Providing OS Support for Wireless Sensor Networks: Challenges and Approaches

Wei Dong, *Student Member, IEEE,* Chun Chen,
Xue Liu, *Member, IEEE* and Jiajun Bu, *Member, IEEE*

***Abstract*—Recently, wireless sensor networks (WSNs) attract a great deal of research attention, and are envisioned to support a variety of applications, including military surveillance, habitat monitoring, and infrastructure protection, etc. Operating system (OS) support for WSNs plays a central role in building scalable distributed applications that are efficient and reliable. Over the years, we have seen a variety of OSes emerging in the sensornet community to facilitate developing WSN applications. Aside from the basic system implementations, there is also a large body of work devoted to improving OS capabilities in different dimensions. In this paper, we provide a comprehensive review of existing work in sensornet OS design. We first examine the challenges in the OS design space. We then introduce the major components of a sensornet OS. Next, we provide an overview of existing work, present a taxonomy of state-of-the-art OSes, and discuss various approaches to address the design challenges. Finally we discuss evaluations of a sensornet OS and present some recommendations from the perspectives of OS developers and OS users. We have also identified several open problems that need further investigation to make the OS provide stronger support for WSNs.**

***Index Terms*—Operating systems, wireless sensor networks, performance metrics.**

I. Introduction

A TYPICAL wireless sensor network (WSN) consists of a large number of small-sized, battery-powered sensor nodes that are limited in battery energy, CPU power, and communication capability. Recently, it has seen an explosive growth in both academia and industry [1], [2]. WSNs attract a great deal of research attention in the past few years [3]–[11], and they are envisioned to support a variety of applications, including military surveillance [12], habitat monitoring [13], and infrastructure protection [14], etc.

Despite the simplicity of the sensor node hardware, WSN applications are diverse and demanding. Infrastructural support for WSN applications in the form of operating systems (OS) is becoming increasingly important. In essence, a sensor network OS (hereafter, sensornet OS) bridges the gap between hardware simplicity and application complexity, and it plays a central role in building scalable distributed applications that are efficient and reliable.

Manuscript received 13 July 2008; revised 25 November 2008 and 5 March 2009.

W. Dong, C. Chen, and J. Bu are with the Zhejiang Key Laboratory of Service Robot, College of Computer Science, Zhejiang University. (e-mail: {dongw, chenc, bjj}@zju.edu.cn).

X. Liu is with the School of Computer Science, McGill University. (e-mail: xueliu@cs.mcgill.ca).

Digital Object Identifier 10.1109/SURV.2010.032610.00045

The basic functionalities of an OS include resource abstractions for various hardware devices, interrupt management and task scheduling, concurrency control, and networking support. Based on the services provided by the OS, application programmers can conveniently use high-level application programming interfaces (APIs) independent of the underlying hardware. The implementation of the abovementioned services for a sensornet OS, however, is non-trivial because most sensor node platforms are severely resource-constrained. Therefore, a sensornet OS should not only provide a variety of system services to facilitate programming WSN applications, but also optimize resource utilizations.

Over the years, we have seen various OSes emerging in the sensornet community. The most prestigious works include TinyOS [15], Contiki [16], SOS [17], Mantis OS [18], Nano-RK [19], RETOS [20] and LiteOS [21]. Aside from the basic system implementations mentioned above, there is also a large body of work devoted to improving OS capabilities in different dimensions, e.g., improving OS reliability [22], [23], providing real-time support [19], [24], [25], and extending the programming model [26]–[29].

To the best of our knowledge, a comprehensive review of the various works related to the sensornet OS design is still lacking mainly due to the following reasons. First, there is still no well-accepted agreement about what constitutes a sensornet OS because the boundary between a *sensornet* OS and *sensornet* applications is not as clear as that of a *general-purpose* OS and *general-purpose* applications. Second, making a taxonomy of the OSes needs a deep understanding of their design goals and underlying mechanisms as different OSes differ drastically. Finally, evaluations of a sensornet OS is difficult as benchmark research for WSNs is still in an early stage [30].

We try to shed light on the abovementioned issues in this paper. We introduce what constitutes a sensornet OS by describing its major components. We provide a taxonomy of state-of-the-art sensornet OSes by examining a set of important OS features. Finally, we discuss evaluations of a sensornet OS by exploring several common performance metrics and the impact of several design factors.

The contributions of our work are two-fold. First, we fill the gap of reviewing existing work, providing a taxonomy of state-of-the-art sensornet OSes, and discussing various approaches in the OS design space as well as their impact to different design goals. This guides us to choose appropriate design approaches when a new OS is to be designed for a specific

TABLE I
TERMS USED IN THIS PAPER

Terms	Definition
Dynamic linking and loading	The process of loading a program on disk into memory, performing dynamic linking if necessary, and starting execution of the program [31].
Reconfigurability	The ability to allow the functionality of an OS to be dynamically changed.
Reprogramming	The process of installing a new application to a network of sensors wirelessly (i.e., over the air) [9].
Run-to-completion task	A task that cannot suspend or block.
Split phase I/O	In this I/O model, to initiate an I/O operation, the programmer needs to first issue an I/O request in one task, and write another task to handle the notification when the requested I/O is complete.
Active message	A communication primitive that each message contains the address of a handler to be executed upon message arrival [32].
dedicated, virtualized, and shared resources	Three types of resource abstractions in TinyOS [33]. A dedicated resource supports a single user. A virtualized resource supports multiple users by maintaining request queues and scheduling concurrent requests in a sequenced manner. A shared resource also supports multiple users, but users must contend for the driver through a lock.

set of goals. Second, we identify several open problems that need further investigation to make the OSes provide stronger support for WSNs, such as real-time and network management support.

The rest of this paper is organized as follows. Section II examines the challenges in the OS design space. Section III introduces the major components of a sensornet OS. Section IV provides a comprehensive review of existing work, provides a taxonomy of state-of-the-art OSes, and discusses various approaches to address the challenges presented in Section II. Section V discusses evaluations of a sensornet OS. Section VI presents some recommendations from the perspectives of OS developers and OS users. Finally, we conclude this paper in Section VII.

II. CHALLENGES

In order to develop a practical and efficient sensornet OS, many challenges have to be addressed, mainly due to the severe resource constraints of the sensor node hardware and the demanding requirements of WSN applications. Table I lists the main terms used in the rest of this paper. The major challenges that influence the OS design are listed as follows.

Small Footprint. The limited memory of only a few kilobytes on a sensor node necessitates the OS to be designed with a very small footprint. It is a fundamental characteristic of a sensornet OS and is the primary reason why so many sophisticated embedded OSes (e.g., μC/OS [34], VxWorks [35]) can not be easily ported to sensor nodes.

Energy Efficiency. Sensor nodes provide very limited battery life-time. On the other hand, guaranteeing sensor networks to operate for 3 to 5 years is a very desirable objective. Such an imbalance implies that a sensornet OS must be energy efficient to prolong WSN life-times.

Reliability. In most applications, sensor networks are deployed once and intended to operate unattended for a long period of time [9]. OS reliability is of great importance to facilitate developing complex WSN software, ensuring the correct functioning of WSN systems [22].

Real-Time Guarantee. As most sensornet applications such as surveillance tend to be time-sensitive in nature where packets must be relayed and forwarded on a timely basis, real-time guarantee is a necessary requirement for such applications.

Reconfigurability. It is desirable that systems on sensor nodes can be dynamically reprogrammed [9] over the air after a WSN is deployed. A high system reconfigurability is a desired OS feature to make the network-wide reprogramming easy and efficient.

Programming Convenience. Sensor network applications are diverse and demanding. Hence programming convenience is of great importance to shorten the development cycle of real-world WSN applications.

Usually it is very hard, if not impossible, to achieve optimality in all aspects. Most current solutions assume specific application scenarios, and make tradeoffs to address the primary design challenges.

III. COMPONENTS

The major components that constitute a sensornet OS are listed as follows.

