Programming the Sensor Nodes in WSN

Anita Patil, Rajashree.V.Biradar

***Abstract---** The present technological era is replacing both physically and logically draining hard-works of the human beings by computerized technologies like Wireless Sensor Network (WSN) and IOT. WSN, being the basis for IoT, share the same set of Operating systems (OSs) with IOT. The numerous sensor nodes that are deployed in the application areas such as wild life study, under water study etc could not be attended by the human beings, so they need well-defined programming. Learning the essential programming approach is the default first step to pass through for every researcher in any research domain. This paper discusses programming concepts for WSN considering four different OSs. The first part of the paper demonstrates execution of one nesC application in detail, as the nesC programming language is the de-facto standard for TinyOS. In the second part of the paper, programming is discussed in brief for the OSs Contiki, RIOT and freeRTOS. TinyOS being a highly documented and popular OS, has the limitations of having only the FIFO scheduling mechanism. This study helps to incorporate the scheduling techniques from other OSs in to TinyOS. This paper can be viewed as an introductory manual for the beginners in WSN programming.*

***Keywords:** Programming in WSN, Contiki OS, nesC programming language, RIOT, FreeRTOS*

I. INTRODUCTION

There are the numerous applications where utmost human intervention is required, but being physically present over there is not possible, for example in wild life study human being presence over the application scenario is a dangerous situation. WSNs can manage and automate such processes to greater extent possible. The basic entities of WSN called sensor nodes, having stringent architecture, concurrently do many tasks like data gathering, processing, communicating with sink node (base station) etc. To make this possible with restricted resources, programming must be done very consciously. Programming in WSN is at two levels. One is at node level, where node is programmed with the support of suitable programming languages and tools, and this is called node-centric programming approach. In the second approach of programming, part of the network or complete network is considered as a single entity and programmed based on application. So, this approach is called as application-centric programming approach [1, 7]. Programming the WSN is not as simple as programming our "traditional" distributed computing systems, because of highly constrained architecture. Basically, the sensor node itself is tiny sized with very limited memory, peripheral devices, battery, processing capability etc. The three basic subsystems of node i.e. sensor subsystem, communication subsystem and processing subsystem need to synchronize with each other and work in parallel [1]. Because of limited power supply, the network topology tends to continuously change which is an ad-hoc network of many sensor nodes. To tackle all such limitations WSN must be well programmed. The author in [7] describes various programming models at different levels as shown in the below figure. But the scope of this paper is restricted to only TinyOS and nesC in detail and programming in Contiki, RIOT and freeRTOS in brief.

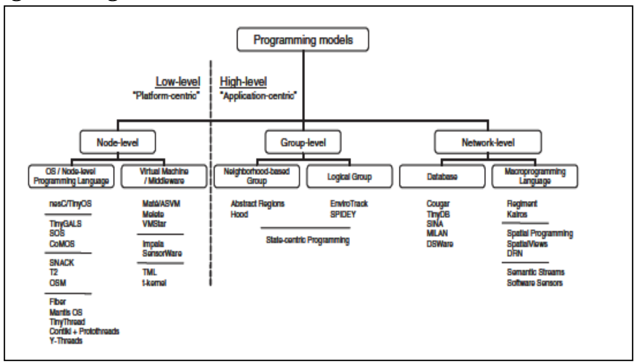

Programming models
Low-level "Platform-centric"
High-level "Application-centric"
Node-level
Group-level
Network-level
OS / Node-level Programming Language
Virtual Machine / Middleware
Neighborhood-based Group
Logical Group
Database
Macroprogramming Language
nesC/TinyOS
TinyGALS SOS CoMOS
SNACK T2 OSM
Fiber Mantis OS TinyThread Contiki + Protothreads Y-Threads
Maté/ASVM Melete VMStar
Impala SensorWare
TML t-kernel
Abstract Regions Hood
EnviroTrack SPIDEY
State-centric Programming
Cougar TinyDB SINA MILAN DSWare
Regiment Kairos
Spatial Programming SpatialViews DRN
Semantic Streams Software Sensors

Figure 1: Taxonomy of programming models for sensor networks

Thus, WSN's numerous nodes and sink node need appropriate OS, for ex TinyOS. Based on the OS installed programming can be done. For TinyOS the programming language is nesC.

