A Virtual Machine-Based Programming Environment for Rapid Sensor Application Development

Jui-Nan Lin and Jiun-Long Huang
Department of Computer Science
National Chiao Tung University
Hsinchu, Taiwan, ROC
E-mail: {jnlin, jlhuang}@cs.nctu.edu.tw

Abstract

In recent years, TinyOS and nesC are gradually becoming the de facto software development platform for implementing sensor applications. However, developing sensor applications is difficult for programmers since the programming paradigm used in nesC is different from that used in other popular programming languages. In view of this, we propose in this paper a virtual machine-based programming environment for rapid sensor application development. With the proposed programming environment, sensor applications are implemented in TinyJava, which is a subset of Java programming language. Developing sensor applications is very similar as developing a traditional Java applications. Therefore, we believe that the proposed programming environment is able to speedup the development of sensor applications.

Keywords: *Programming environment, sensor network, embedded system*

1 Introduction

The growing advance in wireless communications and electronics makes the development of low-cost and low-power sensors possible. These sensors are usually small in size and are able to communicate with other sensors in short distances wirelessly. A sensor network [1] consists of a number of sensors which cooperates with one another to accomplish some tasks. Sensors can be deployed either in a random or in a predetermined manner. Since being self-organized, sensors are able to form a sensor network automatically. Due to the characteristics of wireless communication and configuration-free deployment, sensor networks are suitable for various application areas including inventory management, product quality monitoring and disaster area monitoring.

In recent years, TinyOS [4] and nesC[1] are gradually becoming the de facto software development platform for implementing applications in sensor networks. However, due to employing a programming paradigm (i.e., component-based architecture) different from other popular programming languages such as C++ and Java, nesC is not apprentice-friendly. When leading several students to implement a temperature monitoring application on sensor networks, we observed that some students were reluctant to learn nesC programming. Therefore, designing a programmer-friendly programming environment is important for the success of sensor networks.

In view of this, the community has proposed several programming environments including TinyDT[2], TinyOSIDE[3] and TOSDev[4] to ease the process of programming in nesC. TinyDT and TinyosIDE are plugins of Eclipse. With them, Eclipse can recognize nesC's syntax and perform some advanced source editing functionality such as reserve word highlighting and code folding. TOSDev is a stand-alone integrated development environment (IDE) for nesC. In addition to performing advanced source editing functionality, TOSDev enables programmers to view the whole nesC program in a visual manner. However, programmers still need to learn the syntax and programming paradigm of nesC, and the difficulty of learning sensor application development still remains. In addition, the debuggers of nesC are still under development so that debugging a nesC program is still a difficult job.

In this paper, we propose a new programming environment to address these problems. The proposed programming environment consists of the following three compo-

[1] TinyOS and nesC will be introduced later.
[2] TinyDT, `http://www.tinydt.net`
[3] TinyosIDE, `http://www.tinyoside.ucd.ie`
[4] TOSDev, `http://selab.csuohio.edu/dsnrg/tosdev/`

nents:

- TinyJVM: a lightweight Java virtual machine on top of TinyOS,
- TinyJava: a programming language resulting from reducing Java programming language, and
- Sensorlet: a class library supporting sensor application development.

With the proposed programming environment, a sensor application is implemented in TinyJava. A standard Java compiler is used to compile the TinyJava program into the corresponding Java bytecodes stored in `.class` files. Then, the `.class` files and TinyJVM are installed into sensors running TinyOS. Finally, TinyJVM executes the sensor application by interpreting its `.class` files.

The proposed programming environment has the following advantages:

1. Eliminate the difficulties in learning new programming languages.

   The programming language supported by TinyJVM is TinyJava, which is a programming language resulting from discarding some constructs from Java. Hence, programmers familiar with Java can implement sensor applications easily.

2. Make programming sensor applications more convenient by leveraging existing IDEs.

   Since the syntax of TinyJava is a subset of Java, existing Java IDEs such as Eclipse and Netbeans can support TinyJava without modification. Advanced source editing functionality is hence available for TinyJava programmers.

3. Simplify the debugging process.

   Up to now, there is no mature debugger for nesC. With the proposed programming environment, debugging a TinyJava program is similar to debugging an ordinary Java program.