Task Scheduling. The task scheduling component provides application programmers the *task* environment for executing deferable and long running application code. Currently, there are two categories of task scheduling schemes in the sensornet community: event-driven task scheduling (e.g., TinyOS [15]) and multi-threaded task scheduling (e.g, Mantis OS [18]). In TinyOS, tasks are event-driven and have run-to-completion semantics. They are efficient and responsive to asynchronous activities, but have the drawback of manual state maintenance and split-phase I/Os. In contrast, in Mantis OS, tasks are multi-threaded. They are written with a traditional thread-like programming style, but incur a larger implementation overhead.

Dynamic Linking and Loading. The dynamic linking and loading component allows application modules to be loaded dynamically and improves the OS reconfigurability when reprogramming a network of sensors [36], [37]. It is included in sensornet OSes such as Contiki [16], SOS [17], RETOS [20], and LiteOS [21]. When an application module is to be loaded, the linker and loader allocate memory space for application's data and code sections, link (relocate) external (internal) symbols to their physical addresses, and copy the *linked* module to the reserved memory space. This mechanism is useful when reprogramming an application on sensor nodes: rather than disseminating a monolithic kernel-application image [38], it only requires disseminating the application module [36], which is much smaller and thus more energy-efficient for dissemination.

Memory Management. Memory in current sensor nodes can be broadly classified into: RAM (for fast data storage), internal

flash (for code storage), EEPROM (for data storage), and external flash (for data storage). For example, the MicaZ platform has 4KB RAM, 128KB internal flash, 4KB EEPROM, and 512KB external flash. RAM is important for fast data access while external flash is important for data persistence. RAM management can either be static as in TinyOS [15] or dynamic as in SOS [17]. External flash management can provide high level abstractions such as *file* (e.g., LiteOS [21]) or low level abstractions such as *block* (e.g., TinyOS [15]).

Resource Abstraction. The resource abstraction component provides a safe and high-level accessing interface to the underlying hardware devices. For example, resource abstractions in TinyOS [39] are classified into *dedicated* resource abstractions, *virtualized* resource abstractions, and *shared* resource abstractions [33]. A dedicated resource supports a single user. A virtualized resource supports multiple users by maintaining request queues and scheduling concurrent requests in a sequenced manner. A shared resource also supports multiple users, but users must contend for the driver through a lock.

Sensor Interfaces. Sensor interfaces provide application programmers a way to access sensor readings in an easy and unified form. Low-level details associated with sensor configurations are thus abstracted away from application programmers.

Networking Stack. The networking stack facilitates developing distributed WSN applications. It is also essential to remote system maintenance after a WSN is deployed. A networking stack for the OS should support high-level services such as dissemination and collection. It also handles low-level details such as radio chip configuration, Medium Access Control (MAC) [3], and queue management.

IV. Designs

In this section, we first provide an overview of existing work. Then we present a taxonomy of state-of-the-art sensornet OSes. Finally we discuss various approaches that have been used to address the challenges presented in Section II.

A. Overview

In this subsection, we give a description of state-of-the-art sensornet OSes. It should be noted that most of these OSes are currently under active development, hence we describe features in the current version of these OSes as of the publication of this paper.

TinyOS. TinyOS [15], developed in UC Berkeley, is perhaps the earliest sensornet OS in the literature [42]. To enable a flexible architecture and a low resource consumption, TinyOS programming is based on *components* which are wired together to create an application at design-time. Component interactions happen at two directions, i.e., one component can *use commands* provided by another component; also, one component can *signal events* to another component. The execution model of TinyOS consists of interrupts and tasks. Interrupts execute at a higher priority and can preempt the execution of tasks. Tasks execute at a lower priority and are scheduled in a FIFO manner. Tasks in TinyOS are written in a run-to-completion manner, and they can not be preempted or self-suspended. For this reason, I/Os are done in split-phases, i.e., a *request* is done at the end of a task while *signal* invokes the start of the next task. In order to provide a better support for the component architecture and execution model of TinyOS, the nesC language [26] was designed for programming based on TinyOS.

TinyOS version 2 (T2), enhances version 1 in several aspects. T2 provides *telescoping abstraction*, which is a hybrid of horizontal decomposition (for the lower level to support different kinds of hardware devices) and vertical decomposition (for the higher level to support platform-independent functionality), and makes it easier to support new hardware platforms. Moreover, T2 provides *service distribution*, which limits arbitrary component compositions and provides a group of components (i.e., services) to improve system reliability. Besides architectural improvements, there are a number of important improvements in implementation, including threading support (e.g., TinyThreads [43], and TOSThreads in TinyOS version 2.1 [44]), memory protection support [44], [45].

Contiki. TinyOS components are *statically* wired to create a single kernel/application image. While this approach can optimize resource consumption, it is costly to dynamically reconfigure or update the applications. To address this issue, Contiki [16], developed in Swedish Institute of Computer Science, supports dynamically loadable modules. To address the programming inconvenience of event-based programming (e.g., in TinyOS), Contiki supports *cooperative* multi-threading (i.e., context switches happen at user defined points) via user libraries [16]. Contiki also supports a lightweight threading mechanism, i.e., protothreads [27]. The latest version of Contiki, version 2.2.1, implements a flash ROM file system called Coffee [46], and the Chameleon architecture for the Rime low-power radio stack [47].

SOS. SOS [17], developed in the University of California, Los Angeles, also supports dynamically loadable modules. It adopts a module-based architecture. It is worth noting the difference between TinyOS components and SOS modules. TinyOS components are only visible from programmers' perspective, and they "disappear" when they are compiled into the binary code. SOS modules, on the other hand, still remain the binary information (e.g., entry point of a module, exported functions of a module) after compilation to allow dynamically loading and unloading modules. To support this architecture, SOS also supports dynamic memory allocation. The SOS execution model is only slightly more complex than TinyOS: SOS message handlers (similar to TinyOS tasks) are dispatched according to three different priorities, but preemption is not allowed between two message handlers. The latest version of SOS, version 2.0.1, improves the original version in several aspects, including broader platform support, and memory protection support (e.g., Harbor [23]). However, as of Nov. 24, 2008, SOS is no longer under active development because of the graduation of the core developers.

Mantis OS. To enable full-fledged multi-threading support, Mantis OS [18], developed in Colorado University, implements a traditional *preemptive* time-sliced multi-threading on sensor nodes. To enable the traditional programming paradigm (i.e., thread-like programming), the Mantis kernel also sup-

TABLE II
A TAXONOMY OF STATE-OF-THE-ART SENSORNET OSES

	TinyOS	Contiki	SOS	Mantis	Nano-RK	RETOS	LiteOS
Publication (Year)	ASPLOS (2000)	EmNets (2004)	MobiSys (2005)	MONET (2005)	RTSS (2005)	IPSN (2007)	IPSN (2008)
Static/Dynamic	Static	Dynamic	Dynamic	Dynamic	Static	Dynamic	Dynamic
Event/Thread	Event & Thread (TinyThread, TOSThreads)	Event & Threads & Protothreads	Event	Thread & Event (TinyMOS)	Thread	Thread	Thread & Event (through callback)
Monolithic/Modular	Monolithic	Modular	Modular	Modular	Monolithic	Modular	Modular
Networking	Active Message	uIP, uIPv6, Rime	Message	"comm"	Socket	Three-layer Architecture	File-Assisted
Real-Time Support	No	No	No	No	Yes	POSIX 1003.1b	No
Language Support	nesC	C	C	C	C	C	LiteC++
File System	Single level (1.x only) (ELF, Matchbox)	Coffee	No	No (will be in 1.1)	No	No	Hierarchical Unix-like
Reprogramming	Yes (Deluge, FlexCup)	Yes	Yes (Modular)	No (will be in 1.1)	No	Yes	Yes
Remote Debugging	Yes (Clairvoyant)	No	No	Yes (NodeMD)	No	No	Yes (DT)

ports *synchronous* I/Os (as opposed to *split-phase* I/Os), and a set of concurrency control primitives, e.g., binary semaphores (i.e., mutex) and counting semaphores. The latest version of Mantis, version 1.0 beta, improves system stability and fixes some bugs. Besides, it implements the Collection Tree Protocol (CTP) [48] for MicaZ and TelosB platforms.

