the ability to adapt the system behavior to changes in the application goals or in the network conditions. This feature is essential to enable cross-layer interactions and reduce resource consumption.

Classification. We can therefore distinguish between systems where:

—the configuration parameters are **fixed** at compile-time. Programmers are relieved from the burden of handling details deep down in the stack. However, this prevents adaptation strategies to percolate down the architecture.

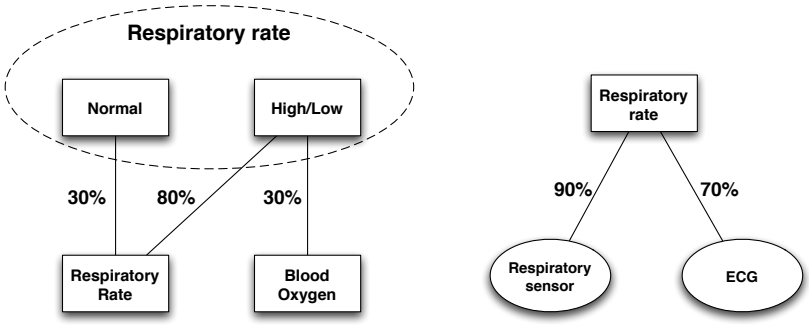
—dedicated **interface**s are provided to access the lowest layers. Programmers can fine-tune various system aspects and explore trade-offs between performance and resource-consumption. However, the responsibility of ensuring that this tuning preserves an acceptable system behavior lies on the programmer's shoulders.

A relevant fraction of the available approaches falls in the former category. For instance, in Pleiades the communication layer is essentially a black box that cannot be tuned at run-time. To exemplify the latter category, we illustrate next the MiLAN middleware [Heinzelman et al. 2004].

6.3.1 *Interface to lower-levels: MiLAN.*

Overview. MiLAN focuses on applications where programmers are to trade off system lifetime for data quality, e.g., health monitoring applications. MiLAN allows applications to specify their requirements using a notion of Quality of Service (QoS) for different variables of interest, where the QoS of a variable is a function of the specific sensors used to compute the variable's value. As these requirements may change over time, the application is described using a state machine with different QoS requirements associated to different states. Programmers also specify the quality of data provided by physical sensors to the evaluation of every variable of interest. Based on this information, MiLAN computes the application *feasible set*, i.e., the subset of nodes that collectively provide a QoS greater than or equal to the minimum acceptable by the application. In the presence of multiple feasible sets, MiLAN chooses the order in which they should be applied to minimize energy costs and maximize the system lifetime. As the application state changes, MiLAN recomputes the feasible sets and possibly performs the reconfiguration needed to gather data from a different subset of physical sensors.

Example. Consider an application to implement a personal health monitor using various WSN devices. Depending on the current patient status, a different QoS is



(a) Variables and required QoS based on the application state.

(b) QoS provided by different sensors to different variables of interest.

Fig. 31. Specifying QoS with MiLAN in a health care application.

required.

Figure 31(a) describes the QoS requirements depending on the patient's respiratory status, adapted from [Heinzelman et al. 2004]. In normal situations, it is sufficient to monitor the respiratory rate with 30% quality. When the patient has some respiratory problem—and the application state changes accordingly—it becomes necessary to gather more information to explain the cause of the problem. Thus, the application requires to obtain the respiratory rate with a minimum of 80% quality, and to measure the percentage of blood oxygen with at least 30% quality. Figure 31(b) describes how the respiratory rate can be obtained from the physical sensors. A respiratory sensor can provide such a reading with 90% quality (in the aforementioned QoS sense), whereas an ECG device can provide the same information with 70% quality. Based on these information, MiLAN determines that under normal conditions the respiratory rate can be obtained with either the respiratory sensor or the ECG. The one maximizing the system lifetime is configured to provide the actual reading. On the contrary, when the patient is experiencing some respiratory problem, MiLAN recognizes that the only device providing a sufficient QoS is the respiratory sensor, and reconfigures the network accordingly.

Implementation highlights. MiLAN is implemented in C. The internal architecture is designed to use different network plug-ins for inter-operating sensors of different nature, ranging from 802.11-based devices to Bluetooth nodes. The specific network plug-in is aware of all the network-specific features (e.g., routing) and tuning parameters that can be exploited to prolong the lifetime of a feasible set. A service discovery protocol such as SDP [Avancha et al. 2002] is used to find new nodes and to trigger a reconfiguration in case of failing devices.

6.4 Execution Environment

WSN implementations are generally difficult to port. Therefore, when it comes to applying a given programming approach in a real application, programmers must consider the *specific hardware platforms that are explicitly supported*. Nevertheless, despite the plethora of available approaches, the range of officially supported

platforms—as reported in the current literature—is surprisingly narrow.

Classification. Existing systems can typically be executed in two different ways:

—on real **hardware**: with the exception of Logical Neighborhoods, which runs also on Contiki, all systems surveyed rely on TinyOS as operating system, and should in principle support any TinyOS-compliant WSN platform. However, efficiently supporting a given hardware platform often requires a considerable effort in custom optimizations.

—through **simulation**: typically used to assess the performance of a given solution, rather than providing direct support to the programmer. TOSSIM [Levis et al. 2002], the TinyOS simulator, or custom-made simulators are common choices. A few works (e.g., [Li et al. 2004]) relied on simulators borrowed from research on mobile ad hoc networks, such as GlomoSim [Zeng et al. 1998].

7. COMPLETING THE PICTURE

In this section we complete our survey of the state of the art in WSN programming approaches by describing systems other than those we used as examples thus far. However, due to space reasons here we cannot provide complete descriptions including code fragments.

Cougar [Yao and Gehrke 2002]. Similarly to TinyDB, Cougar provides programmers with a SQL-like interface to query the WSN, yet it lacks constructs to express fine-grained control flows. Thus, it is geared towards pure data collection applications. At the system level, Cougar is also based on a routing tree rooted at the user base station, like TinyDB. The techniques used to achieve energy efficiency are, however, different. For instance, Cougar tries to push selection operators down the routing tree to reduce the amount of data flowing up, yet it does not consider acquisitional issues, as TinyDB does.

FACTS [Terfloth et al. 2006]. The FACTS middleware provides a rule-based programming model inspired by logical reasoning in expert systems. Data is represented as facts in a dedicated repository. The appearance of new facts triggers the executions of one or more rules, which may generate new facts or remove existing ones. Pure C functions can be used to interact with sensing devices and provide inputs to the creation of new facts. Facts can be shared among different nodes. The basic communication primitives provide 1-hop data sharing, although multi-hop protocols can be employed in collaboration with the basic rule engine.

Flask [Mainland et al. 2008]. A data-flow language is at the core of Flask's programming model. The flow is specified by wiring data operators in an acyclic graph. Each operator is a computational unit taking multiple inputs and producing a single output value. The control flow migrates across operators in a depth-first manner. Different operators can be located on different nodes, and are interconnected using a publish/subscribe infrastructure. The underlying routing scheme can be changed by programmers on a per-application basis. Flask programs are translated into executable nesC code by a dedicated pre-processor. Flask is also designed for building higher-level abstractions in terms of data-flow operators. As

an example, Mainland et al. [2008] show how to map SQL constructs to Flask primitives, building a database abstraction based on Flask.

Kairos [Gummadi et al. 2005]. The Kairos programming model adopts a centralized perspective similar to Pleiades. Three fundamental programming constructs are provided as extensions to the Python language. Programmers use these constructs to read/write variables at nodes, to address specific nodes, and to iterate through the 1-hop neighbors of a node. This enables expressing the required functionality in a way that resembles the high-level descriptions of algorithms used in textbooks. Unlike Pleiades, Kairos only provides an eventual consistency model for data shared among nodes. As a result, most of the complexity required to implement the Kairos programming model can be dealt with by a dedicated pre-processor, while only a minimalistic run-time support is required on the WSN nodes.

