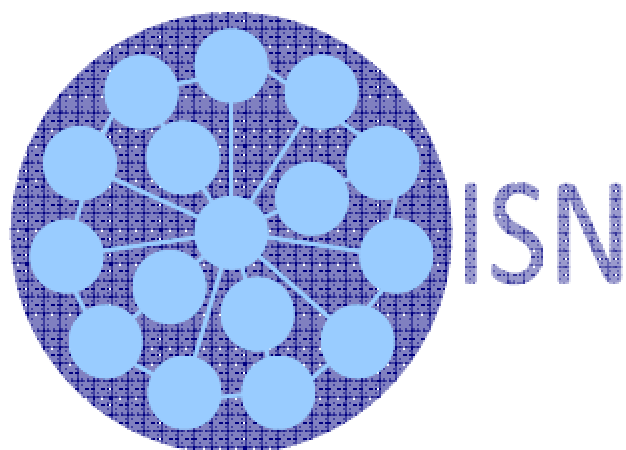

CONTIKI AND TINYOS

Version: 0.4 (August 13, 2012)

Edited by: Edosoft Factory, S.L (Edosoft)

Project Data

Acronym: ISN

Name: Interoperable Sensor Networks

ITEA number: 09034

Consortium:

- Vrije Universiteit Brussel
- Freemind
- MTP
- Edosoft
- MAIS

Document data:

Doc name: Contiki and Tiny OS

Doc version: 0.4

Doc type:

Version	Date	Remarks
0.1	July 20, 2012	First Draft
0.2	July 27, 2012	Added TinyOS
0.3	August 3, 2012	Added Comparative Analysis
0.4	August 13, 2012	Preliminary version

Contents

1. Purpose 4
2. Introduction 4
3. Contiki 4
3.1. Architecture 5
3.2. Programming Model 5
3.3. Scheduling 6
3.4. Memory Management and Protection 6
3.5. Communication Protocol Support 6
3.6. Resource Sharing 7
3.7. Support for Real-Time Applications 7
3.8. 6LowPAN Implementations 7
3.9. Additional Features 7
4. TinyOS 8
4.1. Architecture 8
4.2. Programming Model 9
4.3. Scheduling 10
4.4. Memory Management and Protection 10
4.5. Communication Protocol Support 10
4.6. Resource Sharing 11
4.7. Support for Real-Time Applications 11
4.8. 6LowPAN Implementations 11
4.9. Additional Features 12
5. Comparative Analysis 13
6. References 14

Figures

Figure 1: Contiki Architecture 5
Figure 2: TinyOS Architecture 9
Figure 3: Operating System Summary 13
Figure 4: Miscellaneous Features Summary 13
Figure 5: 6LowPan Implementations 13

1. Purpose

This document contains the State-of-the-art of Contiki and TinyOS Operating Systems in the scope of the Interoperable Sensor Networks (ISN) project. This document integrates the deliverable *State-of-the-art: communication, technology, interoperability*, which addressed the state-of-the-art study on communication protocols done in the first part of the project.

2. Introduction

In complex systems, the Operating System (SO) acts as resource manager. The job of the SO in a typical system is to manage its resources (processors, memories, timers, disks, mice, keyboard, network interfaces, etc.) to users in an orderly and controlled manner. Users and programmers can invoke different OS services via system calls through multiplexing of OS system resources in time and in space. Time multiplexing involves different programs taking turn in using the resources. On the other hand, space multiplexing involves different programs accessing parts of the resource, possibly at the same time.

The development of miniaturized and cheap sensor nodes, capable of communicating wirelessly, sensing and performing computations is due to advances in Micro-Electro Mechanical System (MEMS)-based sensor technology. As we know a wireless sensor node is composed of a microcontroller, transceiver, timer, memory and analog to digital converter. The main and most critical resources in sensor node are energy (is provide by a battery) and memory (is allowed only a few kilobytes). The microcontroller operates al low frequency compared to traditional processing units. Dense deployment of sensor nodes in the sensing field and distributed processing through multi-hop communication among sensor nodes is required to achieve high quality and fault tolerance in WSNs. Considering the resource constraints of typical sensor nodes in a WSN, a new approach is required for OS design in WSN.

