IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received January 8, 2018, accepted February 9, 2018, date of publication February 21, 2018, date of current version March 12, 2018.

Digital Object Identifier 10.1109/ACCESS.2018.2808324

A Survey on Resource Management in IoT Operating Systems

ARSLAN MUSADDIQ[1], YOUSAF BIN ZIKRIA[1], (Senior Member, IEEE), OLIVER HAHM[2], HEEJUNG YU[1], ALI KASHIF BASHIR[3], (Senior Member, IEEE), AND SUNG WON KIM[1]

[1]Department of Information and Communication Engineering, Yeungnam University, Gyeongsan 38541, South Korea
[2]Zuehlke Engineering GmbH, 65760 Eschborn, Germany
[3]Faculty of Science and Technology, University of the Faroe Islands, 100 Faroe Islands, Denmark

Corresponding author: Sung Won Kim (swon@yu.ac.kr)

This work was supported by the 2017 Yeungnam University Research Grant.

ABSTRACT Recently, the Internet of Things (IoT) concept has attracted a lot of attention due to its capability to translate our physical world into a digital cyber world with meaningful information. The IoT devices are smaller in size, sheer in number, contain less memory, use less energy, and have more computational capabilities. These scarce resources for IoT devices are powered by small operating systems (OSs) that are specially designed to support the IoT devices' diverse applications and operational requirements. These IoT OSs are responsible for managing the constrained resources of IoT devices efficiently and in a timely manner. In this paper, discussions on IoT devices and OS resource management are provided. In detail, the resource management mechanisms of the state-of-the-art IoT OSs, such as Contiki, TinyOS, and FreeRTOS, are investigated. The different dimensions of their resource management approaches (including process management, memory management, energy management, communication management, and file management) are studied, and their advantages and limitations are highlighted.

INDEX TERMS Internet of Things, operating systems, resource management, Contiki, TinyOS, FreeRTOS.

I. INTRODUCTION

The demands on Internet of Things (IoT) technologies have grown rapidly due to the various application fields and the advancements in wireless communications technologies [1], [2]. The term *things* in the Internet of Things is a piece of equipment having a sensing, actuating, storage, or processing capability. These devices possess unique characteristics, i.e., little memory, reduced battery capacity, and limited processing power [3]. The IoT has great potential to impact our lives in the future. From home automation to healthcare systems, the IoT has numerous applications to improve industries and society by enabling smart communication between objects and devices in a cost-effective manner [4], [5]. Therefore, it is predicted that there will be about 50 billion IoT devices by 2050 [6]. Due to the expansion of IoT networks in the last decade, various hardware platforms have been developed to support IoT sensors and actuators. Similarly, a number of operating systems (OSs) have gradually been developed to run these tiny sensors [7].

Various IoT communications standards have emerged from different organizations. For example, the Internet Engineering Task Force (IETF) [8], the International Telecommunication Union-Telecommunication (ITU-T) [9], the Institute of Electrical and Electronics Engineers (IEEE), the European Telecommunications Standards Institute (ETSI) [10], the International Organization for Standardization (ISO) and the International Electrotechnical Commission (IEC) [11], One Machine-to-Machine (M2M) [12] and the 3rd Generation Partnership Project (3GPP) [13] are actively working to provide efficient IoT communications protocols. The IETF currently has various working groups (WGs) that deal with IoT-related protocols on any layer above the link layer (e.g., at the network layer). The IETF Routing over Lossy and Low-Power Network (ROLL) WG (RFC 6550) [14] is focused on providing standardization of the IPv6 Routing Protocol for Low-Power and Lossy Networks (RPL). Similarly, the IETF IPv6 over Low-Power Wireless Personal Area Networks (6LoWPAN) (RFC 4944) [15] works on IPv6 networking protocol optimization using IEEE 802.15.4. The IETF 6loBAC WG (RFC Ed Queue) [16] provides specifications for transmission of IPv6 packets on master-slave/token-passing (MS/TP) networks. The IETF 6TiSCH Operation Sublayer (6TOP) WG (RFC Ed Queue) [17] defines the mode of operation

for IPv6 using an IEEE 802.15.4e (6TiSCH) network. Datagram Transport Layer Security (DTLS) in a Constrained Environment (DICE) was drafted by the IETF WG DICE (RFC 7925) [18]. At the application layer, the IETF Constrained Application Protocol (CoAP, RFC 7252) provides web services to constrained devices [19]. Concise Binary Object Representation (CBOR, RFC 7049) provides binary representation of structured data [20]. The Object Signing and Encryption (COSE) WG (RFC Ed Queue) focuses on creating CBOR-based signing and encryption formats [20]. Application layer security for data exchange with CoAP using the COSE format is provided by IETF's Object Security of CoAP (OSCoAP, RFC 7744) [21].

Similarly, ITU-T provided an overview of the IoT and its reference model [22]. ITU-T Task Group 15 (TG-15) is working on smart grid communications aspects of the IoT [23]. Similarly, ITU-T TG 17 is focusing on security and identity management aspects of the IoT [24]. An ISO/IEC joint technical committee does not develop standards but it provides current and future IoT trends and requirements [25]. The IEEE defines an architectural framework for the IoT [26]. The IEEE P2413 working group provided a description of "various IoT domains, definitions of IoT domain abstractions, and identification of commonalities between different IoT domains" [27]. The IoT IEEE 802.15 working group is dealing with medium access control (MAC) and physical layer specifications for wireless personal area networks (WPANS), a mesh topology capability in WPANs, and short-range wireless optical communications using visible light. ETSI has developed a low-throughput network (LTN) as a wide area network (WAN) for the IoT [28]. One M2M is a standardization body that consists of eight world standard development organizations. Their goal is to develop a common standard for M2M communications. 3GPP is also working to meet IoT requirements [29]. LTE Release 12 [30] from 3GPP provides a power-saving mode and a lower overhead signaling procedure to provide energy efficiency [31]. An IoT OS should be flexible enough to support these protocols without violating the needs of resource-constrained tiny devices.

IoT devices have limited memory and power and require real-time capabilities in some scenarios. Additionally, they should support heterogeneous hardware along with efficient connectivity and security mechanisms [32]. Connecting and operating this huge number of devices in an efficient way is one of the most important design goals for the research community. In an IoT system, the fundamental research issue is to manage the available resources in an ordered and controlled manner. The ultimate objective of an IoT resource management mechanism is to satisfy IoT device requirements efficiently [33].

IoT devices are classified into two general categories; i.e., high-end IoT devices and low-end IoT devices [7]. High-end devices contain more processing power and energy, such as smartphones and Raspberry Pi. The low-end devices, on the other hand, are too resource-constrained. Therefore, a traditional OS, such as Linux, cannot run these small resource-constrained devices. Hence, the IoT cannot achieve its full potential until there is a de facto standard OS providing support to run these low-end devices across a heterogeneous network [6]. Moore's law [34] is not applicable to IoT devices in terms of processing power. However, it can be applied to device size and energy efficiency [35]. The low-end devices possess very little random access memory (RAM) and few processing capabilities. The IETF [8] developed the IPv6 over Low-power Wireless Personal Area Networks (6LoWPAN) standard adaptation layer to enable low-power low-data-rate communications [36]. These devices also require real-time capabilities in scenarios like vehicular communications, health care systems, and factory automation and surveillance applications. Providing communications with energy efficiency and reliability is the main objective of the IoT. IoT low-end devices contain small amounts of memory and little processing power. Therefore, in order to satisfy low-end device resources, choosing a suitable lightweight OS is of vital importance. Several OSs have been proposed by numerous companies that offer a different approach to fundamental problems.

The future IoT environment needs to handle and perform tasks independently. Similarly, an ultra-dense network yields computational complexity. In order to cope with IoT low-end–device challenges, such as limited resources, and distributed and dense environments, there is considerable need for an efficient resource management mechanism in an IoT OS [37]. An IoT OS is primarily responsible for managing the device's resources efficiently. Various OSs have presented different solutions to satisfy low-end devices' resource needs. To achieve this goal, various mechanisms are provided by the different OSs to provide proper functioning of sensing nodes. Among various proposed OSs for low-end devices, the Contiki [38], TinyOS [39], and FreeRTOS [40] are most prominent for operating in a resource-starved network. IoT low-end devices usually operate with limited battery power. Consequently, providing an energy-efficient OS is of the utmost importance [41]. These low-end devices transfer the sensed data using a communications protocol. In order to be energy-efficient, the communications protocols should save the maximum amount of energy. Protocols at the transport layer, MAC layer, and network layer need to be energy-efficient [42]–[44].

IoT devices require computational capabilities for their sensing operations. These constrained sensing motes do not offer extensive memory and processing capabilities (usually 100 kB flash memory and 10 kB RAM). For example, Crossbow's Telos B mote provides only 10 kB of RAM and 48 kB of flash memory. Due to this limitation, IoT devices need to manage their resources efficiently. Additionally, densification, randomness, and uncertainty make IoT device resource management a challenging task. An OS acts as a resource manager for this complex IoT system [32]. To handle the limited processing power and memory, an OS requires an effective process and memory management mechanism. IoT

devices are battery-operated and are mostly deployed in remote environments. Thus, energy management provided by an OS is highly important. The main objective of an IoT system is to provide a sensing operation and transfer the sensed data to the base station for further processing. The communications design, signal processing, data reception, data transmission, and radio sleep/wake mechanism need to be efficient in terms of energy and communications. IoT OSs store, catalog, and retrieve the data using a file system. Therefore, the provision of an efficient, robust, and appropriate file system is highly desirable in IoT OSs.

