

LiteOS based Reliable Software Stack and Visible System Architecture for Wireless Sensor Networks

Qing Cao, Ph.D. Candidate, University of Illinois at Urbana-Champaign
Advisor: Professor Tarek Abdelzaher

Abstract

This research summary proposes my research on the LiteOS platform, a UNIX-like, multithreaded operating system for wireless sensor networks. My research focuses on two themes: system failure tolerance to provide reliability, and system visibility through interactive commanding. This research summary outlines my ongoing research efforts in these two directions, as well as a concise description of the LiteOS operating system.

1 Research Motivation

My Ph.D. research proposal focuses on research in two directions, building reliable software for wireless sensor networks, and achieving system visibility through interactive operations. To facilitate these research directions, a UNIX-like operating system, called LiteOS, is implemented as the underlying platform.

Wireless sensor networks are expected to be deployed for prolonged periods of time in an unattended manner. Because of the limited system resource on sensor nodes, debugging and testing wireless sensor network software is particularly challenging. Furthermore, even the most strict debugging still does not guarantee that all bugs are found before deployment. In fact, unexpected changes in the environment where sensor nodes are deployed can also introduce inconsistencies with the assumptions made by the system, which may cause system faults. A systematic approach to detect and recover from system faults is therefore needed, which motivates the first research direction in my PhD thesis, namely, improving software robustness and reliability in wireless sensor networks.

The second challenge we address is system visibility. For the past several years, system visibility, i.e., providing insight on the internal system operations, has been a task of debugging tools and applications. In cases where diagnosis information is not provided by applications or the debugging software, however, the sensor network appears like a black box. In the second research direction, we probe this black box at the operating system level by building interactive services to allow various system information, such as current running threads, to be accessible to the user through Unix-like commands.

In these two directions, more specifically, we propose the following research topics. In the first direction, we propose an architecture to systematically detect a wide range of application faults through memory specification rules. For example, suppose that a node encounters a sensor problem, it can no longer detect any event despite that all its neighbors can. Suppose that the number of event detections is represented by a variable on each node, a memory rule should compare the values of this variable on nodes within one-hop neigh-

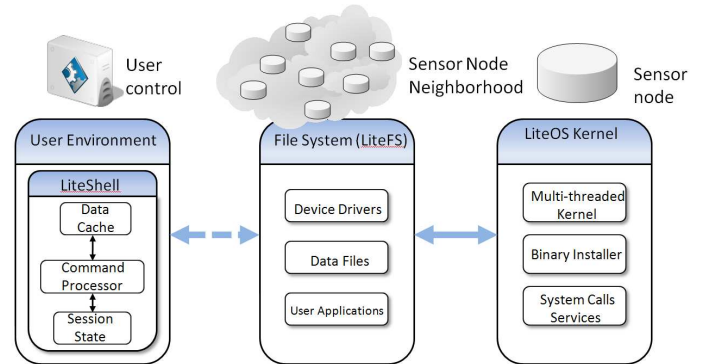


Figure 1. LiteOS Operating System Architecture

borhood, and a difference that is significant enough implies a sensor fault has been detected, assuming that the event to be detected is strong enough that every node in one-hop neighborhood should have similar detection results and that the sensors on nodes have similar sensitivity.

In the second direction, we propose interactive services that are built directly into LiteOS to support user level collection of system information. Such information could range from thread status to energy consumption profiling of different modules. In LiteOS, a kernel serves as a supervisor of all user thread activities. It is therefore intuitive to revise this kernel thread to provide such interactive services.

To facilitate the above research work, we implemented a new operating system platform, called LiteOS, and the work above will serve as extensions. When implementing the LiteOS platform, we consider it beneficial to create a familiar environment for users where they can interactively command the entire sensor network to perform tasks such as reprogramming, data retrieval, or network reconfiguration. To this end, LiteOS implements a UNIX-like environment, which could potentially expand the circle of sensor network application developers by reducing learning curves. Further, LiteOS leverages the knowledge that users may already have, i.e., Unix and threads, an approach not unlike the network directions taken by companies such as Arch Rock (that superimpose a familiar IP space on mote platforms to reduce the learning curve of network programming and management).

The rest of this proposal is organized as follows. In Section 2, we describe the LiteOS platform infrastructure. We then outline in Section 3 the two aforementioned research directions. Finally, in Section 4, we conclude this summary.

2 LiteOS Platform

This section presents the LiteOS platform. It is organized as follows. First, we describe an architectural overview of LiteOS. Second, we present a brief introduction to its subsystems.

