Software Support for Application Development in Wireless Sensor Networks

Chien-Liang Fok, Gruia-Catalin Roman, and Chenyang Lu
Washington University in St. Louis

Abstract

Developing software for Wireless sensor networks (WSNs) is difficult because they consist of tiny, fragile, and resource-deprived embedded devices that communicate over low-bandwidth wireless links. Middleware can simplify WSN programming; many middleware packages have been proposed. This chapter provides an overview of WSN middleware. It starts with middleware providing basic building blocks, and then focuses on middleware for mobility in all of its forms: data, entity, user, and code. The chapter ends with a discussion on emerging WSN middleware strategies.

Contents

1 Introduction

Middleware for wireless sensor networks (WSNs) may seem misplaced in a book on mobile middleware. After all, most existing WSNs are installed on static objects like, for example, the nests of the Leach's Storm Petrels on the Great Duck Island [61], three feet underground at the Pickberry Vineyard in California, and on the housing of semiconductor machinery at Intel. Yet, upon closer inspection, it becomes readily apparent why WSNs are relevant. First, WSNs need not be static, they can be just as easily installed on moving objects. For example, in ZebraNet [48], a WSN was installed on a herd of zebras for tracking their movement. As the zebras move, the nodes opportunistically create wireless ad hoc links for distributing code and data updates. Mobile middleware in the form of Impala [54] was used to ensure flexible and adaptable applications in such a dynamic context. Second, there are many applications that involve a WSN interacting with a *mobile user* [29, 44, 57, 48, 47], or tracking a *mobile entity* [18, 32, 38]. Third, as mobile applications mature, they must exhibit *context awareness*, i.e., they must not only understand what their users are doing, but also perceive properties about their environment. This collection of context information must be done in real-time and at resolutions only WSNs are capable of cost-effectively providing. The combination of user, entity, node, data, and code mobility, along with the reliance of traditional mobile devices on WSNs for information about the surrounding physical environment, render WSN middleware relevant to this book.

WSNs are relatively new, having only recently been made possible by advances in MicroElectro-Mechanical-Systems (MEMS), battery technology, wireless technology, and system-on-chip (SoC) designs. They consist of tiny autonomous nodes each containing a variety of sensors, microprocessor, memory, battery, and wireless communication interface. They are cheaper to deploy than wired networks and can potentially offer greater reliability and agility by routing data through a wireless mesh rather than fixed hard-wired links. Since each node is nominal, WSNs can scale to thousands of nodes offering higher sensing resolutions than was previously possible. WSNs are currently used for habitat monitoring on the Great Duck Island [61] and James Reserve [27], and for microclimate research around redwood trees [20]. Emerging uses include surveillance [38], emergency medical care [55], and structural integrity monitoring [77]. In the future, WSNs may help automate highways [43] and coordinate military operations [53, 72]. As WSNs mature, they will revolutionize the way humans and computers interact.

Programming WSNs is challenging. To ensure seamless integration with the environment, each node must be physically small. Many nodes are the size of a matchbox, if not smaller, severely limiting a node's capabilities. For example, the Tmote Sky [1] node runs on two AA batteries and measures $5.45cm^2$ without the battery pack. It contains a slow 16-bit 8MHz Texas Instruments MSP430 processor with a mere 10KB of data memory, 48KB of instruction memory, and 1MB of external flash memory. It is programmable using an on-board USB port, communicates using an IEEE 802.14.4-compliant Chipcon CC2420 radio, and has built-in light, temperature, and humidity sensors, and SPI and I^2C interfaces for attaching peripherals. The Tmote Sky is relatively new, older nodes like the MICA2 [2], MICAz [3], NMRC [21], M2020 [4], M1010 [5], Smart Dust [6], ESB platform [7], and Intel iMote [8] share similar, if not weaker, specifications.

Many WSN applications like habitat and infrastructure monitoring require long deployment intervals, often without human intervention. Throughout this time, the nodes are continuously subjected to a potentially harsh environment, resulting in some nodes dying early, and new nodes being deployed to replenish those that have failed. This variability further complicates the already unreliable wireless links. In all, programming applications for WSNs is difficult because they need to achieve extraordinarily high levels of *efficiency*, *reliability*, and *autonomy* in an environment that's highly dynamic and resource deprived. Such challenges are further amplified when a WSN must handle all the aforementioned forms of mobility.

Middleware is often relied upon to soothe the difficulties with programming WSNs. It offers more sophisticated abstractions, and often includes very high-level programming languages. This chapter presents an overview of the key principles behind various WSN middleware.

2 Operating System Support

Prior to discussing WSN middleware, we must first understand the basic services provided by the underlying operating system. While there are many operating systems that work in embedded systems like LynxOS [9], ChorusOS [10], Contiki [30], VxWorks [11], NetBSD [12], OSE [13], QNX, OS-9 [14], FreeDOS [15], and eCos [16], not all of them work in the highly resource-constraint setting of WSNs, or are flexible enough to work in a dynamic WSN environment. Instead, we focus on TinyOS [42], which is a representative example and arguably the most popular WSN operating system used today. TinyOS is a highly efficient minimalist operating system developed originally for the Atmel Atmega series of microprocessors, but ported to the TI MP430. It is event-based with a two-level thread hierarchy consisting of tasks and events. Tasks contain long-running computations that may be preempted by time-critical events. To minimize overhead, there is a single task queue, forcing tasks to run sequentially. When an event occurs, other events are disabled until the current event completes. Blocking is avoided using *split-phase operations*, e.g., when the application makes a system call, TinyOS immediately passes control back to the application while it processes the command, and later signals the command's completion using an event.

TinyOS provides a component-based programming language called NesC. NesC divides an application into components that may be *modules* or *configurations*. Program behavior is encapsulated within modules, which are wired together using configurations. Configurations can wire other configurations together, allowing the formation of components trees. An application is implemented as a tree of components where the root is the TinyOS kernel, branches are configurations, and leafs are modules or hardware components. For simplicity, TinyOS does not provide dynamic memory management. Instead, memory is statically allocated to each module and *parameterized interfaces* are used in place of multiple component instantiations. To maximize efficiency, TinyOS uses *active messages* for network communication [25]. Active messages contain an identifier specifying how the receiver should process the message. This, along with TinyOS's event-based execution model, avoids the need for components to block or poll for messages.

While programming WSNs using TinyOS is simpler than using assembly or C, it is still not easy. As pointed out in [52], TinyOS has a high learning curve, particularly for non-programmers. It requires one to become familiar with tasks, events, commands, split-phase operations, and component hierarchy in addition to subtle race conditions involving asynchrony and atomicity between tasks and events. There are many features common to other programming languages like component instantiations and private state that are not provided by NesC, mostly to ensure applications can attain maximum efficiency. Furthermore, TinyOS components tend to encapsulate low-level services. For example, many of TinyOS's components are part of the *hardware presentation layer* (HPL) that interfaces directly with the hardware. Other components encapsulate the network stack (`GenericComm`), flash memory (`ByteEEPROM`), timers (`TimerC`), and kernel (`Main`). In addition, the hard-wiring of TinyOS components makes it difficult to develop flexible applications that can adapt to a changing context. To change a program's behavior, the new behavior must either be pre-coded into the program, or the nodes need to be retrieved and reprogrammed, neither of which is scalable. Middleware is clearly needed to provide higher-level programming abstractions that hide TinyOS's complexities and to allow programmers to quickly implement, test, and deploy their WSN applications.

