A Comparative Analysis of Wireless Sensor Network Operating Systems

Lalit Saraswat\(^1\) and Pankaj Singh Yadav\(^2\)

\(^1\)Department of Computer science, Raj Kumar Goel Institute of Technology, Ghaziabad, India
\(^2\)Department of Information Technology, Raj Kumar Goel Institute of Technology, Ghaziabad, India

ABSTRACT

Wireless sensor networks are composed of large numbers of tiny networked devices that communicate. Many sensor networking applications such as surveillance and environmental monitoring are time-sensitive in nature. Two different operating system types are currently considered for sensor networks: event driven and multithreaded. The Operating systems employed for WSN (wireless sensor networks) are either satisfied with only one or two application classes or unsuitable for strict-constrained resources. In view of a variety of WSN applications, there is a need of developing a self adaptable and self-configurable embedded real-time operating system (RTOS). In recent years, the availability of cheap and small micro sensor node and low power wireless communication give a contribution of enhanced developments of wireless sensor network application in real society. A lightweight embedded resource kernel with rich functionality and timing support is practical and constitutes a simple and alternative paradigm for supporting distributed sensing tasks. This paper will make a comparative study of various operating systems related to wireless sensor network.

KEYWORDS

Mantis, Contiki, TinyOS[2], LIMOS, LiteOS[6], PAVENET OS, Nano-RK, Nano-Qplus[8]

1. INTRODUCTION

Wireless sensor networks are composed of large numbers of tiny sensor devices with wireless communication capabilities. The sensor devices autonomously form networks through which sensor data is transported. The sensor devices are often severely resource constrained. For the designer of an operating system for sensor nodes, the challenge lies in finding lightweight mechanisms and abstractions that provide a rich enough execution environment while staying within the limitations of the constrained devices. Operating systems for sensor nodes follow either one of two different design concepts, event-driven and multi-threaded. In event-driven systems every action an operating system has to perform is triggered by an event (e.g. a timer, an interrupt indicating new sensor readings or an incoming radio packet). An example of such an operating system is TinyOS[2]. The second approach follows the multi-threaded operating system concept.

The operating system multiplexes execution time between the different tasks, implemented as threads. While switching from one thread to another, the current context has to be saved and the new context must be restored. An example of such an operating system for sensor nodes is MANTIS[4]. Based upon two classical system modes, i.e. event-driven and multi-threading, some Operating System can operate in either event-driven mode or multi-threading mode to minimize resource requirement and improve system efficiency according to practical application environments. An example of such an operating system is LIMOS[7]. To map a sensor network into a UNIX-like file system, and supports extremely resource-constrained nodes LiteOS[6] operating system has been developed. To reduce the preemption overhead, PAVENET OS has been formed it uses a characteristic of wireless sensor nodes: tasks can be categorized as real-time tasks or best-effort tasks. This paper is divided in two sections first section will discuss about individual properties of each operating system and further second section will elaborate the comparison of them.

2.1 TINY OS

TinyOS[2] is composed by scheduler and a series of modules. The application programs and modules are compiled together to be a system. TinyOS[2] execute operations based on events, and the event module allows the subsequent operations run in a lesser space. In TinyOS, when an event is triggered, all the tasks related to the event that send out the signal are disposed rapidly. After that event and all the related tasks are disposed over, the untapped CPU will be turned into the SLEEP mode rather than searching the next dynamic event actively. The event-driven mode of TinyOS makes the system use the resource of CPU effectively. TinyOS uses three associated portions to manage the power. First, every of the equipment can stop itself by call the command StdControl.stop. Secondly, TinyOS will check the I/O pin and the control register of the processor to identify the processor’s state through the component HPL Power Management. Last, the timer of TinyOS can work in the lowest power cost mode in which most of the processors run in their power down mode. TinyOS tasks are deferred function calls and are placed in a simple FIFO task-queue for execution. TinyOS tasks are taken sequentially from the queue and are run to completion. Once running, the TinyOS task can not be interrupted (preempted) by another TinyOS task. Event handlers are triggered in response to a hardware interrupt and are able to preempt the execution of a currently running TinyOS task.