This document is focusing on OS for severely resource-constraint WSN nodes such as motes. We have examined the core OS features, such as its Architecture, Programming Model, Scheduling, Memory Management and Protection, Communication Protocols, Resource Sharing, and Support for Real-Time Applications, in both real-time and non-real-time WSN OSs. The document focuses on Contiki and TinyOS.

3. Contiki

Contiki[2] is an open source OS for WSN sensor nodes. It is a lightweight and portable OS written in C language and it is build around an event-driven kernel. This OS provides preemptive multitasking that can be used at the individual process level. A typical Contiki configuration consumes 2 kilobytes of RAM and 40 kilobytes of ROM. A full Contiki installation includes features like: multitasking kernel, preemptive multithreading, proto-threads, TCP/IP networking, IPv6, a Graphical User Interface, a web browser, a personal web server, a simple telnet client, a screensaver, and virtual network computing.

3.1. Architecture

The Contiki OS is based on a modular architecture. At the kernel level it follows the event driven model, but it provides optional threading facilities to individual processes. This kernel comprises of a lightweight event scheduler that dispatches events to running processes. Process execution is triggered by events dispatched by the kernel to the processes or by a polling mechanism. This polling mechanism is used to avoid race conditions. Any scheduled event will run to completion, however, event handlers can use internal mechanisms for preemption.

Asynchronous events and synchronous events are supported by Contiki OS. Synchronous events are dispatched immediately to the target process that causes it to be scheduled. On the other hand asynchronous events are more like deferred procedure calls that are en-queued and dispatched later to the target process.

The polling mechanism can be seen as high-priority events that are scheduled in between each asynchronous event. When a poll is scheduled, all processes that implement a poll handler are called in order of their priority.

All OS facilities: senor data handling, communication, device drivers, etc. are provided in the form of services. Each service has its interface and implementation. Applications using a particular service need to know the service interface and an application is not concerned about the implementation of a service.

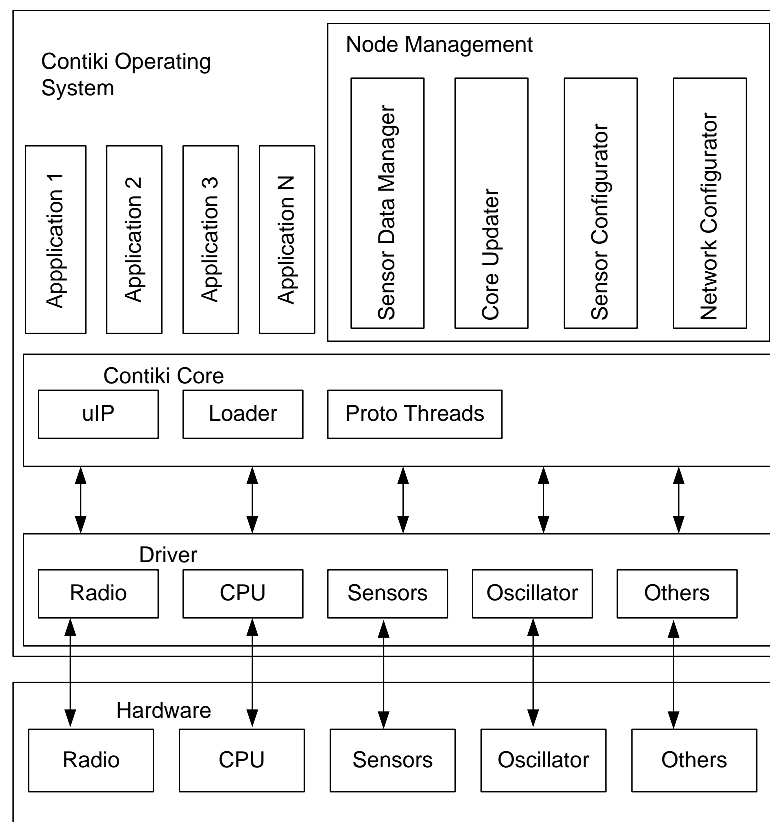

Figure 1: Contiki Architecture

3.2. Programming Model

Contiki uses protothreads [3] for multithreading. Protothreads are designed for severely memory constraint devices because they are stack-less and lightweight. The main features of protothreads are: onlty two bytes per protothread, no extra stack for a thread and

highly portable. Since events run to completion and Contiki does not allow interrupt handlers to post new events, no process synchronization is provided in Contiki.