3 Basic Building Blocks

We now discuss middleware packages that provide basic building blocks. Two middleware packages we focus on are the Sensor Network Application Construction Kit (SNACK) [36] and Hood [76]. SNACK provides a high-level language and a library of components that offer application-level services. Hood provides a neighbor list abstraction commonly used by many applications.

3.1 Sensor Network Application Construction Kit

Object-oriented programming languages are sometimes not used to write software for embedded systems due to concerns about efficiency. In an object-oriented program, a component may have multiple instantiations, each with private state. This results in redundancy of state that could otherwise be shared, increasing memory utilization. To avoid this, many WSN applications are written using non object-oriented languages like NesC and C. These languages do

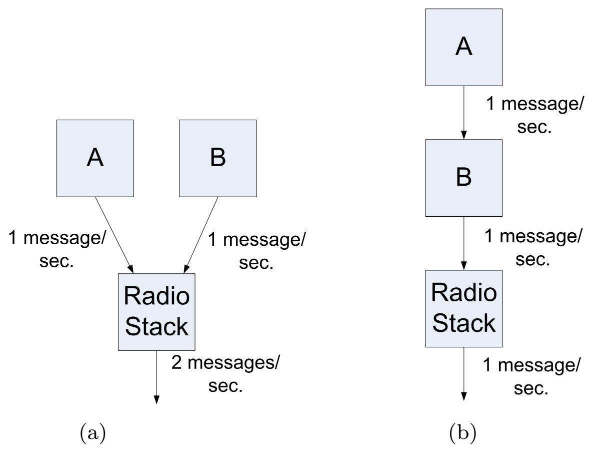

Figure 1: Alternative message control flows: (a) shows two components, A and B, independently sending messages resulting in two messages per second; (b) shows a more efficient flow where the two components share the same message, halving the number of messages broadcasted over the radio.

not allow multiple instantiations of a component. Instead, each component and all of its variables are statically defined. When programming applications using these languages, developers need to maintain each virtual instance's state manually through complex programming constructs like state arrays and parameterized interfaces. A typical example is the timer within TinyOS. The timer is independently used by many components, but since only one timer exists, it relies on a parameterized interface and an internal state array for remembering when each component's timer should fire. The programmer is given fine-grain control over which variables are shared and can thus maximize memory reuse. Doing this manually, however, is tedious and results in complex code.

SNACK is a middleware that provides a high-level language and a library of application-level services. Like object-oriented languages, SNACK allows multiple instantiations of a component. It is implemented on top of TinyOS, but can be ported to other operating systems. To reduce overhead, SNACK's compiler aggressively detects variables that can be shared and reorganizes the program to maximize efficiency. SNACK uses a fairly simple but aggressive mechanism for determining whether state can be shared. When a component is instantiated, all of its state is shared unless the instance is declared using the keyword `my`. If `my` is present, all variables inside the component are private.

The SNACK programming language allots the compiler greater flexibility in re-arranging components for higher efficiency. Many WSN applications consist of numerous independent components. Changing the control flow between them can significantly reduce overhead. For example, consider the application shown in Figure 1(a). The figure shows two components that generate messages at a rate of 1 message per second each. Both components send their messages to the radio stack, which forwards them at a rate of 2 messages per second. If each message contains little data, this will result in high overhead. Re-arranging the message flow to that shown in Figure 1(b) can halve the number of transmissions. However, programming components to adhere to this arrangement is tedious and may not be possible if the two components were developed independently. Instead, SNACK provides a messaging service that works with the compiler to achieve the control flow shown in Figure 1(b).

Allowing the compiler to reorganize an application's control flow is non-trivial. SNACK's programming language enables this by providing a richer syntax for specifying parameter values and using *transitive links*. Instead of specifying a specific value for a parameter, the SNACK programming language allows a range to be specified, e.g., "at most," "at least," or "in between." By relaxing the constraints on the parameter, the compiler is more likely to find values that intersect, allowing it to rearrange the control flow. Transitive links are used to specify links that can be arranged by the compiler. A transitive link simply indicates that one component should be wired to another, but it allows the compiler to insert any number of components in between.

SNACK provides a library of components consisting of a messaging service, timer service, and a hash table data structure. The messaging service provides support for the control flow shown in Figure 1(b). It includes a `MsgSrc`, `Network`, and `AttrM` component. The `MsgSrc` generates empty messages and is transitively linked to the Network. As the message travels from the `MsgSrc` to the `Network`, other components can use `AttrM` to add their data to the message, merging multiple messages into one. The compiler decides how the final program is structured. The timer service reduces the number of independent timers used by an application. By relying on transitive links, a single timer may pass through multiple components. Finally, SNACK provides a hash table data structure keyed by node ID. The SNACK implementation is much simpler since it can be instantiated using the `my` keyword, ensuring all state within it is unshared.

3.2 Hood - A Neighbor List

The lack of fixed infrastructure in WSNs coupled with uncertainties associated with individual sensor readings result in algorithms that exhibit a high degree of locality. Nodes need to collaborate with their neighbors to route and aggregate data. They also have to compare their sensor readings to verify and improve accuracy. The localized algorithms that perform these tasks require neighborhood information, namely, a list containing the addresses of nearby nodes whose attributes match certain application-specific criteria. The neighborhood discovery and maintenance protocol is frequently used across a broad range of applications and is, thus, provided as middleware in the form of Hood.

Hood is a middleware for WSNs that provides a neighbor list abstraction. It handles all of the underlying details about neighbor discovery, message passing, and data caching. By using Hood, an application developer can treat the neighbor list as a programming primitive and operate on data shared by the neighbors, thus simplifying programming. Till now, the messaging protocols and filtering algorithms required to create and maintain this list were done from scratch and customized for each application. This was necessary because each application required different algorithms and data structures. Hood avoids this problem by allowing the developer to plug in application-specific code for distributing a node's profile, called the *push policy*, and for processing discovered profiles, called a *filter*.

Hood allows a node to specify a list of attributes to share and a push policy that determines how these attributes are broadcasted. The attributes may include anything, e.g., sensor readings, application state, or remaining power. The push policy may broadcast a message whenever an attribute is set, perform periodic broadcasts, or perform reliable broadcasts, e.g., broadcast an attribute several times whenever it is set. If attributes are rarely broadcasted, a bootstrap mechanism may be implemented that broadcasts notifications whenever a node joins a neighborhood. Whenever a bootstrap message is received, the node broadcasts its attributes. To ensure maximum flexibility, the push policy is implemented in native NesC and is passed as a parameter to Hood.

To create a neighborhood, a node passes a filter to Hood. The filter receives all of the attributes broadcasted by neighbors and determines which neighbors should be included in the neighbor list. Note that each node independently creates and maintains a neighborhood, meaning neighborhoods are not symmetric. It is possible for node A to consider node B to be a neighbor, but not vice versa. A node does not know who considers it to be a neighbor. This decouples the owner of an attribute from the observers that consider the owner a neighbor. Achieving symmetric neighborhoods would require more complex protocols with transactional semantics which may prove infeasible in an environment as dynamic and unreliable as a WSN. The unreliability of WSNs also means Hood cannot provide any consistency guarantees, i.e., it cannot guarantee that every node within a certain distance will receive every broadcast.