MacroLab [Hnat et al. 2008]. MacroLab is a programming model based on vector operations, similarly to MatLab programs. The authors introduce a notion of macrovector where one dimension in the vector is indexed by node identifiers. For instance, light readings in the system may be stored in a macrovector, and a node identifier may be used to access the reading at a specific device. In addition, a neighborhood-based extension of macrovectors may be used to slice a large macrovector to include only data at neighboring nodes. The system allows programmers to use operators for standard vector arithmetic, and a "dot-product" operator useful to select different elements of a vector at different nodes. MacroLab programs are fed as input to a "decomposer" that outputs executable code operating in a distributed fashion, at a central base station, or halfway between the two extremes. The decision on which implementation to use is taken by a dedicated cost analyzer. This examines the expected system performance based on a cost profile associated to the target deployment, which includes information such as the characteristics of the topology or the power profile of the hardware employed.

Market-based programming [Mainland et al. 2004]. In this programming framework, the objective is to obtain globally efficient behavior under dynamic conditions. Every node is characterized by the actions it can take, the corresponding cost, and a possible reward the node receives in exchange to performing a given action. To maximize its own profit, nodes autonomously decide which actions to take based on the current rewards, the actions' cost, and the surrounding context. The user induces the desired behavior by dynamically changing the rewards given to nodes for each type of action. Communication is delegated to a specific routing scheme chosen by programmers depending on the application's requirements.

Pieces [Liu et al. 2003]. The focus of Pieces is on collaborative applications where multiple, geographically related data must be processed, supported by a programming abstraction based on a notion of group. Similarly to Abstract Regions, system support must be provided on a per-group basis. Constructs are provided to determine the role of nodes in a group (e.g., to determine when a node is to be elected as the leader), and to share information. The application processing is expressed in discrete steps as input/output operations on state variables. The inputs are governed by sensed data, whereas the outputs are the result of some processing

based on the previous values of state variables and the current inputs.

RuleCaster [Bischoff and Kortuem 2007]. The programming model of RuleCaster is centered on a logical partitioning of the network in several spatial regions. Each region is in one discrete state. Rules specify state transitions in each region based on sensed data. Similarly to to Datalog-like languages, each rule consists of a body part specifying the conditions for the rule to fire, and a head part specifying the actions to perform. The RuleCaster compiler takes as input the application rules and a network model describing node locations and capabilities, and decides how to split the actual processing among the nodes. The compiler also determines whether to use a centralized distribution scheme where a single node is in charge of the entire processing, or a distributed strategy with one node per region managing the corresponding processing.

SensorWare [Boulis et al. 2003; Boulis et al. 2007]. Similarly to Agilla, SensorWare allows to move TCL-based scripts from node to node, providing support for multiple applications running concurrently on the same network. Nonetheless, unlike Agilla, it only provides weak mobility (i.e., the state does not move with code), which results in the program execution always restarting from the beginning on arrival on a new node. Coordination is accomplished using direct communication instead of shared memory spaces. When migrating code, policies regarding the energy required by a given script can be specified to determine its acceptance on a node. The current implementation targets fairly powerful devices, e.g., PDAs like Compaq iPAQs or embedded devices with XScale processors.

SINA [Shen et al. 2001]. The SQL dialects used in TinyDB and Cougar do not allow easy integration of custom data operators in addition to the built-in ones. SINA overcomes this limitation by complementing SQL-like declarative constructs with an imperative language called SQTL. This enables the injection of arbitrary code into the network. Support is provided to export the outcome of SQTL-based processing as query results. For instance, an additional aggregation function can be injected in the network at run-time, and made available to programmers for use in later SQL queries. Thus, programmers can dynamically enrich the set of SQL primitives depending on changing requirements. This enables the description of more sophisticated coordination patterns w.r.t. pure SQL. For instance, cluster-based data collection can be implemented using SQTL, and the results gathered using an enhanced SQL query.

snBench [Ocean et al. 2006]. This programming framework targets shared, multi-user sensor networks and provides a strongly-typed, functional language to express the application processing. However, loops and assignments to local variables are also allowed, therefore partially deviating from a purely functional paradigm. In the implementation, a central entity keeps track of the current status of every node in the system, and injects processing units in the network. In doing so, the processing units are optimized to take advantage of programs' shared dependencies, with the goal of making more efficient use of computation, network, and storage resources. Processing occurs on fairly powerful nodes controlling several sensors, as opposed to mote-class nodes usually equipped with a few sensing devices.

Spatial Programming [Borcea et al. 2004]. The system is based upon a logical addressing scheme coupled with a lightweight form of mobile code. In Spatial Programming, the environment is viewed as a single address space, and nodes are accessed using spatial references. These references refer to the expected physical location of a node (e.g., `hill:camera[0]`), and may optionally point to some property of the node itself, e.g., whether the node is currently active. A dedicated run-time system maintains the mapping from spatial references to the physical nodes. Smart Messages, a lightweight scripting language, are used to migrate code and data across nodes. A shared memory space is provided for coordination among Smart Messages, and also used to determine how to route a smart message at each hop. This allows to change the routing policy dynamically.

Virtual Nodes [Ciciriello et al. 2006]. As an extension of Logical Neighborhoods, described in Section 5.1.3, Virtual Nodes abstract programmer-specified subsets of nodes into a single, logical one, which takes the form of a virtual sensor or a virtual actuator. The former abstracts the data sensed by real sensors into the reading of single, fictitious node, whereas the latter provides a single handle to control a distributed set of actuators. Virtual nodes are specified using the Spidey language of Logical Neighborhoods, by binding some node attributes to previously defined neighborhoods. Communication support for virtual actuators is provided by the Logical Neighborhood routing layer. The routing scheme in [Ciciriello et al. 2007] is used to support virtual sensors, with further automatic customizations performed by the Spidey compiler to enable in-network aggregation.

8. APPLYING THE TAXONOMY: A VIEW OF THE STATE OF THE ART & RESEARCH DIRECTIONS

In this section, we take a snapshot of the current state of the art in WSN programming approaches, by classifying the systems we described thus far along the dimensions we identified in our taxonomy. The systems and corresponding references in the literature are summarized in Figure 32. Figure 33 and 34 map the systems on the taxonomy of language issues presented in Section 5. Figure 35 maps the systems on the architectural issues discussed in Section 6. Finally, Figure 36 concludes by mapping the systems back to the application requirements distilled in the taxonomy of WSN applications we presented in Section 2. In discussing these mappings, we take the opportunity to draw some general considerations.

Figures 33 and 34 suggest the following considerations about language aspects:

—Making communication implicit is a common design choice to raise the level of abstraction. Relieving programmers from the burden of dealing with message parsing and serialization is indeed fundamental to make application development more rapid and less error-prone. Not surprisingly, the few proposals forcing programmers to deal directly with communication rely on message passing primitives, and are therefore closer to the communication pattern made available by the operating system. On the other hand, the use of messages in the programming model does not necessarily entail explicit communication. Systems such as Flask and DSWare do embody some notion of message to provide access to data, yet they hide communication from programmers to a great extent. In these systems, messages are used at a higher-level of abstraction, essentially as containers

Programming Abstraction	References
Abstract Regions	[Welsh and Mainland 2004]
Abstract Task Graph	[Bakshi et al. 2005; Mottola et al. 2007]
Active Messages/nesC	[Culler et al. 2001; Gay et al. 2003]
Agilla	[Fok et al. 2005]
Cougar	[Yao and Gehrke 2002]
DSWare	[Li et al. 2004]
EnviroSuite/ EnviroTrack	[Abdelzaher et al. 2004; Luo et al. 2006]
FACTS	[Terfloth et al. 2006]
Flask	[Mainland et al. 2008]
Generic Role Assignment	[Frank and Römer 2005; 2006]
Hood	[Whitehouse et al. 2004]
Kairos	[Gummadi et al. 2005]
Logical Neighborhoods	[Mottola and Picco 2006a; 2006b]
MacroLab	[Hnat et al. 2008]
Market-based programming	[Mainland et al. 2004]
MiLAN	[Heinzelman et al. 2004]
Pieces	[Liu et al. 2003]
Pleiades	[Kothari et al. 2007]
Regiment	[Newton and Welsh 2004; Newton et al. 2007]
RuleCaster	[Bischoff and Kortuem 2007]
SensorWare	[Boulis et al. 2003; Boulis et al. 2007]
Spatial Programming	[Borcea et al. 2004]
SINA	[Shen et al. 2001]
snBench	[Ocean et al. 2006]
Snlog	[Chu et al. 2007]
TeenyLIME	[Costa et al. 2006; 2007]
TinyDB	[Madden et al. 2005]
Virtual Nodes	[Ciciriello et al. 2006]

Fig. 32. Literature references related to the programming approaches surveyed.

of information.