Nano-RK. To support time-sensitive WSN applications such as surveillance and environmental monitoring, Nano-RK [19], developed in Carnegie Mellon University, implements a reservation-based real-time OS for WSNs. Nano-RK supports fixed-priority based preemptive multitasking for guaranteeing that task deadlines are met. It also supports CPU and network bandwidth reservations, i.e., tasks can specify their resource demands and the OS provides timely, guaranteed and controlled access to CPU cycles and network bandwidth. To facilitate network programming, Nano-RK provides APIs for socket-like abstractions, and generic system support for network scheduling and routing [19]. The latest version of Nano-RK fixes some bugs in the previous versions.

RETOS. RETOS [20], developed in Yonsei University, Korea, is designed to improve several aspects of prior work. It improves system resilience by supporting dual mode operation (i.e., kernel mode and user mode) as well as application code checking at design-time and run-time. It optimizes multi-threading implementation [49] and provides support for POSIX 1003.1b real-time scheduling. It also supports loadable modules and provides multi-hop networking services. The latest version of RETOS, version 1.4, supports a broader range of platforms, optimizes the networking layer, and improves system safety.

LiteOS. LiteOS [21], developed in the University of Illinois at Urbana Champaign, is designed to provide a traditional Unix-like environment for programming WSN applications. It includes: (1) a built-in hierarchical file system and a wireless shell for user interaction using Unix-like commands. (2) kernel support for dynamic loading native execution of multi-threaded applications. (3) an object-oriented programming language that uses a subset of C++ as its syntax with class library support. The latest version of LiteOS, version 1.0, adds support for the recent Crossbow IRIS platform [50]. In addition, it provides an energy virtualization mechanism (Virtual Battery [51]) and a remote debugging system (Declarative Tracepoints [52]).

There are some other sensornet OSes that we have not investigated in this paper due to space limit, including MagnetOS [53], Nano-Qplus [54], Pixie OS [55], and SenSpire OS [56].

B. Taxonomy

In this subsection, we provide a taxonomy of state-of-the-art OSes by examining some important OS features (as summarized in Table II).

Static or Dynamic. In static systems (e.g., TinyOS, Nano-RK), application programmers must allocate all of the resources at design-time. On the other hand, in dynamic systems (e.g., SOS, Contiki, Mantis, RETOS, LiteOS), application programmers can allocate and deallocate resources at run-time. Dynamic systems are more flexible, and thus are more suitable for dynamically changing environments. However, they are not reliable as a buggy application could easily acquire excessive resources (at run-time), causing the entire system to crash.

Event-driven or Multi-threaded. In event-driven systems (e.g., TinyOS, SOS), application programmers must manually maintain the application state (in the form of global variables) and use split-phase I/Os. On the other hand, in multi-threaded systems (e.g., Mantis, Nano-RK, RETOS), application programmers can use the traditional thread-like programming style. Hence multi-threaded systems are more familiar to most programmers and are typically considered more user-friendly than event-driven systems. There are number of projects that aim to enhance the event-driven systems by providing multi-threading support. For example, TinyThread [43], TOSThreads

[44] are thread libraries based on TinyOS; Contiki supports preemptive multi-threading via a library on top of the event-driven kernel [16], and it also implements a a lightweight threading mechanism called protothreads [27]. Event-driven systems are also useful: they are suitable for applications that are highly responsive; more importantly, they incur a very small implementation overhead and represent a cost-effective solution for sensor nodes with severe resource constraints. Hence some multi-threaded systems also support events. For example, TinyMOS [57] supports TinyOS events in the Mantis kernel; LiteOS provides event support through callback functions [21].

Monolithic or Modular. An application may be compiled with the OS as a monolithic program (e.g., TinyOS) or may be compiled into an individual program module that is loadable by the OS kernel (e.g., Contiki, SOS, RETOS). The former approach presents an opportunity to do full program analysis and optimizations and is appropriate when the application seldom needs to be modified. The latter approach is favored when the individual application needs to be frequently modified through network reprogramming. Although it adds the OS complexity by including linking and loading support, it greatly reduces the dissemination overhead when reprogramming a network of sensor nodes.

Networking Support. For general-purpose OSes, TCP/IP is the norm. For sensornet OSes, however, there are currently no standard architectures or implementations despite the investigations by Polastree et al. [58], Cheng et al. [59] and Dunkels et al. [47]. It is not surprising because networking for WSN is complicated by the wireless nature and the diversity of transmission patterns (e.g., one-to-one, one-to-all, all-to-one) [60], [61]. Without such a standard architecture, different sensornet OSes adopt quite different mechanisms and support wireless networking to different extent. Most sensornet OSes provide single-hop communications on top of which multi-hop networking can thus be implemented as user-level services. Examples include TinyOS, SOS, Mantis. TinyOS uses a lightweight Active Message (AM) based communication stack. Aside from a data payload, each AM also contains an AM ID which is used for the target node to dispatch the AM to its corresponding handler upon arrival. It is worth noting that a large number of higher level networking protocols are implemented on top of this AM layer, including dissemination protocols [38], [62]–[64], and collection protocols [48]. SOS uses a module based message passing for communication. Each message contains a module ID which specifies the corresponding module to be invoked upon message arrival. Mantis encapsulates MAC protocols in the "comm" layer to provide a unified interface for communications device drivers (e.g., serial, USB, and radio devices). It also manages packet buffering and synchronization functions. Some sensornet OSes directly support multi-hop networking. Examples include Contiki, Nano-RK, RETOS. Contiki contains two communication stacks: μIP and Rime [65]. μIP is a small RFC-compliant TCP/IP stack that makes it possible for Contiki to communicate over the Internet. Recently, Contiki additionally implements μIPv6 that supports IPv6 for WSNs. Rime is a lightweight communication stack designed for low power radios. Rime provides a wide range of communication primitives, from best-effort local area broadcast, to reliable multi-hop bulk data flooding [65]. Nano-RK provides socket-based abstraction that masks MAC scheduling and routing from applications. RETOS provides a three-layer architecture that provides single-hop communication (MAC and buffer management), multi-hop support (neighborhood and routing table management), various multi-hop networking protocols at three different layers (i.e., link layer, networking support layer, and dynamic networking layer). The OS kernel is responsible for the two lower layers while the highest layer is provided by network programmers as reconfigurable kernel modules. This architecture conform more closely to the suggestions of [47], [58], [59], i.e., the *Narrow Waist* of WSN is above the link layer (single-hop) and below the networking layer (multi-hop) and it is advisable that the OS should provide multi-hop support (e.g., neighborhood and routing table management) apart from single-hop communications. LiteOS provides additional support for communications of files among a set of sensor nodes, using traditional Unix-like shell commands such as `cp`, `mv`.

Real-Time Support. Real-time support for sensornet OSes can happen at the node level where the OS should perform priority-based preemptive task scheduling to ensure the real-time performance of a single node. Nano-RK is an RTOS that supports for priority-based preemption and timeliness through off-line schedulability analysis. RETOS supports the POSIX 1003.1b realtime scheduling interface to enable both programmers' explicit priority assignment and kernel's dynamic priority management. Real-time support can also happen at the network level where realtime network behaviors (e.g., end-to-end transmission with delay bounds) are required. This is something beyond the realm of OS research and will be briefly mentioned in Section IV-C.

