# AN EVOLVABLE OPERATING SYSTEM FOR WIRELESS SENSOR NETWORKS

THU-THUY DO, DAEYOUNG KIM, TOMAS SANCHEZ LOPEZ,
HYUNHAK KIM, SEONGKI HONG, MINH-LONG PHAM

*Information and Communications University, 119 Munjiro, Yuseong-gu, Daejeon, 305-732, Korea*
*{dtthuy, kimd, tomas, overflow, white, longpm}@icu.ac.kr*

KWANGYONG LEE and SEONGMIN PARK

*ETRI, 161 Gajeong-dong, Yuseong-gu, Daejeon, 305-350, Korea*
*{kylee, minpark}@etri.re.kr*

With low-power consumption, small code and data size, evolvability as design criteria, we develop an evolvable operating system (EOS) for wireless sensor network applications. The EOS provides memory space efficient thread management, collaborative thread communication model and network stack. It also supports power management of microcontroller and radio transceiver, and network wide time synchronization function. Above all, the most important feature is the concept of evolvability with which the operating system itself can be easily configurable and upgradeable.

*Keywords*: Wireless Sensor Network; Operating System; On-the-fly upgradeability.

## 1. Introduction

Wireless Sensor Network (WSN), as a key enabling technology for ubiquitous computing, has a wide range of applications, which are deeply embedded in the physical world. Since sensor nodes that comprise sensor networks are highly resource constrained, there are many research challenges that must be overcame, such as a light-weight network stack, security mechanisms, localization, operating system, etc. In the recent years, there have been several research efforts on developing operating system for sensor networks. Among them, two notables are Berkeley's TinyOS and Colorado's Mantis Operating System (MOS) [1] [2]. MOS is a layered multithreaded operating system with a layered network stack. Tiny OS is the most popular state machine based operating system for sensor networks. In this paper, we propose our EOS, which shares the legacy multi-threaded architecture with MOS. Our evolvable operating system (EOS) is more space efficient by providing both micro-threads and generic threads. It also eases the writing of programs for most programmers who are familiar with multi-thread programming and C programming language. The EOS includes a flexible Hardware Abstraction Layer (HAL), which helps porting EOS to different hardware platforms; a light-weight and space efficient multithreaded architecture with the co-existence of micro-threads and generic-threads; a message handling engine, which is efficient and seamless for both local and remote communication; ease of use with standard programming language and a sufficient Application Programming Interface (API) for building diversified sensor network applications. Targeting another key direction of achieving evolvability in sensor networks operating system, it also provides self-configurable, fault-tolerant and reliable, transparent, and upgradeable features.

The rest of this paper is organized as follows. Section 2 describes the EOS architecture. In section 3, we briefly mention the concept of evolvability. Section 4 summarizes some implementation and performance issues. Section 5 presents the conclusions and future works.

## 2. EOS architecture

We present the architecture of EOS in Figure 1. It consists of six main components:

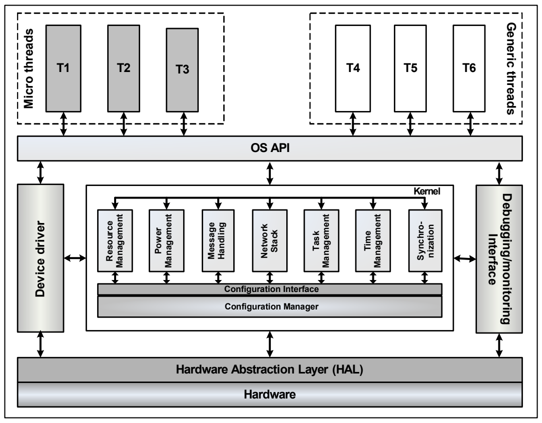

Figure1. Evolvable OS architecture

- *Hardware*: EOS runs on various sensor hardware platforms, mostly with strictly limited resources (e.g.: an 8-bit μC with 128KB Flash, 4 KB RAM).
- *Hardware Abstraction Layer*: It abstracts different types of hardware, eventually providing portability of EOS.
- *Kernel*: As the heart of our operating system, it consists of eight functional blocks: Configuration Manager, Resource Management, Power Management, Message Handling, Network Stack, Task Management, Time Management, and Synchronization.
- *Device driver:* It provides device drivers for all the devices in sensor nodes such as ADC, serial port, eeprom, etc.
- *Debug/monitoring interface:* It provides a command set with which users can debug and monitor the status of a sensor node.
- *OS API*: It provides EOS API to various EOS applications.