Contiki supports preemptive multithreading and this is implemented as a library on top of the event-driven kernel. The library can be linked with applications that require multithreading. The Contiki multithreading library is divided in two parts: a platform independent part and a platform specific part. The platform independent part interfaces to the event kernel and the platform specific part of the library implements stack switching and preemption primitives. Since preemption is supported, preemption is implemented using the timer interrupt and the thread state is stored on a stack.

3.3. Scheduling

Contiki does not employ any sophisticated scheduling algorithm because it is an event-driven OS. Events are fired to the target application as they arrive. In case of interrupts, interrupt handlers of an application runs with regard to their priority.

3.4. Memory Management and Protection

Contiki supports dynamic memory management. Contiki provides memory block management functions and supports dynamic linking of the programs. For providing memory block management functions the library provides simple but powerful memory management functions for blocks of fixed length. A memory block is statically declared using the MEMB() macro. Memory blocks are allocated from the declared memory by the memb_alloc() function, and are de-allocated using the memb_free() function.

Apart from this it also supports dynamic linking of the programs. In order to guard against memory fragmentation Contiki uses a Managed Memory Allocator [4]. The primary responsibility of the managed memory allocator is to keep the allocated memory free from fragmentation by compacting the memory when blocks are free. Therefore, a program using the memory allocator module cannot be sure that allocated memory stays in place.

Contiki does not provide any memory protection mechanism between different applications.

3.5. Communication Protocol Support

Contiki provides an implementation of TCP/IP protocol stack for small 8 bit micro-controllers (uIP). uIP does not require its peers to have a complete protocol stack, but it can communicate with peers running a similar lightweight stack. The uIP implementation is written in C and it has the minimum set of features needed for a full TCP/IP stack. uIP can only support one network interface, and it supports TCP, UDP, ICMP, and IP protocols.

On the other hand, Contiki provides RIME. RIME is another lightweight layered protocol stack for network-based communication. Rime supports both best effort and reliable transmission. In multi-hop communication, Rime allows applications to run their own routing protocols. Rime provides single hop unicast, single hop broadcast, and multi-hop communication support. Applications are allowed to implement protocols that are not present in the Rime stack.

However Contiki does not support multicast. Therefore Contiki does not provide any implementation of group management protocols such as the Internet Group Management Protocol (ICMP), or Multicast Listener Discovery (MLD) protocol. Whenever a packet is received, Contiki places it in the global buffer and notifies the TCP/IP stack. If it is a data packet, TCP/IP notifies the appropriate application. The application needs to copy the data in the secondary buffer or it can immediately process the data. Once the application is done with the received data, Contiki overwrites the global buffer with new incoming data. If an application delays data processing, then data can be overwritten by new incoming data packets.

To provide IPv6 routing protocol for low power and lossy networks exists an implementation of RPL called ContikiRPL[6]. ContikiRPL operates over low power wireless links and lossy power line links.

3.6. Resource Sharing

Contiki provides serialized access to all resources due to events run to completion and Contiki does not allow interrupt handlers to post new events

3.7. Support for Real-Time Applications

Support for real-time applications is not allowed. There is no implementation of any real-time process scheduling algorithm in Contiki. Contiki does not provide any protocol that considers the QoS requirements of multimedia applications on the network protocol stack side. In addition, since Contiki provides an implementation of the micro IP stack, interactions between different layers of the protocol stack are not possible.

3.8. 6LowPAN Implementations

SicslowPAN is the first implementation of 6LoWPAN developed for the Contiki OS. SICSlowPAN is based on the RFC4944, and implements mechanisms for addressing and fragmentation. It also adds a new header compression mechanism. SICSlowPAN does not implement networks "Mesh under" but supports using other routing techniques.

On the other hand, SICSlowPAN is located between IPv6 layer and MAC layer. Therefore, when the MAC layer receives an IPv6 packet, call SICSlowPAN layer to adapt from the MAC layer packet to the IP layer. Furthermore, when uIPv6 layer needs to send a package also uses the SICSlowPAN layer to form the package.

