Demo Abstract: Faithful Reconstruction of Application Behavior based on Event Traces in the LiteOS Operating System

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

Abstract

Visibility has been a key challenge for wireless sensor network applications. Deployed on the extremely resource-constrained mote platform, such applications may fail unexpectedly, or exhibit behavior different from their intended goals. To help understand why such problems occur, we design and implement an event trace logger based on the LiteOS operating system, which allows us to partially reconstruct application behavior after execution, such as which path it took for an IF statement, its invocation history of the kernel system calls, and its dynamics across multiple nodes. We plan to demonstrate the usefulness of this event trace logger and its translator using a multi-hop routing application.

1. Introduction

In the past few years, research in wireless sensor networks has experienced phenomenal growth. Various applications have been developed and deployed, data have been collected and analyzed, and large scale adoption of sensor networks is on the horizon. As applications for wireless sensor networks become more complicated, we find it harder and harder to understand their behavior. Applications fail unexpectedly, or perform differently from their goals. While debugging tools tailored for sensor networks have been proposed in the literature as one promising research direction, we propose to reconstruct application behavior based on carefully collected event traces, an approach that presents us with a detailed map on the dynamics of distributed sensor network applications. We plan to demonstrate this approach through realistic applications.

Our implementation of the event logger is based on the LiteOS operating system, which separates the kernel from user applications. This feature allows us to log event traces at different levels, including the kernel, the system calls API, and the user applications. Based on the analysis of the traces, we find that while it is hard to completely reconstruct the kernel behavior (e.g. certain events, such as how many CPU cycles elapsed before the next interrupt, are almost impossible to log), it is relatively easy to reconstruct the application behavior, such as which path its instructions follow for an IF statement, or how many times it calls the kernel API. Such information is valuable for us to profile the application performance, detect unexpected behavior causes, and faithfully reconstruct its life cycle. We believe such understanding is critical not only for application developers to build robust systems, but also for improving the visibility of sensor network applications.

2. System Design

In this section, we first briefly describe the background of the LiteOS operating system. We next describe how we design the event logger. Finally, we describe how to interpret its output to reconstruct application behavior.

The LiteOS operating system provides a Unix-like interactive environment for wireless sensor networks. It maps nodes into file directories, and allows users to operate such directories using Unix-like commands. For example, to retrieve data collected by a node, LiteOS provides the cp command to copy data from the particular node (now a directory) to the PC (another directory). To support interactive operations, LiteOS implements a kernel that schedules user applications as threads. Therefore, users may install and kill applications dynamically.

LiteOS implements a natural boundary between the kernel and user applications, which is a suite of system calls. Therefore, the application binary consists of instructions that are either application specific or references to system calls, which simplifies recording and analyzing application behavior, an observation that motivates us to design and implement the event logger and its translator to help reconstruct application life cycle after its execution.

We follow an intuitive approach in designing the event logger. We keep an internal buffer (its size is

978-0-7695-3157-1/08 $25.00 © 2008 IEEE
DOI 10.1109/IPSN.2008.56

```
int main()
{
int index;
 for (index = 0; index < 10; index++)
{
  radioSend_string("Hello, world!\n");
  greenToggle();
  sleepThread(100);
}
return 0;
}
```

Figure 1 The Source Code of "Hello, World"

```
17
69 68 7C 68 6C 6A 68 65 93 6B 66 68 65
69 68 7C 68 6C 6A 68 65 93 6B 66 68 65
69 68 7C 68 6C 6A 68 65 93 6B 66 68 65
69 68 7C 68 6C 6A 68 65 93 6B 66 68 65
69 68 7C 68 6C 6A 68 65 93 6B 66 68 65
69 68 7C 68 6C 6A 68 65 93 6B 66 68 65
69 68 7C 68 6C 6A 68 65 93 6B 66 68 65
69 68 7C 68 6C 6A 68 65 93 6B 66 68 65
69 68 7C 68 6C 6A 68 65 93 6B 66 68 65
69 68 7C 68 6C 6A 68 65 93 6B 66 68 65
18
```

Figure 2 Event Traces for "Hello, World"

//user thread starts
Create thread system call

(The following loop ten times)

//Radio_send function
Get radio mutex system call
Get current thread address system call
Get current radio data structure system call

//Mutex_lock function called in Radio_send
Get current thread address system call
Get current thread index system call

//Radio_send function
Radio send operation system call

//SleepThread function called in Radio_send
Get current thread system call
Yield system call

//Radio_send function
Reset radio state system call

//Mutex_unlock function
Mutex unlock system call

//Green_toggle function
Green toggle system call

//SleepThread function
Get current thread address system call
Yield system call

(The above loop ten times)
Destroy thread system call

Figure 3 Translation of Event Traces

decided by the user) to record the most recent application events. We only log one application at a time. Therefore, we only need one byte for most events, up to 256 different types, including nearly 60 different system calls. We also log certain kernel events such as context switches and driver invocations, and application specific events inserted by the user. Every time an event triggers, a corresponding byte is written into the buffer. When the buffer is full, we write its content into a file stored in the external flash, whose size is 512K bytes, more than enough for most applications.

To provide a more detailed view on the application behavior, the event logger also allows recording variable values, using special event types. Such events necessarily occupy more than one byte, but are useful for understanding the transient execution context of the user application, such as, why is this control path taken?

We now describe a simple application and its event traces as an example. The source code of the application is listed in Figure 1.

The behavior for this application is very simple: it sends ten messages containing the string "Hello, world!\n". We then turned on the event logger, and obtained a trace file as follows, shown in Figure 2.

Here, we only log system calls and kernel event of creating/destroying events. A translation of this file is shown Figure 3.

These system call sequences reveal a lot of information regarding how the application executes. For example, the user can easily verify that there are ten loops for the application. As applications become more complicated, the user can insert checkpoints along different paths to get an understanding which paths are taken during application execution.

3. Demo Scenario

We shall demonstrate a more complicated application using the event logger. In the demo, we are going to deploy a multi-hop geographic forwarding application composed of six to eight hops. The first node will send periodic messages to the last node. Because of interferences and unreliable links, we expect some packets to get lost. We record certain radio activities, including sending and receiving data packets, packet sequence numbers, among others. At the end of the experiment, based on analysis of event traces, we are able to reveal which packets got lost and where. We are also able to reconstruct other information regarding the routing protocol, such as the number of context switches and neighbor beacons, the real-time link quality history as measured by LQI and RSSI. Such information will help us reconstruct the application behavior and provide insight on possible communication protocol optimizations.