Hood allows applications to append *scribbles* to each neighbor. A scribble is a locally-derived value like an estimate link quality based on the neighbor's attributes. The filters may include scribbles in their analysis of determining when to include a node in the neighbor list. Complications arise when the filter requires multiple attributes and scribbles since some may be defined while others remain undefined. To address this, a hierarchy of neighborhoods may be created where the members of one neighborhood are dependent on the neighbors in another neighborhood.

Hood is a specific middleware that has been shown to be useful in object-tracking applications. It represents a general class of middleware that provides an abstraction for facilitating neighbor discovery. The key idea of this middleware is that it allows the application developer to specify application-specific details like the push policy and filtering algorithm, but provides a generic infrastructure for putting these policies to work. The middleware with a neighbor list abstraction can be easily generalized to provide multiple neighbor lists each with different attributes, or to provide higher-level operations that operate over all members of a particular list.

4 Data-Centered Abstractions

The primary purpose of a WSN is to gather data about an environment and relay that information to the consumers. Two key challenges to distributing sensor information are 1) mobility and 2) limited resources. The data consumers may include external users, internal nodes, or a combination of both. Many applications like safe-route navigation and a museum tour guide involve multiple external users that transiently connect to the network and physically move through the sensor field. To complicate matters, WSNs operate under severe resource limitations in terms of power, memory, and computational ability, all of which should be sparingly used. This prevents simply porting existing ad hoc routing protocols developed for laptops and PDAs into a WSN. The underlying operating system, in minimizing overhead, provides little support. For example, TinyOS only offers one-hop unicast and broadcast. Clearly, new more efficient and agile algorithms for distributing sensor data, and higher-level programming models to use these algorithms, are required. Many middlewares provide this. We consider two algorithms, geographic hash tables [68] and directed diffusion [46, 39], and two models, WSN as a database [59, 35, 79, 69] and abstract regions [75], all of which are available as middleware.

In a WSN, both the sender and the receiver may be transient and mobile. Traditional routing protocols for wireless ad hoc networks like AODV [66] and DSR [23] are non-ideal because they have high overhead and latency, especially when the network is dynamic. This resulted in the development of many novel routing algorithms that use data caching and forward pointers to efficiently deliver a message to a moving consumer [51]. Forward pointers, however, leave control-state scattered through the network, and may still fail if the receiver is moving too rapidly. A geographic hash table does not suffer from this problem. It takes advantage of the spatial property of a WSN. Whenever a sender wishes to send a message to a particular destination, the destination's address is hashed to a particular location. The data is sent to that location, and any node within one hop stores the message, serving as a mailbox. The destination periodically checks its mailbox to receive the message. A geographic hash table guarantees that the message will be delivered as long as no message loss occurs. By choosing a proper hash function, data can be distributed evenly across the entire network, providing load balancing. Drawbacks include the need for each consumer to periodically query its mailbox, the possibility that the mailbox and consumer may be located on opposite sides of the network, and the additional overhead of dealing with node mobility.

WSNs consist of potentially thousands of nodes. Many of them will fail mid-deployment, and new nodes may be added. This highly dense and dynamic environment calls for applications that place less emphasis on the data collected by each individual node, but rather treats the nodes as an aggregate. Applications for WSNs tend to not require data from a specific sensor but rather from sensors that have certain properties like, for example, being located in a particular region, or having sensor readings above a certain threshold. This observation led to the development of content-based routing where data is not sent to a particular address, but rather is routed based on its attributes. One system that provides this is directed diffusion [46]. Directed Diffusion introduces the idea of routing data down interest gradients. When a consumer is interested in a particular type of data, it broadcasts a description of its interest, which is propagated throughout the network. By propagating the interest, downhill gradient is produced where the consumer is at the bottom. Any data matching the interest is forwarded down the gradient to the consumer. The middleware handles all of the distribution of interests and configuring of "flows" for delivering data down interest gradients. To save power, it also performs in-network data aggregation to reduce the number of packets sent back to the consumer, e.g., instead of receiving all of the raw data, it only receives the average over a certain epoch.

WSN middleware may also change the fundamental model presented to the application programmer. One example is a database. Using this model, a programmer need not program individual sensors or debug complex data aggregation and routing algorithms. Instead, a WSN is treated like a single table in a database consisting of the aggregate data collected across all WSN nodes. Many middlewares that provide this abstraction are available. They include TinyDB [59], SINA [69], Cougar [79] and IrisNet [35]. All of these middlewares implement a distributed query processor that spans the entire network. They provide a SQL-like language tailored to WSNs. A primary difference between implementing a virtual WSN database versus a real database is the fact that the data is continuously generated, often only after a query is issued. Also, in-network data aggregation is necessary in WSNs to reduce message transmissions and save power. To account for this, the database query language is augmented with `EPOCH`, `SAMPLE_PERIOD`, and function modifiers that allow a programmer to specify how aggregate data should be generated within the network. To account for the spatial properties of a network, a `WHERE` modifier is also provided. Recent work in this area has produced more powerful aggregation operations [41] and techniques for compressing aggregate data [71].

Abstract regions [75] is another middleware that provides a new model in place of individual nodes in a WSN. It allows programmers to define regions in the network, and to treat each region as if it were an individual sensor. A region may in reality contain many sensors, but all of them aggregate their data, and the application is only given the aggregate. Abstraction regions still provide application programmers control over resource consumption, and provide feedback on how successfully requested aggregate operations were carried out. It uses a tuple-space like model for distributing data, and provides a thread-like model where applications can perform blocking operations. Together, this greatly simplifies the programming model allowing even non-computer scientists to program a network.

5 Mobility-Centered Abstractions

A common application for WSNs is to track a mobile entity as it travels through a sensor field. The entity can be, among many other things, a soldier in a field or a car in a parking lot. In the simplest implementation, any node that detects an entity would report to a central base station. This implementation is less accurate since the sensors do not collaborate to verify their readings. Some entities are difficult to track and can only be accurately sensed by multiple sensors concurrently [38]. The simple implementation is also not energy efficient since redundant notifications many be routed through the network when multiple sensors detect the same entity. Other challenges arise when the nodes operate on a sleep schedule. If a node is asleep, it will not be able to detect the entity. Thus, complex motion prediction and forewarning algorithms are used to wake up nodes ahead of the entity so they can participate in the event detection process. Tracking mobile entities is a complex process used by many applications. Several middleware packages have been developed to reduce this complexity. One such package is EnviroTrack [18]. Another middleware package that allows mobile users to access the WSN is MobileQuery [57].

5.1 EnviroTrack

EnviroTrack provides a *context-label* abstraction for each entity. A context label is distinguished by type, e.g., "car," and contains user-defined aggregate state about the entity and application-specific code that operates on this state. The context-label is dynamically created when the entity is first detected, and logically follows the entity as it moves through the sensor field. EnviroTrack simplifies programming by hiding the details associated with inter-node communication and group maintenance necessary to detect and track an entity. Programmers interact with the static context label rather than with a continuously changing set of nodes that sense the entity. This is done using a directory service based on a geographic hash table. The context label's type is hashed to a specific location and nodes within one hop of this location serve as directory objects. Each context label periodically updates its state with its directory objects, and programmers interact with the directory objects.