2.2 MANTIS
The Mantis system uses a traditional preemptive multi-threaded model of operation. Mantis enables reprogramming of both the entire operating system and parts of the program memory by downloading a program image onto EEPROM, from where it can be burned into flash ROM. Due to the multi-threaded semantics, every Mantis[4] program must have stack space allocated from the system heap, and locking mechanisms must be used to achieve mutual exclusion of shared variables. The MANTIS[4] scheduler dynamically allocates a memory pool to store the stack and processor registers for each thread. Each task the operating system must support can be implemented - using standard C - as a separate MANTIS thread. In the MANTIS implementation, the packet-processing task has a higher priority than the sensing task.

2.3 CONTIKI

Most operating systems for embedded systems require that a complete binary image of the entire system is built and downloaded into each device. The binary includes the operating system, system libraries, and the actual applications running on top of the system. In contrast, Contiki has the ability to load and unload individual applications or services at run-time. In most cases, an individual application is much smaller than the entire system binary and therefore requires less energy when transmitted through a network. Additionally, the transfer time of an application binary is less than that of an entire system image. The currently available sensor platforms carry completely different sets of sensors and communication devices. The preemptive multi-threading in Contiki is similar to fibers and the lightweight fibers approach by Welsh and Mainland [1]. Unlike the lightweight fibers, Contiki does not limit the number of concurrent threads to two. Furthermore, unlike fibers, threads in Contiki support preemption.

```
Figure 1. Partitioning into core and loaded programs.
```

A Contiki system is partitioned into two parts: the core and the loaded programs as shown in Figure 1. The partitioning is made at compile time and is specific to the deployment in which Contiki is used. 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.

2.4 μC/OS-II

μC/OS-II is a real-time preemptive multitasking embedded OS kernel, which is popular with portable, scalable and ease of use. The μC/OS-II provides a number of key functionalities needed by networked embedded applications, such as multitasking, synchronization, timer management, memory management. The network protocol stack of μC/OS-II is composed of MAC and network layers. The MAC layer of μC/OS-II has a major role controlling the medium access for collision avoidance. The network layer of μC/OS-II supports a variety routing algorithms in wireless sensor networks. the μC/OS-II also enables micro sensor nodes to natively interleave complex tasks with time-sensitive tasks, thereby mitigating the bounded buffer producer-consumer problem. Additionally, the security and reliability is helpful to construct robust wireless sensor networks.

2.5 LIMOS

LIMOS (Lightweight Multi-threading Operating System) is a native configurable hybrid operating system that can thus operate in either event-driven mode or multi-threading mode to minimize resource requirement and improve system efficiency according to practical application environments. LIMOS[7] adopts a component-based multi-level system architecture: action, thread and event. In LIMOS, all kinds of data exchanges, no matter interior interactions (between components, i.e. event and thread) or exterior interactions (generally with external peripherals, including sensors, actuator and a variety of interface devices etc), have been implemented via tuple space. There is thus no interaction between threads and events.

LIMOS is a smart, resource-aware, low-energy and distributed real-time micro-kernel. It adopts the action/event/thread component-based multi-level system architecture. As the result of multi-level structure, LIMOS adopts a two-level scheduling policy: ‘non pre-emption priority’ high level scheduling for events and ‘preemptive priority’ low level scheduling for threads. Since LIMOS is dedicated to strict resource constrained embedded applications, especially for WSN nodes, it consumes tiny resources, i.e. memory, energy and CPU. LIMOS can operate at different operation modes (event-driven, multi-threading), having very little memory requirement (<5Kbytes) comparing with most of RTOSs.

2.6 LiteOS

LiteOS, a multi-threaded operating system that provides Unix-like abstractions for wireless sensor networks. Aiming to be an easy-to-use platform, LiteOS offers a number of novel features, including:

1. a hierarchical file system and a wireless shell interface for user interaction using UNIX-like commands;
(2) kernel support for dynamic loading and native execution of multithreaded applications; and
(3) online debugging, dynamic memory, and file system assisted communication stacks. LiteOS[6] also supports software updates through a separation between the kernel and user applications, which are bridged through a suite of system calls. LiteOS differs from both current sensor network operating systems and more conventional embedded operating systems. Compared to the former category, such as TinyOS, LiteOS provides a more familiar environment to the user. Its features are either not available in existing sensor network operating systems, such as the shell and the hierarchical file system, or are only partially supported.