Language Support. Most sensornet OSes use C as the programming language for both OS development and application development (e.g., Contiki, SOS, Mantis, RETOS). It is because C is efficient, portable, and is the dominant language for embedded system development. However, C has its drawbacks: on one hand, it is too general to perform semantic checking, optimizations, and kernel customizations; on the other hand, it is typically considered less user-friendly than the modern Object-Oriented (OO) languages such as C++ and Java. nesC [26] and LiteC++ [21] represent two solutions to address the two abovementioned issues respectively. The nesC language tightly integrates with TinyOS's execution models and adopts a component-based programming model. Hence it can perform static checking, full program analysis and optimizations, thus generates a monolithic kernel-application image with a very small footprint. However, as nesC is domain-specific, it usually incurs a large learning curve for traditional programmers. LiteC++ is designed for LiteOS for application development. It implements a subset of features of modern OO languages. Hence it is more familiar to traditional programmers and is typically considered more user-friendly than both C and nesC.

File Systems. A typical sensor node has a small storage capacity, e.g., MicaZ nodes have only 4KB EEPROM and

512KB external flash. Therefore, file systems on sensor nodes must have a very small implementation overhead. Matchbox [66] and ELF [67] are both implemented based on TinyOS version 1. They provide single-level file organizations and basic abstractions for file operations such as reading and writing. LiteOS enhances prior work by supporting hierarchical file organization and a richer set of file operation APIs. Recently, Contiki provides a flash-based file system, Coffee [46], for storing data inside the sensor network. The file system allows multiple files to coexist on the same physical on-board flash memory and has a performance that is close to the raw data throughput of the flash chip.

Reprogramming. Reprogramming has been a very active research area in recent years [9], [38], [62], [63], [68]. Typically, applications are programmed onto the sensor nodes via wired connection. With reprogramming support, developers are able to install or update a new application to a network of sensor nodes wirelessly. Deluge [38] is the standard reprogramming mechanism for TinyOS. Because of TinyOS's static design principle, applications are disseminated with the OS kernel as a full image. This approach incurs a large dissemination overhead because of the kernel overhead. To address this issue, FlexCup [69] supports dynamic linking and loading TinyOS binary components, thus allows code update on a modular basis. The dynamic linking and loading mechanism is natively supported by SOS, Contiki, RETOS, and LiteOS.

Remote Debugging. As the software for WSNs becomes more and more complex, the need for remote debugging is increasingly important. Clairvoyant [40], based on TinyOS, provides interactive source-level debugging commands such as `break`, `step`, and `watch` to access program states. NodeMD [41], implemented on Mantis, is designed to diagnose node-level faults in sensor network applications. It also provides remote retrieval of the logged information stored in a circular buffer, so that the probable root of the bug can be traced. Recently, Declarative Tracepoints (DT) [52], implemented on LiteOS, allows the user to insert a group of action-associated checkpoints, or tracepoints, to applications being debugged at run-time. Tracepoints do not require modifying application source code. By triggering the associated actions when these checkpoints are reached, this system automates the debugging process by removing the human from the loop.

C. Approaches to Various Challenges

In Section II, we have described various challenges in the OS design space. In this subsection, we present some specific approaches to address these challenges respectively.

Approaches to Small-footprint OS Design: To make a sensornet OS practically useful on resource-constrained sensor nodes which typically have 4KB–10KB data RAM and 48KB–128KB program flash, special considerations to optimize the code and data sizes must be taken. Current approaches to optimize code and data sizes happen at two levels, i.e., programming level and compilation level. At the programming level, the OS programmers must adopt cost-effective schemes, or, tailor functionalities of general-purpose or embedded OSes for implementation on sensor nodes. For example, TinyOS, Contiki, and SOS all adopt a simple event scheduling scheme instead of a sophisticated thread scheduling scheme; Mantis OS tailors the thread implementation of general-purpose OSes, limiting the thread stack size and running thread number for sensor nodes. At the compilation level, the compiler is responsible to optimize the data and code sizes. Various compilation techniques can be taken. For example, a new language, such as nesC [26], could be designed to facilitate code analysis and optimizations; memory compaction [70] could be carried out to reduce the RAM usage.

Approaches to Energy Efficiency: Since most sensor nodes are battery-powered, energy is an important factor in designing a sensornet OS. Current techniques to save energy on sensor nodes can be classified into microcontroller energy management and peripheral energy management, which will be described below respectively.

Microcontrollers typically have a spectrum of low power modes that support keeping different peripherals enabled. Computing the lowest safe power state of the microcontroller requires system-wide information on whether various peripherals are in use. This computation, however, can have complex dependencies. Klues et al. propose ICEM [33] to do microcontroller energy management. In this scheme, every microcontroller has a sleep function that calculates the lowest safe power state based on which interrupt sources and on-chip devices are enabled. The OS only computes the power state when necessary by keeping track of a dirty bit. The sleep state computation also allows a driver to specify a minimum safe sleep state by an override hook.

Peripheral energy management requires knowledge of the application usage model and turns the individual peripheral off when it is not used. Although for dedicated devices [33], it is fairly simple to control the power states by applications, for virtualized and shared devices, it is the OS's responsibility to do system-wide power management. ICEM [33] exploits the concurrency of application's I/Os and propose the power locks mechanism to automatically minimize energy consumption without additional explicit information from an application. Of all the peripherals on a sensor node, the radio is the most important as it consumes the largest fraction of energy compared to other devices. Hence it needs a special consideration and attracts many research efforts in the sensornet community. MAC scheme is most pertinent to the power management of radio chips. The key insight to save energy is to reduce idle listening and overhearing in WSNs [71]. We have seen various MAC schemes proposed in the sensornet community to address these issues, e.g., the schedule-based MACs such as S-MAC [72] and the contention-based MACs such as B-MAC [73] and X-MAC [74].

Approaches to OS Reliability: Reliability is an important design consideration for both general-purpose OSes and sensornet OSes. Many WSN applications should operate in an unattended manner for a long life-time, it is essential that the OS should provide reliable services to applications. It is, however, really challenging to achieve OS reliability in a sensornet OS because current sensor node hardware lacks advanced hardware features such as CPU privileged mode and MMU. Lack of CPU privileged mode makes it hard to prevent *control* hazards as a buggy program can easily take

CPU cycles away from the OS kernel, causing a sensor node not responsive any more. Lack of MMU makes it difficult to prevent *data* hazards as a buggy program may easily access and corrupt kernel's data, causing the entire system to crash.

There are some works proposed to prevent control hazards, e.g., Gu et al. present t-kernel [22] to modify application code at load-time to ensure OS responsiveness. RETOS also checks the application code via static checks and dynamic checks to prevent illegal memory access and illegal indirect addressing instructions [20].

There are some works proposed to prevent data hazards, e.g., RETOS provides dual mode operation to ensure kernel data integrity: applications in the user mode use the user stack, and the stack is changed to the kernel stack upon system calls and interrupts handling [20]. Kumar et al. present harbor [23] to protect memory access with a software-based solution; various stack estimation techniques [43], [75], [76] are proposed to prevent stack overflow (hence ensure data integrity). Without hardware features mentioned above, most methods combines compile-time instrumentation and run-time checking, i.e., they analyze the program and insert hooks at compile-time, then execute these hooks at run-time to ensure system reliability.

Approaches to Support Real-Time Guarantee: As mentioned in Section IV-B, current work to support real-time guarantee includes node-level task scheduling and network-level packet transmission. In order to support real-time task scheduling, the OS must support priority preemption [19], [77]. Nano-RK supports fixed-priority based preemptive multitasking for guaranteeing that task deadlines are met [19]. RETOS provides support for POSIX 1003.1b real-time scheduling [20]. FIT provides a formal real-time task schedulability analysis and a special optimization for implementation on sensor nodes [24]. On the other hand, in order to support real-time packet transmission, real-time communication protocols must be proposed to meet the end-to-end delay bounds. Several approaches have been proposed to achieve this goal [25], [78]–[81]. SPEED [25] provides end-to-end soft real-time communication by maintaining a desired delivery speed across the sensor network through a combination of feedback control and non-deterministic geographic forwarding. SUPORTS [78] proposes a dynamic scheduling and rate regulation mechanism for supporting real-time, event-based traffic in sensor networks. RTQS [79] propose a novel transmission scheduling approach designed for real-time queries in WSNs.

