State Machine Based Operating System Architecture for Wireless Sensor Networks

Tae-Hyung Kim[1] and Seongsoo Hong[2]

[1] Department of Computer Science and Engineering, Hanyang University, Ansan, Kyunggi-Do, 426-791, South Korea
tkim@cse.hanyang.ac.kr
[2] School of Electrical Engineering and Computer Science, Seoul National University, Seoul 151-741, South Korea,
sshong@redwood.snu.ac.kr

Abstract. A wireless sensor network is characterized as a massively distributed and deeply embedded system. Such a system requires concurrent and asynchronous event handling as a distributed system and resource-consciousness as an embedded system. State machine based software design techniques are capable of satisfying exactly these requirements. In this paper, we present how to design a compact and efficient operating system for wireless sensor nodes based on a finite state machine. We describe how this operating system can operate in an extremely resource constrained sensor node while providing the required concurrency, reactivity, and reconfigurability. We also show some important benefits implied by this architecture.

1 Introduction

Sensor networks consist of a set of sensor nodes, each equipped with one or more sensing units, a wireless communicating unit, and a local processing unit with small memory footprint [1]. In recent advancement of wireless communication and embedded system technologies, the wireless and distributed sensor networks become a prime technical enabler that can provide a way of noble linkage between the computational and the physical worlds. Since the precise delivery of real-time data on the spot is an essential basis for constructing a context-aware computing platform, the recent advancement of low-cost sensor node provides an important opportunity towards the new realm of ubiquitous computing. Positioned at the very end-terminal from the computational world side, wireless sensor nodes convey unique technical challenges and constraints that are unavoidable to system developers, which can be characterized by three aspects. First, they bear extremely limited resources including computing power, memory, and supplied electric power. Nonetheless, a sensor network can be perceived as a traditional distributed computing platform consisting of tens of thousands of autonomously cooperating nodes. Third, the computing platform does not allow recycling of the network, thus is disposable without having reprogrammability.

Such characteristics of a networked sensor node call for a unique operating system architecture that can not only run on an extremely lightweight device with very low power consumption but can also support dynamic reconfigurability to cope with changing environments and applications. Such an operating system should also possess concurrent and asynchronous event handling capabilities and support distributed and data-centric programming models. In order to meet such seemingly contradictory requirements, we propose a state machine based operating system architecture, rather than following a traditional structure of an operating system and adopting it for sensor nodes like TinyOS [2]. To provide re-programmability, TinyOS employs the bytecode interpreter called Maté that runs on it. In a state machine based operating system like ours, each node is allowed to simply reload a new state machine table. Moreover, the state machine based software modeling offers a number of benefits: (1) it enables designers to easily capture a design model and automatically synthesize runtime code through widely available code generation tools; (2) it allows for controlled concurrency and reactivity that are needed to handle input events; and (3) it enables a runtime system to efficiently stop and resume a program since the states are clearly defined in a state machine. In this paper, we explore a state machine based execution model as an ideal operating system design for a networked sensor node and present the end result named SenOS.

2 State Machine Based Execution Environment

While many embedded applications should exhibit a reactive behavior, dealing with such reactivity is considered to be the most problematic. To cope with the complexity of designing such systems, Harel introduced a visual formalism referred to as statecharts [3]. Since then, a state machine has been recognized as a powerful modeling tool for reactive and control-driven embedded applications. Sensor network applications are one of those applications that can mechanize a sequence of actions, and handle discrete inputs and outputs differently according to its operating modes. Being in a state implies that a system reacts only to a predefined set of legal inputs, produces a subset of all possible outputs after performing a given function, and changes its state immediately in a mechanical way. Formally, a finite state machine is described by a finite set of inputs, outputs, states, a state transition function, an output function, and an initial state. When a finite state machine is implemented, a valid input (or event) triggers a state transition and output generation, which moves the machine from the current state to another state. A state transition takes place instantaneously and an output function associated with the state transition is invoked.

A state machine based program environment is not only suitable for modeling sensor network applications but also can be implemented in an efficient and concise way. Since sensor node functionalities are limited, although multi-functional, all those possible node functionalities are defined statically in a callback function library in advance. All we need to do as a programmer is simply to define a legal sequence of actions in tabular forms. To this end, SenOS has four system-level components: (1) an event queue that stores inputs in a FIFO order, (2) a state sequencer that accepts an input from the event queue, (3) a callback function library that defines output func-

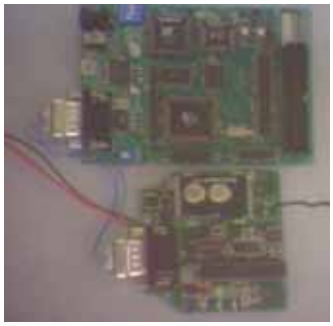
tions, and (4) a re-loadable state transition table that defines each valid state transition and its associated callback function. Each callback function should satisfy the "run-to-completion" semantics to maintain the instantaneous state transition semantics.