Table 1. Shell Commands

Command List	
File Commands	ls, cd, cp, mv, rm, mkdir, touch, chmod, pwd, du
Process Commands	ps, kill, exec
Group Commands	foreach, \$,
Environment Commands	history, who, man, echo
Security Commands	login, logout, passwd

2.1 Architectural Overview

Figure 1 shows the overall architecture of the LiteOS operating system, partitioned into three subsystems: LiteShell, LiteFS, and the kernel. Implemented on a base station, the LiteShell subsystem interacts with sensor nodes only when a user is present. Therefore, LiteShell and LiteFS are connected with a dashed line in this figure.

LiteOS provides a wireless *node mounting mechanism* (to use a UNIX term) through a file system called LiteFS. Much like connecting a USB drive, a LiteOS node mounts itself wirelessly to the root filesystem 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), the wireless mount works only for devices within wireless range. The mount mechanism comes handy, for example, in the lab, when a developer might want to interact temporarily with a set of nodes on a table-top before deployment. While not part of the current version, it is not conceptually difficult to extend this mechanism to a “remote mount service” to allow a network mount. Ideally, a network mount would allow mounting a device as long as a network path existed either via the Internet or via multi-hop wireless communication through the sensor network.

Once mounted, a LiteOS node looks like a *file directory* from the base station. The shell, called LiteShell, supports UNIX commands, such as copy and move, executed on such directories. The external presentation of LiteShell is versatile. While the current version resembles closely a UNIX terminal in appearance, it can be wrapped in a graphical user interface (GUI), appearing as a “sensor network drive” under Windows or Linux.

2.2 LiteOS Subsystems

LiteShell Subsystem: The LiteShell subsystem implements a Unix-style shell for MicaZ-class sensor nodes. Currently, 23 commands, as listed in Table 1, are implemented. We briefly introduce file operation commands as an example.

File Operation Commands: File commands generally maintain their Unix meanings, e.g., the **ls** command lists directory contents. Typing **man ls** in the shell returns the manual information of the **ls** command. It supports the **-l** option to display detailed file information, such as type, size, and protection. To reduce system overhead, LiteOS does not provide any time synchronization service, which is not needed by every application. Hence, there is no time information listed. A **ls -l** command returns the following:

```
$ ls -l
Name      Type      Size      Protection
usrfile  file      100      rwxrwxrwx
usrdir   dir       ---      rwxrwx---
```

In this example, there are two files in the current directory (a directory is also a file): **usrfile** and **usrdir**. LiteOS enforces a simple multilevel access control scheme. All users are classified into three levels, from 0 to 2, and 2 is the highest level.

For instance, the **usrdir** directory can be read or written by users with levels 2 and 3. The **chmod** command can be used to change file permissions.

Once sensor nodes are mounted, a user uses the above commands to navigate the different directories (nodes) as if they were local. The base station PC also has directories, such as drives **C** and **D**. Some common tasks can be greatly simplified. For example, by using the **cp** command, a user can either copy a file from the base to a node to achieve wireless download, or from a node to the base to retrieve data results. The remaining file operation commands are intuitive. Since LiteFS supports a hierarchical file system, it provides **mkdir**, **rm** and **cd** commands.

LiteFS Subsystem: Similar to the Unix-like shell, the interfaces of the file subsystem, LiteFS, resemble Unix closely, providing support for both file and directory operations.

Kernel Subsystem and System Calls: The LiteOS kernel supports threads, and implements two different scheduling policies: priority-based scheduling and round-robin scheduling. The kernel also supports dynamic loading of user threads. It maintains a map of system resource allocation, including both its program flash and RAM. To dispatch a thread, it copies thread information into a free control block. When a thread terminates, it frees allocated resources for this thread, by marking its occupied resource as available. It also forcefully closes previously opened file pointers by this thread, if there are any.

We also introduce lightweight system calls to address software compatibility between different versions. Because the MicaZ CPU does not support soft interrupts or traps, our implementation is based on revised *callgates*, a special type of function pointers. These callgates are the *only* access points through which user applications access system resources. Therefore, they implement a strict separation between the kernel and user applications. As long as the system calls remain supported by future versions of LiteOS, user binaries do not need to be recompiled.

Currently, each system call gate takes 4 bytes, with 1024 bytes of program space allocated for at most 256 system calls. Each system call adds 5 instructions (10 CPU cycles), a low overhead to be supported on MicaZ.

3 Research Directions

This section outlines my research work based on the LiteOS platform, organized into three topics: detection of application failures using memory specification rules, file system assisted communication stacks for fault isolation, and interactive commanding service to improve system visibility.

Cooperative Diagnosis of Application Failures using Memory Specification Rules

The first research direction focuses on detection of application failures. Its key idea is derived from real life. We all have an immune system that protects us against diseases. Further, our human society has created very complicated medical systems, including doctors and medicine, to diagnose and treat diseases. Not every person is a doctor, of course. Therefore, the medical system is inherently cooperative: people not only get help from themselves through medical knowledge and their immune system, but also from

more specialized facilities such as hospitals, to keep healthy. It is beneficial if we could create a similar system for wireless sensor networks to increase its expected system lifetime.