4. Facilitate cheap dynamic code update

   Since TinyOS is a monolithic operating system, dynamic code update is hence costly. With TinyJVM, since performing code update is just transmitting bytecodes, which are very small in size, to sensors, dynamic code update is cheap.

The rest of this paper is organized as follows. An introduction to the execution environments for sensor networks is given in Section 2. In addition, nesC programming will also be introduced. Then, the design of the proposed programming environment will be described in Section 3. An illustrative example of TinyJava is given in Section 4. Finally, Section 5 gives our future work.

2 Preliminaries

The related work of execution environments is described in Section 2.1 while an overview of nesC programming is given in Section 2.2.

2.1 Related Work

Essentially, a sensor node can be seen as a small embedded system with small memory, limited storage, an limited computation power. Several operating systems are designed on top of sensor nodes to provide programmers with friendly execution environments. Han et. al. [2] classified the execution environments for sensor nodes into three categories: monolithic, modular and virtual machine-based.

With a monolithic operating system such as TinyOS, the user program and the operating system are linked together into an image. The image is then loaded to a sensor node to perform its task. Since the user program and the monolithic operating system is tightly coupled, global optimizations can be performed in compilation time. Hence, applications using monolithic operating systems are usually of high performance and consume less memory.

With a modular operating system such as SOS[5] [3], the application is divided into two parts. One is the kernel and the other is loadable components. The kernel provides several APIs to perform I/O, communication, memory management, etc. Since components are loaded dynamically, the interactions between components are usually performed in an indirect manner. Therefore, the performance of applications using modular operating systems are usually less efficient than that using monolithic operations systems. However, modular operating systems enable users to perform some dynamic behavior such as dynamic code update.

Consider the scenario to update application code. With monolithic operating systems, the whole image including operating systems should be transmitted to sensor nodes even the operating systems are not revised. With modular operating systems, only the images of the revised components should be transmitted to sensor nodes. Hence, the cost of code update with modular operating systems is much cheaper than that with monolithic operating systems.

Matè[6] [6] is a virtual machine on TinyOS. The instruction set of Matè is designed for compactness. With Matè, an

[5]SOS, http://nesl.ee.ucla.edu/projects/SOS/
[6]Matè, http://www.cs.berkeley.edu/~pal/mate-web/

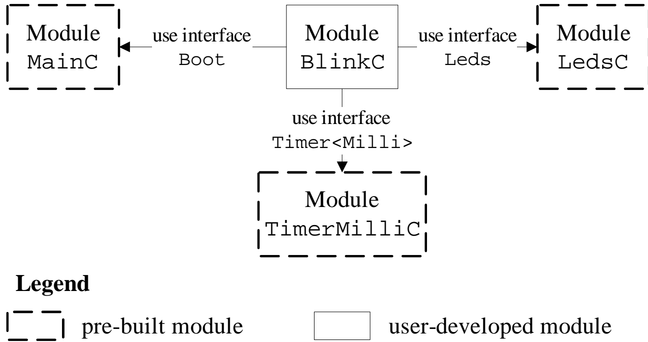

Figure 1. The structure of program BlinkApp

application is implemented in TinyScript, which is designed to hide the event-driven behavior of TinyOS. The application is then compiled into Matè instructions and deployed to sensor nodes. When updating software, only instructions of user programs are needed to be transmitted. Hence, the cost of code update is also cheap. Matè is extended in [7] to support application specific virtual machines to further reduce the size of user programs and the interpretation overhead. However, due to the overhead of virtual machines, virtual machine-based approaches inherently consume higher memory than monolithic and modular ones.

2.2 Overview of nesC Programming

nesC is an extension of C to facilitate event-driven, component-based programming paradigm. Specifically, a nesC program consists of several components and the interactions among these components. A component is called a *module* in nesC. A module may export several interfaces for other modules. Modules will interact with one another via their interfaces. The interactions between modules are called *wirings* in nesC. The wirings of the modules used in a nesC program are specified a *configuration* file. In addition to using the modules pre-built in TinyOS, programmers can develop their own modules.