—There appears to be a relationship between the communication scope (Figure 33) and the data access model (Figure 34) provided by a given programming approach. Essentially all systems providing a system-wide communication scope adopt a database access model. Similarly, systems supporting multi-hop groups usually export a data sharing model, with the only exception of Logical Neighborhoods. Behind the relationship between communication scope and data access model is the objective of providing the programmer with higher levels of abstraction as the communication span approaches the entire system. At an extreme, when the entire system is abstracted away, it can be regarded just like any other data source and therefore accessed like a database. The singularity of Logical Neighborhoods is motivated by its role as a building-block for higher-level functionality. In this respect, it is the operating system interface (i.e., the message passing facility) whose abstraction level is raised, instead of the application programming interface.
Although we noted that the communication scope implies the data access model, the vice versa does not hold. Indeed, the data sharing model (unlike the database one) can be useful also when the communication scope is limited to the physical neighborhood, as witnessed by FACTS, Hood, Kairos, Pleiades, Snlog, and TeenyLime.

—Interestingly, another relationship exists between the communication scope and the computation scope. Looking at the corresponding columns in Figure 33

Programming Abstraction	Communication		
	Scope	Addressing	Awareness
Abstract Regions	Multi-hop group, Non-connected	Logical	Implicit
Abstract Task Graph	Multi-hop group, Non-connected	Logical	Implicit
Active Messages/nesC	Physical neighborhood	Physical	Explicit
Agilla	Routing dependent	Physical	Implicit
Cougar	System-wide	Logical	Implicit
DSWare	System-wide	Physical	Implicit
EnviroSuite/ EnviroTrack	Multi-hop group, Connected	Logical	Implicit
FACTS	Physical neighborhood	Physical	Implicit
Flask	Routing dependent	Routing	Implicit
Generic Role Assignment	Multi-hop group, Connected	Logical	Implicit
Hood	Physical neighborhood	Logical	Implicit
Kairos	Physical neighborhood	Physical	Implicit
Logical Neighborhoods	Multi-hop group, Non-connected	Logical	Explicit
MacroLab	System-wide	Physical	Implicit
Market-based programming	System-wide	Logical	Implicit
MiLAN	Routing dependent	Physical	Explicit
Pieces	Multi-hop group, Connected	Logical	Implicit
Pleiades	Physical neighborhood	Physical	Implicit
Regiment	Multi-hop group, Connected	Logical	Implicit
RuleCaster	Multi-hop group, Connected	Logical	Implicit
SensorWare	Physical neighborhood	Physical	Implicit
Spatial Programming	Multi-hop group Non-connected	Logical	Implicit
SINA	System-wide	Logical	Implicit
snBench	Multi-hop group, Non-connected	Logical	Implicit
Snlog	Physical neighborhood	Physical	Implicit
TeenyLIME	Physical neighborhood	Physical	Implicit
TinyDB	System-wide	Physical	Implicit
Virtual Nodes	Multi-hop group, Non-connected	Logical	Implicit

Fig. 33. Mapping WSN programming abstractions to the taxonomy in Figure 6—language aspects dealing with Communication.

and 34, a global computation scope always implies a system-wide communication scope and, similarly, a group computation scope always implies a multi-hop group communication scope. This is natural, as the ability to affect nodes through computation implies the ability to restrict communication to such nodes. However, once more, the reverse does not necessarily hold. While all systems supporting system-wide communication naturally support a global computation scope, there are systems (i.e., ATaG, GRA, and Logical Neighborhoods) that support a group communication scope but resort to local computation.
It is interesting to note that these aspects are not captured by alternative classifications based on the notion of *macroprogramming* and/or a simple distinction among node-, group-, and network-level approaches [Sugihara and Gupta 2008]. In this case, the two aspects of communication and computation are fused together, resulting in the inability to sharply distinguish between these orthogonal aspects. Therefore, systems with rather distinct characteristics (e.g., Kairos and Regiment, or Abstract Regions and Hood) are classified under the same umbrella.

—Finally, the computation scope also bears some influence on the programming paradigm adopted. All the systems with a global computational scope adopt a declarative approach. Indeed, this choice provides a great expressive power and enables very concise descriptions of the application behavior. At the other extreme, imperative approaches are common when the computation is local and therefore affects only individual nodes, a choice that mirrors the mainstream approach to developing distributed applications. However, declarative approaches

Programming Abstraction	Computation Scope	Data Access Model	Programming Paradigm
Abstract Regions	Group	Data sharing	Imperative, Sequential
Abstract Task Graph	Local	Data sharing	Hybrid
Active Messages/nesC	Local	Message passing	Imperative, Event-driven
Agilla	Local	Mobile code	Imperative, Sequential
Cougar	Global	Database	Declarative, SQL-like
DSWare	Global	Message passing	Declarative, SQL-like
EnviroSuite/ EnviroTrack	Group	Data sharing	Imperative, Event-driven
FACTS	Local	Data sharing	Declarative, Rule-based
Flask	Local	Message passing	Declarative, Functional
Generic Role Assignment	Local	Data sharing	Declarative, Special-purpose
Hood	Local	Data sharing	Imperative, Event-driven
Kairos	Local	Data sharing	Imperative, Sequential
Logical Neighborhoods	Local	Message passing	Declarative, Special-purpose
MacroLab	Global	Data sharing	Imperative, Sequential
Market-based programming	Global	Data sharing	Declarative, Special-purpose
MiLAN	Local	Message passing	Imperative, Event-driven
Pieces	Group	Data sharing	Imperative, Event-driven
Pleiades	Local	Data sharing	Imperative, Sequential
Regiment	Group	Data sharing	Declarative, Functional
RuleCaster	Group	Data sharing	Declarative, Rule-based
SensorWare	Local	Mobile code	Imperative, Sequential
Spatial Programming	Group	Mobile code	Imperative, Sequential
SINA	Global	Database/Mobile code	Hybrid
snBench	Group	Data sharing	Declarative, Functional
Snlog	Local	Data sharing	Declarative, Rule-based
TeenyLIME	Local	Data sharing	Imperative, Event-driven
TinyDB	Global	Database	Declarative, SQL-like
Virtual Nodes	Group	Data sharing	Imperative, Event-driven

Fig. 34. Mapping WSN programming abstractions to the taxonomy in Figure 6—language aspects dealing with Computation Scope, Data Access Model, and Programming Paradigm.

(e.g., FACTS, GRA, and Snlog) targeting local computations also exist.

As for architectural aspects, Figure 35 prompts the following remarks:

—The diversity of WSN applications we pointed out in Section 2 is likely to require an overarching approach where different programming abstractions collaborate into a single, coherent framework [Embedded WiSeNts Project]. Unfortunately, very few programming solutions (i.e., GRA, Hood, and Logical Neighborhoods) are designed as building blocks, meant to work in collaboration with others. Although most existing approaches are well-suited to particular application domains, they lack the ability to be extended by or composed with others. The modularization of functionality into building blocks that provide generic, reusable support for higher-level functionality appears to be an open research issue and, possibly, an enabling factor for a wider adoption of WSN technology.