Approaches to Improve Reconfigurability: System reconfigurability is important in order to facilitate network level reprogramming [9]. It is desirable that the system could be dynamically modified over the air in order to adapt to the changing environments. Applications can be reconfigured at the script level, virtual machine level or native code level [47]. For the former two, an application-specific virtual machine or an generic virtual machine (e.g., Java virtual machine) should be installed on each sensor node. Script and virtual machine code are not efficient to execute for a long time. In contrast, the native code executes much faster. For the native code, an alternative to the full image replacement scheme [38] is to use loadable modules. With loadable modules, only parts of the system need to be modified when a single application is changed. This approach requires dynamic linking and loading support on the OS, which is implemented in Contiki [16], SOS [17], RETOS [20], LiteOS [21].

Approaches to Improve Programming Convenience: The factors that impact application programming are various, e.g., programming models, programming languages, system libraries, etc. We focus our attention on programming models and languages here because their relevance to the OS design. Programming models and languages for sensor networks can further be classified into node level, group level, and network level [8]. We focus on node-level programming models and languages as they are most pertinent to the OS design.

Two well-known programming models in the sensornet community is event-driven programming and multi-threaded programming. As we have already mentioned in Section IV-B, some OSes (e.g., TinyOS) implement the event-driven model for efficiency while other OSes (e.g., Mantis OS) implement the multi-threaded model for programming convenience. Event and thread are not totally orthogonal, there is a large body of work devoted to implementing light-weight multi-threading mechanisms. TinyThread [43] implements multi-threading on top of TinyOS's event-driven kernel. While it supports thread-like programming, it does not support preemption. TinyOS Fiber [82] implements a thread model by a single system stack, and only supports one blocking context. Protothreads [27] can support blocking I/Os with a very low memory requirements, but it does not support local variables and the blocking I/O can only be called inside the top level of the thread. Y-Threads [83] separates the thread execution into two stacks. Each Y-Thread has its own stack upon which blocking calls occur, but all Y-Threads share a single stack where nonblocking calls can occur.

Two representative languages for sensornet OSes are nesC [26] and LiteC++ [21]. As mentioned in Section IV-B, nesC is usually not considered convenient for programming despite its advantages in program code analysis and optimizations. On the other hand, LiteC++ inherits features of modern OO languages. Hence it is typically considered more user-friendly and is convenient for programming than both C and nesC.

V. Evaluations

In this section, we will discuss evaluations of a sensornet OS by exploring several common performance metrics (Section V-A) and the impact of several design factors (Section V-B). We will also briefly discuss the technical support provided by state-of-the-art sensornet OSes (Section V-C).

A. Evaluation Metrics

Several metrics are common to evaluate the performance of a sensornet OS. Among them, reliability, reconfigurability, and programming convenience are difficult to evaluate quantitatively, and they are usually achieved to different extent in different OSes.

- **Reliability.** The OS together with the application should operate correctly for a long life-time.
- **Reconfigurability.** The OS should allow easy and efficient application modifications after deployment.

TABLE III
DIFFERENT DESIGN APPROACHES (↑ IS USED TO INDICATE THAT THE APPROACH HAS A POSITIVE IMPACT ON THE DESIGN GOAL; ↓ IS USED TO INDICATE THAT THE APPROACH HAS A NEGATIVE IMPACT ON THE DESIGN GOAL)

Approach \ Goals	Small Footprint	Energy Efficiency	Reliability	Real-Time	Reconfigurability	Programming Convenience
Static approach		↑	↑	↑	↓	↓
Dynamic approach		↓	↓	↓	↑	↑
Event-driven approach	↑	↑	↓			↓
Multi-threaded approach	↓	↓	↑			↑
Monolithic approach	↑	↓			↓	
Modular approach	↓	↑			↑	
Reliability approach	↓	↓	↑			
Reconfigurability approach	↓	↓			↑	
nesC (domain-specific)	↑	↑	↑			↓
LiteC++ (OO language)	↓	↓				↑

- **Programming Convenience.** The OS should provide a convenient programming environment for application programmers.

There are several other performance metrics to evaluate a sensornet OS quantitatively, which include:

- **CPU Utilization.** The CPU utilization should be kept low to save energy.
- **Energy Consumption.** The energy consumption should be kept low to ensure the longevity of a WSN.
- **OS RAM Usage.** The data RAM usage should be kept small.
- **OS Program Size.** The program size should be kept small.
- **Application Lines of Code (LoC).** Application LoC (after removing all blank lines and comments) should be kept low to facilitate application programming.
- **Application Source Code Size.** Application source code size (in bytes, after removing all blank lines and comments) should also be kept low to facilitate application programming.

B. Performance Comparisons

Current sensornet OSes adopt various approaches as described in Section IV-C to improve performance metrics or trade-off one performance metric for another. We briefly summarize the impact of several design factors in Table III.

The static approach increases system reliability as most system behaviors are predictable at compile-time. Real-time analysis can be carried out more easily on a static system. However, it decreases system reconfigurability and programming convenience as resource requirement cannot be changed dynamically, and, resource allocation must be safely estimated at design-time. The dynamic approach, on the other hand, increases system reconfigurability and programming convenience, but it decreases system reliability and real-time guarantees.

The event-driven approach is favored to generate a system with a small footprint and high energy efficiency. It decreases system reliability as an indefinite loop in an event can easily compromise the whole system. Besides, it sacrifices programming convenience. The multi-threaded approach incurs a larger implementation and run-time overhead. It is less energy-efficient because of the thread scheduling overhead. At the other hand, it can improve system reliability and programming convenience.

The monolithic approach is a good choice to generate a system with a small footprint when the application seldom needs to be changed. But it is less energy efficient to disseminate the native code when reprogramming. The modular approach improves system reconfigurability at the cost of extra implementation overheads. But it is more energy efficient to disseminate the native code when reprogramming.

Most approaches to increase reliability (e.g., preventing control hazards [22] and preventing data hazards [23]) and reconfigurability (e.g., loadable modules [36] and virtual machines [84]) will add the complexity of the system and impair energy efficiency and small-footprint OS design.

nesC [26], a representative of domain-specific languages, is effective in optimizing the program code size but incurs a large learning curve for application programmers. It can also improve reliability and energy efficiency of the system via static checks and static optimizations. LiteC++ [21], a representative of OO languages, on the other hand, incurs an additional implementation overhead but is more familiar to traditional application programmers, and is thus more convenient for application programming.

It is also worth mentioning that some approaches can only be combined to achieve a specific goal. For example, the

TABLE IV
HOMEPAGE OF DIFFERENT SENSORNET OSES

TinyOS	http://www.tinyos.net/
Contiki	http://www.sics.se/contiki/
SOS	https://projects.nesl.ucla.edu/public/sos-2x/
Mantis	http://mantis.cs.colorado.edu/
Nano-RK	http://www.nanork.org/
RETOS	http://retos.yonsei.ac.kr
LiteOS	http://www.liteos.net

TABLE V
HARDWARE PLATFORMS SUPPORTED

TinyOS	Mica2, MicaZ, TelosB/TMote Sky, IntelMote2, eyes, tinynode, IRIS, shimmer, TI CC2430 (testing)
SOS	Mica2, MicaZ, TelosB/Tmote Sky, XYZ
Contiki	ESB, TelosB/Tmote Sky
Mantis	Mica2, MicaZ, Telos, Mantis nymph
Nano-RK	MicaZ, FireFly
RETOS	MicaZ, TelosB/TMote Sky, TI CC2430
LiteOS	MicaZ, IRIS

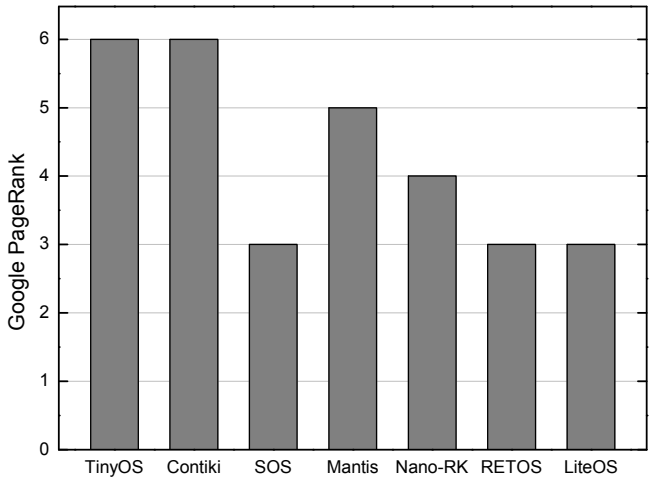

Fig. 1. Google PageRank

modular approach is usually implemented in a dynamic system where resources can be more effectively allocated at run-time.