We use a simple application, called BlinkApp, to describe the structure of a nesC program. This program will ask sensor nodes to blink one of its LEDs with period 2000 ms. The structure of BlinkApp is as shown in Figure 1 while the source code is as follows.

```
configuration BlinkAppC {
}
implementation {
  components MainC, BlinkC, LedsC;
  components new TimerMilliC() as Timer0;
  BlinkC -> MainC.Boot;
  BlinkC.Timer0 -> Timer0;
  BlinkC.Leds -> LedsC;
}
```

Configuration BlinkAppC.nc

```
module BlinkC {
  uses interface Leds;
  uses interface Boot;
  uses interface Timer<TMilli> as Timer0;
}
implementation {
  event void Boot.booted() {
    call Timer0.startPeriodic(1000);
  }
  event void Timer0.fired() {
    call Leds.led2Toggle();
  }
}
```

Module BlinkC.nc

As shown in Figure 1, as the core of the whole application, module BlinkC is wired to three pre-built modules, TimerMillC, MainC and LedsC to perform the task. The used components and the interactions among them are specified in file BlinkApp.nc. In these four components, BlinkC is a user-defined module specified in file BlinkC.nc and the other three components are pre-built in TinyOS.

When booting, event *booted* is generated and the sensor node executes the event handler `Boot.booted` to set up the timer `Timer0` to be fired once per 1000 ms. When `Timer0` is fired, event handler `Timer0.fired` will be executed to toggle the third LED from on to off, or from off to on. As a result, the LED will blink will periodicity 2000 ms. Interested readers can refer to the official site of nesC[7] for the details of the nesC programming language.

3 The Proposed Programming Environment

Figure 2 shows the architecture of the proposed programming environment and the corresponding development process of sensor applications. The proposed tool-kit of the proposed programming environment will be depicted in Section 3.1 while the proposed development process will be described in Section 3.2.

3.1 The Proposed Tool-Kit for Sensor Application Development

3.1.1 TinyJVM: A Lightweight Virtual Machine for TinyOS

To interpret Java bytecodes, we implement a lightweight Java virtual machine called TinyJVM on top of TinyOS.

[7] nesC, `http://nescc.sourceforge.net/`

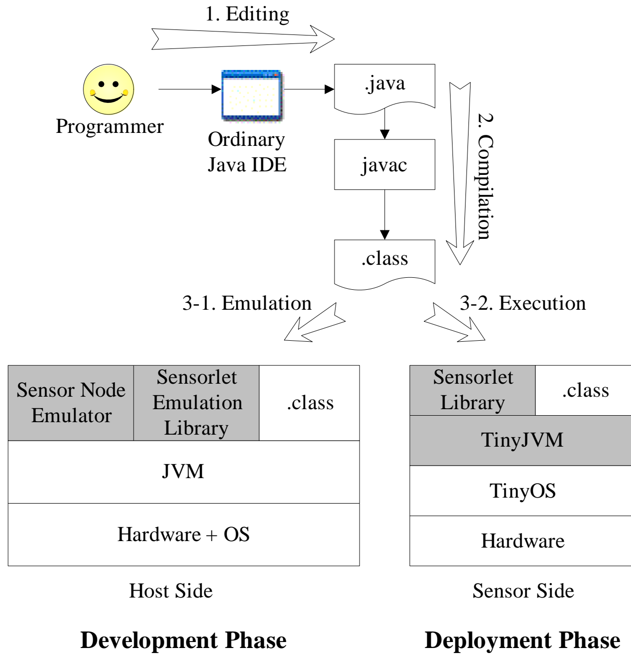

Figure 2. The proposed programming environment and software development process and system architecture

Since sensor nodes are usually of small memory and limited computation power, some advanced functions such as just-in-time compilation, garbage collection, exception handling and multi-threading are not implemented in TinyJVM. In addition, to reduce the footprint of TinyJVM, some infrequently-used bytecodes are not implemented, either. The design rationale of TinyJVM is to implement *just enough* functionality to execute TinyJava programs.

3.1.2 The TinyJava Programming Language

TinyJava is the standard programming language in the proposed programming environment. When designing TinyJava, we keep the following rationale in mind.