—It is often said that WSNs greatly benefit from cross-layer solutions or, more generally, from the ability to manipulate low-level aspects in order to optimize resource consumption [Akyildiz et al. 2002]. Unfortunately, this trend is not reflected in the current state of the art, where only a handful of systems provides some form of access to the layers underlying the programming one. The reason probably lies in the difficulty faced by programmers in understanding how fine-grained tuning of low-level parameters can affect the overall system performance,

Programming Abstraction	Programming Support	Layer Focus	Low-level Configuration	Execution Environment
Abstract Regions	Holistic	Application	Interface	TOSSIM
Abstract Task Graph	Holistic	Application	Fixed	JiST SWANS, SunSPOT
Active Messages/nesC	Holistic	Vertical	Interface	All TinyOS platforms
Agilla	Holistic	Application	Fixed	Mica2
Cougar	Holistic	Application	Fixed	NS-2
DSWare	Holistic	Application	Fixed	GlomoSim
EnviroSuite/ EnviroTrack	Holistic	Application	Fixed	Mica2, XSM
FACTS	Holistic	Vertical	Fixed	ESB, ScatterWeb
Flask	Holistic	Vertical	Fixed	TMote Sky
Generic Role Assignment	Building block	Vertical	Fixed	jSIM
Hood	Building block	Vertical	Interface	Mica2
Kairos	Holistic	Application	Fixed	Mica2, Mica2Dot
Logical Neighborhoods	Building block	Vertical	Interface	TMote Sky, Mica2
MacroLab	Holistic	Application	Fixed	TMote Sky
Market-based programming	Holistic	Application	Fixed	TOSSIM
MiLAN	Holistic	Application	Interface	N/A
Pieces	Holistic	Application	Fixed	Custom simulator
Pleiades	Holistic	Application	Fixed	TelosB
Regiment	Holistic	Application	Fixed	Custom simulator
RuleCaster	Holistic	Application	Fixed	N/A
SensorWare	Holistic	Application	Fixed	iPAQ, XScale
Spatial Programming	Holistic	Application	Interface	iPAQ
SINA	Holistic	Application	Fixed	Custom simulator
snBench	Holistic	Application	Fixed	Custom simulator
Snlog	Holistic	Vertical	Fixed	TMote Sky
TeenyLIME	Holistic	Vertical	Fixed	TMote Sky Chess MyriaNed
TinyDB	Holistic	Application	Fixed	Mica2
Virtual Nodes	Holistic	Application	Fixed	Mica2

Fig. 35. Mapping WSN programming abstractions to the taxonomy in Figure 27—architectural aspects.

and by the absence of a general, agreed-upon interface to low-level layers. However, the absence of such "hooks" is likely to yield systems that are too rigid, and that may render the programming solutions unable to adapt to different needs.

Architectural issues, however, are not entirely separated from language ones:

—All systems meant to support "vertical" development across all layers feature a local computation scope. Indeed, developing low-level mechanisms demands control over the behavior of individual nodes (e.g., as in the case of routing), therefore inherently clashing with the perspective adopted by group and global computation, where nodes tend to disappear into higher-level aggregates.

—Dually, none of the approaches characterized by a global computation scope feature hooks into the lower levels of the stack. The level of abstraction provided in these cases is usually too high to accommodate a similar functionality without affecting the overall programming framework.

An important question is to what extent the current state of the art in programming covers the needs of WSN applications. As we pointed out in the introduction, as of today only few real-world deployments leverage high-level programming abstractions [Whitehouse et al. 2004; Buonadonna et al. 2005], and among these only

Programming Abstraction	Goal	Interaction Pattern	Mobility	Space	Time
Abstract Regions	Sense-only	Depending on region	Static	Regional	Periodic/ Event-triggered
Abstract Task Graph	Sense-and-react	Many-to-many	Static	Regional	Periodic
Active Messages/ nesC	Sense-only	Many-to-many	Static	Regional	Periodic/ Event-triggered
Agilla	Sense-only	Many-to-many	Static	Regional	Event-triggered
Cougar	Sense-only	Many-to-one	Static	Global	Periodic
DSWare	Sense-only	Many-to-one	Static	Regional	Event-triggered
EnviroSuite/ EnviroTrack	Sense-only	Many-to-one	Static	Regional	Event-triggered
FACTS	Sense-only	Many-to-many	Static	Regional	Event-triggered
Flask	Sense-only	Many-to-many	Static	Global	Periodic/ Event-triggered
Generic Role Assignment	Sense-only	Many-to-many	Static	Global	Periodic/ Event-triggered
Hood	Sense-only	Many-to-many	Static	Regional	Periodic
Kairos	Sense-only	Many-to-many	Static	Global	Periodic/ Event-triggered
Logical Neighborhoods	Sense-and-react	One-to-many	Static	Regional	Periodic/ Event-triggered
MacroLab	Sense-only	Many-to-one	Static	Global	Periodic/ Event-triggered
Market-based programming	Sense-only	Many-to-one	Static	Global/ Regional	Periodic
MiLAN	Sense-only	Many-to-one	Static	Global	Periodic
Pieces	Sense-only	Many-to-one	Static	Regional	Event-triggered
Pleiades	Sense-only	Many-to-many	Static	Global	Periodic/ Event-triggered
Regiment	Sense-only	Many-to-one	Static	Regional	Event-triggered
RuleCaster	Sense-and-react	Many-to-many	Static	Regional	Event-triggered
SensorWare	Sense-only	Many-to-one	Static	Global	Periodic
Spatial Programming	Sense-only	Many-to-one	Static	Regional	Event-triggered
SINA	Sense-only	Many-to-one	Static	Global	Periodic
snBench	Sense-only	Many-to-one	Static	Regional	Event-triggered
Snlog	Sense-only	Many-to-many	Static	Global	Periodic/ Event-triggered
TeenyLIME	Sense-only/ Sense-and-react	Many-to-many	Static	Regional	Periodic/ Event-triggered
TinyDB	Sense-only	Many-to-one	Static	Global	Periodic
Virtual Nodes	Sense-and-react	Many-to-many	Static	Regional	Periodic/ Event-triggered

Fig. 36. Mapping WSN programming abstractions to the application taxonomy in Figure 1.

the deployment by Ceriotti et al. [2009] has been running continuously for months. However, the mapping of programming abstraction onto applications we provide in Figure 36 helps understanding what kinds of applications have been targeted so far. Albeit somewhat academical in nature, this helps identifying areas not fully covered by the current state of the art. Indeed, the mapping highlights how current approaches are definitely skewed in the applications they target:

—Only a small fraction of existing solutions appears to be appropriate for sense-and-react applications. As we already pointed out, the latter usually require many-to-many interactions, as well as continuous monitoring limited to specific portions of the system. The current state of the art appears in general ill-suited to these requirements, in that most systems privilege many-to-one interactions and/or a rather rigid definition of communication scope, appropriate for applica-

tions revolving around pure data collection. Interestingly, the systems surveyed appear to support regional interactions to a greater extent in applications with event-triggered processing.

—A large body of research on communication issues in mobile sensor networks has been carried out [Wang et al. 2007; Al-Karaki and Kamal 2004]. Nonetheless, none of the programming solutions considered in our survey specifically addresses applications with mobile nodes or sinks. The requirements to meet in these scenarios are, however, quite different from the challenges in static applications. Location is usually of paramount importance, the network topology is even more dynamic, and delay-tolerant interactions often are the only way to achieve communication. Therefore, programmers must implement, on a per-application basis, mechanisms such as neighbor discovery as well as store-and-forward mechanisms. Ideally, higher-level programming abstractions should be developed to shield programmers from these aspects.

9. CONCLUSIONS AND OUTLOOK

Wireless sensor networks are a powerful technology with the potential to make the vision of a truly pervasive computing environment become a reality. However, despite the advancements bringing smaller devices, more computation and communication power, and an ever-increasing range of sensors and actuators, *programming* these myriads of devices remains the weakest link in the chain that leads to rapid and reliable WSN deployments. This situation, currently hampering a wider acceptance of this technology, will be overcome only when programming platforms will be simple enough to be used by a domain expert, and yet provide acceptable and predictable levels of performance and reliability.

The research field is still relatively far from this goal, although a number of approaches have already been proposed. In this paper, we provided a systematic treatment of the topic, proposing a taxonomy that identifies the fundamental dimensions characterizing and distinguishing the various approaches. We extensively surveyed the state of the art in programming WSNs, by classifying existing solutions against our taxonomy as well as the application requirements typically posed by WSNs. This comprehensive view of current efforts in simplifying the programming of WSNs was also the opportunity to identify at a glance areas that require more research effort.