Moreover, an IoT OS should be highly concurrent to support these low-end–device sensing operations. Hence, the importance of efficient resource allocation in the OS for low-end devices motivates us to write this paper. In this paper, we consider the IoT low-end device resource management solutions offered by various OSs. To the best of our knowledge, this is the first paper that encompasses detailed information about the resource management mechanisms in Contiki, TinyOS, and FreeRTOS. Various resource management operations, including process management, memory management, energy management, communications management, and file management (and their advantages) are discussed in order to make the low-end devices more and more resource-efficient and flexible. Thus, this study has taken all the resource management mechanisms into consideration. A list of abbreviations is provided in Table 1, whereas Table 2 provides a comparison of this study and already existing surveys on tiny sensor device OSs.

The contributions of this paper compared to the recent literature in the field are as follows.

a. It provides a literature review related to IoT OSs.
b. It covers the resource management aspects of Contiki, TinyOS, and FreeRTOS, including:
   - process management
   - memory management
   - energy management
   - communication management, and
   - file management
c. It provides future research directions and challenges in resource management of IoT OSs.

The remainder of this paper is structured as follows. Section II provides an overview of related work. Section III discusses resource management classifications in detail. Section IV provides open research issues and recommendations, followed by Section V, which concludes the paper.

II. RELATED WORK

Over the years, several OSs for the IoT have emerged. Contiki, TinyOS, and FreeRTOS emerged as predominant OSs to provide support to IoT devices. This section discusses the recent survey papers related to IoT OSs, e.g., Hahm et al. provided a detailed analysis of various requirements to satisfy low-end IoT devices [7]. The survey discussed various OSs that could become the de facto standard. OSs in this survey are categorized into three types, including event-driven OSs, multithreading OSs, and pure real-time operating systems (RTOSs). Along with the key design choices, the characteristics of each category are presented

TABLE 1. List of abbreviations.

Symbols	Description
3GPP	3rd Generation Partnership Project
6LoWPAN	Low-power Wireless Personal Area Networks
6TOP	6TiSCH Operation Sublayer
AEON	Accurate Prediction of Power Consumption
BVR	Beacon Vector Routing
CBOR	Concise Binary Object Representation
CCA	Clear Channel Assignment
CFS	Coffee File System
CH	Cluster Head
CoAP	Constrained Application Protocol
COSE	Object Signing and Encryption
DICE	DTLS in Constrained Environment
DIO	DODAG Information Message
DODAG	Destination-oriented Directed Acyclic Graph
DSVR	Destination Sequence Vector Routing
DTLS	Datagram Transport Layer Security
DYMO	Dynamic MANET on Demand
EATT	Energy-aware Target Tracking
ELF	Efficient Log-structured Flash
ERTP	Energy-efficient and Reliable Transport Protocol
ETSI	European Telecommunications Standards Institute
HAL	Hardware Abstraction Layer
HDRTP	Hybrid and Dynamic Reliable Transport Protocol
ICMP	Internet Control Message Protocol
IETF	Internet Engineering Task Force
IGMP	Internet Group Management Protocol
ISO/IEC	International Organization for Standardization and International Electrotechnical Commission
ITU-T	International Telecommunication Union-Telecommunication
LAD	Location Aided Routing
LEACH	Low Energy Adaptation Clustering Hierarchy
LPL	Low Power Listening
LTN	Low Throughput Network
MS/TP	Master-Slave/Token-Passing
OF	Objective Function
OSCoAP	Object Security of CoAP
POSIX	Portable Operating System Interface for Unix
RCRT	Rate Controlled Reliable Transport Protocol
ROLL	Routing over Lossy and Low-Power Network
RPL	Routing Protocol for Low-Power and Lossy Networks
STCP	Stream Control Transmission Protocol
STI	Software Thread Integration
TORP	TinyOS Opportunistic Routing Protocol
TOS-PRO	TinyOS Preemptive Original
TOSSTI	TinyOS Software Thread Integration
TOSThread	TinyOS Thread
UTOS	Untrusted Extension for TinyOS

TABLE 2. Overview of comparison between this study and available surveys.

Approaches	Key Concepts	This Study			Hahm et al. [7]			Amjad et al. [47]			Strazdins et al. [50]			Reusing et al. [51]			Farooq et al. [53]		
		C	T	F	C	T	F	C	T	F	C	T	F	C	T	F	C	T	F
Process Management	Programming Model	✓	✓	✓	✓	✓	✓		✓			✓		✓	✓		✓	✓	
	Scheduling Model	✓	✓	✓	✓	✓	✓		✓					✓	✓		✓	✓	
Memory Management	Memory Allocation/De-allocation	✓	✓	✓	✓		✓		✓						✓		✓	✓	
	Memory Fragmentation and Safety	✓	✓	✓			✓		✓								✓	✓	
Energy Management	Software-level Energy Management	✓	✓	✓					✓		✓	✓			✓				
	Energy Tracking		✓	✓					✓						✓				
Communication Management	Supported MAC Layer Protocols	✓	✓	✓	✓		✓		✓										
	Supported Network Layer Protocols	✓	✓	✓	✓		✓		✓								✓	✓	
	Supported Transport Layer Protocols	✓	✓	✓					✓								✓	✓	
File Management	Storage Abstraction	✓	✓	✓													✓	✓	
	File Storage and Organization	✓	✓	✓													✓	✓	

C: Contiki, T: TinyOS, F: FreeRTOS

in the study. Based on the key design choices and low-end device requirements, the most prominent OS representing each category is identified.

Similarly, Amjad et al. discussed several aspects of TinyOS design in detail [45]. This survey encompassed the design paradigm and main features of TinyOS. It has event-driven concurrency, a programming layout based on NesC (a dialect of the C programming language), a monolithic architecture, and a non-preemptive task scheduler. TinyOS memory management, energy management, and energy-efficient communications protocols were presented. TinyOS uses a software thread integration (TOSSTI) mechanism for energy conservation, which helps an OS utilize busy–wait time in an efficient manner [46]. Similarly, an energy tracking mechanism is also utilized by TinyOS [47]. To maintain network stability and lifetime, TinyOS supports several communications protocols at the transport layer, MAC layer, and network layer. In addition, simulators for TinyOS and its various sensing applications are also discussed in the paper.

Strazdins et al. surveyed wireless sensor network (WSN) deployments and analyzed the collected data to study the design rules for a WSN OS using 40 deployment scenarios [48]. Deployments from 2002 to 2011 are reviewed to study different WSN applications, including environmental monitoring, animal monitoring, human-centric applications, infrastructure monitoring, smart buildings, and military applications. The authors studied Contiki, TinyOS, LiteOS, and MansOS, and proposed 25 design rules. The rules include suggestions related to the task scheduler, networking protocol, and energy-efficiency mechanism.

TinyOS and Contiki are the two best-known OSs for low-end devices. A comparison between these two OSs is presented in a survey by Reusing [49]. The main requirements an OS should fulfill for a sensor network include concurrency, flexibility, and energy efficiency [50]. The contrast between TinyOS and Contiki is shown based on these requirements. Special emphasis was placed on a programming model and execution model, along with the hardware platforms supported by both OSs. This survey indicates that TinyOS might be more useful in a resource-constrained environment, whereas Contiki provides more flexibility in the network.

Farooq and Kunz highlighted major challenges for an OS design, and identified the advantages and limitations in an OS for a WSN [51]. For example, Contiki follows a modular kernel concept. It is a layered approach in which application modules are independent and can be linked with a kernel at boot time. In this way, the kernel provides only core services, while other services can be added when required. Hence, it reduces the memory footprint and decreases boot time. However, the kernel may crash due to modules that contain bugs. Similarly, TinyOS follows a monolithic architecture similar to Linux. The monolithic architecture helps to reduce modular interaction costs. However, it may make the OS unreliable and hard to maintain, because no clear boundaries are provided between modules. The alternative is a microkernel architecture.