C. Technical Support

In this subsection, we briefly discuss the technical support provided by different sensornet OSes. All OSes investigated in Section IV-A provide installation manuals and programming tutorials, which can be found in the corresponding homepages listed in Table IV. It is worth noting that TinyOS, as the de facto OS in the literature, provides the best technical support. Many real-world WSN projects are built upon TinyOS. Table V lists the hardware platforms currently supported in these OSes, which crudely reflects the efforts of OS developers. Note that Contiki provides abstractions for MCUs such as Atmel AVR and TI MSP430. Support for Mica series mote platforms still requires a port by implementing additional device drivers. Figure 1 shows the Google PageRank values of the homepages listed in Table IV, which crudely reflects the number of interested OS users. It is worth mentioning that, as of Nov. 24, 2008, SOS is no longer under active development because of the graduation of the core developers.

VI. RECOMMENDATIONS

This section presents some recommendations from the perspectives of OS developers and OS users.

A. Developers' Perspective

Start with Simple Designs. Most sensor nodes are severely resource constrained. In addition, with energy, form factor, and cost being emphasized considerations in designing sensor hardware platforms, we will continue to see highly constrained devices for many WSNs [22]. Hence, a sensornet OS should be small with simple designs. Developers should carefully evaluate the suitability of sophisticated OS features for PCs, and incorporate them into the design only when the benefits deserve the implementation overheads.

Build A Flexible Architecture. There is a variety of sensor node platforms, such as MicaZ, TelosB [85], inode, eyes, CC2430, etc. Therefore, it is necessary to build a flexible OS architecture in order to make it evolvable with hardware developments. On the other hand, there is also a variety of WSN applications, such as military surveillance [12], habitat monitoring [13], and infrastructure protection [14], etc. Hence, it is also necessary to build a flexible OS architecture in order to make it capable of supporting different applications.

Consider Application Scenarios and Potential Users. Many WSN research projects are highly application-driven [86]. The design of a sensornet OS should also consider its main application scenarios and potential users. OS designers should first setup the major design goals, such as described in Section V-B, according to application needs, e.g., if sensing data should be forwarded in a timely basis, then real-time guarantee is a desired OS feature. When the OS design goals are setup, designers can then refer to Table III to select the appropriate design approaches.

B. Users' Perspective

Consider Hardware Requirements. For large-scale applications, an important consideration is the cost factor, which impacts the choice of hardware platforms. OS users must therefore choose an OS that supports the specific hardware (refer to Table V). For a hybrid network application, the OS must then provides multi-platform support.

Understand Application Needs. Application requirement is another important consideration in selecting an appropriate OS. If high reliability is required, then static OSes, such as TinyOS, Nano-RK, are good choices; if the application needs frequent update, then dynamic modular OSes, such as Contiki, SOS, are good choices.

Evaluate Developing Cost. There are many different ways to develop a WSN application. If the application programmers have the legacy code, they would probably like to port the code onto a similar system. For example, if the legacy code is written in a thread-like style, then multi-threaded OSes, such as Mantis, RETOS, LiteOS, are good choices. If the application programmers begin from scratch, then they would probably like to seek for OSes with good technical support. In

this case, OSes with good technical support, such as TinyOS, Contiki, Mantis, are good choices.

VII. Conclusions and Future Work

OS support is important to facilitate the development and maintenance of WSNs. In this paper, we first examine the challenges of in the OS design space. We then introduce what constitutes a sensornet OS by describing its major components. Next, we provide an overview of existing work, present a taxonomy of state-of-the-art sensornet OSes, and discuss various approaches to address the design challenges. Finally we discuss evaluations of a sensornet OS and present some recommendations from the perspectives of OS developers and OS users.

Currently, sensornet OS research and development are still in the state of active development. In the process of WSN revolution, we will continue to see sensornet OS evolve with the emergence of new sensor hardware and new sensornet applications.

There are lots of open problems that need further investigation to make the OS provide stronger support for WSNs. Providing a convenient programming model and a suite of useful system services on the resource constrained sensor nodes is a continuing focus of current research. However, design trade-offs among small footprint, energy efficiency, reliability, real-time guarantees, reconfigurability, and programming convenience, need to be made with respect to different application scenarios. Approaches to address reliability issues and provide real-time support need further study to improve OS reliability and real-time performance. OS support for network management is currently under active research and development. While there is a large body of work devoted to wireless reprogramming and remote debugging, how to incorporate them into a unified OS framework is still to be considered.

With a stronger support from sensornet OSes, we envision that WSN application development will be greatly simplified.

Acknowledgment

The authors would like to thank the anonymous reviewers for their insightful comments. This work is supported by the National Basic Research Program of China (973 Program) under grant No. 2006CB303000, and in part by an NSERC Discovery Grant under grant No. 341823-07, NSERC Strategic Grant STPGP 364910-08, FQRNT 2010-NC-131844.

References

[1] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci, "Wireless sensor networks: A survey," *Computer Netw.*, vol. 38, pp. 393–422, 2002.

[2] Jiming Chen, Shibo He, Youxian Sun, Preetha Thulasiramanz, and Xuemin (Sherman) Shen. "Optimal flow control for utility-lifetime tradeoff in wireless sensor networks," *Comput. Netw.* (Elsevier), vol. 53, no. 18, pp. 3031-3041, 2009.

[3] C. E. Ilker Demirkol and F. Alagöz, "MAC protocols for wireless sensor networks: A survey," *IEEE Commun. Mag.*, vol. 44, no. 4, pp. 115–121, 2006.

[4] R. Rajagopalan and P. K. Varshney, "Data-aggregation techniques in sensor networks: A survey," *IEEE Commun. Surveys Tutorials*, vol. 8, no. 4, pp. 48–63, 2006.

[5] N. A. Pantazis and D. D. Vergados, "A survey on power control issues in wireless sensor networks," *IEEE Commun. Surveys Tutorials*, vol. 9, no. 4, pp. 86–107, 2007.

[6] Y. Zhou, Y. Fang, and Y. Zhang, "Securing wireless sensor networks: A survey," *IEEE Commun. Surveys Tutorials*, vol. 10, no. 3, pp. 6–28, 2008.

[7] X. Chen, K. Makki, K. Yen, and N. Pissinou, "Sensor network security: A survey," *IEEE Commun. Surveys Tutorials*, vol. 11, no. 2, pp. 52–73, 2009.

[8] R. Sugihara and R. K. Gupta, "Programming models for sensor networks: A survey," *ACM Trans. Sensor Netw. (TOSN)*, (to appear).

[9] Q. Wang, Y. Zhu, and L. Cheng, "Reprogramming wireless sensor networks: Challenges and approaches," *IEEE Netw. Mag.*, vol. 20, no. 3, pp. 48–55, 2006.

[10] M. Li and Y. Liu, "Underground structure monitoring with wireless sensor networks," in *Proc. ACM/IEEE IPSN*, 2007.

[11] Z. Yang, M. Li, and Y. Liu, "Sea depth measurement with restricted floating sensors," in *Proc. IEEE RTSS*, 2007.