### 2.1. *Task management*

We propose our task management scheme which is named ***Hybrid Task management.*** All the tasks can be classified into two categories: the ***generic threads*** with their own separate stacks and the ***micro threads*** sharing a single stack within the group. The approaches of using a single stack in the system and the fixed priority scheduler with the Preemption Threshold mechanism have been used. To control the levels of preemption, we use the concept of Non-Preemption Group [3], which is a collection of tasks that are not permitted to preempt each other in the same group. Since this concept is applied to the micro-threads, each thread can have two kinds of priorities. **Base (Ready)** priority is the priority that is used to enqueue a thread in the ready queue. **Dispatch** priority is assigned to a thread when it becomes a running thread. This priority is the key to control the stack usage level in the system.

Since generic thread has its own stack, the thread's priority can be also considered as a preemption level. So the generic thread has only the Base priority. With the proposed task management mechanism, EOS can offer more flexibility in designing sensor network applications. Moreover, it guarantees high concurrency and efficient memory usage.

### 2.2. *Message Handling*

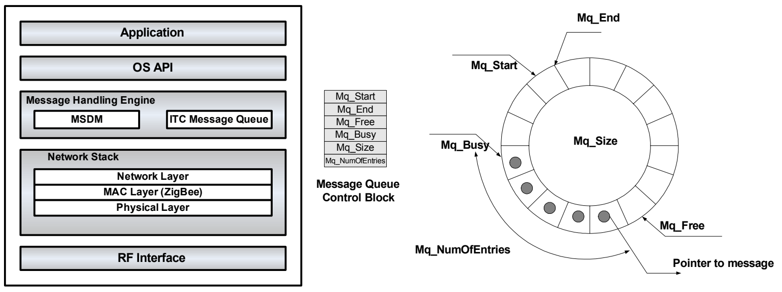

Figure 2. MHE- General Architecture and ITC Message Queue Structure

The general architecture of Message Handling Engine (MHE) with the details in message queue structure is shown in Figure 2. The Message Send/Dispatch Module (MSDM) receives requests from user threads. If the request is for inter-thread communication, the message will be forwarded to the ITC message queue; otherwise, the data is packed into a packet through the network stack and then, sent to the network destined to the target.

- ***Inter-Thread Communication (ITC)***

The basic idea of the Inter-Thread Communication in EOS is to use a message queue to store data exchanged among tasks. The message queue is implemented as a circular buffer. The way a message is inserted to the queue and then, retrieved, depends on the adopted policy. Parameters specifying a message queue are defined in the Message Queue Control Block. The structure of a Message queue is shown in Figure 2.

- ***Remote-Thread Communication (RTC)***

For RTC, we adopt three communication models: ***Base station Centered Communication (BCC)***: Base station (BS) can flood message(s) to all the sensor nodes or send message(s) to an arbitrary node in the network. Also sensor node can send message to BS. ***Publish / Subscribe Communication (PSC)****:* BS or an arbitrary node can be either a publisher or a subscriber of specific message. ***Collaboration / Group Communication (CGC)****:* A group of sensor nodes can collaborate using this communication model.

## **2.3.** *Network Stack*

The network stack is implemented as a pluggable module, which can be updated or changed easily. It supports three kinds of communication models as discussed. Currently, a tree-based topology on top of IEEE 802.15.4 PHY/MAC is used for routing. For BCC, the base station can broadcast the message to all nodes. To send the message to an arbitrary node in the network, the base station sends the message with the list of intermediate nodes on the way to the destination. From sensor node backward to the base station, each node just uses the parent node to reach the base station. For PSC, nodes send the advertisement and subscription message to the base station. The base station then forwards the published data to appropriate subscribers. For CGC, communication between nodes is performed through the base station. First the sender sends the message to the base station. The base station then forwards the message to the corresponding receivers.

## **2.4.** *Time Synchronization*

The time synchronization module has been implemented using "Sender-Receiver" method. The synchronization procedure is initiated by the base station. And we let the RF module timestamp just before passing the packet to the CC2420 RF chip for the sake of the more precise time stamping. After base station initiates a synchronization procedure, the children in the next level exchange two time-stamped packets with the base station. Then the child can calculate the own offset with exchanged packets, correct the own local clock. In the meantime the children of the next level can overhear its own parent's radio activity. Thus after detecting the parent's synchronization activity, the children can start to exchange time-stamped packets with the own parent. Whenever any node requests to synchronize with its own parent, it should wait a random amount of time to avoid the collision. Eventually the procedure of synchronization propagated from the base station up to the end of the synchronization tree.

## **2.5.** *Power Management*

To increase the lifetime of a sensor node, the power management module of EOS monitors the behaviors of threads and other parts of the kernel, and determines whether transition to an appropriate power mode is necessary or not. To be concrete, when all threads on a sensor node stop to run, power management module is activated to transfer the sensor node to sleep state. As soon as the power management module is invoked, it gathers the wake up sources which each thread wait to wake from sleep state such as timer expiration, or network packet arrival. Under performing such procedure, the power management module selects one sleep mode which meets conditions that include wake up sources at that time and consume lowest power among several provided sleep modes. In the end, the sensor node transfers into selected sleep mode. And it stays on that mode until one of expected interrupts is raised.