II. WSN OPERATING SYSTEMS

2.1 Tinyos OS

As the name itself indicates TinyOS has a very small footprint i.e. 400 bytes [2]. This OS is specially designed for sensor nodes, those have limited memory. Along with limited memory, TinyOS supports other constrained resources like limited battery, low computation capabilities etc. TinyOS has a component-based architecture, so it adopts component-based programming language like nesC. Compared to other WSN Oss like Contiki, MANTIS, RTOS, NANORK etc TinyOS is more advantageous, highly documented and widely used OS [4, 5, 6].

2.1.1 Programming language and Simulator

NesC is the programming language for application development, in fact the tinyOS itself is developed in nesC. While coming to the discussion of simulator,

Manuscript published on 30 December 2018.
* Correspondence Author (s)
Anita Patil, Department of Computer Science and Engineering, Ballari Institute of Technology and Management, Ballari, Karnataka, India (Email: srithap@bitsathy.ac.in)
Rajashree.V.Biradar, Department of Computer Science and Engineering, Ballari Institute of Technology and Management, Ballari, Karnataka, India.

TinyOS has a built-in simulator called TOSSIM, that simulates thousands of nodes which is not an easy task. This is a discrete event-based simulator and simulates TinyOS applications. It has TinyOS code compiling, running, analyzing, and debugging capabilities. The architecture of TOSSIM is simple, but it provides more powerful emulation for WSN applications. Every node can be evaluated under perfect transmission conditions with the hidden node's problems captured [10].

2.1.2 nesC programming structure

The pair of Network Embedded System C (nesC) and TinyOS is a de-facto standard. TinyOS itself is written in nesC language. The nesC was purposely developed to support component-based architecture of TinyOS. It is the improved version of C, developed with more concern towards the unique challenges of WSN [1]. As a component-based language, nesC programs contain two types of components. One is module component, which provides application code to implement one or more interfaces. Another one is configuration component that wires multiple components in the program and looks after programming control flow. The interfaces within the module component declare the services provided and the services used. There could be built-in interfaces or user defined interfaces. The interface consists of commands and events. Commands are requests for some services, and events are signals regarding the completion of requested service. Higher level components send the commands to lower level components and handle the events when they get signals from lower level components. In turn lower level components processes the commands received from higher level components then signal the events. Within a program such multiple components get wired through configurations. [1,7].

So, the overall structure looks like

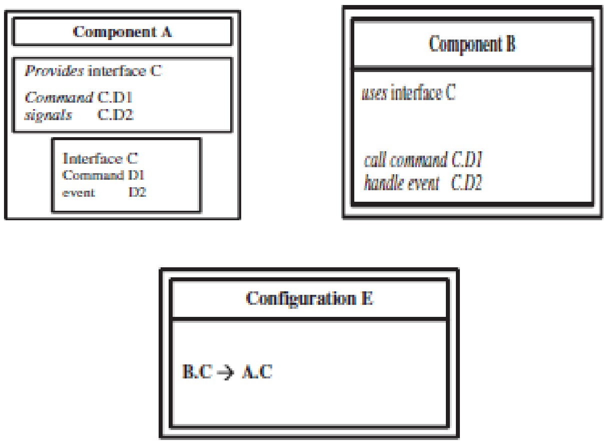


Figure 2. Two components with interfaces and their configuration component.

Here is an example application for detailing the programming structure.

Programming the node for Blink application:

nesC code needs configuration file, module file and make file as described below.