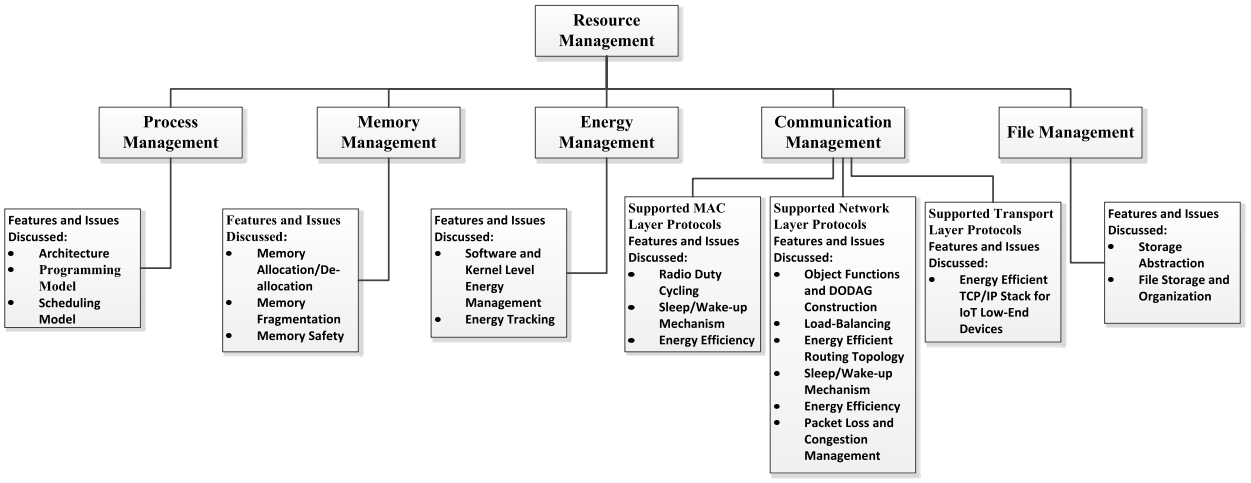

FIGURE 1. Resources management classification.

FreeRTOS is an example of a microkernel structure which provides robustness against bugs in the components. The microkernel provides minimum functionalities to the kernel. Hence, the kernel size is reduced significantly. All other functions are provided by servers running at the user level. Therefore, OS functionalities are extendable, and failure in one user-level server does not crash the kernel itself. The disadvantage is poor performance due to user-to-kernel boundary crossing. The OS architecture, programming model, scheduling mechanism, memory management, and communications protocol support a major design goal for an IoT OS. Along with resource sharing, support for real-time applications is of vital importance. It is pointed out that some OSs support a priority application capability, whereas few provide real-time applications. Some miscellaneous features, including communication security, file system support, simulation support, and programming language, are also discussed.

Dariz et al. compared Contiki and TinyOS with a real-time OS named ChibiOS to study the safety-relevant application in a WSN [52]. Another OS called LiveOS was introduced to conserve memory and energy for a WSN [53]. Over the years, various OSs have emerged in the WSN community. One of the critical issues for an OS is dealing with a large number of resources to provide ubiquitous services to IoT low-end devices [54]. In this study, we focus on three well-known OSs: Contiki, TinyOS, and FreeRTOS. This study aims to cover all the resource management aspects of a low-end–device OS.

III. RESOURCE MANAGEMENT CLASSIFICATION

OS provides a layer of abstraction for the hardware by managing the resources on each IoT device [55]. The OS provides a programming interface and manages processor time. IoT devices operate in resource-constrained concurrent environments, and to handle this concurrent application, a suitable execution model must be provided by OS. The execution model must provide memory efficiency [56]. Similarly, OS (being battery-powered) must provide a sleep mode when no application is running [57]. Providing energy efficiency to the communication components is more challenging for an OS. The communication components must wake up during a communication period. Therefore, an OS handles energy efficiency during communication using various mechanisms, for example, a separate radio duty cycling procedure [58], a virtual carrier-sensing mechanism with a network allocation vector, and time-division multiple access (TDMA)-based methods. Not all IoT devices have storage like flash memory. Therefore, an appropriate file system is required to provide storage needs for some applications. The file system needs to efficiently map the data into sectors to make writing and reading of data more efficient. Therefore, an OS must provide a full file system interface [59], [60].

The communications needs of diverse applications are handled by a communications architecture. Considering the device's resource scarcity, the communications protocols must be energy- and memory-efficient during data collection, event detection or tracking, device synchronization, neighbor discovery, and data delivery [61], [62]. To address the resource management challenges for low-end IoT devices, various resource management mechanisms and schemes have been proposed. These resource management schemes fall into five subsections.

The flow chart of resource management in OS for low-end devices is shown in Figure 1.

A. PROCESS MANAGEMENT

In the context of resource management, the kernel manages processes and threads to share information, protect process resources, and assign system resources in a safe way. In the IoT environment, multiple activities may occur during a certain time period. Managing these activities and processes by fairly sharing resources is essential, and it depends on the OS execution model.

The Contiki and TinyOS follow an event-driven execution model to provide memory efficiency and low complexity of

state machines in the event-driven TCP/IP stack [63], [64]. Event handlers continuously wait for internal or external events, such as an interrupt. The kernel allocates the memory stack to the process, and an event handler follows the run-to-completion mechanism. All the processes effectively share the same stack and utilize limited memory efficiently. Some events are queued and processed in first in, first out (FIFO) fashion. The event-driven concurrency model introduces certain complexities if multiple events occur. The task in an event-driven model cannot be blocked during run-time. Sometimes, time-critical tasks need to be executed first. Therefore, real-time performance of the event-driven approach is poor. Hence, an OS needs multiple event handlers.

Low-end IoT devices offer only few kilobytes of RAM. The multithreading approach allocates a stack of memory to each thread even if the thread is not utilizing memory. Hence, most of this memory is unused. Therefore, a more effective hybrid model is required for better memory efficiency and low programming complexity. The Contiki supports a novel, lightweight, stackless threading mechanism called a protothread [65]. The protothread utilizes a multithreaded model without increasing multiple-stack overhead. In an event-driven approach, the program runs to completion, which is not desirable in some scenarios, especially in a system where a high-priority task is present. A protothread simplifies the event-driven programming model by providing a conditional locking wait statement that enables a program to execute a blocking wait without introducing an additional stack for each protothread. Between the beginning and end of each protothread, there is a conditional wait statement. This conditional wait statement blocks the program if there is an interruption. In other words, the thread is blocked only if an explicit blocking wait statement is used. In this way, the number of explicit state machines in the event-driven approach is reduced, with memory overhead of only two bytes per protothread. The protothread is a better alternative for memory efficiency. However, providing process synchronization between protothreads is not possible.

Sometimes blocking certain components may interrupt the whole sensing application. TinyOS Thread (TOSThread) is a complete implementation of a preemptive application-level thread library to achieve maximum concurrency without increasing resource usage [66]. TOSThreads categorize all event-based code into kernel-level threads and application-level threads. Kernel-level threads are given the highest priority, and cannot be interrupted by application-level threads. An application-level thread makes a system call application programming interface (API) that does not interrupt the TinyOS code itself; rather, it sends a message to the kernel thread. Application-level threads execute only if kernel-level threads are not active. The basic architecture of a TOSThread is shown in Figure 2. The overall structure consists of five elements: a single kernel-level thread, a number of application-level threads, a task scheduler, a thread scheduler, and system-call APIs. A number of application threads run

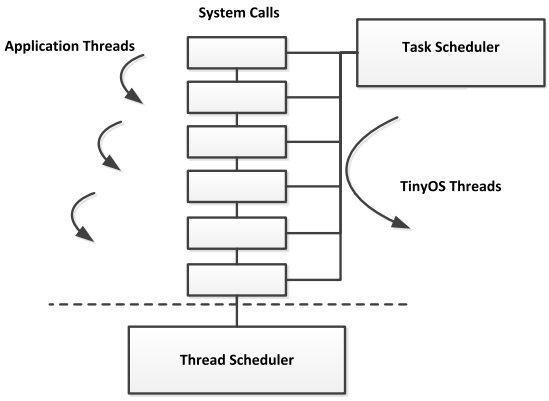

FIGURE 2. TOSThreads architecture (adapted from [66]).

concurrently and make a call to the kernel-level threads through API slots. The thread scheduler provides concurrency between application-level threads and system-call APIs. TOSThread provides a preemptive behavior to TinyOS but increases the computational complexity. To provide preemptive execution in a simple manner, the TinyOS preemptive original (TOS-PRO) approach was introduced [67]. This approach provides increased flexibility for scheduling without introducing extra complexity into TinyOS.

FreeRTOS is based on a microkernel architecture and utilizes a multithreading approach [68]. Each process can be interrupted, and the scheduler can switch between threads [69]. It provides a real-time, preemptive multitasking environment for low-end devices. It ensures execution of a higher priority task in any given time period. If two tasks are given equal priority, the scheduler divides execution time between them. This execution follows a priority-based round-robin implementation. The FreeRTOS kernel is structured using four C files (*task.c, list.c queue.c and croutine.c*), where *task.c* provides scheduling functionalities by using structures and functions in the *list.c* file. The *queue.c* file provides a thread-safe queue to implement inter-task communication and synchronization, and *croutine.c* implements simple lightweight tasks [70]. The IoT OS process management overview is given in Table 3.

B. MEMORY MANAGEMENT

Memory management provides techniques for allocating and deallocating memory for various processes and threads. OS offers two common methods for memory allocation, i.e., static allocation and dynamic allocation. In static memory management, OS allocates memory to the system that cannot be altered during run-time. But a dynamic management technique provides flexibility in memory acquisition at run-time. Static allocation cannot predict how much memory will be needed, especially in real-time scenarios. Similarly, memory over-provisioning may result in memory overhead. With dynamic allocation, if the allocated memory is not freed, it may result in a memory leak.