Our proposed approach, which relies on memory specification rules to detect application bugs (illness), works in a two-tiered way. The first tier works at the node scale, where the user creates memory rules to detect unhealthy (buggy) state. Such rules are analogous to human medical knowledge. The second tier, on the other hand, allows nodes to cooperate with each other to detect more delicate bugs that would not otherwise be detected. For example, if a node finds that in the past ten seconds it has not detected any event, but all its neighbors have reported multiple detections, then either this node is located in a void area or it has a bad sensor. Such a scenario may need to be logged as a warning. Another example is in group-management protocols, such as EnviroTrack, at most one leader node is allowed to be elected in one-hop neighborhoods. If more than one node sets its leader flag as true in one-hop neighborhood, an application fault is detected. Both examples can be expressed using memory rules that are checked against at runtime.

Normally memory rules are stored in LiteFS. The kernel reads memory rules when needed to detect failures. Once a failure is found, the kernel may use one of the following approaches as treatment. First, it could give the thread “medicine”, by forcefully modifying certain variables back to the normal state. Second, the user may want to collect early warnings of a failing system to find bugs. To do this, the kernel continuously snapshots thread information until it fails. Third, A user may have anticipated this bug and provided alternative modules, such as the second communication stack. The kernel then loads the new stack into the memory as a backup.

File System Assisted Communication Stacks for Fault Isolation

It has been challenging to design energy-efficient and flexible communication stacks for wireless sensor networks, due to both hardware limitations and energy constraints. In this research effort, we propose to implement a file system assisted communication stack for wireless sensor networks. Instead of hard-wiring the communication stack into application logic as a layer, the new approach allows different stacks to be dynamically chosen and loaded at run time in an adaptive manner. More specifically, an entire communication stack is implemented as a file. The application specifies which file to use, which is in turn loaded at run-time, making it particularly flexible to respond to environment changes.

This approach has the following advantages. First, because communication stacks can be dynamically loaded, they achieve natural fault isolation, because bugs in a communication protocol can be safely removed by replacing one communication stack with a backup without changes to the application. Second, this approach provides an avenue where different communication stacks can be directly compared in terms of their performance and overhead, which was previously much harder if communication stacks were implemented as part of the application.

Interactive Commanding Service for System Visibility

In this research topic, we explore how to make a system

information more visible at the operating system level. Currently, the LiteOS platform already allows the user to perform tasks when a node is located within one-hop neighborhood of the base station. In this research direction, we aim to provide a more powerful commanding service that achieves the following goals.

First, we intend to implement this commanding service over multiple hops. With this service, a user can task the entire sensor network without being physically within one hop radius of each node. Second, we aim to optimize the commanding service when tasking a group of nodes. One optimization goal is to minimize the communication energy cost. Under such a scenario, the problem of reliably delivering commanding packets to multiple nodes becomes a multicast problem, whose solution requires careful tradeoffs between energy consumption, delay, and throughput. Third, we explore how to improve system level visibility through this interactive commanding service. While certain information, such as the current variable values on several nodes, can be easily retrieved, other information, such as the underlying mechanism of a protocol behavior, requires multiple nodes to log certain critical state information at runtime. Tasking such logging behavior could be far more complicated, and requires careful runtime energy cost optimizations to balance its cost and performance.

4 Conclusions

Above all, this research summary outlines the two research directions we intend to pursue based on the LiteOS platform. These research directions are challenging for the following reasons. First, we need to provide extensive evaluation of the research directions as well as the LiteOS platform. Comparison with existing similar platforms, such as TinyOS, Mantis, and SOS, is also important. Because Mantis and SOS both use C as the main programming language, comparing with them rather than TinyOS might be more appropriate. Second, we need to evaluate the energy consumption of different services carefully, and explore conservation approaches. Because LiteOS uses threads as the basic building block, it may consume more energy in context switches. Profiling such energy usage will be particularly important to develop energy conservation protocols and prolong system lifetime.

Acknowledgements

I gratefully acknowledge my advisor, Professor Tarek Abdelzaher, and my shepherd, Professor Philip Levis, on their insightful comments during revision of this manuscript.

Brief Biography Qing Cao is a graduate student at the computer science department of University of Illinois at Urbana-Champaign. He got his Masters degree from University of Virginia in 2004. His advisor is Tarek Abdelzaher. His research interest is wireless sensor networks and embedded systems. He is currently working on his Ph.D. thesis, as well as focusing on the development of the LiteOS project. He is the author and co-author of more than fifteen publications in peer-reviewed conferences and journals. His expected date of dissertation submission is August 2008 or later.