LiteOS operating system, partitioned into three subsystems: LiteShell, LiteFS, and the kernel. LiteOS provides a wireless node mounting mechanism through a file system called LiteFS. Much like connecting a USB drive, a LiteOS node mounts itself wirelessly to the root file system of a nearby base station. Moreover, analogously to connecting a USB device (which implies that the device has to be less than a USB-cable-length away), our wireless mount currently supports devices within wireless range, where security concerns are relevant, an authentication mechanism is needed between the base station and mounted motes. Low-cost authentication mechanisms for sensor networks are provided by this operating system.

2.7 Nano-Qplus

Nano-Qplus[8] is a new multi-threaded, lightweight, and low-power sensor network operating system integrated with a general-purpose single-board hardware platform to enable flexible and rapid prototyping of WSN. Nano-Qplus uses the notion of task to describe a piece of code that needs to be executed. Usually, there are several tasks activities at the same time and a task scheduler decides the run order. Nano-Qplus optionally provide a variety of schedulers such as simple non-preemptive FIFO scheduler, timed FIFO scheduler. Nano-Qplus contains the MAC protocol algorithm based on IEEE 802.15.4 formulating multi-hop peer-to-peer network formation.

2.8 Nano-RK

Many sensor networking applications such as surveillance and environmental monitoring are time-sensitive in nature. To support such applications, we design and implement Nano-RK[5], a reservation-based real-time operating system (RTOS) with multi-hop networking support for use in wireless sensor networks. Nano-RK, a small-footprint embedded real-time operating system with networking support.

Nano-RK supports the classical operating system multitasking abstractions allowing sensor application developers to work in a familiar paradigm resulting in short learning curves, quicker application development times and improved productivity. Nano-RK[5] provide these kind of support (1) Multitasking (2) Networking Stack Support (3) Priority-based Preemption (4) Timeliness and Schedulability (5)Battery Lifetime Requirements (6)Enforcement of Resource Usage Limits (7)Unified Sensor Interface Abstraction (8)Small Footprint

2.9 PAVENET OS

PAVENET OS provides hybrid multithreading: preemptive multithreading and cooperative multithreading. Both of the multithreading is optimized for two kinds of task on wireless sensor networks, and the kinds are real-time tasks and best-effort tasks. PAVENET OS can efficiently perform hard real-time tasks that cannot be performed by TinyOS. To realize the hard real time feature, PAVENET OS is designed with a thread model and enabling preemption. The enabling preemption causes two problems. First, the preemption induces huge overhead for checking task priorities and saving CPU context. Second, the preemption induces a conflict management problem among tasks. To reduce the preemption overhead, PAVENET OS uses a characteristic of wireless sensor nodes: tasks can be categorized as real-time tasks or best-effort tasks. PAVENET OS provides two kinds of multithreading, which are preemptive multithreading and cooperative multithreading. The preemptive multithreading is optimized for the real-time tasks with a CPU specific design, and the cooperative multithreading is optimized for the best-effort tasks. To mitigate the conflict management problem, PAVENET OS uses another characteristic of wireless sensor nodes: most conflicts occur between communication layers. PAVENET OS provides a wireless communication stack for hiding the exclusive controls to users. PAVENET OS sacrifices portability because PAVENET OS has a design specific to Microchip PIC18. Lack of portability is a significant problem.

3 COMPARATIVE ANALYSES

3.1) Due to the multi-threaded semantics, every Mantis program must have stack space allocated from the system heap, and locking mechanisms must be used to achieve mutual exclusion of shared variables. In contrast, Contiki uses an event based scheduler without preemption, thus avoiding allocation of multiple stacks and locking mechanisms. Preemptive multithreading is provided by a library that can be linked with programs that explicitly require it.