Contiki operating system implements a default non-IP protocol on the MAC layer, called RIME. uIPv6 and uIPv4 stacks have recently been added. uIPv6 stack is independent of any of the lower layers, making it possible to send data over standard 802.15.4, 802.11 or 6LoWPAN. uIPv6 stack supports ICMPv6 packets, implements the neighbour discovery protocol (Neighbour Discovery) and supports both UDP and TCP.

3.9. Additional Features

- **Cofee File System**: Contiki uses the Coffee file system. This file system gives support for flash-based sensor devices. Coffee file system provides a programming interface for building efficient and portable storage abstractions

and provides provides a platform independent storage abstraction through an expressive programming interface. Coffee uses a small and constant memory footprint per file (in default setup uses 5 Kb ROM for the code and 0.5 Kb RAM at run-time), making it scalable. Coffee also introduces the concept of micro logs to handle file modifications without using a spanning log structure. Because of the contiguous page structure file metadata, Coffee uses a small and constant footprint for each file. Flash memory is divided into logical pages and the size of the page typically matches the underlying flash memory pages. If the file size is not known beforehand, Coffee allocates a predefined amount of pages to the file. Later on, if the reserved size turns out to be insufficient, Coffee creates a new larger file and copies the old file data into it. To boost the file system performance, by default Coffee uses a metadata cache of 8 entries in the RAM. Coffee also provides an implementation of a garbage collector that reclaims obsolete pages when a file reservation request cannot be satisfied. To allocate pages to a file, Coffee uses a first fit algorithm. Flash memories suffer from wear, i.e., every time a page is erased it increases the chances of memory corruption. Coffee uses wear levelling and its purpose is to spread sector erasures evenly to minimize the risk of damaging some sectors much earlier than others. Coffee provides the following APIs to the application programmers. open(), read(), modify(), seek(), append(),close().

- **Security Support**: Contiki does not provide support for secure communication. There is a proposal and implementation of a secure communication protocol with the name ContikiSec
- **Simulation Support**: Contiki provides sensor network simulations through Cooja [8].
- **Languaje Support:** Cotiki supports application development in the C language.
- **Supported Platforms:** Contiki supports the following sensing platforms: Tmote [9], AVR series MCU [10].
- **Documentation Support:** Contiki documentation can be found on the Contiki home page at: http://www.sics.se/contiki.

4. TinyOS

TinyOS [3] is an open source, flexible, component based, and application specific operating system designed for sensor networks. TinyOS can support concurrent programs with very low memory requirements. The TinyOS has a footprint that fits in 400 bytes. The TinyOS component library includes network protocols, distributed services, sensor drivers and data acquisition tools.

4.1. Architecture

TinyOs is based on monolithic architecture class. TinyOS uses the component model and different components are glued together with the scheduler to compose a static image that runs on the mote. A component is an independent computational entity that exposes one or more interfaces. Components have three computational abstractions: commands, events, and tasks. Mechanisms for inter-component communication are commands and events. Tasks are used to express intra component concurrency. A command is a request to perform some service, while the event signals the completion

of the service. TinyOS provides a single shared stack and there is no separation between kernel space and user space.