To use EnviroTrack, the programmer must specify the possible types of context-labels. For each type, the programmer must provide a `sense` function, a `state` function, and, optionally, *tracking objects*. The `sense` function takes the raw sensor readings and determines whether the entity is present. It includes a *freshness* constraint that determines the maximum age of a sensor reading for it to be considered, and a *critical mass* constraint that determines the minimum number of sensors that sense the entity for the detection to be considered valid. The `state` function produces the aggregate data to maintain within the context label. The *tracking objects* perform local processing on the nodes that comprise the context-label, allowing them to perform context-sensitive tasks that may directly affect the entity, e.g., forming a barrier if the entity is an intruder or activating a sprinkler system if the entity is a fire.

EnviroTrack provides a group maintenance algorithm for organizing nodes that can sense the entity. Ideally, the algorithm will produce a single context-label per entity, each with a single *leader* who is responsible for aggregating the data and updating the context label's directory service. A context-label is formed whenever a node detects an entity using the `sense` function and is not already part of, or aware of, a group for the entity. Whenever this occurs, it creates a context label and declares itself leader. As the leader, it periodically broadcasts a heartbeat that is propagated a certain number of hops beyond the group's boundary. The heartbeat serves the dual purpose of telling members the leader is still alive, and providing a forewarning system. If a member node can sense an entity but does not hear a heartbeat within a certain time, it creates a new group. Whenever a node receives a heartbeat, it remembers the last time it received it. If the entity is sensed within a certain time of receiving the last heartbeat, the node joins the group by periodically sending its sensor readings to the leader. Multiple leaders may occur when the entity is initially detected by multiple nodes, or when the heartbeat fails to propagate. To account for this, each leader maintains a weight corresponding to the number of updates it has received from group members. The leader with the lower weight secedes to the one with the higher weight. As the entity moves, the leader may loose the ability to sense the entity. When this it occurs, it hands off its leadership. The node the becomes the new leader inherits the former leader's weight, thereby reducing the probability that spurious leaders will remain active.

EnviroTrack has been implemented on top of TinyOS. To further simplify programming, it provides a preprocessor that translate EnviroTrack code into NesC. Through simulations and actual implementations on Mica motes, EnviroTrack has been shown to simplify WSN programming.

5.2 MobileQuery

While EnviroTrack represents a middleware supporting entity mobility, MobileQuery [57] is a middleware supporting user mobility. MobileQuery is best motivated using an example. Imagine a user carrying a PDA traveling through a sensor field. As the user travels, what is the simplest task the user may want to do that is useful to many applications? One such task is to simply query the sensor readings of nodes within a certain vicinity, e.g., a possible query may be "what is the average of all temperature readings within a one mile area?" This requires the dissemination of a query to a certain geographic region, and routing and aggregation algorithms for delivering the results back to the user. Since the user continuously moves, this query process should be periodic. MobileQuery provides this service.

Implementing MobileQuery may at first appear simple: just broadcast a query, and wait for a reply. In networks that do not operate on a sleep schedule and applications that do not require real-time operation, this is probably sufficient. However, many WSN applications like hazard detection and safe-route navigation software require real-time context information. Also, WSNs often operate on a sleep schedule to prolong network lifetime where the nodes remain asleep for the majority of the time, only briefly waking up to perform application-specific tasks. If the user issues a query when the nodes just went to sleep, the latency will be high. Furthermore, as the query area expands, a naive flooding solution will result in greater energy consumption due to duplicate broadcasts and more message loss due to contention. To avoid these problems, MobileQuery adapts a pre-fetching and tree-building scheme where *prefetching* messages are sent ahead of the user to pre-defined pickup points (i.e., the location where the user expects to receive query results), and a routing tree rooted at the pickup point is created to collect and aggregate the results.

MobileQuery assumes that a user's movements are predictable and that prefetch message can travel faster than the user. A user's movements may be predictable based on history, or in certain scenarios like driving on the highway or hiking on a trail, the motions may follow a pre-defined map. To ensure prefetch messages can travel faster than the user, a backbone overlay network [28, 74, 78] is used.

The are two types of pre-fetching: all-at-once and just-in-time. All-at-once pre-fetching sends the pre-fetch messages as far as the user will travel. It assumes that the user travels a fixed distance along a known path. By sending the prefetch messages immediately, it can guarantee that all of the nodes in future query areas will be ready to supply sensor readings by the time the user arrives.

6 Dynamic Reprogramming

The aforementioned middleware systems simplify programming by providing commonly-used services. They are integrated into the application at compiletime and do little to increase network run-time flexibility. Flexibility is important in WSNs because of their highly dynamic topology and lengthy deployment intervals throughout which the user requirements, or the users themselves, may change. Knowing all possible uses of a WSN prior to deployment is not possible. A network initially deployed for habitat monitoring may later be used for wildfire detection. Large WSN networks covering a wide geographic area may be deployed for a single use initially, but later be divided into regions, each running software specialized to features within each environment. Anticipating and incorporating all possible behaviors into an application prior to deployment is not feasible. There are many aspects like algorithms and data structures that are difficult to parameterize. Memory constraints prevent including extraneous behavior. To address this, middleware packages have been created to facilitate the dynamic reprogramming of a pre-deployed network. By enabling dynamic reprogramming, the network can assume any behavior, and can serve transient users as they arrive and move through the sensor field.

Embedded systems often adhere to a Harvard architecture with separates data and instruction memories. This naturally leads to the ability to reprogram a node by re-flashing the instruction memory and can be provided at the operating system level [45]. The problem is that 1) some nodes do not employ rewritable instruction memory due to cost, 2) re-flashing memory consumes a

Table 1: Middleware Supporting Network Reprogramming

Name	**Execution Unit**	**Execution Model**	**Network Reprog.**	**Communication Model**
Agilla	agent	agent thread	strong migration	tuple space
Impala	modules	event	flood code	message passing
Maté	capsules	event	flood code	message passing
SensorWare	script	event	weak migration	message passing
Smart Messages	smart message	single thread	strong migration	remote: migration local: tag space

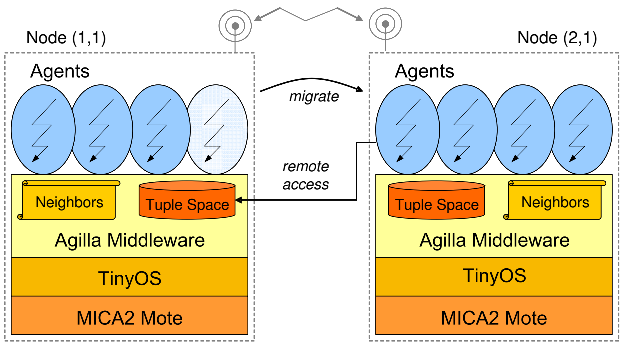

Figure 2: The Agilla model

lot of power and is unreliable when the batteries are not fully charged, and 3) it requires the transmission of the entire operating system and application over a lossy low-bandwidth radio. Some middleware packages provide a more efficient way to reprogram a WSN. They either allow the instruction memory to be partially reprogrammed, or, more often, provide a virtual machine that interprets lightweight mobile control scripts. Middleware packages of this sort include Agilla [33], Impala [54], Maté [52], SensorWare [22], and Smart Messages [50]. A summary of their features is shown in Table 1.

