A Comparative Study on Operating System for Wireless Sensor Networks

Thang Vu Chien, Hung Nguyen Chan, and Thanh Nguyen Huu
Thai Nguyen University of Information and Communication Technology Quyet Thang Commune, Thai Nguyen City, Vietnam

E-mail: vcthang@ictu.edu.vn, chanhung@mail.hut.edu.vn, thanhnh@mail.hut.edu.vn

Abstract—**Wireless Sensor Networks (WSNs) have been the subject of intensive research over the last years. WSNs consist of a large number of sensor nodes, and are used for various applications such as building monitoring, environment control, wild-life habitat monitoring, forest fire detection, industry automation, military, security, and health-care. 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 operating systems (OSes) emerging in the sensor network community to facilitate developing WSN applications. In this paper, we present OS for WSNs. We begin by presenting the major issues for the design of OS for WSNs. Then, we examine some existing OSes for WSNs, including TinyOS, Contiki, and LiteOS. Finally, we present a comparison of these OSes by examining some important OS features. We believe that our work will help both OS developers and OS users. With OS developers, they will know what has worked in previous OSes and what has not. With OS users, they know the features of existing sensor network OSes, so they can select a sensor network OS that is the most appropriate for their application.**

I. Introduction

A WSN is generally composed of a centralized station (sink) and tens, hundreds, or perhaps thousands of tiny sensor nodes. With the integration of information sensing, computation, and wireless communication, these devices can sense the physical phenomenon, (pre-)process the raw information, and share the processed information with their neighboring nodes.

Typical sensor nodes are equipped with a sensor, a microprocessor or microcontroller, a memory, a radio transceiver, and a battery. Therefore, these hardware components should be organized in a way that makes them work correctly and effectively without a conflict in support of the specific applications for which they are designed. Each sensor node needs an OS that can control the hardware, provide hardware abstraction to application software, and fill in the gap between applications and the underlying hardware.

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 traditional OS is system software that operates between application software and hardware and is often designed for workstations and PCs with plenty of resources. This is usually not the case with sensor nodes in WSNs. There are also embedded OSes such as VxWorks [1] and WinCE [2], none of which is specially designed for data-centric WSNs with constrained resources. Sensors usually have a slow processor and small memory, different from most current systems. These parameters should be kept in mind in the process of OS design for WSN nodes.

In this paper, we identify several major issues for the design of OS for WSNs. By examining some existing OSes for WSNs, we hope that our work may allow research community to know the strengths and weaknesses of a number of different OSes.

The rest of this paper is organized as follows. Section II presents the major issues for the design of sensor network OS. Section III examines some existing OSes for WSNs, including TinyOS, Contiki, and LiteOS. Section IV presents a comparison of these OSes. Finally, we conclude this paper in Section V.

II. Operating System Design Issues

Traditional OSes are system software, including programs that manage computing resources, control peripheral devices, and provide software abstraction to the application software. Traditional OS functions are therefore to manage processes, memory, CPU time, file system, and devices. This is often implemented in a modular and layered fashion, including a lower layer of kernels and a higher layer of system libraries. Traditional OSes are not suitable for WSNs because WSNs have constrained resources and diverse data-centric applications, in addition to a variable topology. WSNs need a new type of operating system, considering their special characteristics. There are several issues to consider when designing sensor network OS.

A. Process Management and Scheduling

The traditional OS provides process protection by allocating a separate memory space (stack) for each process. Each process maintains data and information in its own space. But this approach usually causes multiple data copying and context switching between processes. This is obviously not energy efficient for WSNs. Sensor network OSes should provide efficient resource management mechanisms in order to allocate microprocessor time and limited memory. The CPU time and limited memory must be scheduled and allocated for processes carefully to guarantee fairness (or priority if required).

B. Memory Management

Memory is often allocated exclusively for each process/task in traditional OSes, which is helpful for protection and security of the tasks. Since sensor nodes have small memory, another approach, sharing, can reduce memory requirements.

C. Kernel Model

The event-driven and finite state machine (FSM) models have been used to design microkernels for WSNs. The event-driven model may serve WSNs well because they look like event-driven systems. An event may comprise receiving a packet, transmitting a packet, detection of an event of interest, alarms about energy depletion of a sensor node, and so on. The FSM-based model is convenient to realize concurrency, reactivity, and synchronization.