## 3. EOS advanced features

Evolvability is a distinguished concept of EOS. We are on our way to have a comprehensive definition of the so-called *evolvability* in the WSN in general and the operating system in particular. As our first step, evolvability offers our operating system some features such as scalability, high-modularity, and *on-the-fly* upgradeability.

*On-the-fly* upgradeability is the ability of the operating system to be upgraded without interfering user applications. The basic idea of *on-the-fly* upgradeability is to use one special module named ***Configuration Manager (CM).*** This module can manage and support the ability to load or unload other *repluggable modules* without requiring EOS to be rebooted. The structure of CM is presented in Figure 3 as follows.

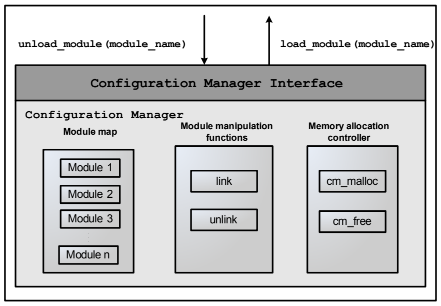
unload_module(module_name)
load_module(module_name)
Configuration Manager Interface
Configuration Manager
Module map
Module manipulation functions
Memory allocation controller
Module 1
Module 2
Module 3
Module n
link
unlink
cm_malloc
cm_free

Figure 3. Configuration Manager Structure

To load/unload *repluggable modules*, CM uses the *Dynamic Object Module Linking/Unlinking* technique. This is done by the *Module manipulation functions.* The information of one repluggable module (*symbols and string tables, .text segment, .data segment, .stack segment, etc*) is provided by the *Module map.* As the memory resource in a sensor node is very limited, *Memory allocation controller* ensures safe module load/unload operations and avoids confliction among different segments in the memory.

To upgrade a new module, the follow steps occur: CM allocates memory for different segments of the new object module. The new object module is, then, read into memory. CM has to ensure that the module to be upgraded is not in use. It initializes data for the new module and links it to the current kernel data. The old module is, then, unlinked and the memory space for the old module is freed. With the support of this *on-the-fly upgrading* mechanism, EOS can be evolved transparently to users.

## 4. EOS implementation

EOS is implemented on our *ANTS-H2* hardware platform with 8 bit ATMEGA128 CPU, 4KB RAM, 128 KB flash memory on chip, 4 Mb extended flash memory, and ZigBee CC2420 transceiver for the CPU board, various sensors such as light, accelerometer, magnetometer, gas, IR, etc. for the sensor board as shown in Figure 4.

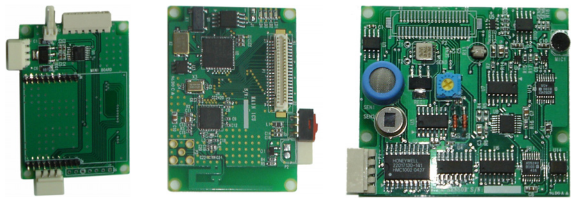
Figure 4. ANTS H2 Hardware. From left to right, programming board, CPU board and sensor board

Among the various experiments we have done after development of the EOS, we describe one of them; the ability of *on-the-fly upgrading* mechanism. The scenario to upgrade *task management* module is as follows: The system first operates with a simple FIFO task scheduling algorithm. It is required then an upgrade to the ***Hybrid Task management*** scheme. The most important point in our *on-the-fly upgrading* mechanism is to decide when the new object module is linked to kernel data and the old object module is unlinked. For the task management module, it is done when the task queue is empty.

## 5. Conclusions

In this paper, we proposed an evolvable operating system for wireless sensor networks. Each component in our architecture is designed to perform its functionality concerned with power consumption, highly limited resources, and evolvability. For the future work, we will consider other topics such as security, real-time constraints, etc.

## References

1. J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister, “System architecture directions for network sensors” (ASPLOS 2000) pp. 93 – 104.
2. H. Abrach, S. Bhatti, J. Carlson, H. Dai, J. Rose, A. Sheth, B. Shucker, J. Deng, R. Han, "MANTIS: System Support For MultimodAl NeTworks of In-situ Sensors" (ACM International Workshop on Wireless Sensor Networks and Applications , 2003), pp. 50-59.
3. R. Davis, N. Merriam, and N. Tracey, “How embedded applications using an RTOS can stay within on-chip memory limits”, (Proc. of the Work in Progress and Industrial Experience Sessions, 12th Euromicro Workshop on Real-Time Systems, Stockholm, 2000) pp. 43–50.
4. R.S Fabry, “How to design systems in which modules can be changed on the fly” ($2^{nd}$ International Conference Software Engineering, 1976), pp. 470-476.