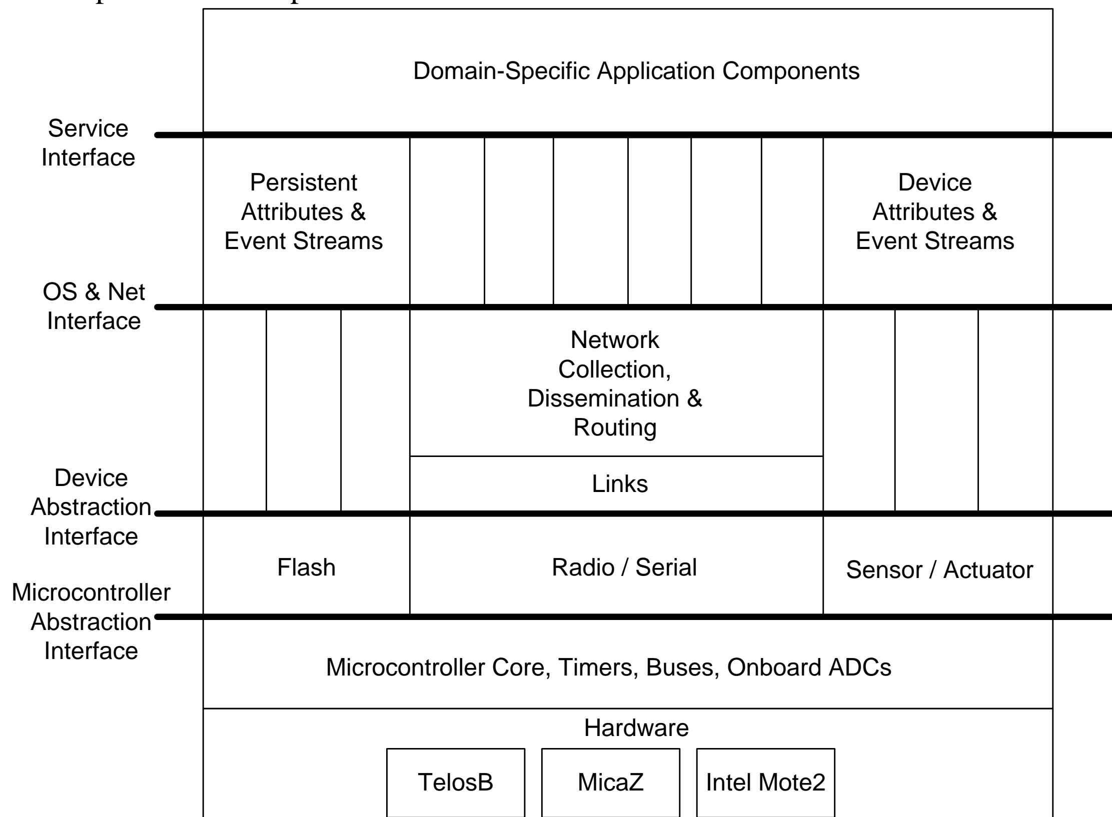

Figure 2: TinyOS Architecture

4.2. Programming Model

TinyOS version 2.1 provides support for multithreading and these TinyOS threads are called TOS Threads. Given the motes' resource constraints, an event-based OS permits greater concurrency. However, preemptive threads offer an intuitive programming paradigm. The TOS threading package provides the ease of a threading programming model coupled with the efficiency of an event driven kernel. TOS threads are backward compatible with existing TinyOS code. TOS threads use a cooperative threading approach, i.e., TOS threads rely on applications to explicitly yield the processor. This adds an additional burden on the programmer to explicitly manage the concurrency. Application level threads in TinyOS can preempt other application level threads but they cannot preempt tasks and interrupt handlers. A high priority kernel thread is dedicated to running the TinyOS scheduler. For communication between the application threads and the kernel, TinyOS 2.1 provides message passing. When an application program makes a system call, it does not directly execute the code. Rather it posts a message to the kernel thread by posting a task. Afterwards, the kernel thread preempts the running thread and executes the system call. This mechanism ensures that only the kernel directly executes TinyOS code. System calls like Create, Destroy, Pause, Resume and Join are provided by the TOS threading library.

TOS threads dynamically allocate Thread Control Blocks (TCB) with space for a fixed size stack that does not grow over time. TOS Threads context switches and system calls introduce an overhead of less than 0.92%.

Earlier versions of TinyOS imposed atomicity by disabling the interrupts, i.e., telling the hardware to delay handing the external events until after the application completed an atomic operation. This scheme works well on uniprocessor systems. Critical section can occur in the user level threads and the designer of the OS does not want the user to disable the interrupts due to system performance and usability issues. This problem is circumvented in TinyOS version 2.1. It provides synchronization support with the help of condition variables and mutexes. These synchronization primitives are implemented with the help of special hardware instructions e.g., test & set instruction.