D. Energy Efficiency

Sensor nodes provide very limited battery lifetime. On the other hand, guaranteeing sensor networks to operate for 3 to 5 years is a very desirable objective. Sensor network OS should support power management, which helps to extend the system lifetime and improve its performance. For example, the operating system may schedule the process to sleep when the system is idle, and to wake up with the advent of an incoming event or an interrupt from the hardware.

E. Application Program Interface

Sensor nodes need to provide modular and general APIs for their applications. The APIs should enable applications access the underlying hardware. This may allow access and control of hardware directly, to optimize system performance.

F. Code Upgrading and Reprogramming

Since the behavior of sensor nodes and their algorithms may need to be adjusted either for their functionality or for energy conservation, the operating system should be able to reprogram and upgrade.

G. 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 sensor network OS and is the primary reason why so many sophisticated embedded OSes can not be easily ported to sensor nodes.

H. Real-Time Guarantee

As most sensor network 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.

I. Reliability

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

III. Existing Operating Systems for Wireless Sensor Networks

Over the years, we have seen various OSes emerging in the sensor network community [3]. The most prestigious works include TinyOS [4], Contiki [5], SOS [6], Mantis OS [7], Nano-RK [8], RETOS [9] and LiteOS [10]. In this paper, we present only TinyOS, Contiki (many interested OS users), and LiteOS (the newest sensor network OS).

A. TinyOS

TinyOS [4], developed in UC Berkeley, is perhaps the earliest sensor network OS in the literature [11]. The design of TinyOS allows application software to access hardware directly when required. TinyOS is a tiny microthreaded OS that attempts to address two issues: how to guarantee concurrent data flows among hardware devices, and how to provide modularized components with little processing and storage overhead. These issues are important since TinyOS is required to manage hardware capabilities and resources effectively while supporting concurrent operation in an efficient manner. TinyOS uses an event-based model to support high levels of concurrent application in a very small amount of memory. Compared with a stack-based threaded approach, which would require that stack space be reserved for each execution context, and because the switching rate of execution context is slower than in an event-based approach, TinyOS achieves higher throughput. It can rapidly create tasks associated with an event, with no blocking or polling. When CPU is idle, the process is maintained in a sleep state to conserve energy.

Fig. 1 illustrates the basic architecture of TinyOS. TinyOS includes a tiny scheduler and a set of components.

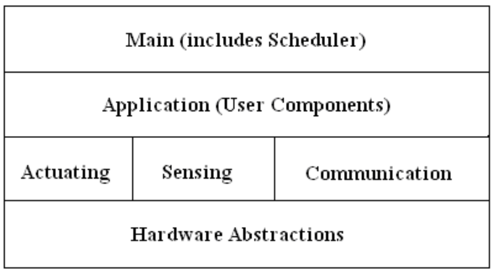

Fig. 1. TinyOS architecture.

The scheduler schedules operation of those components. Each component consists of four parts: command handlers, event handlers, an encapsulated fixed-size frame, and a group of tasks. Commands and tasks are executed in the context of the frame and operate on its state. Each component will declare its commands and events to enable modularity and easy interaction with other components. The current task scheduler in TinyOS is a simple FIFO mechanism whose scheduling data structure is very small, but it is power efficient since it allows a processor to sleep when the task queue is empty and while the peripheral devices are still running. The frame is fixed in size and is assigned statically. It specifies the memory requirements of a component at compile time and removes the overhead from dynamic assignment. Commands are nonblocking requests made to the low-level components. Therefore, commands do not have to wait a long time to be executed. A command provides feedback by returning status indicating whether it was successful (e.g., in the case of buffer overrun or of timeout). A command often stores request parameters into its frame and conditionally assigns a task for later execution. The occurrence of a hardware event will invoke event handlers. An event handler can store information in its frame, assign tasks, and issue high-level events or call low-level commands. Both commands and events can be used to perform a small and usually fixed amount of work as well as to preempt tasks. Tasks are a major part of components. Like events, tasks can call low-level commands, issue high-level events, and assign other tasks. Through groups of tasks, TinyOS can realize arbitrary computation in an event-based model. The design of components makes it easy to connect various components in the form of function calls. In order to provide a better support for the component architecture and execution model of TinyOS, the nesC language [12] was designed for programming based on TinyOS. TinyOS has a component-based programming model, codified by the nesC language.