We conclude this paper by pointing out a few additional open research issues that, albeit not germane to our taxonomy, are however strongly related to programming WSNs and for which solutions are sorely missing. Here we briefly comment on those we believe are most significant:

—*Tolerance to failures.* Various types of hardware faults are often observed in real deployments [Werner-Allen et al. 2006]. However, most of the programming approaches we examined provide only limited guarantees in these exceptional circumstances. Nodes running out of battery power, for instance, are eventually recognized and excluded from processing, although no time bounds are provided w.r.t. when this happens. Transient faults, e.g., those arising from sensors temporarily providing erroneous readings [Sharma et al. 2007], are usually not considered. Little or no support is offered to programmers for dealing with these

situations. As a result, they are frequently forced to implement dedicated mechanisms on a per-application basis. High-level programming frameworks where faults are a first-class notion are necessary to ease development of WSN applications targeted to harsh environments. For instance, programming constructs to identify erroneous sensor readings and temporarily exclude a node from the processing may help programmers in improving the fidelity of data.

—*Debugging and testing.* A few works recently addressed the problem of debugging WSN applications. However, these systems are usually tied to a specific operating system [Krunic et al. 2007; Yang et al. 2007], or are independent of the programming language [Ringwald et al. 2007]. Consequently, they may signal to programmers that something is not working, but without any detailed clue regarding the cause of the problem or what part of the application might be the culprit. On the other hand, none of the programming systems we considered in this paper provides dedicated support for testing the behavior of applications built using them. Further research is required to augment high-level programming abstractions with the mechanisms necessary to validate the system.

—*Evaluation methodology.* In the current state of the art, programming frameworks are usually evaluated quantitatively w.r.t. system performance. Most often, this is achieved using ad-hoc examples of application- or system-level functionality, and some form of simulation. This practice, however, is largely unsatisfactory, along two complementary dimensions:

  —The examples used to evaluate the system performance span drastically different functionality and are implemented at considerably different levels of details. This may bear great impact on the outcome of the evaluation, ultimately rendering the experiments not reproducible and the results not comparable. To address this issue, the WSN community may take inspiration from other fields (e.g., databases) and conceive a set of clearly-defined metrics and benchmark functionality to be used in the evaluation of the system performance. These may be based on staple WSN mechanisms, e.g., data collection and time synchronization, to widen the validity of the results obtained.

  —Although system performance is an important aspect, it is only half of what it takes to assess the effectiveness of a programming approach. The gains brought to the programmers' productivity must also be evaluated. However, to investigate this aspect, the only quantitative metric used in most of the existing approaches is the number of lines of code. Among other things, this makes it almost impossible to compare solutions based on different programming paradigms. This is a problem by itself, and the software engineering community has long been working on code metrics [Fenton and Pfleeger 1998]. Because of the specific characteristics of WSN programming, however, dedicated metrics are likely to be required.

—*Real-world use.* As already pointed out, real-world deployments based on high-level programming frameworks are rarely reported in the literature. When it comes to developing real-life WSN applications, programmers—often computer science or networking researchers themselves—prefer to spend the additional effort required to use low-level abstractions and keep every single bit under control, rather than endeavor to use programming frameworks they cannot fully trust.

However, this is not a sustainable strategy, especially if application development is to be placed directly into the hands of the domain experts. Therefore, more effort is required in moving WSN programming abstractions from the labs to real deployments, not only to evaluate their effectiveness concretely, but also by gathering fundamental feedback in steering the design of the next generation of programming solutions.

Acknowledgments. This work is partially supported by the Autonomous Province of Trento under the call for proposals "Major Projects 2006" (project ACube), by the Cooperating Objects Network of Excellence (CONET) under EU contract FP7-2007-2-224053, and by the Swedish Foundation for Strategic Research (SSF).

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 *Proc. of the* 24^{th} *Int. Conf. on Distributed Computing Systems (ICDCS).*

Abrach, H., Bhatti, S., Carlson, J., Dai, H., Rose, J., Sheth, A., Shucker, B., Deng, J., and Han, R. 2003. MANTIS: System support for MultimodAl NeTworks of In-situ Sensors. In *Proc. of the* 2^{nd} *Int. Conf. on Wireless Sensor Networks and Applications (WSNA).*

Aduino Sensor Node Platform. `www.arduino.cc`.

Akyildiz, I., Su, W., Sankarasubramaniam, Y., and Cayirci, E. 2002. A survey on sensor networks. *IEEE Communication Mag. 40,* 8.

Akyildiz, I. F. and Kasimoglu, I. H. 2004. Wireless sensor and actor networks: Research challenges. *Ad Hoc Networks Journal 2,* 4.

Al-Karaki, J. and Kamal, A. E. 2004. Routing techniques in wireless sensor networks: A survey. *IEEE Wireless Communications 11,* 6.

Arora, A., Dutta, P., Bapat, S., Kulathumani, V., Zhang, H., Naik, V., Mittal, V., Cao, H., Demirbas, M., Gouda, M., Choi, Y., Herman, T., Kulkarni, S., Arumugam, U., Nesterenko, M., Vora, A., and Miyashita, M. 2004. A line in the sand: A wireless sensor network for target detection, classification, and tracking. *Computer Networks 46,* 5.

Avancha, S., Joshi, A., and Finin, T. 2002. Enhanced service discovery in bluetooth. *IEEE Computer 35,* 6.

Bakshi, A., Pathak, A., and Prasanna, V. K. 2005. System-level support for macroprogramming of networked sensing applications. In *Int. Conf. on Pervasive Systems and Computing (PSC).*

Bakshi, A., Prasanna, V. K., Reich, J., and Larner, D. 2005. The Abstract Task Graph: A methodology for architecture-independent programming of networked sensor systems. In *Workshop on End-to-end Sense-and-respond Systems (EESR).*

Baldus, H., Klabunde, K., and Müsch, G. 2004. Reliable set-up of medical body-sensor networks. In *Proc. of* 1^{st} *European Workshop on Wireless Sensor Networks (EWSN).*

Baronti, P., Pillai, P., Chook, V. W. C., Chessa, S., Gotta, A., and Hu, Y. F. 2007. Wireless sensor networks: A survey on the state of the art and the 802.15.4 and ZigBee standards. *Comput. Commun. 30,* 7.

Batalin, M. A., Sukhatme, G. S., and Hattig, M. 2004. Mobile robot navigation using a sensor network. In *Proc. of the Int. Conf. on Robotics and Automation (ICRA).*

Bischoff, U. and Kortuem, G. 2007. A state-based programming model and system for wireless sensor networks. In *Proc. of the* 3^{rd} *Int. Workshop on Sensor Networks and Systems for Pervasive Computing (PerSens).*

Blum, B., Nagaraddi, P., Wood, A., Abdelzaher, T., Son, S., and Stankovic, J. 2003. An entity maintenance and connection service for sensor networks. In *Proc. of the* 1^{st} *Int. Conf. on Mobile Systems, Applications and Services (MobiSys).*

Body Sensor Network Nodes. vip.doc.ic.ac.uk/bsn/index.php?article=926.

Borcea, C., Intanagonwiwat, C., Kang, P., Kremer, U., and Iftode, L. 2004. Spatial programming using smart messages: Design and implementation. In *Proc. of the* 24^{th} *Int. Conf. on Distributed Computing Systems (ICDCS).*

Boulis, A., Han, C.-C., Shea, R., and Srivastava, M. B. 2007. SensorWare: Programming sensor networks beyond code update and querying. *Elsevier Pervasive and Mobile Computing Journal 3,* 4.

Boulis, A., Han, C.-C., and Srivastava, M. B. 2003. Design and implementation of a framework for efficient and programmable sensor networks. In *Proc. of the* 1^{st} *Int. Conf. on Mobile Systems, Applications and Services (MobiSys).*

BTNode. www.btnode.ethz.ch.

Buonadonna, P., Gay, D., Hellerstein, J., Hong, W., and Madden, S. 2005. TASK: Sensor network in a box. In *Proc. of the* 2^{th} *European Conf. on Wireless Sensor Networks (EWSN).*

Burrell, J., Brooke, T., and Beckwith, R. 2004. Vineyard computing: Sensor networks in agricultural production. *IEEE Pervasive Computing 3,* 1.