TABLE 3. An overview of process management.

Relevant References	OS	Architecture	Category	Scheduling	Programming Language	Key Concept	Advantages	Limitations
[65], [67],	Contiki	Monolithic	Event-Driven	Cooperative	C	Protothread: Provides conditional locking wait statement and utilizes multithreaded model without increasing multiple stack overhead.	Module interaction cost is low.	Need proper process synchronization.
[66], [68], [69]	TinyOS	Monolithic	Event-Driven	Cooperative	NesC	TOSThreads: Categorizes all event-based code into synchronous (tasks) and asynchronous (interrupts) execution context.	Provides maximum concurrency without increasing resources usage.	Cannot provide proper execution of abnormal tasks.
[70], [71], [72]	FreeRTOS	Microkernel	Real-time	Preemptive	C	Uses microkernel to provide OS reliability.	Time tracking mechanism to disable a periodic tick is beneficial.	Cannot handle large IoT system tasks.

The memory size for sensor devices is constrained due to the device's physical size and the cost. Static memory contains the program code, and dynamic memory contains runtime variables, the buffer, and the stack. IoT low-end devices as classified by IETF require about 10 kB of RAM and about 100 kB of flash memory. The Contiki C library provides a set of functions for allocation and de-allocation of memory for the heap. For example, the memb macro(), and memb_alloc(), and memb_free() functions are used for memory declaration, allocation, and de-allocation, respectively [71]. The memory allocation function needs to handle memory fragmentation. If memory is fragmented, allocation may fail to allocate all the unused memory. The managed memory allocator function mmem() in the Contiki frees the allocated memory from fragmentation by compacting it when blocks are unused. However, dynamic allocation may lead to stack overflow, and requires more space. TinyOS is based on the NesC programming language [72]. To cope with sensor node hardware constraints, the language does not support dynamic memory allocation, the program states and memory are declared at compile time. In this way, memory fragmentation and runtime allocation failure are prevented. Similarly, maintaining an additional data stack to manage the dynamic heap is not required [73]. In the earlier version of TinyOS, the basic building block (i.e., memory safety) was not available [74]. However, new updates and revisions provided memory safety and memory safety–check features. Safe TinyOS was developed mainly to provide memory safety to sensor nodes [75]. Similarly, Untrusted Extension for TinyOS (UTOS) utilizes a sandboxing concept to provide enhanced memory safety features, compared to Safe TinyOS [76]. To provide memory safety features to memory-constrained devices, CCured is leveraged [77]. CCured provides a red line that draws a boundary between trusted and untrusted extensions. The untrusted extensions cannot access the hardware and network resources directly. An extension communicates with the rest of the system through a proper UTOS system call interface. The extension is terminated if it violates the safety model of the system. The CCured compiler inserts dynamic safety checks before every operation.

Restarting an extension is still faster than rebooting a TinyOS application. To make the memory more efficient, unstacked C is used, which is a source-to-source transformation to translate a TinyOS multithread program into stackless threads. Since these programs do not have a separate stack, their memory overhead is reduced significantly. Dynamic memory-like capabilities can be offered in TinyOS by using a component named TinyAlloc through an interface called MemAlloc. Additional memory management and capacity are provided through a TinyPaging mechanism, which makes use of flash storage [75]. TinyAlloc allows double referencing, which means that the memory region is referenced indirectly through another array that contains it current address. Hence, TinyAlloc can alter the memory address in the

TABLE 4. An overview of memory management.

Relevant References	OS	Type	Key Concept	Advantages	Limitations
[39], [51], [73]	Contiki	Dynamic	MEMB() macro, memb_alloc(), memb_free() functions are used to declare, allocate and deallocate memory. mmem() function frees the memory from defragmentation.	Offers memory size, dynamically adjusting capabilities for changing requirements during run-time.	Does not provide Memory Protection Unit (MPU).
[74], [75], [76], [77], [78], [79]	TinyOS	Static/Dynamic	Program transformation system (CCured) is used to provide memory safety. Unstacked C translates multithread into stackless threads. TinyAlloc component is used to provide dynamic allocation capability. TinyPaging mechanism is used to provide additional space.	Static allocation prevents memory fragmentation and run-time allocation failures.	Does not provide memory usage prediction. Memory may get wasted if program is unused.
[80], [81]	FreeRTOS	Dynamic	pvPortMalloc() and vPortFree() funtions are used to provide three heap implementations to allocate and de-allocate. Heap_1: Does not de-allocate the allocated memory. Heap_2: Frees the allocated memory. Heap_3: Allocates and de-allocates the memory similar to Contiki mechanism.	Offers several heap management schemes, depending on the application requirements.	Memory is not safe thread nor deterministic.

intermediate array, and move the memory region freely with in the heap. The MemAlloc interface in TinyAlloc returns a pointer handle to the newly assigned memory region, and also frees the memory region and returns the handle pointer to allocated memory. Tinypaging uses virtual addresses. The memory region is allocated a virtual address. Before using it, a dereferencing function takes the virtual address and returns the physical address for that memory. It also reduces the need to use an additional intermediate array. Hence, Tinypaging combines these concepts and works with virtual addresses to exchange parts of memory into flash.

The additional threads in TinyOS that provide more execution and concurrency support may require more memory usage. Therefore, memory usage prediction is required for TinyOS applications. With a real-time operating system (FreeRTOS), the kernel allocates memory dynamically for every event. The malloc() and free() functions are not desirable in a real-time operating system due to the fact that dynamic memory allocation has typically deterministic run-times, needs extra code space, and suffers from memory fragmentation. To eliminate these problems, FreeRTOS introduced two new functions: pvPortMalloc() and vPortFree() [78]. These functions provide three heap implementations for memory allocation, depending on the system design [79]. Heap_1 does not allow de-allocation of memory once it is allocated. It is suitable for a system where allocated memory size always remains the same (for example, with application tasks that do not vary with time and that are created before the kernel is started). Heap_2, in contrast to heap_1, allows previously allocated memory to be freed. It does not combine adjacent free blocks into a larger memory block. This scheme is suitable for systems where tasks are created dynamically. Heap_3 is similar to the malloc() and free() function allocations, and make a safe thread. This scheme is not memory-efficient, and may increase the kernel code size. The memory management aspect of IoT OS is summarized in Table 4.

C. ENERGY MANAGEMENT

IoT devices consume energy during sensing, data processing, and data transfer. The management of limited energy has been a key issue for these devices due to the fact that these sensors are deployed mostly in remote environments and function without human intervention. Therefore, OS should provide an energy-efficient mechanism to prolong the life of an IoT network [80]. The management of a limited energy budget is rudimentary, and can be accomplished through both hardware and software techniques [81]. Hardware-based approaches require additional hardware, which increases system cost. Software-based techniques are more practical, but may introduce additional overhead. Energy efficiency can be achieved through network protocol design and OS scheduling aspects, e.g., sleep/wake and duty-cycle modes are employed in most OSs to conserve energy [62]. Reducing energy consumption through a software mechanism requires a comprehensive view of the application at a different layer of the system, and is an essential condition for OS.

The Contiki kernel offers no explicit power-saving mechanism. The applications provide a power-saving mode by

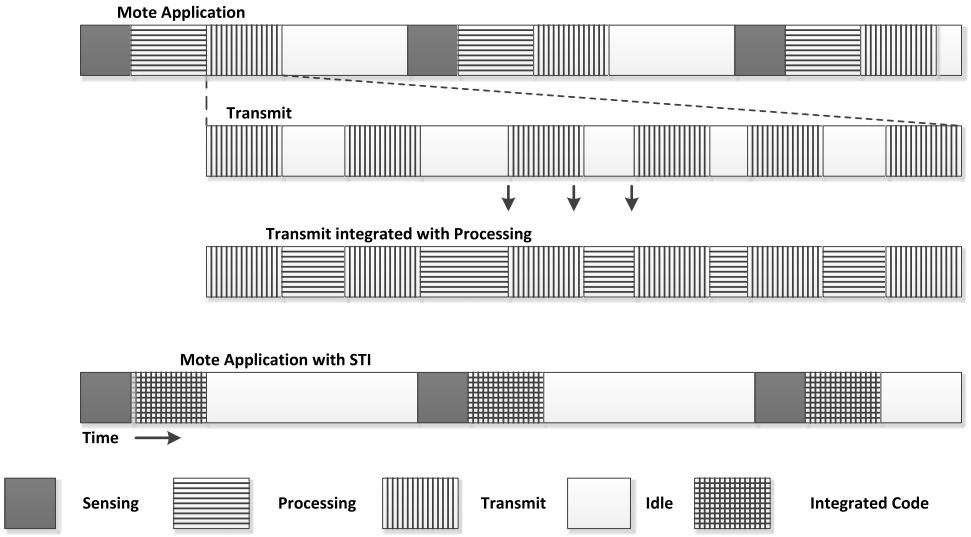

FIGURE 3. TinyOS software thread integration (TOSSTI) (adapted from [46]).

