

Demo Abstract: A Virtualizing OS Kernel for Wireless Sensor Networks

Lin Gu
Department of Computer Science
University of Virginia
lingu@cs.virginia.edu

John A. Stankovic
Department of Computer Science
University of Virginia
stankovic@cs.virginia.edu

Categories and Subject Descriptors

D.4.7 [Operating systems]: Organization and Design;
D.4.8 [Operating Systems]: Performance

General Terms

Design, Reliability, Performance

Keywords

Wireless Sensor networks, OS Protection, Virtual Memory

1 Introduction

The experience with several recent real-world wireless sensor networks (WSNs) shows that it is very difficult to develop a distributed, adaptive, and reliable network system with a large number of low-power sensor nodes. Reasons include stringent resource constraints, a tight energy budget, and demanding application requirements. Sensor nodes typically employ low-power microcontrollers and very small amounts of data memory (3KB–10KB RAM). But systems being built with large numbers of these devices can be quite sophisticated. There is a gap between the relatively low hardware abstraction and relatively high software complexity.

Three OS features—OS protection (boundary between the OS and application), virtual memory, and preemptive scheduling—can significantly improve the reliability of WSN systems and facilitate developing sophisticated WSN software. However, the lack of hardware support for privileged execution and address translation makes it impossible to implement these features with traditional OS design techniques.

We design a new OS kernel, the *t-kernel*, to close the gap between hardware simplicity and software complexity. It virtualizes the simple sensor hardware to be a platform with a much higher capability. Without relying on hardware or compilers, the kernel performs extensive code modification when it loads and dispatches the application for execution.

The modified code and the OS collaborate to support OS protection, virtual memory, and preemptive scheduling. Hence, the code modification, called *naturalization* in the *t-kernel*, enhances the system abstraction visible to programmers.

In this demonstration, we run several applications to exhibit the features of the *t-kernel*. Specifically, we use a routing application to show that the *t-kernel*'s ability to handle a high interrupt rate and perform radio communication. A “wild” program shows how the *t-kernel* resists application errors. Finally, a program with a large memory footprint shows how *t-kernel* enables applications to use 64KB virtual memory space.

2 About the kernel

Our regular paper describes the design and implementation of the *t-kernel* in detail [3]. In this demonstration description, we give an overview of the kernel design and highlight some parts related to this demonstration, namely, interrupt handling and radio, OS protection, and virtual memory.

Figure 2 shows the hardware and software architecture of a MICA2 node running the *t-kernel*. The *t-kernel* resides in the “kernel space” in the program memory. The *application* is written in native instructions of the sensor hardware's instruction set. It is usually generated by a compiler (e.g., nesC compiler [2] and avr-gcc), or by an assembler if the application is written in assembly language, or a combination of the two. When an application is installed in a network (e.g., through wireless programming), it resides in the external flash on sensor nodes.

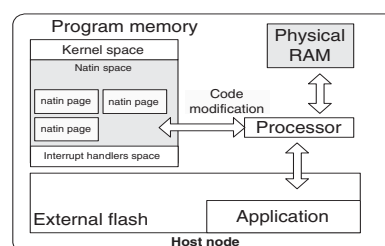


Figure 1. The sensor node and *t-kernel*

When the sensor node boots up, the *t-kernel* initializes its own working environment, then loads, modifies, and runs the application. The modification and execution are performed page by page. When the control flow reaches a new page of application code, the *t-kernel* reads that page and modifies

some of its instructions. We call instructions in the modified code *natins*. A fixed number of natins (256 bytes on MICA2) form a natin page, and the natin pages constitute a *naturalized program*. The naturalized program executes in a collaborative way with the OS kernel, and supports the enhanced system abstraction.

In the *t-kernel*, interrupts are also handled in a collaborative manner—the kernel detects an interrupt, and invokes the application’s event handler for this interrupt. The event handler for an interrupt can be at a low level, directly manipulating the hardware, or at a higher level, leaving part of the logic to be handled by the kernel. This design gives the application developers flexibility for handling interrupts. The application’s interrupt (event) handlers are modified and executed the same way as usual application code.

The naturalization process ensures OS protection, so that application errors cannot compromise the kernel or the sensor node. Two key issues in OS protection are CPU control by the OS kernel, and the OS data integrity. The naturalized program yields CPU control to the kernel frequently, and does not write the kernel data area. Hence, it guarantees not to compromise the sensor node, and is trusted and executed without constraints.