Among them, Agilla [33] provides a virtual machine that supports *mobile agents* and highly flexible capabilities for in-network reprogramming. Its middleware architecture is shown in Figure 2. Unlike traditional programs that are statically installed on a specific node, mobile agents can move and clone themselves across the network performing application-specific tasks. Two types of migration operations are provided: strong and weak. Strong migration captures the entire state of an agent including its program counter and stack. The agent resumes execution at the destination uninterrupted. A weak migration, on the other hand, only captures data state, all execution state is lost and the agent resumes running at the beginning when it arrives at the destination. While strong migration may simplify programming, it entails higher overhead.

An application often consists of multiple agents. For example, in a fire detection application, multiple mapping agents may be used to form a perimeter around the fire [32]. When multiple agents are used to carry out an application's task, they must coordinate with each other. In Agilla, coordination is achieved through local Linda-like tuple spaces [34]. A tuple space is a special type of shared memory where the data is addressed by content using templates. One agent can insert a tuple, and another can later read or remove it using pattern matching via a template. To prevent polling, Agilla augments the tuple space with *reactions* [31, 49, 64, 26] that notify an agent of a tuple matching a particular template when it appears in the tuple space. Tuple spaces decouple agents ensuring that they remain autonomous. An agent's autonomy is vital in a dynamic environment such as a WSN since inter-agent interactions tend to be highly transient.

The tuple space also serves as a convenient mechanism for agents to discover their context. For example, in Agilla, the tuple space stores the types of sensors available and the identities of the other co-located agents. Tradeoffs regarding what to store in the tuple space versus through a dedicated data structure must be made. For example, since the address of neighboring nodes is frequently accessed by many applications, Agilla provides an *acquaintance list* abstraction. However, a list of available sensors is accessed less frequently, so it is stored in the tuple space. Providing a dedicated data structure reduces latency, but increases memory overhead.

Unlike other mobile middleware [64], Agilla does *not* support a global tuple space that spans across multiple nodes primarily due to bandwidth and energy constraints. Instead, it supports *local* tuple spaces where each node maintains a distinct and separate tuple space. If agents were restricted to only operate on the local tuple space, they would only be able to coordinate with co-located agents. But in many applications, agents need to communicate with agents residing on different hosts. While this may be accomplished by having the agent migrate to the other agent's host, agent migration is relatively expensive. Thus, Agilla provides special instructions that allow an agent to perform operations on a remote host's tuple space. These instructions rely on simple muti-hop unicast communication and are thus scalable. Sequentially accessing each neighbors' tuple space, however, entails significant overhead. To address this, Agilla provides a group instruction that uses single-hop multicast to query the tuple spaces of all one hop neighbors. Scalability is ensured since this operation only operates over one hop.

WSNs are unique in that spatial properties are important. Sensor nodes detect certain properties of the environment. The location at which they take the measurement is necessary, for example, when determining where the intruder is. In other words, many WSN applications must know their spatial placement to make sense of the sensor data they collect. Agilla embraces this reliance on spatial information by addressing nodes by their geographic location. Sensors can get their location through GPS, or any number of other localization schemes [67, 24, 62]. As a side benefit, addressing nodes by location enables Agilla primitives to be easily extended to operate over geographic regions, and to use geographic routing for multi-hop interactions.

Agilla is implemented on top of TinyOS and tested on MICA2 motes. It demonstrated that careful engineering of the middleware makes programming flexible applications consisting of mobile agents not only feasible but easier. In its current implementation, standard coordination mechanisms like tuple spaces and acquaintance lists are used. In the future, agents may be able to communicate directly with each other, thereby further reducing overhead, or they may be able to mutate their code taking on additional capabilities as they gain experience within the network. The possibilities are endless.

Impala [54] and Maté [52] are two similar middleware systems that divide an application's code into capsules, that are then distributed throughout the network. The main difference is that Impala uses native code whereas Maté uses a virtual machine. Unlike Agilla agents, Impala and Maté capsules have no control over where they execute. When an updated capsule is issued, it is flooded throughout a network. This prevents multiple applications from running concurrently and different areas of the network from running different code. Both Impala and Maté use an event-based programming model where each capsule remains inactive unless an event to which it is sensitive occurs. For example, one capsule within Maté is a timer capsule. The code within this capsule is executed whenever the timer fires. Another capsule is sensitive to the arrival of a message. By using an event-based model, these middlewares achieve high efficiency by avoiding polling and allowing the execution unit to remain dormant during periods of no events.

Smart Messages [50] and SensorWare [22] are similar to Agilla in that they allow their execution units to control where they are located. Smart Messages allow its execution units, known as a *smart message*, to perform strong migrations. SensorWare uses mobile scripts as its execution unit, but only supports weak migration. Both systems are implemented as virtual machines on relatively powerful PDAs. Unlike Agilla, Smart Messages only provides a single thread per node and separates local communication (via a tag space) from remote communication (via migration). By having a single thread, multiple applications cannot run concurrently on a node, and the need for a smart message to migrate to communicate with a remote node incurs higher overhead than simply remotely accessing the node. SensorWare, on the other hand, uses an event-based execution model like Impala and Maté. Also like Impala and Maté, SensorWare uses direct message passing for communication between its mobile scripts.

7 Emerging Strategies

WSNs are continuously evolving as technology improves and new applications become feasible. In recent years, WSNs have evolved from rigid application-specific deployments to flexible embedded computing platforms. As WSN nodes improve, they will run more sophisticated applications that demand better middleware support. There are several emerging strategies that middleware designers are currently investigating. They include providing quality of service (QoS), macroprogramming, and connecting WSNs with traditional networks.

7.1 Quality of Service Management

As WSNs mature, they will run more sophisticated applications, and multiple applications at a time. Existing middleware like Agilla already allows multiple applications to be dynamically loaded into a network. Little attention, however, has been paid to quality of service (QoS), specifically those related to message delivery latency, sensing accuracy, energy consumption, and data throughput. Most existing middleware packages provide services on a best-effort basis and do not consider application-specific semantics when making trade-offs between QoS and resource consumption. Many applications share the same types of trade-offs, e.g., decrease sensing accuracy or increase message latency for additional power savings. Programming these trade-offs within each application is tedious and error-prone. Furthermore, when multiple applications share the same network, interactions across applications need to be accounted for. For example, if one application is tracking a raging wildfire, it should be given better QoS, than an application monitoring the migration patterns of monarch butterflies. Middleware provides a convenient mechanism for adding these QoS provisions.

Two middleware projects that provide QoS are MiLan [63, 40] and AutoSec [37]. MiLan takes a specification on the minimum QoS an application requires, and adapts the network to achieve this QoS while minimizing resource consumption. Instead of simply observing network parameters and adapting the application, MiLan attempts to control the network. For example, consider a habitat monitoring application running on a WSN deployed throughout a forest. The majority of the time, there isn't anything interesting so the middleware selects a sparse subset of the nodes to monitor the environment at a low resolution to save power. However, when an interesting event is detected, the middleware increases the QoS by activating additional nodes near the phenomena of interest. This proactive approach allows MiLan to provide high QoS while still consuming low resources. AutoSec differs from MiLan in that it focuses on resource allocation to ensure maximum system throughput. It relies on a *directory service* that stores information about the network's current state, and, based on this information, chooses a resource allocation policy that divvies up the resources such that each application achieves its desired QoS. Part of the challenge with AutoSec lies in determining how to maintain the directory service, and what resource allocation policies should be provided.