TABLE 5. Predicted energy consumption (in mJ) and node lifetime for TinyOS 1.1.7 component (adapted from [82]).

Test Application	CPU				LEDs	Sensor Board	Total Energy	Lifetime
	Active	Idle	Rx	Tx				
Blink	0.37	601.6	0	0	196.2	-	798.2	25.8
CntToLeds	0.77	601.5	0	0	590.6	-	1193	17.4
CntToLedsAndRfm	93	560.7	1651	130	589.6	-	3025	6.9
CntToRfm	92.7	560.8	1651	130	0	-	2435	8.5
RfmToLeds	82.9	565.2	1727	0.6	589.0	-	2965	7.0
SenseToLeds	1.85	601	0	0	0[1]	126	728.8	28.5
SenseToRfm	4.39	560.3	1651	130	0	126	1937	10.7

utilizing an event queue size. The application can put the CPU into sleep mode when the event queue is empty [63]. The Contiki network-level energy-saving mechanisms are discussed in more detail in Section D.

TinyOS utilizes software thread integration (STI) for energy conservation [46]. The node faces idle–busy time during sensing, processing, and transmission. The idle time is too short to perform traditional context switching. With STI, the processor can reclaim this time to perform other useful tasks, as depicted in Figure 3. Similarly, the processor can boost battery life by switching to low-power mode sooner. The TinyOS overall system response time also improves, which supports higher priority task processing. In this way, the scheduler can provide the effects of pre-emption at the task level. Hence, it enhances the concurrency model of the scheduler.

Allocating the energy dynamically by predicting the power consumption of nodes can be helpful to conserve energy. For example, accurate prediction of power consumption (AEON) is an energy prediction tool for sensor nodes [82]. TinyOS application energy prediction based on AEON is shown in Table 5. Table 5 shows the amount of energy consumed by each component. For example, the radio consumes most of the energy, thus, the CPU idle–active mode duration can be altered to extend node lifetime. Similarly, the TinyOS programming mode supports an energy-tracking mechanism to track energy consumption of various components. An energy-aware target tracking (EATT) algorithm is implemented in TinyOS using a clustering and data aggregation technique [83]. The tracking algorithm is executed by the cluster head (CH) that performs data collection, aggregation tracking, and result propagation to send the results to the desired location. Through energy tracking optimization, the number of CPU cycles can be minimized. However, this mechanism is not suitable for mobile devices, and may introduce additional memory usage. A distributed energy-aware wake-up counter was tested in TinyOS to provide updated link status in real time [84].

In an event-driven system, the threads of executions or tasks spend a portion of their time waiting for an interrupt, or for a time period to expire. In FreeRTOS, these tasks are referred to as being in a blocked state [85]. If all the tasks are in a blocked state, FreeRTOS creates and runs a task called idle task. Therefore, when the processor is idle, it can go into power-saving mode. This is implemented in FreeRTOS using an idle task hook function [86]. The idle task is given the lowest priority, and the idle hook function gets called only if there is no higher priority task available [87].

TABLE 6. An overview of energy management.

Relevant References	OS	Key Concept	Advantages	Limitations
[65]	Contiki	Application-specific energy conservation implementation.	The applications provide a sleep mode by observing event-queue size.	No specific kernel-level power-saving mechanism is provided.
[48], [84], [85], [86]	TinyOS	Software Thread Integration (STI). Energy-Aware Target Tracking (EATT).	With STI, the processor can boost battery life. The number of CPU cycles can be minimized.	This mechanism is not suitable for mobile devices and may introduce additional memory usage.
[88], [89], [90]	FreeRTOS	Tickless Idle.	It disables a periodic tick source for a period of time to put the processor into deep sleep mode for more energy savings.	Introduces run-time overhead.

Hence, this function provides an automatic power-saving mechanism to the FreeRTOS processor. This mechanism may be beneficial in some scenarios, but if the frequency of the ticks is too high, the processor will waste energy and time in entering and exiting idle mode. Hence, the power savings through this mechanism are not beneficial. Therefore, to provide an appropriate power-saving mechanism, a tickless idle technique was introduced [88]. Tickless idle is a power management technique for FreeRTOS that provides more power saving during processor idle states. It uses a time-tracking mechanism to disable a periodic tick source for a period of time to put the processor into deep sleep mode until a higher priority external or kernel interrupt occurs. However, it introduces run-time overhead. The energy management aspects of IoT OSs is presented in Table 6.

D. COMMUNICATION MANAGEMENT

Providing seamless continuous and ubiquitous communication between IoT devices is the ultimate goal of an IoT OS. IoT networking is complicated by the devices' wireless nature, heterogeneity, density, and diverse transmission patterns [89]. Therefore, communications support at the MAC layer, the transport layer, and the network layer impacts overall IoT network performance. There is a plethora of IoT communication protocols available in the literature [90]. Some of these protocols are widely accepted and standardized. The IoT communications protocols should focus on energy efficiency rather than providing higher throughput. The networking stack for an IoT OS must support higher-level services, including data dissemination and accumulation. It also requires managing low-level services, including radio management, queue management, and MAC support [91]. Apart from these requirements, there is a need to consider the devices' unique traffic characteristics, and consequently, a need to manage the quality of service (QoS). For example, in the smart metering scenario, devices periodically transmit a small burst of data. A detailed tabular overview of communication management section is provided in Table 7.

1) CONTIKI SUPPORT FOR COMMUNICATION PROTOCOLS

The Contiki provides two networking stacks, i.e., a uIPv6 net-stack and a Rime communications stack [92]. uIPv6 is the implementation of the TCP/IP protocol stack for eight-bit microcontrollers, and can be configured with 6LowPAN, RPL routing for low-power and lossy networks, User Datagram Protocol (UDP) and Constrained Application Protocol (CoAP) [93]. Similarly, the Rime communications stack is designed for low-power radio. It supports single-hop unicast, single-hop broadcast, and multi-hop communications. In multi-hop scenarios, Rime allows applications to implement routing protocols other than the Rime stack–implemented protocols. The Contiki network stack layer model is shown in Figure 4. The Contiki network stack layer is a little bit different than the traditional OSI layer. It covers all the OSI layers; however, there is a radio layer, a radio duty cycle layer, and a MAC layer present in between the network layer and the physical layer [94].

a: CONTIKI SUPPORT FOR MAC LAYER PROTOCOLS

IoT resource management under a MAC protocol is usually achieved in terms of energy efficiency [42]. The MAC protocol approach developed for a duty-cycle IoT aims to reduce radio idle listening duration to minimize energy consumption. Idle listening is the time the node spends listening to the medium, even if no packet is present. The X-MAC protocol is implemented in the Contiki, and it provides a low-power listening mechanism [95]. If a node sends data, it transmits a preamble. The receiver wakes up, detects the preamble, and stays in the idle state to receive the data. In this basic approach, the receiver stays in the wake-up state until the preamble is finished, and it then starts the data- and acknowledge (ACK)-packet exchanges (Figure 5). The receiver may have woken up at the start of the preamble. This results in wasted energy. X-MAC replaces the low preamble with short strobe frames [96]. The receiver receives one strobe and transmits a strobe-ACK. The sender then proceeds with data transmission. Hence, a short preamble further decreases the time and energy consumption. However, X-MAC wakes up each node for a short active period in this procedure. The node goes to sleep mode again after an active period, which is 5% to 10% of the wake-up interval.

Contiki 2.4 introduced a carrier sense multiple access (CSMA) MAC protocol that simply detects a collision and

TABLE 7. An overview of communication management.