Cao, Q., Abdelzaher, T., , Stankovic, J., and He, T. 2008. The LiteOS operating system: Towards unix-like abstractions for wireless sensor networks. In *Proc. of the* 7^{th} *ACM/IEEE Int. Conf. on Information Processing in Sensor Networks (IPSN).*

Ceriotti, M., Mottola, L., Picco, G. P., Murphy, A. L., Guna, S., Corrà, M., Pozzi, M., Zonta, D., and Zanon, P. 2009. Monitoring heritage buildings with wireless sensor networks: The Torre Aquila deployment. In *Proc. of the* 8^{th} *ACM/IEEE Int. Conf. on Information Processing in Sensor Networks (IPSN).* Best Paper Award.

Cha, H., Choi, S., Jung, I., Kim, H., Shin, H., Yoo, J., and Yoon, C. 2007. RETOS: Resilient, expandable, and threaded operating system for wireless sensor networks. In *Proc. of the* 6^{th} *Int. Conf. on Information Processing in Sensor Networks (IPSN).*

Chatzigiannakis, I., Mylonas, G., and Nikoletseas, S. 2007. 50 ways to build your application: A survey of middleware and systems for wireless sensor networks. In *Proc. of the Int. Conf. on Emerging Technologies and Factory Automation (ETFA).*

Chu, D., Popa, L., Tavakoli, A., Hellerstein, J., Levis, P., Shenker, S., and Stoica, I. 2007. The design and implementation of a declarative sensor network system. In *Proc. of the* 5^{th} *Int. Conf. on Embedded Networked Sensor Systems (SenSys),.*

Ciciriello, P., Mottola, L., and G.P. Picco. 2006. Building virtual sensors and actuator over Logical Neighborhoods. In *Proc. of the* 1^{st} *ACM Int. Workshop on Middleware for Sensor Networks (MidSens).*

Ciciriello, P., Mottola, L., and Picco, G. P. 2007. Efficient routing from multiple sources to multiple sinks in wireless sensor networks. In *Proc. of* 4^{th} *European Conf. on Wireless Sensor Networks (EWSN).*

CONET. Research roadmap of the cooperating objects network of excellence. www.cooperating-objects.eu/roadmap/.

Costa, P., Coulson, G., Gold, R., Lad, M., Mascolo, C., Mottola, L., Picco, G. P., Sivaharan, T., Weerasinghe, N., and Zachariadis, S. 2007. The RUNES middleware for networked embedded systems and its application in a disaster management scenario. In *Proc. of the* 5^{th} *Int. Conf. on Pervasive Communications (PerCom).*

Costa, P., Mottola, L., Murphy, A. L., and Picco, G. P. 2006. TeenyLime: Transiently shared tuple space middleware for wireless sensor networks. In *Proc. of the* 1^{st} *Int. Workshop on Middleware for Sensor Networks (MidSens).*

Costa, P., Mottola, L., Murphy, A. L., and Picco, G. P. 2007. Programming wireless sensor networks with the TeenyLime middleware. In *Proc. of the* 8^{th} *ACM/USENIX Int. Middleware Conf.*

Crossbow Tech. www.xbow.com.

Culler, D., Hill, J., Buonadonna, P., Szewczyk, R., and Woo, A. 2001. A network-centric approach to embedded software for tiny devices. In *Proc. of the* 1^{st} *Int. Workshop on Embedded Software (EMSOFT).*

Demirkol, I., Ersoy, C., and Alagoz, F. 2006. MAC protocols for wireless sensor networks: A survey. *IEEE Communications Magazine 44,* 4.

Dermibas, M. 2005. Wireless sensor networks for monitoring of large public buildings. Tech. Report, University of Buffalo. Available at `www.cse.buffalo.edu/tech-reports/2005-26.pdf`.

Deshpande, A., Guestrin, C., and Madden, S. 2005. Resource-aware wireless sensor-actuator networks. *IEEE Data Engineering 28,* 1.

Dunkels, A., Grönvall, B., and Voigt, T. 2004. Contiki - A lightweight and flexible operating system for tiny networked sensors. In *Proc. of* 1^{st} *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 *Proc. of the* 4^{th} *Int. Conf. on Embedded Networked Sensor Systems (SenSys).*

Elson, J. and Roemer, K. 2003. Wireless sensor networks: A new regime for time synchronization. *SIGCOMM Comput. Commun. Rev. 33,* 1.

Embedded WiSeNts Project. Embedded WiSeNts Research Roadmap. `www.embedded-wisents.org/dissemination/roadmap.html`.

Eswaran, A., Rowe, A., and Rajkumar, R. 2005. Nano-rk: An energy-aware resource-centric rtos for sensor networks. In *Proc. of the* 26^{th} *International Real-Time Systems Symposium (RTSS).*

Eugster, P., Felber, P., Guerraoui, R., and Kermarrec, A.-M. 2003. The many faces of Publish/Subscribe. *ACM Computing Surveys 2,* 35.

Eyes WSN Nodes. `www.eyes.eu.org`.

Fenton, N. E. and Pfleeger, S. L. 1998. *Software Metrics: A Rigorous and Practical Approach.* PWS Publishing Co., Boston, MA, USA.

Fok, C.-L., Roman, G.-C., and Lu, C. 2005. Rapid development and flexible deployment of adaptive wireless sensor network applications. In *Proc. of the* 25^{th} *Int. Conf. on Distributed Computing Systems (ICDCS).*

Frank, C. and Römer, K. 2005. Algorithms for generic role assignment in wireless sensor networks. In *Proc. of the* 3^{rd} *ACM Conf. on Embedded Networked Sensor Systems (SenSys).*

Frank, C. and Römer, K. 2006. Solving generic role assignment exactly. In *Proc. of the* 14^{th} *Int. Workshop on Parallel and Distributed Real-Time Systems (WPDRTS).*

Fuggetta, A., Picco, G. P., and Vigna, G. 1998. Understanding code mobility. *IEEE Transactions Softw. Eng. 24,* 5.

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 *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI).*

Gelernter, D. 1985. Generative communication in Linda. *ACM Computing Surveys 7,* 1.

Gu, L. and Stankovic, J. A. 2006. T-Kernel: Providing reliable OS support to wireless sensor networks. In *Proc. of the* 4^{th} *Int. Conf. on Embedded Networked Sensor Systems (SenSys).*

Gummadi, R., Gnawali, O., and Govindan, R. 2005. Macro-programming wireless sensor networks using Kairos. In *Proc. of the* 1^{st} *Int. Conf. on Distributed Computing in Sensor Systems (DCOSS).*

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

Han, C.-C., Kumar, R., Shea, R., Kohler, E., and Srivastava, M. 2005. A dynamic operating system for sensor nodes. In *Proc. of the* 3^{rd} *Int. Conf. on Mobile Systems, Applications, and Services (MobiSys).*

Hartung, C., Han, R., Seielstad, C., and Holbrook, S. 2006. FireWxNet: A multi-tiered portable wireless system for monitoring weather conditions in wildland fire environments. In *Proc. of the* 4^{th} *Int. Conf. on Mobile Systems, Applications and Services (MobiSys).*

Heinzelman, W. B., Murphy, A. L., Carvalho, H. S., and Perillo, M. A. 2004. Middleware linking applications and networks. *IEEE Network 18.*

Henricksen, K. and Robinson, R. 2006. A survey of middleware for sensor networks: state-of-the-art and future directions. In *Proc. of the 1st ACM Int. Workshop on Middleware for Sensor Networks (MidSens)*.

Hill, J., Szewczyk, R., Woo, A., Hollar, S., Culler, D., and Pister, K. 2000. System architecture directions for networked sensors. In *Proc. of the 9th Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*.

Hnat, T., Sookoor, T., Hooimeijer, P., Weimer, W., and Whitehouse, K. 2008. Macrolab: A vector-based macroprogramming framework for cyber-physical systems. In *Proc. of the 6th Int. Conf. on Embedded Networked Sensor Systems (SenSys)*.

Hughes, D., Greenwood, P., Blair, G., Coulson, G., Grace, P., Pappenberger, F., Smith, F., and Beven, K. 2007. An experiment with reflective middleware to support grid-based flood monitoring. *Concurrency and Computation: Practice and Experience 23,* 4.