1. Configuration file BlinkAppC has three components

```
configuration BlinkAppC
{
//if user defined interfaces are needed they can be written
here
}
implementation
{
  components MainC, BlinkC, LedsC;
  components new TimerMilliC() as Timer0;
  components new TimerMilliC() as Timer1;
  components new TimerMilliC() as Timer2;

  BlinkC -> MainC.Boot;

  BlinkC.Timer0 -> Timer0;
  BlinkC.Timer1 -> Timer1;
  BlinkC.Timer2 -> Timer2;

  BlinkC.Leds -> LedsC;
}
```

2. Module component implements the interfaces with commands and events.

```
module BlinkC @safe()
{
  uses interface Timer<TMilli> as Timer0;
  uses interface Timer<TMilli> as Timer1;
  uses interface Timer<TMilli> as Timer2;
  uses interface Leds;
  uses interface Boot;
}
implementation
{
  event void Boot.booted()
  {
    call Timer0.startPeriodic( 250 );
    call Timer1.startPeriodic( 500 );
    call Timer2.startPeriodic( 1000 );
  }

  event void Timer0.fired()
  {
    dbg("BlinkC", "Timer 0 fired @ %s.\n",
sim_time_string());
    call Leds.led0Toggle();
  }

  event void Timer1.fired()
  {
    dbg("BlinkC", "Timer 1 fired @ %s \n",
sim_time_string());
    call Leds.led1Toggle();
  }

  event void Timer2.fired()
  {
   dbg("BlinkC", "Timer 2 fired @ %s.\n", sim_time_string());
     call Leds.led2Toggle();
   }
 }
```

3. Make file: The basic structure for a Main Application is Makefile, in which we indicate the name of the configuration file. The compiler will search in the folder for configuration file.

```
COMPONENT=BlinkAppC
include $(MAKERULES)
```

III. RESULTS

The Blink application is executed on different platforms i.e. mica2dot, micaz, and telosb. Along with TOSSIM the different simulators are also used for taking the results. Just for minimizing the size of the output, only one screen shot is shown and only two sensor nodes are taken while executing the program. The Micaz platform shows the command line results consequently Telosb results are visualized. No doubt, the results of actual hardware platforms will give exact results. But the simulated results, as shown in this paper are also accurate to the possible extent. Detailed study of programming and platforms will help to decide to select appropriate platform for any WSN application.

a. Micaz platform

1. Build Blink application using Micaz.
2. Using the command
   avr-objdump -zhD main.exe >main.od
   Create.od file
3. Command to use simulator to display the results.

```
java -jar /home/anithapatil/avrora/avrora-beta-1.7.117.jar -simulation=sensor-network
-seconds=5.0 -nodecount=2 main.od

anithapatil@anitha:/opt
/tinyos-2.1.2/apps/Blink/build/micaz$ java -jar /home/anithapatil/avrora/avrora-beta-1.7.117.jar -simulation=sensor-network -seconds=5.0 -nodecount=2 main.od
```

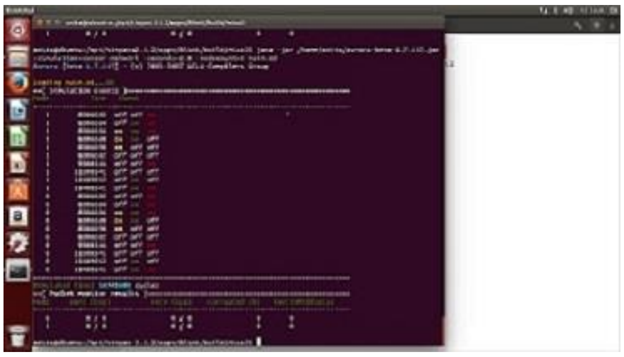


Figure 3. Result for Blink Application.

b. Telosb platform

1. Build Blink application using telosb platform.
2. Telosb Execution command requirements
   1. MSPSIM msp430
   2. EXTRACT mspsim.jar from MSPSIM to /opt/mspsim/mspsim.jar
3. Command to use simulator to display the results.

```
java -jar /opt/mspsim/mspsim.jar main.exe
anithapatil@anitha:/opt/tinyos-2.1.2/apps/Blink/build/telosb$ java -jar /opt/mspsim/mspsim.jar main.exe
```