This WNS operating system defines three types of components: hardware abstractions, synthetic hardware, and high-level software components. Hardware abstraction components are the lowest-level components. They are actually the mapping of physical hardware such as Input/Output (I/O) devices, a radio transceiver, and sensors. Each component is mapped to a certain hardware abstraction. Synthetic hardware components are used to map the behavior of advanced hardware and often sit on the hardware abstraction components. TinyOS designs a hardware abstract component called the Radio-Frequency Module (RFM) for the radio transceiver, and a synthetic hardware component called radio byte, which handles data into or out of the underlying RFM.

TinyOS supports a wide range of hardware platforms and has been used on several generations of sensor nodes. Supported processors include the Texas Instruments MSP430 and the Atmel AVR. TinyOS applications may be compiled to run on any of these platforms without modification.

B. *Contiki Operating System*

The Contiki operating system [5] is an open source operating system for networked embedded systems in general, and wireless sensor nodes in particular. It is developed by a team of developers from the industry and academia. The Contiki project is lead by Adam Dunkels.

Typically, a running Contiki system consists of the kernel, libraries, the program loader, and a set of processes. Communication between processes always goes through the kernel, which does not provide a hardware abstraction layer, but lets device drivers and applications communicate directly with the hardware.

A process is defined by an event handler function and an optional poll handler function. The process state is held in the process' private memory and the kernel only keeps a pointer to the process state. All processes share the same address space and do not run in different protection domains. Interprocess communication is done by posting events.

Looking at it from a higher perspective, the Contiki system is partitioned into two parts: the core and the loaded programs as shown in Fig. 2. Typically, the core consists of the Contiki kernel, the program loader, the most commonly used parts of the language run-time and support libraries, and a communication stack with device drivers for the communication hardware.

The core is compiled into a single binary image and is usually not modified after deployment, although it is possible to use a special boot loader to overwrite or patch the core. Programs are loaded into the system by the program loader. The program loader is in charge of loading/unloading the programs into the system either by using the communication stack or directly attached storage (such as EEPROM).

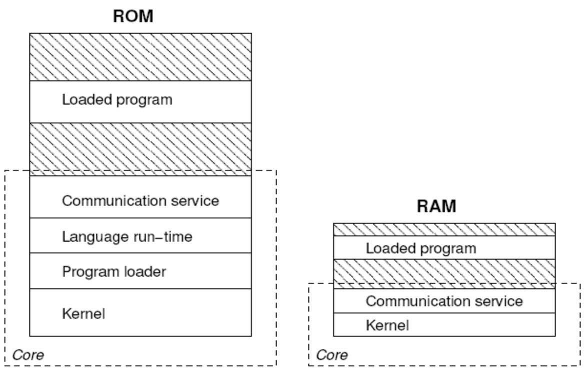

Fig. 2. Contiki system.

The Contiki kernel consists of a lightweight event scheduler that dispatches events to running processes and periodically calls processes' polling handlers. All program execution is triggered either by events dispatched by the kernel or through the polling mechanism. The kernel does not preempt an event handler once it has been scheduled. The kernel supports two kinds of events: asynchronous and synchronous events. In addition to the events, the kernel provides a polling mechanism. Polling can be seen as high priority events that are scheduled in-between each asynchronous event.

Contiki was the first operating system for wireless sensor nodes that provided IP communication with the uIP TCP/IP stack. In 2008, the Contiki system incorporated uIPv6, the world's smallest IPv6 stack. The footprints of the uIP and uIPv6 stacks are small: less than 5 kB for the uIP stack and approximately 11 kB for uIPv6. This makes them suitable for use in the constrained environment of a wireless sensor node.

Both the Contiki system and applications for the system are implemented in the C programming language. Because Contiki is implemented in C, it is highly portable. Contiki has been ported to a number of microcontroller architectures, including the Texas Instruments MSP430 and the Atmel AVR.

C. LiteOS

LiteOS [10], developed in the University of Illinois at Urbana Champaign, is designed to provide a traditional Unix-like environment for programming WSN applications. It includes: a hierarchical file system and a wireless shell interface for user interaction using UNIX-like commands; kernel support for dynamic loading and native execution of multithreaded applications; and online debugging, dynamic memory, and file system assisted communication stacks. LiteOS also supports software updates through a separation between the kernel and user applications, which are bridged through a suite of system calls.

