Demo Abstract: An Interactive UNIX Shell for Low-End Sensor Nodes with LiteOS

Qing Cao, Tarek Abdelzaher
University of Illinois, Urbana-Champaign
{qcao2,zaher}@cs.uiuc.edu

John Stankovic
University of Virginia
stankovic@cs.virginia.edu

Tian He
University of Minnesota
tianhe@cs.umn.edu

Abstract

This demonstration highlights an interactive Unix-like shell for operating wireless sensor networks, where the user uses familiar Unix commands to complete tasks ranging from wireless installation of user applications to retrieval of data reports with the help of a built-in Unix-like file system.

Categories and Subject Descriptors

D.4.7 [**Operating Systems**]: Organization and Design—*Distributed systems, Interactive systems, Real-time systems and embedded systems*

General Terms

Design, Experimentation

Keywords

Unix shell, Sensor networks, LiteOS

1 Overview and Motivation

While TinyOS has become a *de facto* standard in the sensor networks community, the concepts of event-based programming and NesC wiring abstractions are not as commonplace in embedded systems and networking communities. It may be argued that Linux and its many embedded variants enjoy a larger following among embedded systems and other software developers. A premise of the authors is therefore that software development for sensor networks can benefit from the introduction of a UNIX-like operating system interface for common platforms such as MicaZ motes. This demo highlights such an interface.

Motivated by the need to simplify sensor network development, the authors demonstrated in Sensys 2006 [1] a lightweight operating system kernel, called LiteOS, and a programming environment, called LiteC, that allow thread-based programming in C++ on low-end motes. This year's demonstration presents the next stage in the evolution of LiteOS; namely, its newly developed UNIX-like interface.

Sensor network software developers who run LiteOS can now make use of a subset of UNIX commands from within LiteOS programs on individual sensor nodes. A UNIX-like local file system is exported to these programs. Local sensors and actuators appear as UNIX I/O devices that are handled as special files per UNIX conventions. The multi-threaded nature of LiteOS greatly simplifies the implementation of the aforementioned interactive Unix-like environment by creating a separate thread for handling different UNIX commands. Finally, the system supports an implicit mount-like operation whereby a mote within radio range from an authorized base-station (that runs a LiteOS shell) is mounted to the base-station's file system.

SenSys'07, November 6–9, 2007, Sydney, Australia.
ACM 1-59593-763-6/07/0011

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 Introduction of an Interactive Unix Shell

The interactive Unix shell is designed to support single-node functionality. As mentioned above, it also provides a wireless *node mounting mechanism* (to use a UNIX term) based on the built-in file system of LiteOS. Much like connecting a USB drive, a node mounts itself wirelessly to the 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.

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 node looks like a *file directory* from the perspective of the interactive shell. We call this Unix-like shell LiteShell, and implement 23 commands, as listed in Table 1. They fall into five categories: file commands, process commands, group commands, environment commands, and security commands. Due to space limitations, we only briefly describe the first three categories of commands in this section.

File Operation Commands: File commands generally maintain their Unix meanings, e.g., the **ls** command lists directory contents. It provides a **-l** option to display detailed file information, such as type, size, and protection. A **ls -l** command returns the following in a screenshot:

```
$ 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**. The shell enforces a simple multilevel access control scheme. All users are classified into three levels, from 0 to 2, and 2 is the highest level. Each level is represented by three bits, stored on sensor nodes. For instance, the **usrdir** directory can be read or written by users with levels 1 and 2. The **chmod** command can be used to change file permissions.

Once sensor nodes are mounted, a user navigates different directories (nodes) as if they are 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 [1]. The remaining file operation commands are intuitive. Since LiteOS supports a hierarchical file system, it provides **mkdir**, **rm** and **cd** commands.

Process Operation Commands: LiteShell also allows users to control processes. When an application is first loaded, it is executed as a thread that can spawn new threads. LiteShell provides three commands to control thread behavior: **exec**, **ps**, and **kill**. We illustrate these commands through an application called **Blink**, which blinks LEDs periodically. Suppose that this application has been compiled into a binary file called Blink.lhex[2], and is located under the C drive of the base station. To install it on a node named **node101** (that maps to a directory with the same name) in a sensor network named **sn01** (the names of the network and nodes are specified by the user), the user types the following commands:

```
$ cp /c/Blink.lhex /sn01/node101/apps/Blink.lhex
    Copy complete
$ cd /sn01/node101
$ exec ./apps/Blink.lhex
    File Blink.lhex started
$ ps
    Name   PID
    Blink  1
```

Group Commands: It is usually convenient to support certain group operations. For example, prior to deployment, all nodes are usually kept together for programming and testing. Operations in this phase are usually group-based by nature, such as "install an application on all nodes". We have implemented two such group mechanisms, the **foreach** command and the **$** variable. When used together with regular expressions and pipes, these two commands loop over a set of nodes (directories) to perform group operations. We illustrate their usage through two examples.

Suppose that we have ten nodes labeled **201**, **202**, up to **210**. We have installed and started an application **Tracking.lhex** on each of them (located in the directory **bin**, and having a PID of 1), and have copied another application **Report.lhex** to each node (in the directory **apps**). To retask each node with the **Report.lhex** application, the user types the following command:

```
$ echo [201-210] | foreach $ \
  {kill 1; exec $/apps/Report.lhex}
```

In this example, the **echo** command generates a list of directories using a regular expression. Instead of being displayed on the screen, these directories are provided to the variable **$** through a pipe (spaces between strings are used as separators), which is in turn combined with the **foreach** command. As its name suggests, **foreach** loops over these directories, terminates the **Tracking** application, and invokes the **Report** application.

In the next example, the **$** variable is concatenated with other strings to generate customized commands. The scenario is as follows. Suppose that the **report** application writes data results into a local file called **data.txt** on each node (in its root directory). To retrieve such data, the user types the **copy** command as follows:

```
$ echo [201-210] | foreach $ \
  cp $/data.txt /c/sensordata/$.txt
```

Here, the command **cp $/data.txt /c/sensordata/$.txt** customizes the destination file for each node. Because normal file names only allow letters, digits, underscores(_), and dots(.), LiteShell recognizes that **$.txt** needs to be translated before execution. Hence, no confusion will be introduced.

3 Demonstration Scenario

To illustrate the interactive Unix shell, we are going to deploy a grid of 20 MicaZ nodes. A laptop computer will serve as the base station, where the user can interact with the whole network through the interactive shell. All the aforementioned commands will be demonstrated. The demonstration will include the following:

1. Demonstration of basic file, process, shell, group, and security commands.
2. Control of sensing devices, such as light sensors, demonstrated through the shell by reading real-time data from such sensors.
3. Control of radio using shell commands to change frequency, power setting, and send/receive raw radio messages.
4. Upload, install, and terminate individual user applications (which may themselves be multi-threaded).
5. LiteOS hands-on demo where audience can try their own command sequences.

Several applications, ranging from data collection to spanning tree formation, will also be presented as demonstration examples.

4 References

[1] Q. Cao and T. Abdelzaher. Demo abstract: Liteos - a lightweight operating system for c++ software development in sensor networks. In *The 4th ACM Conference on Embedded Networked Sensor Systems*, 2006.

[1] The implementation of copy requires that this file is not concurrently opened by applications for writing. Otherwise, the copy operation will return a failure.

[2] LiteOS uses a revised version of the Intel hex format, called lhex, to store binary applications. lhex stands for LitOS Hex.