Real-time behavior is a specific type of QoS that many applications require. In a real-time application (e.g., surveillance, fire monitoring, and intruder detection), messages and actions need to be precisely choreographed for the application to function correctly. Messages must be delivered on time at the right place carrying data of a certain freshness. Two middleware projects that provide real-time functionality are DSWare [58] and RAP [56]. DSWare is a publish/subscribe middleware that relies on standard real-time packet scheduling mechanisms, e.g., earliest deadline first (EDF). It also provides group management, event detection, data caching and data storage services, reducing the burden of application developers. RAP is a real-time query service for WSNs. It introduces *velocity monotonic scheduling*, which takes advantage of the spatial properties of the network to provide real-time message delivery. RAP allows a user to issue a query with certain period, deadline and data freshness requirements. As data is delivered, its velocity, as measured by how far it has traveled over how long, is used as a local indicator of how urgent the packet needs to be forwarded. A message that will barely make or miss its deadline traveling at its current velocity will have higher priority than a message that will easily make its deadline. Both middleware projects are still in prototype phase having only been evaluated in simulators.

7.2 Macroprogramming

Another emerging strategy being embraced by developers of WSN middleware is the idea of *macroprogramming.* Macroprogramming relies on new programming languages that allow a programmer to describe, at a high level, what the sensor network should do. The middleware and compiler would then determine the low-level code that executes on each individual node. By hiding the distributed nature of the network, programming it is greatly simplified. Agilla can be viewed as a form of macroprogramming where developers create agents without worrying about precisely where they are installed. Other middleware projects based on macroprogramming include abstract regions [75], virtual markets [60], Regiment [65], and MagnetOS [19].

Abstract regions were discussed in Section 4. They allow programmers to reason about abstract regions that map to collections of nodes. Virtual markets take a unique approach to achieving new behavior in a WSN. Instead of introducing new code into the network, a virtual market simply changes the value of performing certain tasks. In a virtual market, intelligent agents are distributed throughout the network that can perform certain actions, e.g., take a sensor reading, aggregate data, and forward data. Each action has a value associated with it. By programming the agents to seek maximum profits, the overall system behavior can be controlled by simply changing the value of each action. Regiment and MagnetOS are both high-level programming languages that allow a developer to program a WSN application as if it ran on a single node. The underlying middleware and compiler takes the program and determines how it can be divided and distributed across multiple nodes within the WSN. They differ in that Regiment provides a functional language whereas MagnetOS is written in Java.

7.3 Integration with Traditional Networks

Another emerging area of WSN middleware research involves developing platforms that allow the seamless integration of WSNs with traditional networks. In order for WSNs to gain widespread use, they must be easily integrated with the existing computing infrastructure. Currently, the code that bridges WSNs with traditional networks is proprietary relying on custom protocols tailored to each application. Furthermore, as WSNs gain widespread use, more sophisticated applications will want to harness the power of multiple potentially heterogeneous WSNs. These applications are often distributed across multiple administrative domains. For example, a company's inventory management system may reside on servers belonging to the company, supplier, and shipping company. Applications running on traditional networks are mature. They operate across administrative domains by adhering to common protocols and languages like those proposed by the Open Grid Service Architecture (OSGA) [17]. OSGA, however, introduces too much overhead for use in WSNs. Developing and maintaining custom protocols that facilitate the interactions between WSNs and the fixed infrastructure is a formidable task. It is a service that WSN middleware is just beginning to provide.

One middleware package that links WSNs with traditional networks is Agilla. In Agilla, mobile agents can easily migrate between WSNs and traditional networks. Another project is Hourglass [70]. Hourglass operates over the Internet and handles the delivery of data between consumers and sources located across multiple WSNs. In creates an overlay network over the fixed Internet infrastructure, and provides a *circuit* abstraction that connects the sources with the destinations. Circuits are tailored to handling WSN data by allowing various services to operate over the data flowing through them. For example, some of the services provided by Hourglass include filtering, aggregating, compressing, and buffering. Since Hourglass nodes are more powerful than WSN nodes, they are capable of performing more complex operations on the data. An application developer simply tells Hourglass its data needs, and the middleware takes care of discovering the networks that provide the raw data and assembling the necessary services to produce the required data. Hourglass is still in the prototype stage, having only been simulated in ModelNet [73].

8 Conclusion

WSNs promise to revolutionize the way humans interact with their physical environment. They will soon gain widespread use because they provide many benefits and are relatively inexpensive to deploy. However, in order to gain widespread use, new middleware solutions are required. Programming WSNs is difficult because they have extremely limited resources and exhibit many forms of mobility involving the users, entities being sensed, sensor nodes, data, and code. To help simplify application development, many middleware packages have been created. Initial WSN middleware provided basic building blocks like high-level programming languages, neighbor lists, and libraries of components that provide application-level services. As applications matured and gained complexity, new middleware for handling the various forms of mobility and increasing network flexibility through in-network reprogramming were developed.

WSNs are relatively new and are rapidly evolving, forcing middleware designers to embrace new strategies. These emerging strategies include providing QoS, macroprogramming, and providing a foundation for connecting WSNs to traditional networks.

Acknowledgement
This research was supported in part by the Office of Naval Research under MURI research contract N00014-02-1-0715 and by the the NSF under ITR contract CCR-0325529. Any opinions, findings, and conclusions expressed in this paper are those of the authors and do not necessarily represent the views of the research sponsors.

References

[1] http://www.moteiv.com.
[2] http://www.xbow.com/Products/productsdetails.aspx?sid=72.
[3] http://www.xbow.com/Products/productsdetails.aspx?sid=101.
[4] http://www.dustnetworks.com/PDF/M2020_Mote.pdf.
[5] http://www.dustnetworks.com/PDF/M1010_Mote.pdf.
[6] http://robotics.eecs.berkeley.edu/~pister/SmartDust/.
[7] http://www.scatterweb.com/.
[8] http://www.intel.com/research/exploratory/motes.htm.
[9] http://www.lynuxworks.com/.
[10] http://www.experimentalstuff.com/Technologies/ChorusOS/.
[11] http://www.windriver.com/.
[12] http://www.netbsd.org/.
[13] http://www.ose.com/.
[14] http://www.microware.com/.
[15] http://www.freedos.org/.
[16] http://sourceware.org/ecos/.
[17] http://www.globus.org/ogsa/.
[18] T. Abdelzaher, B. Blum, Q. Cao, Y. Chen, D. Evans, J. George, S. George, L. Gu, T. He, S. Krishnamurthy, L. Luo, S. Son, J. Stankovic, R. Stoleru, and A. Wood. Envirotrack: Towards an environmental computing paradigm for distributed sensor networks. In *Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS'04)*, pages 582–589. IEEE Computer Society, 2004.
[19] R. Barr, J. C. Bicket, D. S. Dantas, B. Du, T. W. D. Kim, B. Zhou, and E. G. Sirer. On the need for system-level support for ad hoc and sensor networks. *SIGOPS Oper. Syst. Rev.*, 36(2):1–5, 2002.
[20] M. A. Batalin, M. Rahimi, Y.Yu, D.Liu, A.Kansal, G. Sukhatme, W. Kaiser, M.Hansen, G. J. Pottie, M. Srivastava, and D. Estrin. Towards event-aware adaptive sampling using static and mobile nodes. Technical Report 38, Center for Embedded Networked Sensing, 2004.
[21] S. Bellis, K. Delaney, B. O'Flynn, J. Barton, K. Razeeb, and C. O'Mathuna. Development of field programmable modular wireless sensor network nodes for ambient systems. In *Computer Communications, Special Issue on Wireless Sensor Networks, to appear 2005.*, 2005.