SenOS exposes another important opportunity for developers. There exist quite a few CASE tools that help designers capture state machine based system models and automatically synthesize executable code for them. UML-RT is one such tool widely used in the embedded systems industry [4]. Under our state machine based operating system, application programmers can take advantage of high-level CASE tools like UML-RT to synthesize executable code for a sensor node.

3 Implementing SenOS Architecture

The SenOS kernel architecture is comprised of three components: the Kernel consisting of a state sequencer and an event queue, a state transition table, and a callback library. The Kernel continuously checks the event queue for event arrivals; if there are one or more inputs in the queue, it takes the first one out of the queue and triggers a state transition if the input is valid. It then invokes an output function associated with the state transition. To do so, the Kernel keeps track of the state of the machine and guards the execution of a callback function with a mutex that can guarantee the run-to-completion semantics. The callback library provides a set of built-in functions for application programmers, thus determining the capability of a sensor node. The Kernel and callback library should be statically built and stored in the flash ROM of a sensor node whereas the state transition table can be reloaded or modified at runtime. The SenOS can host multiple applications by means of multiple co-existing state transition tables and provide concurrency among applications by switching state transition tables. Note that each state transition table defines an application. During preemption, the Kernel saves the present state of the current application, restores the state of the next application, and changes the current state transition table. The SenOS architecture also contains a runtime monitor that serves as a dynamic application loader. Considering the sheer number of sensor nodes, this is essential to dynamically reconfigure a new sensor network management scheme like dynamic power management. When the SenOS receives an application reload message via an interrupt from a communication adapter, the Monitor puts the Kernel into a safe state, stops the Kernel, and reloads a new state transition table. Note that the Monitor is allowed to interrupt the Kernel at any time unless it is in state transition. Since state transition is guarded by a mutex, the safety of a state machine is not compromised by such an interruption.

We have implemented our SenOS on 8-bit MCU AT89S8252 equipped with Radiometrix's BIM433 RF module that has a reliable 30m in-building range. A sensor node has four independent memory banks, each of which has 32KB flash memory as shown in Fig. 1. In our experimental implementation, we hire three sensor nodes and one sink node (PC). The SenOS is downloaded onto the sensor node that is directly connected to the host PC via a serial communication initially, and then all other nodes obtain the same OS via wireless RF communication. The SenOS was written in about 700 C lines of code, and compiled using Keil 8051 v7.0 compiler, which is compact enough to reside in a 32KB memory bank. We used four FSM tables (FSM_Serial,



8051 Board Specifications

- ATMEL89S8252
- 8 KB Flash ROM, 2 KB EEPROM
- 32KB SRAM
- Serial Port Interface

RF Module (BIM433) Specifications

- 433.92 MHz (ETS 300-220)
- FM Transmission at -6dBm ERP
- -107dBm receive sensitivity
- Half duplex data at up to 40 kbit/s
- Reliable 30m in-building range
- Single 4.5 to 5.5 V < 15mA

Fig. 2. Sensor node module used for our implementation and its specifications

FSM_Network, FSM_Timer, FSM_Sensor) and defined nine output functions for wireless and serial communications, sensor and timer operations, and network management in our implementation. We confirmed the compactness and efficiency of a state machine based operating system by this implementation.

4 Conclusion

We have presented SenOS, a state machine based operating system for a wireless sensor node. Programmers can easily write a SenOS application via techniques on state machines and load the executable code at runtime using the Monitor as an agent. SenOS offers a number of benefits. First, its implementation is very compact and efficient since it is based on a state machine model. Second, it supports dynamic node reconfigurability in a very effective manner using replaceable state transition tables and callback libraries. Third, it can be extended to implement a sensor network management protocol, which is one of the largely untouched regions of sensor network research. Without having reconfigurability, the sensor network boils down to a hard-wired system because it is hard to reprogram that many nodes manually. These benefits render SenOS ideal for a networked sensor node. The associated tools with SenOS are underway and we further explore the applicability of reconfiguration.

References

1. Hill, J., Szewczyk, R., Woo, A., Hollar, S., Culler, D., and Pister, K.: System Architecture Directions for Networked Sensors. Proceedings of International Conference on Architecture Support for Programming Languages and Operating Systems (2000)
2. Levis, P. and Culler, D.: Maté: A tiny virtual machine for sensor networks. Proceedings of International Conference on Architecture Support for Programming Languages and Operating Systems (2002)
3. Harel, D.: Statecharts: A Visual Formalism for Complex Systems, The Science of Computer Programming, pp. 231-274 (1987)
4. IBM (former Rational Software): http://www.rational.com