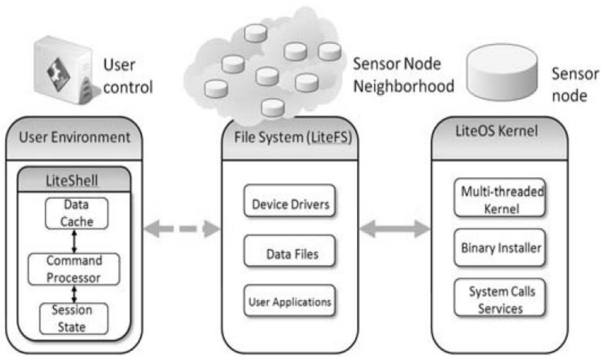

Fig. 3. LiteOS architecture.

Fig. 3 shows the overall architecture of the LiteOS operating system, partitioned into three subsystems: LiteShell, LiteFS, and the Kernel. Implemented on the base station PC side, the LiteShell subsystem interacts with sensor nodes (motes) only when a user is present. Therefore, LiteShell and LiteFS are connected with a dashed line in this figure.

The LiteShell subsystem provides Unix-like commandline interface to motes. This shell runs on the base station PC side. Therefore, it is a front-end that interacts with the user. The motes do not maintain command-specific state, and only respond to translated messages (represented by compressed tokens) from the shell, which are sufficiently simple to parse.

The interfaces of LiteFS provide support for both file and directory operations. The APIs of LiteFS can be exploited in two ways; either by using shell commands interactively, or by using application development libraries.

The kernel subsystem of LiteOS takes the thread approach, but it also allows user applications to handle events using callback functions for efficiency. It implements both priority-based scheduling and round-robin scheduling in the kernel. It also support dynamic loading and un-loading of user applications, as well as a suite of system calls for the separation between kernel and applications.

The LiteOS 2.0 is the latest version of LiteOS. It runs on the following platforms: MicaZ as target board, and MIB510/MIB520 as programming boards. Unlike 1.0, LiteOS 2.0 is closely integrated with AVR Studio 5.0. This brings multiple advantages, such as IDE editing, debugging, and built-in JTAG support. Due to a problem of compatiability between IRIS and AVR Studio, IRIS mote support will be added in version 2.1.

IV. Comparison

In this section, we will present a comparison between TinyOS, Contiki, and LiteOS by examining some important OS features (as summarized in Table I).

A. Static/Dynamic System

TinyOS is a static system, so application programmers must allocate all of the resources at design-time. On the other hand, Contiki and LiteOS are dynamic systems, and application programmers can allocate and deallocate resources at run-time. Dynamic systems are more flexible, and thus are more suitable for dynamically changing environments.

B. Monolithic/Modular System

Whereas TinyOS is a monolithic system, Contiki and LiteOS are modular systems. In monolithic systems, an application is compiled with the OS as a monolithic program. On the other hand, in modular systems, it is compiled into an individual program module that is loadable by the OS kernel. Modular systems are more flexible when the individual application needs to be frequently modified through network reprogramming.

C. Networking Support

Networking support in sensor network OSes provides communication schemes between sensor nodes. TinyOS uses a lightweight Active Message (AM) based communication stack. Contiki contains two communication stacks: uIP and Rime [13]. uIP is a small RFC-compliant TCP/IP stack that makes it possible for Contiki to communicate over the Internet. Recently, Contiki additionally implements uIPv6 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. LiteOS provides additional support for communications of files among a set of sensor nodes, using traditional Unix-like shell commands.

D. Event Based Programming

In event based programming systems, application programmers must manually maintain the application state and use split-phase I/Os. Event based programming systems 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. TinyOS, Contiki and LiteOS support event based programming.

E. Multi-Threading Support

In multi-threaded systems, 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 [14], TOSThreads [15] are thread libraries based on TinyOS; Contiki supports preemptive multi-threading via a library on top of the event-driven kernel [5], and it also implements a lightweight threading mechanism called protothreads [16]. LiteOS has a multi-threaded kernel to run applications as threads concurrently [10].