4.3. Scheduling

Earlier versions of TinyOS supported a non-preemptive First-In-First-Out (FIFO) scheduling algorithm and those versions do not support real-time application. The core of the TinyOS execution model are tasks that run to completion in a FIFO manner. Since TinyOS supports only non preemptive scheduling, task must obey run to completion semantics. Tasks run to completion with respect to other task but they are not atomic with respect to interrupt handlers, commands, and events they invoke. Since TinyOS uses FIFO scheduling, disadvantages associated with FIFO scheduling are also associated with the TinyOS scheduler. The wait time for a task depends on the task's arrival time. FIFO scheduling can be unfair to latter tasks especially when short tasks are waiting behind longer ones.

4.4. Memory Management and Protection

Hardware-based memory protection is not available and the resources are scarce in sensor nodes.Resource constraints necessitate the use of unsafe, low level languages like nesC [12]. Memory safety is incorporated in TinyOs version 2.1 [13]. The goals for memory safety are: trap all pointer and array errors, provide useful diagnostics, and provide recovery strategies. Implementations of memory-safe TinyOS exploit the concept of a Deputy. The Deputy is a resource to resource compiler that ensures type and memory safety for C code. Code compiled by Deputy relies on a mix of compile and run-time checks to ensure memory safety. Safe TinyOS is backward compatible with earlier version of TinyOS. The Safe TinyOS tool chain inserts checks into the application code to ensure safety at run-time. When a check detects that safety is about to be violated, code inserted by Safe TinyOS takes remedial actions. TinyOS uses a static memory management approach.

4.5. Communication Protocol Support

Earlier versions of TinyOS provide two multi-hop protocols: dissemination and TYMO [14,15]. The dissemination protocol reliably delivers data to every node in the network. This protocol enables administrators to reconfigure queries and to reprogram a network. The dissemination protocol provides two interfaces: DisseminationValue and DisseminationUpdate. A producer calls DisseminationUpdate. The command DisseminationUpdate.change() should be called each time the producers wants to disseminate a new value. On the other hand, the DisseminationValue interface is provided for the consumer. The event DisseminationValue.changed() is signaled each time the dissemination value is changed. TYMO is the implementation of the DYMO

protocol, a routing protocol for mobile ad hoc networks. In TYMO, packet formats have changed and it has been implemented on top of the active messaging stack.

TinyOS version 2.1.1 now also provides support for 6lowpan [16], an IPv6 networking layer within a TinyOS network.

At the MAC layer, TinyOS provides an implementation of the following protocols: a single hop TDMA protocol, a TDMA/CSMA hybrid protocol which implements Z-MAC's slot stealing optimization, B-MAC, and an optional implementation of an IEEE 802.15.4 complaint MAC.

4.6. Resource Sharing

TinyOS has two mechanisms for managing shared resources: Virtualization and Completion Events. A virtualized resource appears as an independent instance. i.e., the application uses it independent of other applications. Resources that cannot be virtualized are handled through completion events. The GenericComm communication stack of TinyOS is shared among different threads and it cannot be virtualized. GenericComm can only send one packet at a time, send operations of other threads fail during this time. Such shared resources are handled through completion events that inform waiting threads about the completion of a particular task.

4.7. Support for Real-Time Applications

TinyOS does not provide any explicit support for real-time applications. Tasks in TinyOS observe run to completion semantics in a FIFO manner, hence in its original form, TinyOS is not a good choice for sensor networks that are being deployed to monitor real-time phenomena. An effort has been made to implement an Earliest Deadline First (EDF) process scheduling algorithm and it has been made available in newer versions of TinyOS. However, it has been shown that the EDF algorithm cannot produce a feasible schedule when tasks content for resources. In the nutshell, TinyOS is not a strong choice for real-time applications.

In relation to Quality of Service requirements of real-time multimedia streams, TinyOS does not provide any specific MAC, network, or transport layers protocol implementations. At the MAC layer, TinyOS supports TDMA, which can be fine-tuned depending upon the requirements of an application to support multimedia traffic streams.