Intanagonwiwat, C., Govindan, R., Estrin, D., Heidemann, J., and Silva, F. 2003. Directed Diffusion for wireless sensor networking. *IEEE/ACM Transactions Networking 11,* 1.

IST CRUISE Project. Flood detection using sensor networks. `www.ist-cruise.eu/cruise/business-deck/wsns-applications/flood-detection-1`.

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. *SIGPLAN Not. 37,* 10.

Kim, S., Son, S. H., Stankovic, J. A., Li, S., and Choi, Y. 2003. Safe: A data dissemination protocol for periodic updates in sensor networks. In *Proc. of the Int. Workshop on Data Distribution for Real-time Systems*.

Kothari, N., Gummadi, R., Millstein, T., and Govindan, R. 2007. Reliable and efficient programming abstractions for wireless sensor networks. In *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*.

Krishnamurthy, L., Adler, R., Buonadonna, P., Chhabra, J., Flanigan, M., Kushalnagar, N., Nachman, L., and Yarvis, M. 2005. Design and deployment of industrial sensor networks: experiences from a semiconductor plant and the north sea. In *Proc. of the 3rd Int. Conf. on Embedded Networked Sensor Systems (SenSys)*.

Krunic, V., Trumpler, E., and Han, R. 2007. NodeMD: Diagnosing node-level faults in remote wireless sensor systems. In *Proc. of the 5th Int. Conf. on Mobile Systems, Applications and Services (MobiSys)*.

Lampe, M. and Strassner, M. 2003. The potential of RFID for moveable asset management. In *Proc. of the Workshop on Ubiquitous Commerce at UbiComp*.

Langendoen, K. and Reijers, N. 2003. Distributed localization in wireless sensor networks: A quantitative comparison. *Computer Networks 43,* 4.

Levis, P. and Culler, D. 2002. Maté: A tiny virtual machine for sensor networks. In *Proc. of the 10th Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X)*.

Levis, P., Lee, N., Welsh, M., and Culler, D. 2002. TOSSIM: Accurate and scalable simulation of entire TinyOS applications. In *Proc. of 5th Symp. on Operating Systems Design and Implementation (OSDI)*.

Levis, P., Madden, S., Gay, D., Polastre, J., Szewczyk, R., Woo, A., Brewer, E., and Culler, D. 2004. The emergence of networking abstractions and techniques in TinyOS. In *Proc. of 1st Symp. on Networked System Design and Implementation (NSDI)*.

Li, S., Lin, Y., S.H. Son, J.A. Stankovic, and Wei, Y. 2004. Event detection services using data service middleware in distributed sensor networks. *Telecommunication Systems 26,* 2.

Li, X.-Y., Wan, P.-J., Wang, Y., and Frieder, O. 2002. Sparse power efficient topology for wireless networks. In *Proc. of the 35th Annual Hawaii International Conference on System Sciences (HICSS)*.

Liu, J., Cheung, P., Zhao, F., and Guibas, L. 2002. A dual-space approach to tracking and sensor management in wireless sensor networks. In *Proc. of the 1st Int. Workshop on Wireless Sensor Networks and Applications (WSNA)*.

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

Liu, T. and Martonosi, M. 2003. Impala: A middleware system for managing autonomic, parallel sensor systems. In *Proc. of the* 9^{th} *SIGPLAN Symposium on Principles and Practice of Parallel Programming.*

Lorincz, K., Malan, D., Fulford-Jones, T., Nawoj, A., Clavel, A., Shnayder, V., Mainland, G., Welsh, M., and Moulton, S. 2004. Sensor networks for emergency response: Challenges and opportunities. *IEEE Pervasive Computing 3,* 4.

Luo, L., Abdelzaher, T. F., He, T., and Stankovic, J. A. 2006. EnviroSuite: An environmentally immersive programming framework for sensor networks. *IEEE Transactions on Embedded Computing Systems 5,* 3.

Luo, L., Huand, C., Abdelzaher, T., and Stankovic, J. 2007. EnviroStore: A cooperative storage system for disconnected operation in sensor networks. In *Proc. of the* 26^{th} *Int. Conf. on Computer Communications (INFOCOM).*

Lymberopoulos, D. and Savvides, A. 2005. XYZ: A motion-enabled, power-aware sensor node platform for distributed sensor network applications. In *Proc. of the* 4^{th} *Int. Symp. on Information Processing in Sensor Networks (IPSN).*

Lynch, J. P. and Loh, K. J. 2006. A summary review of wireless sensors and sensor networks for structural health monitoring. *Shock and Vibration Digest.*

Madden, S., Franklin, M. J., Hellerstein, J. M., and Hong, W. 2003. TAG: A tiny aggregation service for ad-hoc sensor networks. In *Proc. of* 1^{st} *Int. Conf. on Embedded Networked Sensor Systems (SenSys).*

Madden, S., M.J. Franklin, J.M. Hellerstein, and Hong, W. 2005. TinyDB: An acquisitional query processing system for sensor networks. *ACM Transactions on Database Systems 30,* 1.

Mainland, G., Kang, L., Lahaie, S., Parkes, D. C., and Welsh, M. 2004. Using virtual markets to program global behavior in sensor networks. In *Proc. of the* 11^{th} *ACM SIGOPS European Workshop.*

Mainland, G., Morrisett, G., and Welsh, M. 2008. Flask: Staged functional programming for sensor networks. In *Proc. of the* 13^{th} *Int. Conf. on Functional Programming.*

Mainwaring, A., Culler, D., Polastre, J., Szewczyk, R., and Anderson, J. 2002. Wireless sensor networks for habitat monitoring. In *Proc. of the* 1^{st} *ACM Int. Workshop on Wireless Sensor Networks and Applications (WSNA).*

Manzie, C., Watson, H. C., Halgamuge, S. K., and Lim, K. 2005. On the potential for improving fuel economy using a traffic flow sensor network. In *Proc. of the Int. Conf. on Intelligent Sensing and Information Processing.*

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

MeshNetics Tech. `www.meshnetics.com`.

Michahelles, F., Matter, P., Schmidt, A., and Schiele, B. 2003. Applying wearable sensors to avalanche rescue. *Computer and Graphics 27,* 6.

MoteIV. `www.moteiv.com`.

Mottola, L., Pathak, A., Bakshi, A., Picco, G. P., and Prasanna, V. K. 2007. Enabling scope-based interactions in sensor network macroprogramming. In *Proc. of the the* 4^{th} *Int. Conf. on Mobile Ad-Hoc and Sensor Systems (MASS).*

Mottola, L. and Picco, G. P. 2006a. Logical Neighborhoods: A programming abstraction for wireless sensor networks. In *Proc. of the* 2^{nd} *Int. Conf. on Distributed Computing on Sensor Systems (DCOSS).*

Mottola, L. and Picco, G. P. 2006b. Programming wireless sensor networks with Logical Neighborhoods. In *Proc. of the* 1^{st} *Int. Conf. on Integrated Internet Ad hoc and Sensor Networks (InterSense).*

Naik, P. and Sivalingam, K. 2004. A survey of mac protocols for sensor networks. *Chapter in Wireless sensor networks, Kluwer Academic Publishers.*

Newton, R., Arvind, and Welsh, M. 2005. Building up to macroprogramming: An intermediate language for sensor networks. In *Proc. of the* 4^{th} *Int. Symp. on Information Processing in Sensor Networks (IPSN).*

Newton, R., Morrisett, G., and Welsh, M. 2007. The Regiment macroprogramming system. In *Proc. of the* 6^{th} *Int. Conf. on Information Processing in Sensor Networks (IPSN).*

Newton, R. and Welsh, M. 2004. Region streams: Functional macroprogramming for sensor networks. In *Proc. of the* 1^{st} *Int. Workshop on Data Management for Sensor Networks.*

Nitta, C., Pandey, R., and Ramin, Y. 2006. Y-threads: Supporting concurrency in wireless sensor networks. In *Proc. of the* 2^{nd} *Int. Conf. on Distributed Computing on Sensor Systems (DCOSS).*

Ocean, M. J., Bestavros, A., and Kfoury, A. J. 2006. snBench: Programming and virtualization framework for distributed multitasking sensor networks. In *Proc. of the* 2^{nd} *Int. Conf. on Virtual Execution Environments (VEE).*

OnWorld. Emerging wireless research. `www.onworld.com`.