Relevant References	OS	Communication Layer	Key Concept	Advantages	Limitations
[97], [98], [100], [102], [105], [106], [107], [108], [109], [110], [111], [112], [113]	Contiki	MAC Layer	ContikiMAC	Provides an excellent sleep/wake-up mechanism.	Can face false and unnecessary wake-ups. Phase-lock needs to be improved.
			X-MAC	Provides better retransmissions and archives higher PDR as compared to ContikiMAC.	No proper collision avoidance mechanism is provided, i.e., it requires a CCA mechanism. As a result, the ETX is very high.
			CSMA-MAC	Provides a collision avoidance mechanism.	If collision is detected; it does not pass this information the upper layer, which may affect the overall routing operation.
		Network Layer	ContikiRPL	IPv6 forwarding table mechanism.	ContikiRPL uses MRHOF as an OF, which is not suitable in all application scenarios. The RPL protocol requires a proper routing metric and OF for parent selection. Interoperability is also required. Similarly, mobility is not supported.
			RER_{BDI}	For DODAG construction, it takes both the residual energy ratio (RER) of the nodes and their battery discharge index (BDI).	Have not provided the overhead cost, load balancing, and memory footprint information.
			BRPL	Supports node mobility and varying traffic.	Trickletimer adjustment is an issue, especially in mobile node scenarios.
		Transport Layer	uIP	Suitable for simple TCP and UDP scenarios.	Does not support multi-streaming and multi-homing features.
[60], [114], [115], [116], [117], [118], [119], [120] [121], [122], [123], [124], [125], [127], [128], [129], [130], [131], [132]	TinyOS	MAC Layer	TinyLPL	Allows sleep/wake-up implementation with user-defined intervals. Uses short preambles for better energy efficiency.	May suffer from false alarms, including false positives and false negatives in the presence of external interference and the hidden terminal problem.
			MultiMAC	Introduces the concept of a virtual gateway, which allows sensor network interoperability using heterogeneous MAC protocols.	Energy efficiency is more important than interoperability. LPL performs better in terms of energy utilization.
		Network Layer	TinyRPL	Implements an IPv6 stack based on 6LoWPAN specifications.	Similar to ContikiRPL, TinyRPL uses OF0 and MRHOF for parent selection and routing construction, it is not a desired solution, considering diverse IoT applications and network scenarios.
			QU-RPL	Achieves load-balancing in LLN networks. Provides a congestion detection mechanism for parent selection and better PRR.	The DIO overhead cost may result in overall delay in the large IoT network. Similarly, the mentioned parent selection procedure does not take node energy into consideration in switching from one parent to another. In the same way, the Tickletimer resetting strategy depends on network size. Needs a more efficient Tickletimer for better output. The last problem is that it might not be suitable for multimedia applications.
		Transport Layer	HDRTP, STCP, ERTP, RCRT	Unlike Contiki, TinyOS supports a variety of transport layer protocols, including HDRTP, STCP, ERTP and RCRT.	Needs to provide a proper congestion control mechanism. Must implement TFRC and DCCP.
[133], [134], [135], [137]	FreeRTOS	MAC Layer	FreeRTOS MAC	Offers three MAC implementations. CSMA-MAC: Provides a collision avoidance mechanism. TDMA-MAC: Useful to handle a large number of nodes. X-MAC: Provides low-power duty cycling.	CSMA MAC: May affect the routing operation. TDMA MAC: Requires tight time synchronization and is very sensitive to underlying mobility and topology changes. X-MAC: No proper collision avoidance is provided.
		Network Layer	6LoWPAN Nanostack	Provides an ICMP implementation along with NanoMesh, which covers multiple hops.	Does not provide an RPL implementation.
		Transport Layer	FreeRTOS TCP/IP and lwIP stack	FreeRTOS TCP/IP stack is based on uIP which simplifies the TCP and UDP operation for low-end IoT devices. IwIP is based on IPv6 and 6LoWPAN to provide better energy management.	FreeRTOS TCP/IP is still under development, features like multi-streaming and multi-homing are not present yet.

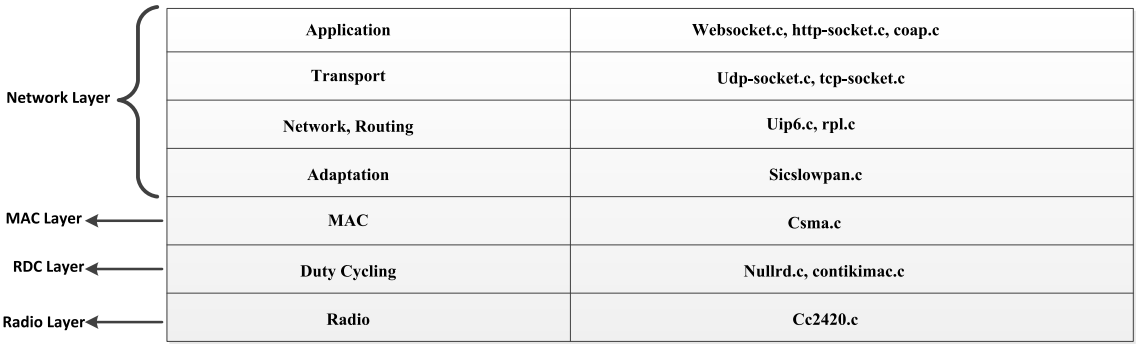

FIGURE 4. Contiki network stack (adapted from [94]).

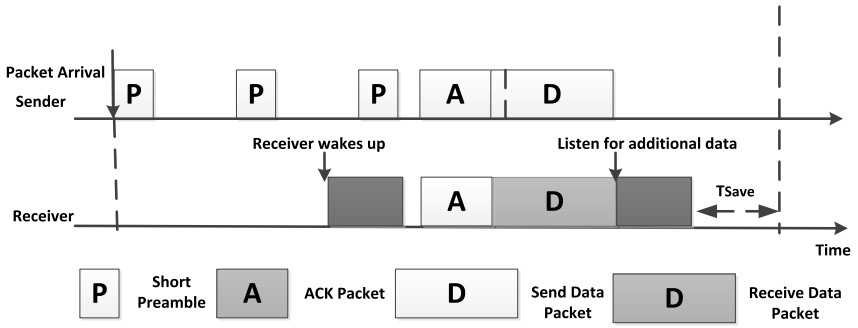

FIGURE 5. X-MAC medium access (adapted from [96]).

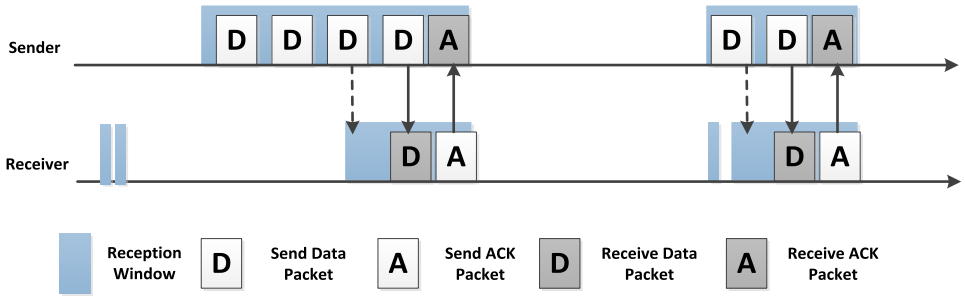

FIGURE 6. ContikiMAC mechanism of sending data packet (adapted from [97]).

retransmits the packets. However, this retransmission information is not passed to the upper layers in order to save computational costs. Hiding this information may affect the overall routing operation. Therefore, a new power-saving mechanism called the ContikiMAC radio duty cycling protocol was introduced in Contiki 2.5 [97]. The ContikiMAC radio duty cycle mechanism was inspired by the X-MAC duty cycling procedure [98].

ContikiMAC periodically wakes up the radio to listen for a packet transmission. The sending node continuously sends the data frame to the receiver until it gets an acknowledgment. The packet's destination field reduces overhearing, i.e., the node can go into sleep mode if it is not the packet destination. The receiver wakes its radio to listen for packet transmission. After detecting the packets, the receiver stays awake to receive the full transmission. Once reception of packets is done, it sends a link layer acknowledgment. This mechanism is illustrated in Figure 6.

The wake-up duration timing needs to be precise. To provide power-efficient wake-up timing, Contiki uses a mechanism called clear channel assignment (CCA), which utilizes the received signal strength indicator (RSSI) value to predict channel availability. An RSSI value lower than a given threshold returns "CCA positive," indicating the channel is free. Similarly, an RSSI value greater than the threshold amount returns "CCA negative," indicating the channel is busy. ContikiMAC follows precise timing constraints. ContikiMAC timing is illustrated in Figure 7; t_i is the time duration between two data packet transmissions, which must be greater than the time required to transmit and receive the ACK, i.e., $t_a + t_d$.

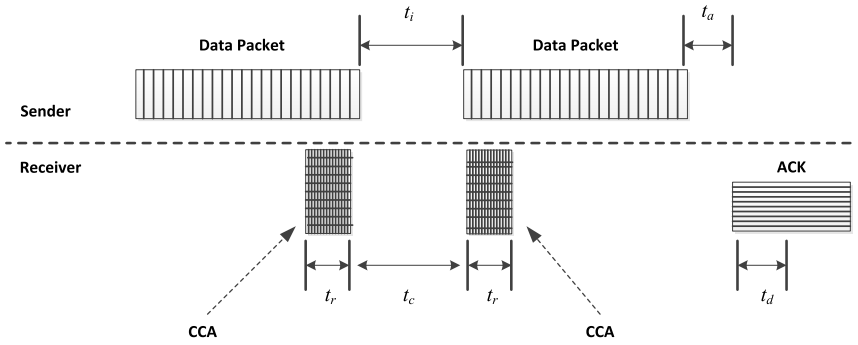

FIGURE 7. The ContikiMAC transmission and CCA timing (adapted from [97]).

The interval t_c between two CCAs (t_r) must be greater than t_i to ensure two CCAs detect a frame. ContikiMAC uses a phaselock mechanism introduced by WiseMAC [99]. In this mechanism, the transmitter can estimate the wake-up schedule of the receiver with the ACK packet and can transmit data frames repeatedly just before the receiver is expected to be in the wake-up state. This phaselock mechanism in ContikiMAC reduces both energy and channel utilization, but at the risk of collision.