4.8. 6LowPAN Implementations

The first implementation is called 6lowpancli, was released in 2007 and is now integrated as a native application in TinyOS-2.x. 6lowpancli supports header compression, fragmentation and addressed. Regarding the higher level protocols, 6lowpancli supports UDP, holding it in the form HC_UDP and ICMPv6 messages. Therefore it is possible to ping the motes and use UDP communication from anywhere on the Internet to a small sensor node. To perform a bridge between the sensor network and conventional network, TinyOS-2.x provides a small application to emulate a tunnel IPv6 through the USB port through which you can connect the sensor node. The main drawback of 6lowpancli is that it is completely static and needs to be configured manually. Not support any mechanism for discovery of neighboring nodes (Neighbor Discovery Protocol) or mobility. Besides configuring mesh network (mesh networks) is

not supported and when a packet with a destination address different from the node is received, it is omitted.

The second implementation of 6LoWPAN developed for TinyOS-2.x is called BLIP (Berkeley Low-Power IP stack). In this case the code is not included natively in TinyOS-2.x but can be found on the contributions of the University of Berkeley. BLIP implements the basic features defined in RFC4944, i.e., header compression, fragmentation and addressing. Support for ICMPv6 and UDP packets. Moreover, the latest version adds the first prototype of the TCP stack, which is still experimental. In addition, BLIP incorporates a "light" version of Discovery Protocol neighboring nodes (Neighbor Discovery Protocol), being able to configure a link local address or global in neighboring nodes, depending on whether or not it has received a frame of notice from the router. As 6lowpancli, BLIP includes an application to create an IPv6 tunnel to connect a conventional network to wireless sensor network. BLIP addition, unlike 6lowpancli supports mesh networks (mesh networks) as defined in RFC4944 also called "mesh under". "Mesh under" indicates that from the point of view of the IP layer, all nodes are one hop away, since multiple jumps are performed by the MAC layer.

4.9. Additional Features

- **File System**: TinyOS provides a single level file system. The rationale behind providing a single level file system is the assumption that only a single application runs on the node at any given point in time. As node memory is scarce, having a single level file system is therefore sufficient.
- **Database Support**: The purpose of sensor nodes is to sense, perform computations, store and transmit data; therefore TinyOS provides database support in the form of TinyDB.
- **Security Support**: Communication security in wireless broadcast medium is always required. TinyOS provides its communication security solution in the form of TinySec [17].
- **Simulation Support**: TinyOS provides simulation support in the form of TOSSIM [18]. The simulation code is written in NesC and consequently can also be deployed to actual motes.
- **Language Support**: TinyOS supports application development in the NesC programming language. NesC is a dialect of the C language.
- **Supported Platforms**: TinyOS supports the following sensing platforms: Mica, Mica2, Micaz, Telos, Tmote and a few others.
- **Documentatation Support:** TinyOS is a well documented OS and extensive documentation can be found on the TinyOS home page at http://www.tinyos.net.

5. Comparative Analysis

OS	Architecture	Programming Model	Sheduling	Memory Management and Protection	Communication Protocol Support	Resource Sharing	Support for Real-Time Applications	6LowPan Implementations
Contiki	Modular	Protothreads and events	Events are fired as they occur. Interrupts execute w.r.t. priority	Dynamic memory management and linking. No process address space protection.	uIP and Rime	Serialized Access	No	Sicslowpan
TinyOS	Monolitic	Primarily event Driven, support for TOS threads has been added	FIFO	Static Memory Management with memory protection	Active Message	Virtualization and Completion Events	No	6lowpancli and BLIP

Figure 3: Operating System Summary

OS	Communication Security	File System Support	Simulation Support	Programming Languaje	Shell
Contiki	ContikiSec	Coffee file system	Cooja	C	Unix-like shell runs on sensor mote
TinySO	TinySec	Single level file system	TOSSIM	NesC	Not available

Figure 4: Miscellaneous Features Summary

6LowPAN Implementation	ICMPv6 Echo	UDP	TCP	Neighbor Discovery	Mesh Header	Route Over
SiscLowPAN(Contiki)	YES	YES	YES	YES	YES	YES
6Lowpancli(Tinyos-2x)	YES	YES	NO	NO	NO	NO
BLIP (TinyOS-2.x)	YES	YES	PROTOTIPE	YES	YES	YES

Figure 5: 6LowPan Implementations

6. References

[1] http://www.itea2.org/

[2] Dunkels, A.; Gronvall, B.; Voigt, T. Contiki a Lightweight and Flexible Operating System for Tiny Networked Sensors. In *Proceedings of the 9th Annual IEEE International Conference on Local Computer Networks*, Washington, DC, USA, October 2004; pp. 455-462.