Padhy, P., Dash, R. K., Martinez, K., and Jennings, N. R. 2006. A utility-based sensing and communication model for a glacial sensor network. In *Proc. of the* 5^{th} *Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS).*

Pathak, A., Mottola, L., Bakshi, A., Prasanna, V. K., and Picco, G. P. 2007. Expressing sensor network interaction patterns using data-driven macroprogramming. In *Proc. of the* 3^{rd} *Int. Workshop on Sensor Networks and Systems for Pervasive Computing (PerSens).*

Petriu, E., Georganas, N., Petriu, D., Makrakis, D., and Groza, V. 2000. Sensor-based information appliances. *IEEE Instrumentation and Measurement Mag. 3.*

Polastre, J., Hill, J., and Culler, D. 2004. Versatile low power media access for wireless sensor networks. In *Proc. of the* 2^{nd} *Int. Conf. on Embedded Networked Sensor Systems (SenSys).*

Project SunSPOT. `www.sunspotworld.com`.

Rajendran, V., Obraczka, K., and Garcia-Luna-Aceves, J. J. 2003. Energy-efficient collision-free medium access control for wireless sensor networks. In *Proc. of the* 1^{st} *Int. Conf. on Embedded Networked Sensor Systems (SenSys).*

Rajendran, V., Obraczka, K., and Garcia-Luna-Aceves, J. J. 2006. Energy-efficient, collision-free medium access control for wireless sensor networks. *Wireless Networks 12,* 1.

Ratnasamy, S., Karp, B., Yin, L., Yu, F., Estrin, D., Govindan, R., and Shenker, S. 2002. GHT: A geographic hash table for data-centric storage. In *Proc. of the* 1^{st} *Int. Workshop on Wireless Sensor Networks and Applications (WSNA).*

Ringwald, M., Römer, K., and Vitaletti, A. 2007. Passive inspection of sensor networks. In *Proc. of the the* 3^{rd} *Int. Conf. on Distributed Computing on Sensor Systems (DCOSS).*

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

Rubio, B., Diaz, M., and Troya, J. M. 2007. Programming approaches and challenges for wireless sensor networks. In *Proc. of the* 2^{nd} *Int. Conf. on Systems and Networks Communications (ICSNC).*

ScatterWeb Inc. `www.scatterweb.com`.

Shah, R., Roy, S., Jain, S., and Brunette, W. 2003. Data MULEs: Modeling and analysis of a three-tier architecture for sparse sensor networks. *Elsevier Ad Hoc Networks Journal 1,* 2–3.

Sharma, A., Golubchik, L., and Govindan, R. 2007. On the prevalence of sensor faults in real-world deployments. *Proc. of the* 4^{th} *Sensor, Mesh and Ad Hoc Communications and Networks Conference (SECON).*

Shen, C.-C., Srisathapornphat, C., and Jaikaeo, C. 2001. Sensor information networking architecture and applications. *IEEE Personal Communications 8,* 4.

Sheth, A., Tejaswi, K., Mehta, P., Parekh, C., Bansal, R., Merchant, S., Singh, T., Desai, U. B., Thekkath, C. A., and Toyama, K. 2005. Senslide: A sensor network based landslide prediction system. In *Proc. of the* 3^{rd} *Int. Conf. on Embedded Networked Sensor Systems (SenSys).*

Simon, G., Maróti, M., Lédeczi, A., Balogh, G., Kusy, B., Nádas, A., Pap, G., Sallai, J., and Frampton, K. 2004. Sensor network-based countersniper system. In *Proc. of the* 2^{nd} *Int. Conf. on Embedded Networked Sensor Systems (SenSys).*

Stankovic, J. A., Cao, Q., Doan, T., Fang, L., He, Z., Kiran, R., Lin, S., Son, S., Stoleru, R., and Wood, A. 2005. Wireless sensor networks for in-home healthcare: Potential and challenges. In *Proc. of High Confidence Medical Device Software and SystemsWorkshop (HCMDSS)*.

Sugihara, R. and Gupta, R. K. 2008. Programming models for sensor networks: A survey. *ACM Transactions on Sensor Networks 4,* 2.

Sundararaman, B., Buy, U., and Kshemkalyani, A. D. 2005. Clock synchronization for wireless sensor networks: A survey. *Ad Hoc Networks 3,* 3.

Terfloth, K., Wittenburg, G., and Schiller, J. 2006. FACTS - A rule-based middleware architecture for wireless sensor networks. In *Proc. of the* 1^{st} *Int. Conf. on Communication System Software and Middleware (COMSWARE)*.

Thorstensen, B., Syversen, T., Bjornvold, T., and Walseth, T. 2004. Electronic shepherd—A low-cost, low-bandwidth, wireless network system. In *Proc. of the* 2^{nd} *Int. Conf. on Mobile Systems, Applications, and Services (MobiSys)*.

TinyOS Community Forum. TinyOS TEP 109 - Sensors and Sensor Boards. `www.tinyos.net/tinyos-2.x/doc/txt/tep109.html`.

TinyOS Community Forum. TinyOS TEP 118 - Dissemination. `www.tinyos.net/tinyos-2.x/doc/txt/tep118.html`.

TinyOS Community Forum. TinyOS TEP 119 - Collection. `www.tinyos.net/tinyos-2.x/doc/txt/tep119.html`.

TinyOS Community Forum. TinyOS TEP 126 - CC2420 radio stack. `www.tinyos.net/tinyos-2.x/doc/html/tep126.html`.

van Dam, T. and Langendoen, K. 2003. An adaptive energy-efficient MAC protocol for wireless sensor networks. In *Proc. of the* 1^{st} *Conf. on Embedded Networked Sensor Systems (SenSys)*.

Wang, Q., Zhu, Y., and Cheng, L. 2006. Reprogramming wireless sensor networks: Challenges and approaches. *IEEE Network 20,* 3.

Wang, Y., Dang, H., and Wu, H. 2007. A survey on analytic studies of delay-tolerant mobile sensor networks. *Wireless Communication and Mobile Computing 7,* 10.

WASP Project. `www.wasp-project.org`.

Welsh, M. and Mainland, G. 2004. Programming sensor networks using abstract regions. In *Proc. of* 1^{st} *Symp. on Networked Systems Design and Implementation (NSDI)*.

Werner-Allen, G., Lorincz, K., Johnson, J., Lees, J., and Welsh, M. 2006. Fidelity and yield in a volcano monitoring sensor network. In *Proc. of* 7^{th} *Symp. on Operating Systems Design and Implementation (OSDI)*.

Whitehouse, K., Sharp, C., Brewer, E., and Culler, D. 2004. Hood: A neighborhood abstraction for sensor networks. In *Proc. of* 2^{nd} *Int. Conf. on Mobile Systems, Applications, and Services (MobiSys)*.

WildSensing Project. `www.dcs.bbk.ac.uk/~assent/WILDSENSING/index.html`.

Wittenburg, G., Terfloth, K., Villafuerte, F. L., Naumowicz, T., Ritter, H., and Schiller, J. 2007. Fence monitoring - Experimental evaluation of a use case for wireless sensor networks. In *Proc. of the* 4^{th} *European Conf. on Wireless Sensor Networks (EWSN)*.

Yang, J., Soffa, M. L., Selavo, L., and Whitehouse, K. 2007. Clairvoyant: A comprehensive source-level debugger for wireless sensor networks. In *Proc. of the* 5^{th} *Int. Conf. on Embedded Networked Sensor Systems (SenSys)*.

Yao, Y. and Gehrke, J. 2002. The Cougar approach to in-network query processing in sensor networks. *SIGMOD Rec. 31,* 3.

Ye, W., Heidemann, J., and Estrin, D. 2002. An energy-efficient MAC protocol for wireless sensor networks. In *Proc. of the* 21^{st} *Int. Conf. on Computer Communications (INFOCOM)*.

Zeng, X., Bagrodia, R., and Gerla, M. 1998. GloMoSim: A library for parallel simulation of large-scale wireless networks. In *Proc. of the* 12^{th} *Workshop on Parallel and Distributed Simulation (PADS)*.