3.2) Contiki’s event kernel is significantly larger than that of TinyOS because of the different services provided. While the TinyOS event kernel only provides a FIFO event queue scheduler, the Contiki kernel supports both FIFO events and poll holders with priorities.

3.3) The experimental results show that MANTIS is more predictable than TinyOS. Specifically, the packet forwarding task execution time in MANTIS has a low variation and is independent of other activity such as the sensing task. Thus, MANTIS would be preferable in situations that need deterministic and timely processing. However, as the experiments show, the MANTIS system is not as power-
efficient as TinyOS. Thus, TinyOS would seem preferable if energy consumption is deemed to be of primary importance.

3.4) The dynamic loading mechanism of LiteOS follows the line of several previous efforts that did not involve virtual memory such as TinyOS (using XNP), SOS, and Contiki.

3.5) TinyOS’ latency is much smaller than the others because TinyOS’ task creation simply means assigning function pointer of a task to a ready queue. It does not need memory to be allocated or copied because TinyOS’ scheduler is FIFO (non-preemptive). However, MANTIS and Nano-Qplus[8] operating systems requires memory allocation of task control block.

3.6) Nano-RK[5] supports power management techniques and provides several power-aware APIs for system use. While low-footprint operating systems such as μC/OS support real-time scheduling, they do not have support for wireless networking.

3.7) The Nano-RK is the most closely related work to PAVENET OS. Nano-RK is a preemptive multitask operating system supporting real-time tasks. Additionally, Nano-RK is more portable than PAVENET OS. However, Nano-RK has more context switch overhead than PAVENET OS because Nano-RK has to preserve CPU context by software. Nano-RK needs several dozens of μs for task switching whereas PAVENET OS needs several ms.

3.8) Like PAVENET OS, MANTIS is a thread model operating system. The difference between MANTIS and PAVENET OS is the implementation of the thread model. MANTIS uses time-sliced multithreading, whereas the threading of PAVENET OS is not time-sliced.

3.9) TinyOS can port to PAVENET modules, but PAVENET OS cannot port to MICA2[3] because of its CPU specific architecture.

<table>
<thead>
<tr>
<th>Features</th>
<th>TinyOS</th>
<th>SOS</th>
<th>MANTIS</th>
<th>Nano-Qplus</th>
<th>Nano-RK</th>
<th>Improved μC/OS-II</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low power mode</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
<td></td>
</tr>
<tr>
<td>Multimodal sensing/tasking</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
<td></td>
</tr>
<tr>
<td>Dynamic reprogramming</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
<td></td>
</tr>
<tr>
<td>Priority-based scheduling</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
<td></td>
</tr>
<tr>
<td>Real-time guarantee</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
<td></td>
</tr>
<tr>
<td>Execution model</td>
<td>Component-based</td>
<td>Module-based</td>
<td>Thread-based</td>
<td>Thread-based</td>
<td>Thread-based</td>
<td>Thread-based</td>
</tr>
</tbody>
</table>

Table 1: Comparison table for wireless sensor network operating system

CONCLUSION AND FURTHER WORK
This paper has made a analysis of various operating systems that are in use for wireless sensor networks. It is found that techniques change according to use and demand, various threading, abstraction, portability, power consumption options are available but the future work is expected in the underground sensing and underwater sensor field. Underground Wireless Sensor Networks Research Challenges Dynamic Channel Power Constraints, Very Low Data Rates, Extremely Lossy Environment, New Communication Protocols needed. Research Challenges for Underwater Sensor Networks are Available bandwidth is severely limited, UW channel is severely impaired (in particular due to multi-path and fading), Very long and extremely variable propagation delays, Very high bit error rates and temporary losses of connectivity, Battery power is limited and usually batteries cannot be recharged, no solar energy, Very prone to failures because of fouling, corrosion, etc, New communication protocols needed.

REFERENCES

A Comparative Analysis of Wireless Sensor Network Operating Systems

[8] Seungmin Park+, Jin Won Kim*, Kwangyong Lee*, Kee-
Young Shin*, Daeyoung Kim∇ Proceedings of the Ninth IEEE
International Symposium on Object and Component-Oriented
Real-Time Distributed Computing.