1. TinyJava should be a subset of Java.

   To avoid the burden of learning new programming language, TinyJava is designed to be a subset of Java. That is, TinyJava is designed by removing some unnecessary language constructs from Java.

2. The implementation of the language constructs of TinyJava cannot consume too much memory and computation power, and should adhere to the constraints imposed by TinyJVM.

Since TinyJava is a subset of Java, programmer can use ordinary Java IDEs to debug their sensor applications easily. The differences between TinyJava and Java are as follows.

1. TinyJava only supports integer data types: 32-bit integer (`int`), 16-bit integer (`short`) and 8-bit integer (`byte`). Since most processors of sensor nodes do not support floating point numbers, floating point data types such as `float` and `double` are not supported right now.

2. Exception handling related language constructs such as `try`, `throw`, `catch` and `finally` are not included in TinyJava.

3. Multi-threading related language constructs such as `synchronized` are not included in TinyJava.

4. Since TinyJVM does not implement garbage collection, programmers should free memory by themselves explicitly.

3.1.3 Sensorlet: A Package for Sensor Application Development

In the proposed programming environment, package `Sensorlet` is implemented for a TinyJava program to control the underlying platform (hardware and operating system) by encapsulating the modules in TinyOS. In `Sensorlet`, the wirings among modules are implemented as registering event handlers by *listeners*. TinyOS consists of many pre-built modules. To reduce memory consumption, only some frequently-used modules are supported in `Sensorlet`. Since most sensor applications are implemented to monitor some phenomena, `Sensorlet` currently supports the following four functions: timer, led controller, sensing module controller and network transmitter. The interfaces of the classes in `Sensorlet` are listed below. Note that the term *sensor* in `Sensorlet` is used to indicate sensing modules.

```
static class Sensorlet {
  static LED[3];
  static Timer[3];
  static Sensor[3];
}
final class LED {
  static void toggle() {...}
  static void turnOn() {...}
  static void turnOff() {...}
}
final class Timer {
  void setup(int interval, bool isPeriodic)
```

```
  {...}
  void addTimerListener(TimerListener l)
  {...}
  void start() {...}
  void stop() {...}
}
interface TimerListener {
  void fired(void);
}
final class Sensor {
  int read() {...}
}
final class Memory {
  static void delete(class Object) {...}
}
final class Network {
  ...
}
```

The proposed programming environment consists of two sides: the host-side and the sensor-side. TinyJVM is implemented in sensor-side and `Sensorlet` is integrated into TinyJVM for better performance. In addition, to make development of sensor applications more convenient, we also implement a sensor node emulator and an emulation library of `Sensorlet` in Java. The usage of host-side and sensor-side components is described in Section 3.2.

3.2 The Proposed Sensor Application Development Process

The corresponding development process consists of two phases: development phase and deployment phase. In development phase, programmers develop (including debug) their applications using the host-side part of the proposed programming environment. In deployment phase, the application is executed in sensor nodes with the sensor-side part.

3.2.1 Development Phase

In development phase, programmers use ordinary Java IDEs to develop sensor applications in TinyJava. Since TinyJava is a subset of Java, ordinary Java IDEs are able to perform advanced source editing functionality for TinyJava. However, programmers can only use language constructs supported by TinyJava and `Sensorlet` package. After coding, programmers can invoke a java compiler such as `javac` to compile their code into Java bytecodes. With an emulation library of `Sensorlet` and an emulator of sensor node, the sensor application can be executed on top of Java virtual machines. The `Sensorlet` emulation library and the sensor node emulator can work together to emulate the execution environment on sensors. The readings of sensing modules in sensor node emulator can be obtained from existing files (such as reading logs) or from random number generators. In addition, sensor node emulator also emulates the network transmission among sensor nodes by UDP. Thus, in development phase, programmers can implement and debug sensor applications in resource-rich platforms such PC. These designs make programming sensor applications very similar to programming in simplified Java with more constraints.

3.2.2 Deployment Phase

After the sensor application has been implemented and tested in development phase, the development process steps into deployment phase. In deployment phase, the bytecodes of the sensor application, TinyJVM and TinyOS are compiled into one image and the image is then written to the sensor node. When TinyOS is booted, it invokes the initialization function of TinyJVM and then asks TinyJVM to interpret the bytecodes of the sensor application.