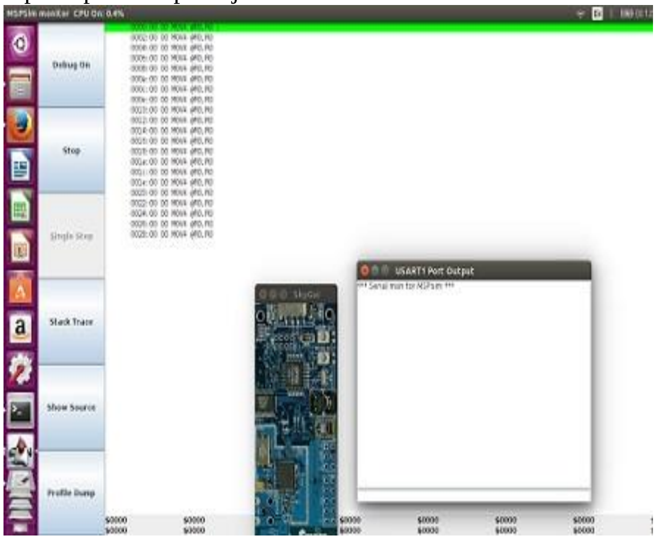


Figure 4. Blink application result on Telosb

2.2 Contiki

Contiki, is a lightweight open source OS suitably designed for the low power and low memory like resource constrained devices. So, this OS fulfils the needs of WSN sensor nodes as well as IOT devices. It is Contiki is a highly portable OS and it is built around an event-driven kernel. Having an event driven kernel, it supports multithreaded programming with the help of protothreads. Contiki provides pre-emptive multitasking schedular that can be used at the individual process level. Plenty of nodes deployed in application area keeps changing the network size and topology because of the death of many nodes due to loss of limited energy. The dynamic linking and loading feature of Contiki OS handles such situations effectively. These features made Contiki OS more suitable for many specific applications. [12,13,14,15]

Programming language and Simulator

The programming language here supported is the well-known C and the simulator used is Cooja.

Contiki installation we done on Ubuntu is not as difficult as TinyOS installation and not as simple as RIOT OS installation. The source code for Contiki OS is available at GitHub link https://github.com/contiki-os/contiki for download. For the Contiki OS installation and example execution the video we referred is https://www.youtube.com/watch?v=BQWU73nKvX8.

2.3 RIOT

RIOT (Real time operating system for IOT) is an operating system for low-powered resource constrained wireless devices which are mostly used in IoT applications.

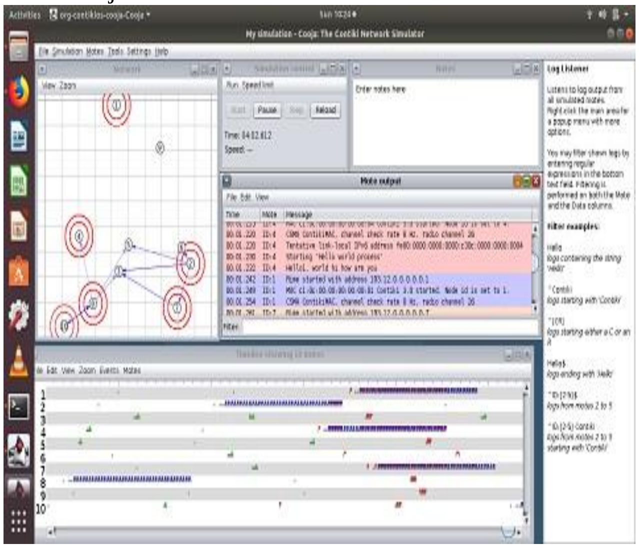
It is important to note that unlike most of other operating systems for IoT applications where event-driven model is used, RIOT is a microkernel-based OS with truly real-time process scheduling capability. RIOT allows programmer to create as threads needed without any restriction just with memory constraint [12]. This feature is the strength for the strong support to real time service. RIOT is developer friendly, resource friendly and IOT friendly OS. [12,16,17,18]

Programming language and Simulator

The programming languages here supported are the well-known C and C++. The simulator is Cooja.