[22] A. Boulis, C.-C. Han, and M. Srivastava. Design and implementation of a framework for efficient and programmable sensor networks. In *Proc. of MobiSys*, pages 187–200. USENIX, May 2003.

[23] J. Broch, D. B. Johnson, and D. A. Maltz. The Dynamic Source Routing Protocol for Mobile Ad Hoc Networks. Internet Draft, March 1998. IETF Mobile Ad Hoc Networking Working Group.

[24] N. Bulusu, J. Heidemann, and D. Estrin. Gps-less low cost outdoor localization for very small devices. Technical Report 00-729, University of Southern California, April 2000.

[25] P. Buonadonna, J. Hill, and D. Culler. Active message communication for tiny networked sensors. In *TinyOS Website*, `http://www.tinyos.net/papers/ammote.pdf`.

[26] G. Cabri, L. Leonardi, and F. Zambonelli. Reactive tuple spaces for mobile agent coordination. *Lecture Notes in Computer Science*, 1477:237–252, 1998.

[27] A. Cerpa, J. Elson, D. Estrin, L. Girod, M. Hamilton, and J. Zhao. Habitat monitoring: application driver for wireless communications technology. *SIGCOMM Comput. Commun. Rev.*, 31(2 supplement):20–41, 2001.

[28] B. Chen, K. Jamieson, H. Balakrishnan, and R. Morris. Span: An energy-efficient coordination algorithm for topology maintenance in ad hoc wireless networks. In *Mobile Computing and Networking*, pages 85–96, 2001.

[29] C. Curino, M. Giani, M. Giorgetta, A. Giusti, A. L. Murphy, and G. P. Picco. TinyLime: Bridging Mobile and Sensor Networks through Middleware. In *Proceedings of the 3rd IEEE International Conference on Pervasive Computing and Communications (PerCom 2005)*, pages 61–72, Kauai Island (Hawaii, USA), Mar. 2005. IEEE Computer Society.

[30] A. Dunkels, B. Grnvall, and T. Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *Proceedings of the First IEEE Workshop on Embedded Networked Sensors 2004 (IEEE EmNetS-I)*, November 2004.

[31] C.-L. Fok, G.-C. Roman, and G. Hackmann. A Lightweight Coordination Middleware for Mobile Computing. In R. DeNicola, G. Ferrari, and G. Meredith, editors, *Proceedings of the 6th Internation Conference on Coordination Models and Languages (Coordination 2004)*, number 2949 in Lecture Notes in Computer Science, pages 135–151. Springer-Verlag, February 2004.

[32] C.-L. Fok, G.-C. Roman, and C. Lu. Mobile agent middleware for sensor networks: An application case study. In *Proc. of the 4th Int. Conf. on Information Processing in Sensor Networks (IPSN'05)*, pages 382–387. IEEE, April 2005.

[33] C.-L. Fok, G.-C. Roman, and C. Lu. Rapid development and flexible deployment of adaptive wireless sensor network applications. In *Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS'05)*, pages 653–662. IEEE, June 2005.

[34] D. Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, January 1985.

[35] P. Gibbons, B. Carp, Y. Ke, S. Nath, and S. Seshan. Irisnet: An architecture for a worldwide sensor web. *IEEE Pervasive Computing*, pages 22–33, October-December 2003.

[36] B. Greenstein, E. Kohler, and D. Estrin. A sensor network application construction kit (snack). In *Proceedings of the 2nd Int. Conf. on Embedded Networked Sensor Systems (SenSys '04)*, pages 69–80, New York, NY, USA, 2004. ACM Press.

[37] Q. Han and N. Venkatasubramanian. Autosec: An integrated middleware framework for dynamic service brokering. In *IEEE Distributed Systems Online*, volume 2:7, Nov. 2001.

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

[39] J. Heidemann, F. Silva, and D. Estrin. Matching data dissemination algorithms to application requirements. In *Proceedings of the ACM SenSys Conference*, pages 218–229. ACM, November 2003.

[40] W. Heinzelman, A. Murphy, H. Carvalho, and M. Perillo. Middleware to support sensor network applications. In *IEEE Network Magazine Special Issue*, volume 18, pages 6–14, Jan. 2004.

[41] J. Hellerstein, W. Hong, S. Madden, and K. Stanek. Beyond average: Towards sophisticated sensing with queries. In *Workshop on Information Processing In Sensor Networks (IPSN)*, 2003.

[42] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. In *Architectural Support for Programming Languages and Operating Systems*, pages 93–104, 2000.

[43] T. T. Hsieh. Using sensor networks for highway and traffic applications. *IEEE Potentials*, 23(2):13–16, Apr-May 2004.

[44] Q. Huang, C. Lu, and G.-C. Roman. Spatiotemporal Multicast in Sensor Networks. In *Proc. of the 1st Int. Conf. on Embedded Networked Sensor Systems (SenSys 2003)*, pages 205–217. ACM Press, November 2003.

[45] J. Hui and D. Culler. The dynamic behavior of a data dissemination protocol for network programming at scale. In *Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 81–94. ACM Press, 2004.

[46] C. Intanagonwiwat, R. Govindan, and D. Estrin. Directed diffusion: a scalable and robust communication paradigm for sensor networks. In *Proc. of MobiCom 2000*, pages 56–67, 2000.

[47] D. Jea, A. Somasundara, and M. Srivastava. Multiple controlled mobile elements (data mules) for data collection in sensor networks. In *Proc. of the Int. Conf. on Distributed Computing in Sensor Systems (DCOSS)*, June 2005.

[48] P. Juang, H. Oki, Y. Wang, M. Martonosi, L. S. Peh, and D. Rubenstein. Energy-efficient computing for wildlife tracking: design tradeoffs and early experiences with zebranet. *SIGPLAN Not.*, 37(10):96–107, 2002.

[49] C. Julien and G.-C. Roman. Egocentric Context-Aware Programming in Ad hoc Mobile Environments. In *Pro. of the 10th Int. Symp. on the Foundations of Software Engineering*, pages 21–30, Nov. 2002.

[50] P. Kang, C. Borcea, G. Xu, A. Saxena, U. Kremer, and L. Iftode. Smart messages: A distributed computing platform for networks of embedded systems. *Special Issue on Mobile and Pervasive Computing, The Computer Journal*, 47:475–494, 2004.