4 An Illustrative Example

In this section, we take BlinkApp mentioned above as an example application. The TinyJava version of BlinkApp is below.

```
import Sensorlet.*;
public class BlinkApp extends Sensorlet
  implements TimerListener {
  public static void main(String[] argv) {
     new BlinkApp();
  }
  BlinkApp() {
    Sensorlet.Timer[0].setup(1000, true);
    Sensorlet.Timer[0].addTimerListener(this);
    Sensorlet.Timer[0].start();
  }
  public void fired() {
    Sensorlet.LED[2].toggle();
  }
}
```

The constructor of BlinkApp sets up the timer to be fired with period 1000 ms by invoking `setup` and sets up the instance of BlinkApp as the listener of the timer by function `addTimerListener`. Hence, BlinkApp should implement interface `TimerListener`. When BlinkApp is executed in sensor node emulator, the `Sensorlet` emulation library will configure the sensor node emulator to fire the

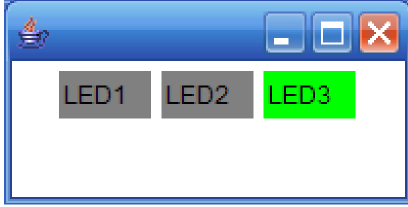

Figure 3. A screenshot of sensor node emulator

Version	ROM	RAM
TinyJava	5558 bytes	992 bytes
nesC	2680 bytes	41 bytes

Table 1. Memory consumption

timer with period 1000 ms. Each time the timer is fired, function `fired` is called to toggle the third LED of the sensor node. Thus, the LED icon of the sensor node emulator will blink with period 2000 ms. A screenshot of the sensor node emulator is shown in Figure 3.

We also build the nesC version and the TinyJava version of BlinkApp to sensor nodes and their footprints in ROM and RAM are listed in Table 1. Due to the overhead of the virtual machine, the memory consumption of the TinyJava version is higher than the nesC version. This is an inherent cost of virtual machine-based programming environments and is a necessary cost to facilitate code update in TinyOS.

5 Future Work

The implementation of the proposed programming environment is still in progress. Currently, we have finished the implementation of the LED and Timer modules in TinyJVM, sensor node emulator, `Sensorlet` and the emulation library of `Sensorlet`. Other modules are still under development. Our future work is as follows.

- Implement other modules such as sensing module and network transmitter.
- Optimize TinyJVM.

  We will try to optimize TinyJVM to reduce its footprint in ROM and RAM. We will also consider to employ some techniques to speedup the performance of interpretation.
- Compact `.class` files.

  According to [5] and [8], method bytecodes occupy only 20% of an average `.class` file, and the constant pool is a good target for size reduction. We will try to remove some unused information in `.class` files to reduce their sizes.

References

[1] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. A Survey on Sensor Networks. *IEEE Communications Magazine*, August 2002.

[2] C.-C. Han, R. Rengaswamy, R. Shea, and M. Srivastava. Sensor Network Software Update Management: A Survey. *Internation Journal of Network Management*, July 2005.

[3] C.-C. Han, R. K. Rengaswamy, R. Shea, E. Kohler, and M. Srivastava. SOS: A Dynamic Operating System for Sensor Networks. In *Proceedings of the 3rd ACM International Conference on Mobile Systems, Applications, And Services*, June 2005.

[4] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System Architecture Directions for Network Sensors. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, November 2000.

[5] J. Koshy and R. Pandey. VM*: Synthesizing Scalable Runtime Environments for Sensor Networks. In *Proceedings of the 3rd ACM Conference on Embedded Networked Sensor Systems*, Novemver 2005.

[6] P. Levis and D. Culler. Matè: A Tiny Virtual Machine for Sensor Networks. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2002.

[7] P. Levis, D. Gay, and D. Culler. Active Sensor Networks. In *Proceedings of the 2nd USENIX/ACM Symposium on Networked Systems Design and Implementation*, May 2005.

[8] W. Pugh. Compresing Java Class Files. In *Proceedings of the ACM International Conference on Programming Language Design and Implementation*, May 1999.