The *t-kernel* provides differentiated virtual memory (DVM), which is a software based virtual memory system that differentiates memory accesses to different areas in order to make the common case fast. On MICA2 motes, DVM supports 64KB virtual data memory over 4KB RAM.

3 Demonstration scenarios

We have implemented the *t-kernel* on MICA2 family motes, including MICA2, XSMv1, and XSMv3 (ExScal) motes [1], as shown in Figure 2. Using these sensor nodes, this demonstration presents three scenarios to exhibit the *t-kernel*’s capability in interrupt handling, OS protection, and virtual memory.

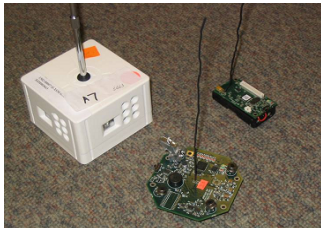


Figure 2. The hardware platforms for the current *t-kernel* implementation

3.1 Routing application

Taking a pure software approach to support OS features that traditionally require hardware support, the *t-kernel* necessarily increases the CPU utilization and slows down CPU-bound computation. However, most of the WSN applications are not CPU bound. Moreover, performance is an emphasized goal in the *t-kernel* design, and the kernel overhead is kept low for common cases. Particularly, the *t-kernel* is able to handle interrupts at a very high rate to perform radio communication.

We demonstrate a routing application which has a strict timing requirement – the application must process wireless

packets quickly enough to achieve a high packet reception ratio. The *t-kernel* provides a byte-level interface to communicate with the application. When a radio interrupt (an SPI interrupt on MICA2) occurs, the kernel only reads and writes the radio transceiver’s data register, and triggers the byte-level event handler (SPI interrupt handler on MICA2) in the application for the new byte read from the radio transceiver. The application conducts the major work involved in this interrupt, including byte processing, transceiver controlling, framing, packet parsing, and traffic regulating, in its event (interrupt) handler. We demonstrate the performance of the radio communication by showing the packet reception ratio.

3.2 Resistance to application errors

The *t-kernel* is resistant to application errors. We use a “wild” program to demonstrate how the *t-kernel* handles application errors and guarantees control over the sensor node despite these errors. The “wild” program contains a number of serious programming errors, such as left recursion, wrong pointers, and an infinite loop with interrupt disabled. Without the *t-kernel*, these errors would make a sensor node crash, or stop responding to system commands. With the *t-kernel*, the kernel still controls the sensor node, and can accept and execute system commands (e.g., to perform wireless reprogramming) to fix the application errors. We show the system behavior through a display interface supported by the *t-kernel*.

3.3 An application with large memory

We will demonstrate DVM with an application with a large memory footprint. It stores and reads large chunks of data, and shows the execution status of the program through the display interface supported by the *t-kernel*. This application showcases how the *t-kernel* efficiently handles virtual memory accesses and performs swapping to and from the external flash device.

4 Availability

By providing OS protection, virtual memory, and preemptive scheduling, the *t-kernel* significantly enhances developers’ ability to design sophisticated applications on a virtualized, reliable, and easy-to-use platform. We intend to release the *t-kernel* to the WSN research community. Both the kernel and the test applications will be made available. As an OS kernel, the *t-kernel* can work independently on a sensor node, and run applications designed for the *t-kernel*. The kernel can also work together with TinyOS, and provide an enhanced abstraction (e.g., more memory) to TinyOS applications. We are currently improving the kernel for TinyOS compatibility to make it easy to use, and a beta version release of the *t-kernel* is expected in near future.

5 References

- [1] P. Dutta, M. Grimmer, A. Arora, S. Bibyk, and D. Culler. Design of a wireless sensor network platform for detecting rare, random, and ephemeral events. In *Proc. of the 4th Intl. Conf. on Information Processing in Sensor Networks (IPSN’05)*, pages 497–502, Los Angeles, CA, 2005.
- [2] D. Gay, P. Levis, R. Behren, M. Welsh, E. Brewer, and D. Culler. The nesC language: A holistic approach to networked embedded systems. In *Proc. of Programming Language Design and Implementation*, pages 1–11, San Diego, CA, June 2003.
- [3] L. Gu and J. A. Stankovic. *t-kernel*: Providing reliable os support to wireless sensor networks. In *Proc. of the 4th ACM Intl. Conf. on Embedded Networked Sensor Systems*, Boulder, CO, Nov. 2006.