RIOT OS installation is not that complicated as TinyOS installation. Just we have to download RIOT OS source code from Github link https://github.com/RIOT-OS/RIOT.git on ubuntu. RIOT uses COOJA simulator which comes in Contiki OS package. As we had already installed Contiki OS, the COOJA was ready to use. The Example directory of the RIOT source has some example applications. To run an application on COOJA the steps are as follows

1. In the terminal first, we need to build the RIOT application (in which project folder we are) for any Cooja supported board/platform (e.g. make BOARD=z1)
2. In that RIOT project folder, we need to go to bin/boardname (e.g. z1) folder and rename projectname.elf file to projectname.z1(boardname)
3. Now in the terminal go to the Contiki/tools/Cooja and run 'sudo ant run' command to open the Cooja simulator. Now we can create new simulation and add any number of this mote type (having your application) by navigating in RIOT directory to projectname/bin/boardname folder and selecting this file (projectname.z1) for creating mote in Cooja.



```
Time       Mote  Message
00:01.220  ID:4  CSMA ContikiMAC, channel check rate 8 Hz, radio channel 26
00:01.220  ID:4  Tentative link-local IPv6 address fe80:0000:0000:0000:c30c:0000:0000:0004
00:01.230  ID:4  Starting 'Hello world process'
00:01.222  ID:4  Hello, world hi how are you
00:01.242  ID:1  Mote started with address 193.12.0.0.0.0.0.1
00:01.249  ID:1  MAC c1:0c:00:00:00:00:00:01 Contiki 3.0 started. Node id is set to 1.
00:01.254  ID:1  CSMA ContikiMAC, channel check rate 8 Hz, radio channel 26
```

Figure 5: COOJA simulator results

2.4 FreeRTOS

FreeRTOS is a real-time operating system for embedded devices like tiny constrained microcontrollers with small memory footprint, low overhead, and fast execution features. Preemptive and cooperative scheduling is the main highlight of freeRTOS to fulfill real-time requirement of the applications. For less power consumption freeRTOS has Tick-less options. FreeRTOS has simple and lightweight tasks called coroutines for limited use of call stack.FreeRTOS hasTrace support through generic trace macros. Tools such as Tracealyzer by FreeRTOS partner Percepio can thereby record and visualize the runtime behavior of FreeRTOS-based systems. This includes task scheduling and kernel calls for semaphore and queue operations. The official website of freeRTOS https://freertos.org/ provide the detailed directory structure of the freeRTOS [12,19,20,21].

Programming language and Simulator:

The programming language here supported is the well-known C and the simulation results are shown in Microsoft Visual Studio – 2017.

We downloaded the latest version FreeRTOSv10.0.1 from the official website. The link https://www.coursera.org/lecture/autonomous-runway-detection/freertos-hello-world-tutorial-3vALN shows the example application execution on Microsoft Visual Studio – 2017.

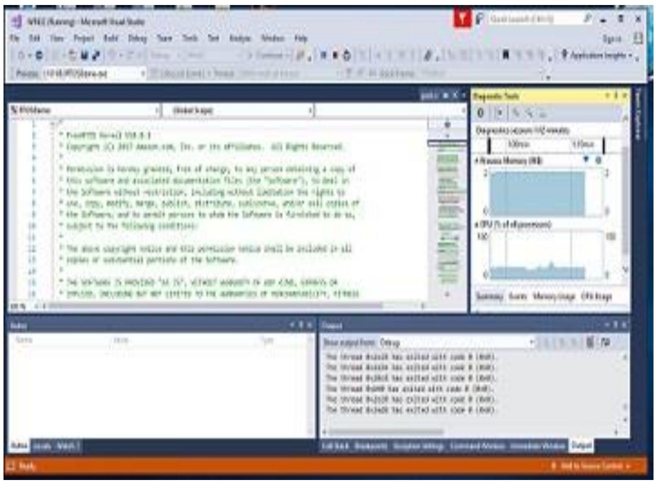


Figure 6: Visual Studio results for freeRTOS applications

IV. COMPARATIVE ANALYSIS

This comparative study of, 'programming the nodes in different OSs' helps us to incorporate the suitable features of one language in to another language according to our task processing requirement. For example, TinyOS being highly documented and popular OS has only FIFO scheduling, if we would like to write priority scheduling code for this in nesC language, then we can take input from RIOT or FreeRTOS priority scheduling codes.