F. Wireless Reprogramming

Reprogramming support allows developers are able to install or update a new application to a network of sensor nodes wirelessly. Deluge [17] 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 [18] 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 Contiki [5] and LiteOS [10].

G. File Systems

With TinyOS, Matchbox [19] and ELF [20] provide single-level file organizations and basic abstractions for file operations such as reading and writing. Contiki provides a flash-based file system, Coffee [21], for storing data inside the sensor network. The file system allows multiple files to coexist on the same physical on-board flash memory. LiteOS supports hierarchical file organization and wireless shell interface for user interaction using UNIX-like commands [10].

V. Conclusion

In this paper, we presented OS for WSNs and several major issues for the design of sensor network OS. By examining some existing sensor network OSes, we know the strengths and weaknesses of a number of different OSes. The contributions of this paper are twofold. First, we identify several major issues for the design of sensor network OS, such as memory requirement, process management and scheduling, kernel model, generic application programming interfaces, effective code distribution and upgrades, energy-efficient, real-time guarantee, and reliability. Second, our work may allow research community to know the features of a number of different OSes. This work is valuable with both OS developers and OS users. With OS developers, they will know what has worked in previous OSes and what has not. With OS users, they know the features of existing sensor

TABLE I.
A COMPARISON BETWEEN TINYOS, CONTIKI AND LITEOS

Features	TinyOS	Contiki	LiteOS
Publication (Year)	ASPLOS (2000)	EmNets (2004)	IPSN (2008)
Website	www.tinyos.net	www.sics.se/ contiki	www.liteos.net
Static/Dynamic System	Static	Dynamic	Dynamic
Monolithic/Modular System	Monolithic	Modular	Modular
Networking Support	Active Message	uIP. uIPv6, Rime	File-Assisted
Real-Time Guarantee	No	No	No
Language Support	nesC	C	LiteC++
Event Based Programming	Yes	Yes	Yes (through callback functions)
Multi-Threading Support	Partial (through TinyThreads)	Yes (also supports Protothreads)	Yes
Wireless reprogramming	Yes	Yes	Yes
File Sytem	Single level (ELF, Matchbox)	Coffee	Hierarchical Unix-like
Platform Support	Mica, Mica2, MicaZ, TelosB, Tmote, XYZ, IRIS, Tinynode, Eyes, Shimmer	Tmote, TelosB, ESB, AVR MCU, MSP430 MCU	MicaZ, IRIS, AVR MCU
Simulator	TOSSIM, Power Tossim	Cooja, MSPSim, Netsim	Through AVRORA

network OSes, so they can select a sensor network OS that is the most appropriate for their application.

REFERENCES

[1] Vxwork. [Online]. Available: http://www.windriver.com/products/vxworks/

[2] WinCE. [Online]. Available: http://www.microsoft.com/windowsembedded/en-us/windows-embedded.aspx

[3] Lalit Saraswat, Pankaj Singh Yadav, "A comparative analysis of wireless sensor network operating systems," Int J Engg Techsci Vol 1(1) 2010, 41-47.

[4] TinyOS. [Online]. Available: http://tinyos.net/

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

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

[7] 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.

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

[9] 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.

[10] 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.

[11] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister, "System Architecture Directions for Networked Sensors," in *Proc. ACM ASPLOS*, 2000.

[12] 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.

[13] Contiki [Online]. Available: http://www.sics.se/~adam/contiki/docs/

[14] W. P. McCartney and N. Sridhar, "Abstractions for safe concurrent programming in networked embedded systems," in *Proc. ACM SenSys*, 2006.

[15] T. T. Alliance, "TinyOS 2.1: Adding threads and memory protection to TinyOS (poster)," in *Proc. ACM SenSys*, 2008.

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

[17] J. W. Hui and D. Culler, "The dynamic behavior of a data dissemination protocol for network programming at scale," in *Proc. ACM SenSys*, 2004.

[18] 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.

[19] D. Gay, "Design of matchbox, the simple filing system for motes." [Online]. Available: http://www.tinyos.net/tinyos-1.x/doc/matchbox-design.pdf

[20] H. Dai, M. Neufeld, and R. Han, "ELF: An effcient log-structured flash system for micro sensor nodes," in *Proc. ACM SenSys*, 2004.

[21] 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*, 2009.