[3] *Protothreads: Lightweight, Stackless Threads in C*; Available online: http://www.sics.se/ ~adam/pt/

[4] *Contiki Documentation*; Available online: http://www.sics.se/~adam/contiki/docs/

[5] Winter, T.; Thubert, P. *RPL: Ipv6 Routing Protocol for Low Power and Lossy Networks*, *Draft-ietf-roll-rpl-11*; Available online: http://tools.ietf.org/html/draft-ietf-roll-rpl-06 (accessed on 17 April 2011).

[6] Tsiftes, N.; Eriksson, J.; Dunkels, A. Low-Power Wireless Ipv6 Routing with ContikiRPL. In *Proceedings of ACM/IEEE IPSN*, Stockholm, Sweden, 12–16 April 2010.

[7] Tisftes, N.; Dunkels, A.; He, Z.; Voigt, T. Enabling Large Scale Storage in Sensor Networks with the Coffee File System. In *Proceedings of the 9th International Conference on Information Processing in Sensor Networks*, Francisco, CA, USA, 13–16 April 2009.

[8] Osterlind, F.; Dunkels, A.; Eriksson, J.; Finne, N.; Voigt, T. Cross Level Sensor Network Simulation with Cooja. In *Proceedings of the 31st IEEE Conference on Local Computer Networks (LCN)*, Tampa, FL, USA, 14–16 November 2006.

[9] *MoteIV Cooperation*; Available online: http://www. Moteiv.com

[10] *Atmel AVR Devices*.; Available online: http://www.atmel.com/products/avr/

[11] Levis, P.; Madden, S.; Polastre, J.; Szewczyk, R.; Whitehouse, K.; Woo, A.; Gay, D.; Hill, J.; Welsh, M.; Brewer, E.; Culler, D. *Tinyos: An Operating System for Sensor Networks*; Available online: http://dx.doi.org/10.1007/3-540-27139-2_7.

[12] Gay, D.; Levis, P.; von Behren, R.; Welsh, M.; Brewer, E.; Culler, D. The NesC Language: A Holistic Approach to Networked Embedded Systems. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, New York, NY, USA, May 2003.

[13] Cooprider, N.; Archer, W.; Eide, E.; Gay, D.; Regehr, J. Efficient Memory Safety for Tinyos. In *Proceedings of the 5th International Conference on Embedded Networked Sensor Systems (SenSys'07)*, New York, NY, USA, November 2007; pp. 205-218.

[14] *TinyOS Network Working Group*; Available online: http://docs.tinyos.net/index.php/TinyOS_ Tutorials#Network_Protocols

[15] *Network Protocols TinyOS documentation Wiki*; Available online: http://docs.tinyos.net/ index.php/Network_Protocols

[16] Montenegro, G.; Kushalnagar, N.; Hui, J.; Culler, D. *Transmission of Ipv6 Packets over IEEE 802.15.4 Networks*, *RFC 4944*; Available online: http://tools.ietf.org/html/rfc4944

[17] Karlof, C.; Sastry, N.; Wagner, D. TinySec: A Link Layer Security for Wireless Sensor Networks. In *Proceedings of the 2th ACM SenSys*, Baltimore, MD, USA, 3–5 November 2004.

[18] Levis, P.; Lee, N.; Welsh, M.; Culler, D. ToSSIM: Accurate and Scaleable Simulation of Entire TinyOS Applications. In *Proceedings of the 1st ACM SenSys*, Los Angeles, CA, USA, 5–7 November 2003.

[19] 6LoWPAN, the wireless embedded Internet. Zach Selby, Carsten Bormann. WILEY

[20] Interconnecting Smart Objects with IP. Jean-Philippe Vasseur, Adam Dunkels. MK.

[21] Evaluating 6lowPAN implementations in WSNs. Ricardo Silva. Jorge Sá Silva and Fernando Boavida, Department of Informatics Engineering, University of Coimbra.

[22] Evaluation of IPv6 over low power- Wireless personal area networks implementations. Kevin Dominik Korte,Iyad Tumar,Jürgen Schönwälder, Computer Science. Jacobs University Bremen.