Some other MAC layer protocols were designed and tested with the Contiki OS. RAWMAC, a cross-layer approach is implemented in Contiki [100]. It exploits the Contiki RPL [101] protocol at the routing layer, and ContikiMAC at the MAC layer. It uses RPL's directed acyclic graph (DAG) and aligns node wake-up internally estimated by the ContikiMAC phase lock mechanism with its parent node to minimize data collection delay. Another MAC protocol implemented in Contiki is called GinLITE [102].

b: CONTIKI SUPPORT FOR NETWORK LAYER PROTOCOLS

IETF provides IPv6 routing in low-power and lossy networks. RPL specifies how to construct a destination-oriented directed acyclic graph (DODAG). Each node is given a rank based on an objective function (OF). The rank provides the position of the node in the network. The OF calculates the rank of the node using a path calculation in a low-power and lossy network (RFC 6551) [103]. The node joining the RPL network first listens to a DODAG information object (DIO) message. If a node is unable to receive the DIO message, it will broadcast a DODAG information solicitation (DIS) message, which compels the neighboring node to broadcast the DIO message. Using the DIO message, the OF selects the parent node. The packet is forwarded to each parent node until the packet reaches the sink. When traffic is required to flow in the opposite direction, the routing state at every node is built using a DODAG destination advertisement object (DAO) message. The node sends a DAO message to its parent node, which will forward it through the parent's parent node to the sink [104]. Tsiftes et al. proposed a mechanism to implement RPL protocols inside uIPv6 [101]. Similarly, Ko et al. [105] tested the ContikiRPL implementation using two OFs, i.e., OF0 and a minimum rank objective function with hysteresis (MRHOF). ContikiRPL separates the OF into various modules. First, the protocol logic module maintains DODAG information and the node's parent-associated information. Second, the message-construction and parsing module provides the RPL ICMPv6 message format and data structure to ContikiRPL. Third, the OF modules provide an OF API. ContikiRPL provides a forwarding table mechanism for uIPV6 instead of taking a forwarding packet decision per packet. The link cost is estimated by a neighbor information module and is updated to the forwarding table. The uIPv6 layer forwards the outgoing packets to the 6LoWPAN layer, which provides header compression and fragmentation, and then, the packet is forwarded to ContikiMAC.

RPL faces congestion and packet loss problems during heavy traffic. Similarly, RPL has a fixed traffic configuration, it cannot adapt to IoT applications' varying traffic patterns. Mobility is another crucial issue that causes link breakage and invalid routes in DAGs. To address these problems, Tahir et al. proposed an extension of RPL called backpressure RPL [106], which combines the RPL OF with backpressure routing. Congestion issues in the 6LowPAN layer are evaluated in Contiki using a non-cooperative game theory mechanism [107].

Some other routing protocols have been implemented in the Contiki, e.g, the Mesh_under Cluster_based Routing (MUCBR) protocol proposed by Al-Nidawi et al. [108] reduces the node energy consumption and radio duty cycle by implementing a clustering structure under the 802.15.4 standard. Similarly, another mechanism is proposed to provide an improved RPL routing metric [109]. It combines a node residual energy ratio (RER) and a battery discharge index (BDI) along with expected transmission count (ETX) for parent selection and rank computation.

c: CONTIKI SUPPORT FOR TRANSPORT LAYER PROTOCOLS

A traditional TCP/IP cannot be implemented in limited-resources devices; uIP provides the minimum features needed to implement the full TCP/IP stack [110]. It contains simple TCP and UDP transport layer protocols. However, UDP in uIP does not support broadcast or multicast transmission. Similarly, UDP checksums are also not provided in uIP.

2) TINYOS SUPPORT FOR COMMUNICATION PROTOCOLS

The basic communications paradigm of TinyOS is the active message (AM), a single-hop protocol [111]. AM is a simple networking primitive where each message includes an identifier to be invoked on the target node to pass the AM to its handler. In this way, this event-based communication between nodes provides a TinyOS publish/subscribe-based communications architecture. The GenericComm component in TinyOS 1.x provides AM communications interfaces, which provide single-hop unicast, and broadcast communications. TinyOS supports two multi-hop communications protocols called dissemination and TYMO (which is an implementation of the Dynamic Manet on Demand Routing Protocol (DYMO)) [112], [113]. Dissemination protocols are designed to reduce temporary disconnections and packet losses by ensuring the reliable delivery of data to every node in the network.

a: TinyOS SUPPORT FOR MAC LAYER PROTOCOLS

Various MAC protocols have been tested with TinyOS. For example, B-MAC provides a low-power operation interface [114]. It introduced an adaptive preamble sampling mechanism, which reduces radio duty cycling and idle listening. Similarly, X-MAC is a low-power listening approach that uses a short preamble to conserve energy [98]. The receiving node's address information is embedded in the preamble to resolve the overhearing problem, which in turn, saves energy for the non-receiving node. It also uses the idea of a short strobed preamble, which is created by adding pauses in the short preamble. The strobe preamble enables the receiving node to interrupt a preamble as soon as it wakes up and immediately recognizes its own address. Then, it transmits an ACK in the next pause after the preamble. In this way, instead of waiting for the entire preamble to complete, a node can start receiving packets without wasting time and energy. Similarly, a transmitter also does not need to send the remaining short preambles.

Similarly, TinyOS provides a MAC mechanism called TinyOS LPL, which is similar to the ConikiMAC technique [58]. TinyOS LPL allows the radio to implement sleep/wake cycles at user-defined intervals. The LPL receiver energy-saving mechanism saves energy by performing short, periodic receive checks. A node wakes up during every LPL period to sense the channel. If there is activity on the channel, the receiver node will switch its radio on to receive packets and send an ACK. The transmitter stops packet transmission upon receiving an ACK. The transmitter sends a packet only during the receive check interval of the receiver. In a traditional LPL mechanism, the sending node transmits a very long preamble to span a complete receive check period. The receiver node radio stays awake during the full duration of the sending node preamble, and waits for the data packets after that. Staying awake for a long period and reading bits consumes lots of energy. TinyOS LPL improved this mechanism by replacing the long preamble with a smaller packet transmission. Along with that, a low-power interface was introduced, which allows the user to deploy nodes with a pre-defined duty cycle percentage or sleep time. The TinyOS LPL mechanism provides energy efficiency through a radio duty–cycling mechanism. However, the default inter-packet spacing (IPS) in TinyOS LPL is 8 ms, which may result in lower throughput. With the higher IPS (8 ms) the average packet reception ratio (PRR) is 11.67%.

A wide variety of MAC protocols were designed in order to provide energy efficiency to a sensor network. Due to compatibility problems, the sensor networks are not interoperable. To overcome this problem; a MultiMAC network stack to run multiple MAC protocols using a single radio interface was introduced [115]. The MultiMAC protocol stack utilizes three known protocols, CSMA with collision avoidance (CSMA/CA), LPL MAC, and TDMA MAC, on top of the same radio driver. It introduces the concept of using a virtual gateway to enable sensor network interoperability using heterogeneous MAC protocols. The MultiMAC network stack architecture is depicted in Figure 8.

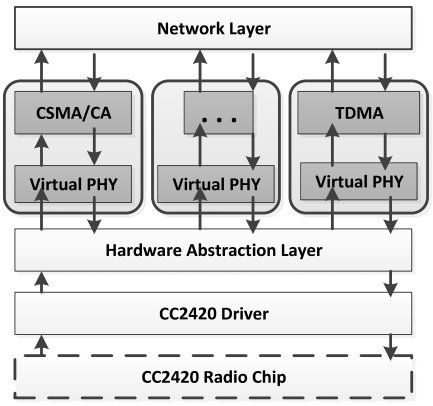

FIGURE 8. The architecture of MultiMAC network stack (adapted from [115]).

The CC2420 driver manages the data transmission, reception, and frame transmission timing. Similarly, the hardware abstraction layer (HAL) provides multiplexing of different MAC protocols. The HAL is responsible for dispatching the received frame to the correct MAC protocols using a MAC-id in the frame, performs address recognition, and sends automatic ACK packets for each MAC. Each MAC protocol is provided a virtual physical layer address to isolate them from one another.

b: TinyOS SUPPORT FOR NETWORK LAYER PROTOCOLS

ContikiRPL, TinyRPL is based on the IETF RPL (RFC 6550) [13]. TinyOS 2.x utilizes an interface provided by the Berkeley low-power IP stack (blip), which implements an IPv6 stack based on 6LoWPAN specifications [116]; blip utilizes 6LoWPAN header compression, neighbor discovery, and DHCPv6 to provide IPv6 in the upper layer. The blip architecture is shown in Figure 9. The IP forwarding abstraction allows RPL implementation on top of ICMv6.

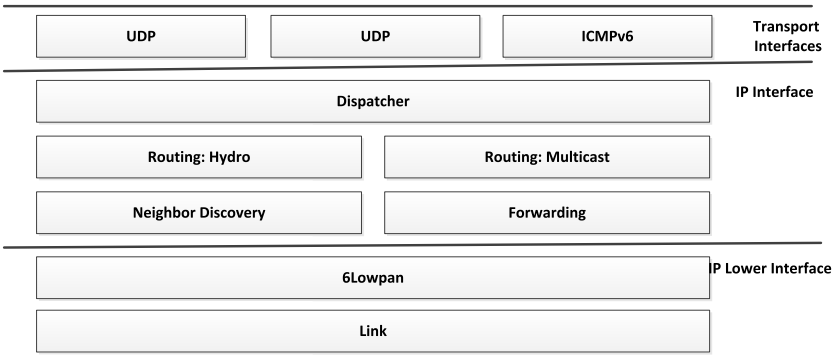

FIGURE 9. The basic architecture of the blip stack.