Table 1: basic elements of programming

OS	Language support	Simulator support	Processor Scheduling
Tiny OS	nesC	TOSSIM	FIFO
Contiki	C	COOJA	Partial priority
RIOT	C, C++	COOJA	Priority
FreeRT OS	C	Visual studio, eclipse	Priority

V. CONCLUSION

Programming is required in every level of WSN hierarchy, but the scope of this paper is limited to node level programming only. This paper not only depicts the contemporary state of the art for WSN programming, but also describes about the programming environments for the OSs TinyOS, Contiki, RIOT and freeRTOS. The discussion on programming in these multiple Oss, helps to understand and compare different programming approaches as well as to incorporate the better features like priority scheduling from FreeRTOS into TinyOS. Hence our contribution in this paper can be viewed as an introductory manual for the beginners in WSN programming.

REFERENCES

1. Waltenegus Dargie, Christian Poellabauer, Book, “Fundamentals of wireless sensor networks theory and practice”,2010 John Wiley & Sons Ltd.
2. PHILIP LEVIS, DAVID GAY, “TinyOS Programming book”, Intel Research © Cambridge University Press 2009, ISBN-13 978-0-511-50730-4 eBook (EBL).
3. David Gay,Philip Levis, Robert von Behren, Welsh, M.; Brewer, E.; Culler, D, “ The nesC Language: A Holistic Approach to Networked Embedded Systems”, 2003 Conference on Programming Language Design and Implementation, New York, NY, USA, May 2003.
4. Muhammad Amjad, Muhammad Sharif, Muhammad Khalil Afzal, and Sung Won,Kim, “TinyOS-New Trends, Comparative Views, and Supported Sensing Applications:A Review “, IEEE SENSORS JOURNAL, VOL.16, NO. 9, MAY 1, 2016.
5. Zhang Jing, Xue Leng, Fan Hongbo, Cui Yi, “TQS-DP: A Lightweight and Active Mechanism for Fast Scheduling Based on WSN Operating System TinyOS “,978-1-4799-7016-2/15/$31.00 2015 IEEE.
6. Anita Patil, Dr. Rajashree.V.Biradar ,” Scheduling Techniques for TinyOS: A Review “, 2016 International Conference on Computational Systems and Information Systems for Sustainable Solutions 978-1-5090-1022-6/16/$31.00 ©2016 IEEE
7. RYO SUGIHARA and RAJESH K. GUPTA, “Programming Models for Sensor Networks: A Survey”, University of California, San Diego ACM Transactions on Sensor Networks, Vol. 4, No. 2, Article 8, Publication date: March 2008.
8. Tobias Reusing, Betreuer: Christoph Söllner,” Comparison of Operating Systems TinyOS and Contiki”, Seminar: Sensorknoten - Betrieb, Netze & Anwendungen SS2012 Network Architectures and Services, August 2012
9. Tej Bahadur Chandra, Anuj Kumar Dwivedi,” Programming Languages for Wireless Sensor Networks: A Comparative Study”,978-9-3805-4416-8/15/$31.00_c 2015 IEEE.
10. Rudradeep Nath, “TOSSIM Based Implementation and Analysis of Collection Tree Protocol in Wireless Sensor Networks”, International conference on Communication and Signal Processing, April 3-5, 2013, India 978- 1-4673-4866-9/13/$31.00 ©2013 IEEE.
11. Zeenat Rehena, Krishanu Kumar, Sarbani Roy, “Nandini Mukherjee,SPIN Implementation in TinyOS Environment using nesC”, Second International conference on Computing, Communication and Networking Technologies, 2010.
12. Aleksandar Milinković, Stevan Milinković, Ljubomir Lazić Metropolitan University, Faculty of Information Technology Belgrade, Serbia, “Choosing the right RTOS for IoT platform”, INFOTEH-JAHORINA Vol. 14, March 2015.
13. https://github.com/contiki-os/contiki
14. http://contiki-os.org/
15. https://en.wikipedia.org/wiki/Contiki
16. https://www.riot-os.org/,
17. https://en.wikipedia.org/wiki/RIOT(operating system)
18. https://github.com/RIOT-OS/RIOT
19. https://www.freertos.org
20. https://en.wikipedia.org/wiki/FreeRTOS
21. https://www.iot-lab.info/operating-systems/