[12] T. He, S. Krishnamurthy, J. A. Stankovic, T. A. L. Luo, R. Stoleru, T. Yan, L. Gu, and J. H. B. Krogh, "Energy-efficient surveillance system using wireless sensor networks," in *Proc. ACM MobiSys*, 2004.

[13] R. Szewczyk, A. Mainwaring, J. Polastre, and J. A. D. Culler, "An analysis of a large scale habitat monitoring application," in *Proc. ACM SenSys*, 2004.

[14] N. Xu, S. Rangwala, K. Chintalapudi, D. Ganesan, A. Broad, R. Govindan, and D. Estrin, "A wireless sensor network for structural monitoring," in *Proc. ACM SenSys*, 2004.

[15] TinyOS. [Online]. Available: http://www.tinyos.net

[16] A. Dunkels, B. Grönvall, and T. Voigt, "Contiki—a lightweight and flexible operating system for tiny networked sensors," in *Proc. EmNets*, 2004.

[17] C.-C. Han, R. Kumar, R. Shea, E. Kohler, and M. Srivastava, "A dynamic operating system for sensor nodes," in *Proc. ACM MobiSys*, 2005.

[18] S. Bhatti, J. Carlson, H. Dai, J. Deng, J. Rose, A. Sheth, B. Shucker, C. Gruenwald, A. Torgerson, and R. Han, "MANTIS OS: An embedded multithreaded operating system for wireless micro sensor platforms," *ACM/Kluwer Mobile Netw. Appl. J. (MONET), Special Issue Wireless Sensor Netw.*, vol. 10, pp. 563–579, 2005.

[19] A. Eswaran, A. Rowe, and R. Rajkumar, "Nano-RK: An energy-aware resource-centric RTOS for sensor networks," in *Proc. IEEE RTSS*, 2005.

[20] H. Cha, S. Choi, I. Jung, H. Kim, and H. Shin, "RETOS: Resilient, expandable, and threaded operating system for wireless sensor networks," in *Proc. ACM/IEEE IPSN*, 2007.

[21] Q. Cao, T. F. Adbelzaher, and J. A. Stankovic, "The LiteOS operating system: towards Unix-like abstractions for wireless sensor networks," in *Proc. ACM/IEEE IPSN*, 2008.

[22] L. Gu and J. A. Stankovic, "t-kernel: Providing reliable OS support to wireless sensor networks," in *Proc. ACM SenSys*, 2006.

[23] R. Kumar, E. Kohler, and M. Srivastava, "Harbor: software-based memory protection for sensor nodes (Poster)," in *Proc. ACM/IEEE IPSN*, 2007.

[24] W. Dong, C. Chen, X. Liu, K. Zheng, R. Chu, and J. Bu, "FIT: A flexible, lightweight, and real-time scheduling system for wireless sensor platforms," *IEEE Trans. Parallel Distributed Syst. (TPDS)*, vol. 21, no. 1, pp. 126-138, 2010.

[25] T. He, J. Stankovic, C. Lu, and T. Abdelzaher, "SPEED: A stateless protocol for real-time communication in sensor networks," in *Proc. IEEE ICDCS*, 2003.

[26] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler, "The nesC language: A holistic approach to networked embedded systems," in *Proc. ACM PLDI*, 2003.

[27] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali, "Protothreads: Simplifying event-driven programming of memory-constrained embedded systems," in *Proc. ACM SenSys*, 2006.

[28] A. Awan, S. Jagannathan, and A. Grama, "Macroprogramming heterogeneous sensor networks using cosmos," *ACM SIGOPS Operating Syst. Rev.*, vol. 41, no. 3, pp. 159–172, 2007.

[29] N. Kothari, R. Gummadi, T. Millstein, and R. Govindan, "Reliable and efficient programming abstractions for wireless sensor networks," in *Proc. ACM PLDI*, 2007.

[30] M. Hempstead, M. Welsh, and D. Brooks, "TinyBench: The case for a standardized benchmark suite for TinyOS based wireless sensor network devices (poster)," in *Proc. IEEE LCN*, 2004.

[31] J. R. Levine, *Linkers and Loaders*. Morgan Kaufmann, 2000.

[32] P. Buonadonna, J. Hill, and D. Culler, "Active message communication for tiny networked sensors," in *Proc. 20th Annual Joint Conf. IEEE Comput. Commun. Societies*, 2001.