[51] H. S. Kim, T. F. Abdelzaher, and W. H. Kwon. Minimum-energy asynchronous dissemination to mobile sinks in wireless sensor networks. In *Proc. of Sensys 2003*, pages 193–204, 2003.

[52] P. Levis and D. Culler. Maté: a tiny virtual machine for sensor networks. In *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pages 85–95, New York, NY, USA, 2002. ACM Press.

[53] T.-H. Lin, H. Sanchez, W. J. Kaiser, and H. Marcy. Wireless integrated network sensors (wins) for tactical information systems. In *Proc. of the 1998 Government Microcircuit Applications Conference*, 1998.

[54] T. Liu and M. Martonosi. Impala: A middleware system for managing autonomic, parallel sensor systems. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2003.

[55] K. Lorincz, D. Malan, T. R. F. Fulford-Jones, A. Nawoj, A. Clavel, V. Shnayder, G. Mainland, S. Moulton, and M. Welsh. Sensor networks for emergency response: Challenges and opportunities. *IEEE Pervasive Computing, Special Issue on Pervasive Computing for First Response*, pages 16–23, Oct-Dec 2004.

[56] C. Lu, B. M. Blum, T. F. Abdelzaher, J. A. Stankovic, and T. He. Rap: A real-time communication architecture for large-scale wireless sensor networks. In *Proc. of the 8th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'02)*. IEEE Computer Society, 2002.

[57] C. Lu, G. Xing, O. Chipara, C.-L. Fok, and S. Bhattacharya. A spatiotemporal query service for mobile users in sensor networks. In *Proc. of the Int. Conf. on Distributed Computing Systems (ICDCS)*, pages 381–390. IEEE Press, June 2005.

[58] S. Lu, S. Son, and J. Stankovic. Event detection services using data service middleware in distributed sensor network. In *Proc. of the 2nd Int. Workshop on Information Processing in Sensor Networks (IPSN)*, April 2003.

[59] S. Madden, M. Franklin, J. Hellerstein, and W. Hong. The design of an acquisitional query processor for sensor networks. In *Proceedings of the 2003 ACM SIGMOD Int. Conf. on Management of Data*, pages 491 – 502, 2003.

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

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

[62] D. Moore, J. Leonard, D. Rus, and S. Teller. Robust distributed network localization with noisy range measurements. In *The Second ACM Conference on Embedded Networked Sensor Systems (Sensys 04)*, November 2004.

[63] A. Murphy and W. Heinzelman. MiLAN: Middleware Linking Applications and Networks. Technical Report TR-795, University of Rochester, 2002.

[64] A. L. Murphy, G. P. Picco, and G.-C. Roman. Lime: A Middleware for Physical and Logical Mobility. In *Proceedings of the 21^{st} International Conference on Distributed Computing Systems*, pages 524–533, April 2001.

[65] R. Newton and M. Welsh. Region streams: Functional macroprogramming for sensor networks. In *Proc. of the 1st Int. Workshop on Data Management for Sensor Networks (DMSN)*, August 2004.

[66] C. Perkins and E. Royer. Ad Hoc On-Demand Distance Vector Routing. In *Proceedings of the Second IEEE Workshop on Mobile Computing Systems and Applications*, pages 90–100, February 1999.

[67] N. Priyantha, A. Chakraborty, and H. Balakrishnan. The cricket location-support system. In *Mobile Computing and Networking*, pages 32–43, 2000.

[68] S. Ratnasamy, B. Karp, L. Yin, F. Yu, D. Estrin, R. Govindan, and S. Shenker. Ght: A geographic hash table for data-centric storage in sensornets. In *Proceedings of the First ACM International Workshop on Wireless Sensor Networks and Applications (WSNA)*, September 2002.

[69] C.-C. Shen, C. Srisathapornphat, and C. Jaikaeo. Sensor Information Networking Architecture and Applications. *IEEE Personel Communication Magazine*, 8(4):52–59, August 2001.

[70] J. Shneidman, P. Pietzuch, J. Ledlie, M. Roussopoulos, M. Seltzer, and M. Welsh. Hourglass: An Infrastructure for Connecting Sensor Networks and Applications. Technical Report TR-21-04, Harvard, 2004.

[71] N. Shrivastava, C. Buragohain, D. Agrawal, and S. Suri. Medians and beyond: new aggregation techniques for sensor networks. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 239–249, New York, NY, USA, 2004. ACM Press.

[72] G. Simon, M. Maroti, and A. Ledeczi. Sensor network-based countersniper system. In *The Second ACM Conference on Embedded Networked Sensor Systems (Sensys'04)*, November 2004.

[73] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kostic, J. Chase, and D. Becker. Scalability and accuracy in a large-scale network emulator. In *Proceedings of 5th Symp. on Operating Systems Design and Implementation (OSDI)*, December 2002.

[74] X. Wang, G. Xing, Y. Zhang, C. Lu, R. Pless, and C. Gill. Integrated coverage and connectivity configuration in wireless sensor networks. In *Proc. of the 1st Int. Conf. on Embedded networked sensor systems (Sensys'03)*, pages 28–39. ACM Press, 2003.

[75] M. Welsh and G. Mainland. Programming sensor networks using abstract regions. In *Proceedings of the First USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI)*, March 2004.

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

[77] N. Xu, S. Rangwala, K. K. Chintalapudi, D. Ganesan, A. Broad, R. Govindan, and D. Estrin. A wireless sensor network for structural monitoring. In *Proc. of the 2nd Int. Conf. on Embedded Networked Sensor Systems (SenSys '04)*, pages 13–24. ACM Press, 2004.

[78] Y. Xu, J. Heidemann, and D. Estrin. Geography-informed energy conservation for ad hoc routing. In *MobiCom '01: Proc. of the 7th Annual Int. Conf. on Mobile computing and networking*, pages 70–84, New York, NY, USA, 2001. ACM Press.

[79] Y. Yao and J. Gehrke. The cougar approach to in-network query processing in sensor networks. *SIGMOD Rec.*, 31(2):9–18, 2002.

Chien-Liang Fok received B.S. degrees in computer science and computer engineering from Washington University in St. Louis in 2002. He is a PhD candidate in the Department of Computer Science and Engineering at Washington University in Saint Louis where he is a member of the Mobile Computing Laboratory. His research interests include wireless ad hoc networks and sensor networks.

Gruia-Catalin Roman is a Professor and the Chairman of the Department of Computer Science and Engineering at Washington University in St. Louis. He was a Fulbright Scholar at the University of Pennsylvania, where he received a B.S. degree (1973), an M.S. degree (1974), and a Ph.D. degree (1976), all in computer science. His current research interests include the study of formal models, algorithms, design methods, and middleware for mobile computing. Roman was an associate editor for ACM TOSEM and served as the general chair of ICSE 2005. He is a member of Tau Beta Pi, ACM, and IEEE Computer Society.

Chenyang Lu is an Assistant Professor in the Department of Computer Science and Engineering at Washington University in St. Louis. He received the NSF CAREER Award in 2005. He has published more than forty refereed research papers. His research interests include wireless sensor networks, distributed real-time systems and middleware, and adaptive QoS control. He received the Ph.D. degree from University of Virginia in 2001, the M.S. degree from Chinese Academy of Sciences in 1997, and the B.S. degree from University of Science and Technology of China in 1995, all in computer science.