[33] K. Klues, V. Handziski, C. Lu, A. Wolisz, D. Culler, D. Gay, and P. Levis, "Integrating concurrency control and energy management in device drivers," in *Proc. ACM SOSP*, 2007.
[34] J. Labrosse, *MicroC/OS-II, the real-time kernel, 2nd edition*. CMP Books, 2002.
[35] I. Wind River Systems, "Vxworks 5.4 datasheet." [Online]. Available: http://www.windriver.com/products/html/vxwks54_ds.html
[36] A. Dunkels, N. Finne, J. Eriksson, and T. Voigt, "Run-time dynamic linking for reprogramming wireless sensor networks," in *Proc. ACM SenSys*, 2006.
[37] W. Dong, C. Chen, X. Liu, J. Bu, and Y. Liu, "Dynamic linking and loading in networked embedded systems," in *Proc. IEEE MASS*, 2009.
[38] J. W. Hui and D. Culler, "The dynamic behavior of a data dissemination protocol for network programming at scale," in *Proc. ACM SenSys*, 2004.
[39] P. Levis, D. Gay, V. Handziski, J.-H. Hauer, B. Greenstein, M. Turon, J. Hui, K. Klues, C. Sharp, R. Szewczyk, J. Polastre, P. Buonadonna, L. Nachman, G. Tolle, D. Culler, and A. Wolisz, "T2: A SECOND GENeration OS for embedded sensor networks," TKN-05-007, Telecommunication Networks Group, Technical University Berlin, Tech. Rep., 2005.
[40] J. Yang, M. L. Soffa, L. Selavo, and K. Whitehouse, "Clairvoyant: A comprehensive source-level debugger for wireless sensor networks," in *Proc. ACM SenSys*, 2007.
[41] V. Krunic, E. Trumpler, and R. Han, "NodeMD: diagnosing node-level faults in remote wireless sensor systems," in *Proc. ACM MobiSys*, 2007.
[42] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister, "System Architecture Directions for Networked Sensors," in *Proc. ACM ASPLOS*, 2000.
[43] W. P. McCartney and N. Sridhar, "Abstractions for safe concurrent programming in networked embedded systems," in *Proc. ACM SenSys*, 2006.
[44] T. T. Alliance, "TinyOS 2.1: Adding threads and memory protection to TinyOS (poster)," in *Proc. ACM SenSys*, 2008.
[45] N. Cooprider, W. Archer, E. Eide, D. Gay, and J. Regehr, "Efficient Memory Safety for TinyOS," in *Proc. ACM SenSys*, 2007.
[46] N. Tsiftes, A. Dunkels, Z. He, and T. Voigt, "Enabling large-scale storage in sensor networks with the coffee file system," in *Proc. ACM/IEEE IPSN/SPOTS*, 2009.
[47] A. Dunkels, F. Österlind, and Z. He, "An adaptive communication architecture for wireless sensor networks," in *Proc. ACM SenSys*, 2007.
[48] R. Fonseca, O. Gnawali, K. Jamieson, S. Kim, P. Levis, and A. Woo, "The collection tree protocol (CTP), TinyOS extension proposals (TEP) 123," Aug. 2006.
[49] H. Kim and H. Cha, "Multithreading optimization techniques for sensor network operating systems," in *Proc. EWSN*, 2007.
[50] Crossbow Inc. [Online]. Available: http://www.xbow.com.cn/
[51] Q. Cao, D. Fesehaye, N. Pham, Y. Sarwar, and T. F. Adbelzaher, "Virtual battery: An energy reserve abstraction for embedded sensor networks," in *Proc. IEEE RTSS*, 2008.
[52] Q. Cao, T. F. Adbelzaher, J. Stankovic, K. Whitehouse, and L. Luo, "Declarative tracepoints: A programmable and application independent debugging system for wireless sensor networks," in *Proc. ACM SenSys*, 2008.
[53] 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," *ACM SIGOPS Operating Syst. Rev.*, vol. 36, no. 2, pp. 1–5, 2002.
[54] S. Park, J. W. Kim, K.-Y. Shin, and D. Kim, "A nano operating system for wireless sensor networks," in *Proc. 8th International Conf. Advanced Commun. Technol.*, 2006.
[55] K. Lorincz, B. rong Chen, J. Waterman, G. Werner-Allen, and M. Welsh, "Resource Aware Programming in the Pixie OS," in *Proc. ACM SenSys*, 2008.
[56] SenSpire OS [Online]. Available: http://eagle.zju.edu.cn/home/eos/senspire
[57] E. Trumpler and R. Han, "A Systematic framework for evolving TinyOS," in *Proc. EmNets*, 2006.
[58] J. Polastre, J. Hui, P. Levis, J. Zhao, D. Culler, S. Shenker, and I. Stoica, "A unifying link abstraction for wireless sensor networks," in *Proc. ACM SenSys*, 2005.
[59] C. T. Ee, R. Fonseca, S. Kim, D. Moon, A. Tavakoli, D. Culler, S. Shenker, and I. Stoica, "A modular network layer for sensornets," in *Proc. USENIX OSDI*, 2006.
[60] J. N. Al-Karaki and A. E. Kamal, "Routing techniques in wireless sensor networks: a survey," *IEEE Wireless Commun.*, vol. 11, pp. 6–28, 2004.
[61] J. Luo, D. Ye, X. Liu, and M. Fan, "A survey of multicast routing protocols for mobile ad-hoc networks," *IEEE Commun. Surveys Tutorials*, vol. 11, no. 1, pp. 78–91, 2009.
[62] S. S. Kulkarni and L. Wang, "MNP: Multihop network reprogramming service for sensor networks," in *Proc. IEEE ICDCS*, 2005.
[63] R. K. Panta, I. Khalil, and S. Bagchi, "Stream: Low overhead wireless reprogramming for sensor networks," in *Proc. IEEE INFOCOM*, 2007.
[64] K. Lin and P. Levis, "Data discovery and dissemination with DIP," in *Proc. ACM/IEEE IPSN*, 2008.
[65] C. . R. Manual. [Online]. Available: http://www.sics.se/~adam/contiki/contiki-2.1-doc/
[66] D. Gay, "Design of matchbox, the simple filing system for motes." [Online]. Available: http://www.tinyos.net/tinyos-1.x/doc/matchbox-design.pdf
[67] H. Dai, M. Neufeld, and R. Han, "ELF: An effcient log-structured flash system for micro sensor nodes," in *Proc. ACM SenSys*, 2004.
[68] W. Dong, C. Chen, X. Liu, and Y. Liu, "Performance of bulk data dissemination in wireless sensor networks," in *Proc. IEEE/ACM DCOSS*, 2009.
[69] P. J. Marrón, M. Gauger, A. Lachenmann, D. Minder, O. Saukh, and K. Rothermel, "FlexCup: A flexible and efficient code update mechanism for sensor networks," in *Proc. EWSN*, 2006.
[70] B. L. Titzer and J. Palsberg, "Vertical object layout and compression for fixed heaps," in *Proc. CASES*, 2007.
[71] B. Krishnamachari, *Networking Wireless Sensors*. Cambrige Uniersity Press, 2005.
[72] W. Ye, J. Heidemann, and D. Estrin, "An energy-efficient MAC protocol for wireless sensor networks," in *Proc. IEEE INFOCOM*, 2002.
[73] J. Polastre, J. Hill, and D. Culler, "Versatile low power media access for wireless sensor networks," in *Proc. ACM SenSys*, 2004.
[74] M. Buettner, G. V. Yee, E. Anderson, and R. Han, "X-MAC: A short preamble MAC protocol for duty-cycled wireless sensor networks," in *Proc. ACM SenSys*, 2006.
[75] J. Regehr, A. Reid, and K. Webb, "Eliminating stack overflow by abstract interpretation," *ACM Trans. Embedded Comput. Syst. (TECS)*, vol. 4, no. 4, pp. 6–28, 2005.
[76] K. Hänninen, J. Mäki-Turja, M. Bohlin, J. Carlson, and M. Nolin, "Determining Maximum Stack Usage in Preemptive Shared Stack Systems," in *Proc. IEEE RTSS*, 2006.
[77] C. Duffy, U. Roedig, J. Herbert, and C. J. Sreenan, "Adding preemption to TinyOS," in *Proc. EmNets*, 2007.
[78] K. Karenos and V. Kalogeraki, "Real-time traffic management in sensor networks," in *Proc. IEEE RTSS*, 2006.
[79] O. Chipara, C. Lu, and G.-C. Roman, "Real-time query scheduling for wireless sensor networks," in *Proc. IEEE RTSS*, 2007.
[80] X. Liu, Q. Wang, L. Sha, and W. He, "Optimal QoS sampling frequency assignment for real-time wireless sensor networks," in *Proc. IEEE RTSS*, 2003.
[81] W. Shu, X. Liu, Z. Gu, and S. Gopalakrishnan, "Optimal sampling rate assignment with dynamic route selection for real-time wireless sensor networks," in *Proc. IEEE RTSS*, 2008.
[82] M. Welsh and G. Mainland, "Programming sensor networks using abstract regions," in *Proc. USENIX NSDI*, 2004.
[83] C. Nitta, R. Pandey, and Y. Ramin, "Y-Threads: Supporting Concurrency in Wireless Sensor Networks," in *Proc. IEEE/ACM DCOSS*, 2006.
[84] P. Levis and D. Culler, "Maté: a tiny virtual machine for sensor networks," in *Proc. ASPLOS*, 2002.
[85] J. Polastre, R. Szewczyk, and D. Culler, "Telos: Enabling Ultra-Low Power Wireless Research," in *Proc. ACM/IEEE IPSN/SPOTS*, 2005.
[86] L. M. Ni, Y. Liu, and Y. Zhu, "China's National Research Project on Wireless Sensor Networks," *IEEE Wireless Commun.*, vol. 14, pp. 78–83, 2007.

Wei Dong received his BS degree in the College of Computer Science from Zhejiang University, and achieved all credits in the Advanced Class of Engineering Education (ACEE) of Chu Kechen Honors College from Zhejiang University in 2005. He is currently a Ph.D. student in the College of Computer Science of Zhejiang University. His research interests include networked embedded systems and wireless sensor networks. He is a student member of the IEEE.

Chun Chen received his Bachelor of Mathematics degree from Xiamen University, China, in 1981, and his Masters and Ph.D. degrees in Computer Science from Zhejiang University, China, in 1984 and 1990 respectively. He is a professor in College of Computer Science, the Dean of College of Software, and the Director of Institute of Computer Software at Zhejiang University. His research activity is in image processing, computer vision, CAD/CAM, CSCW and embedded system.

Xue Liu received the BS degree in applied mathematics and the MEng degree in control theory and applications from Tsinghua University and the Ph.D. degree in computer science from the University of Illinois, Urbana-Champaign, in 2006. He is currently an assistant professor in the School of Computer Science, McGill University. He was briefly with the Hewlett-Packard Laboratories (HP Labs) and IBM T.J. Watson Research Center. His research interests include real-time and embedded computing, performance and power management of server systems, sensor networks, fault tolerance, and control. He has filed 5 patents, and published more than 50 research papers in international journals and major peer-reviewed conference proceedings. He is a member of the IEEE and the ACM.

Jiajun Bu received the B.S and Ph.D. degrees in Computer Science from Zhejiang University, China, in 1995 and 2000, respectively. He is a professor in College of Computer Science and the deputy dean of the Department of Digital Media and Network Technology at Zhejiang University. His research interests include embedded system, mobile multimedia, and data mining. He is a member of the IEEE and the ACM.