As explained in ContikiRPL, the packet is forwarded once DODAG is constructed using a DIO message. Then, TinyRPL saves its route in the blip forwarding module. Both ContikiRPL and TinyRPL implement OF0 and MRHOF objective functions for route selection. However, the routing in RPL depends on other layer functions, as well. For example, the MAC layer retransmission timeout affects the RPL link discovery function. It is been shown that the ContikiRPL retransmission timeout is two times the TinyRPL retransmission timeout because of the MAC layer retransmission timeout difference between them [105].

Load balancing under heavy traffic in RPL is carried out efficiently by queue utilization (QU) [117]. The proposed QU-RPL mechanism aims to achieve load balancing by offering a better parent-selection method, which results in less congestion at both the link level and the queue level, along with a better packet reception ratio. During DODAG construction, the DIO message contains information about the RPL node rank and routing metrics. The QU-RPL procedure added new QU information in DIO. During heavy traffic and fast Tickletime, the amount of overhead can cause severe delay in a large IoT network. Similarly, the proposed algorithm is required to add the battery's remaining energy information during parent selection. In the same way, the Tickletimer resetting strategy depends on network size. It requires a more efficient Tickletimer strategy for better output. Providing multimedia application support in this scenario is also very challenging due to constrained environment and overhead costs.

TinyOS Opportunistic Routing Protocol (TORP) for WSNs is designed to conserve energy by implementing an efficient forwarding mechanism [118]. Similarly, the Low-Energy Adaptation Clustering Hierarchy (LEACH) protocol was tested with TinyOS [119]. LEACH forms node clusters using their RSSI values. Then, the local cluster head (CH) is declared a router to communicate with the base station. In this protocol, energy conservation is based on the idea of preventing long distance communications by each node, since only the CH is responsible for communicating with the base station. Another energy-efficient routing protocol named Beacon Vector Routing (BVR) is implemented in TinyOS [120]. It defines a beacon vector routing metric and routes the packets in a greedy manner to the next closest hop, which is calculated by a beacon vector distance metric. A proactive distance-vector protocol (Babel) is implemented in TinyOS to support low-power sensor operation [121]. Babel is based on Destination-Sequenced Distance-Vector (DSDV) routing and Ad Hoc On-Demand Distance Vector (AODV) protocols. Location-aided routing (LAR), Destination-Sequenced Vector Routing (DSVR) and an event-driven data-centric routing protocol are also implemented in TinyOS to support sensors' battery lifetime [122].

c: TinyOS SUPPORT FOR TRANSPORT LAYER PROTOCOLS

IoT applications demand a certain level in quality of service plus specific resource requirements. TinyOS does not provide any specific transport layer protocol implementation. The blip interface in TinyOS provides a UDP socket layer as a basic transport layer implementation [116]. The UDPShell provided by blip contains simple commands, including help, echo, uptime, ping, and ident, to provide debugging commands to the sensor node. Similarly, blip also offers a very simple TCP stack. In order not to exhaust the sensor resources, the TinyOS TCP implementation does not do receive-side buffering. The new packets are dispatched does not do transmitter-side buffering; the dropped segment is automatically retransmitted instantly [123].

Besides the above-mentioned transport layer support, TinyOS supports a variety of transport layer protocols to conserve energy. For example, Sensor Transmission Control Protocol (STCP) was implemented and tested in TinyOS [124]. In STCP, most of the functionalities are implemented at the base station. Thus, a considerable amount of energy is conserved in the sensor nodes. The sensor nodes associate themselves with the base station using a session initiation packet, which informs the base station about the number of flows, data types, and transmission types, and reliability requirements. Likewise, the Hybrid and Dynamic Reliable Transport Protocol (HDRTP) was proposed using TinyOS [44]. It has been shown that HDRTP enhances sensor node performance in terms of success rate, average latency, and average delivery ratio. Similarly, various transport layer

protocols were tested on TinyOS, e.g., a post-order–based protocol [125], the Energy-efficient and Reliable Transport Protocol (ERTP) [126] and the Rate-Controlled Reliable Transport Protocol (RCRT) were implemented and tested in TinyOS [127].

3) FREERTOS SUPPORT FOR COMMUNICATION PROTOCOLS

FreeRTOS utilizes third-party additional tools and applications to provide networking support, e.g., Real Time Engineers Ltd. provides the FreeRTOS+TCP configuration [128]. The FreeRTOS+TCP configuration allows Ethernet-based IPv4 protocol networking solutions. Similarly, a third-party networking port stack e.g., uIP and IwIP, is utilized in a constrained environment [129].

a: FreeRTOS SUPPORT FOR TRANSPORT LAYER PROTOCOLS

FreeRTOS, being a minimalistic OS, does not provide a native MAC layer implementation. However, a third-party implementation is available. For example, IoT-LAB provided a FreeRTOS-based MAC layer implementation to provide better real-time support [130]. They implemented three MAC layers, including CSMA, TDMA, and X-MAC. This CSMA implementation provides a user-defined number of transmitting requests before packet dropping. TDMA is useful to handle a large number of nodes operating on the same channel. X-MAC provides a low-power duty cycling MAC mechanism that is suited to low-traffic networks. Schoofs et al. provided an 802.15.4 MAC implementation on a FreeRTOS microkernel [131]. In this configuration, MAC is considered an application that is run by a FreeRTOS task as a FreeRTOS MAC task and is given the highest priority.

b: FreeRTOS SUPPORT FOR NETWORK LAYER PROTOCOLS

Real Time Engineers Ltd. provides FreeRTOS+TCP, which is an open-source TCP/IP stack [128]. It is based on a Berkeley sockets interface supporting an Ethernet-based IPv4 stack that offers support for UDP and TCP, and lwIP is based on a TCP/IP protocol suite with low RAM usage. IwIP is suitable for devices with 10 kB RAM. Hence, it is suitable for IoT low-end devices. Most of the FreeRTOS demos make use of an old IwIP version. At the network level, it supports Internet Protocol, Internet Control Message Protocol (ICMP) and Internet Group Management Protocol (IGMP). IwIP utilizes basic IP functionalities, it sends, receives, and forwards packets, but does not handle fragmented IP packets with IP options. The OS does not use function calls and data structures directly in the code. In order to make IwIP more portable, an OS uses an emulation layer to provide the IwIP functions. The emulation layer provides a common interface between the kernel and the IwIP code. This interface provides services that include a timer used by TCP/IP. It processes synchronization (semaphores) and has a message processing mechanism that uses an abstraction called mailboxes. The uIP implemented in Contiki does not support all UDP and multicast features, i.e., uIP can send UDP multicast messages, but is unable to join multicast groups and receive multicast messages [110]. Unlike uIP, IwIP provides the necessary UDP and multicast components. FreeRTOS also supports ports of an embedded network stack called Nanostack [132], which was developed by Sensinode. It is based on a 6LowPAN implementation and decreases RAM usage by executing as a single task under FreeRTOS. Similarly, 6LowPAN compresses the IPv6 headers to make them useful for resource-constrained sensor devices.

c: FreeRTOS SUPPORT FOR TRANSPORT LAYER PROTOCOLS

FreeRTOS uses TCP and UDP as transport protocols. FreeRTOS+UDP is a socket-based fully thread–aware stack for FreeRTOS. It provides a Berkeley socket–like interface with compact code size that makes it useful for communication between limited-resource IoT devices. The FreeRTOS+TCP stack, on the other hand, provides a more reliable stream service. Therefore, TCP contains 50% of the total code size of the lwIP stack [129].

E. FILE MANAGEMENT

A typical IoT network consists of thousands of tiny devices that sense the environment and process raw information. This information sometimes needs to be stored. In the past, a wireless sensor network was communication-centric and used to transfer sensed data to one or more sensor devices or a base station. However, in recent years, the presence of onboard flash storage in IoT hardware platforms has provided a storage capability to the sensor network. As a sensor node's memory is a scarce resource, an efficient file system is required, although not every IoT scenario requires a file system. Contiki provides a flash-based file system called Coffee, which gives support to flash-based sensor devices [133]. A typical IoT device contains a few kilobytes of RAM. The onboard flash-based storage provides more memory capabilities. The file system must support storage-centric sensor applications and networking components' storage needs. In other words, how to store and retrieve the data in an efficient manner is handled by the file system. The OS in this scenario is required to support the file system in order to satisfy the IoT resource requirements. TinyOS, on the other hand, uses a single-level file based on the assumption that a node runs a single application at a time [134]. Similarly, FreeRTOS+FAT is a DOS-compatible, open source file system for FreeRTOS [135]. Table 8 summarizes the overview of file management.

1) CONTIKI FILE SYSTEM

Contiki File System (CFS) is a virtual file system to provide an interface to different file systems [133]. Building storage abstraction is a challenging task in a limited-resource environment (little code and a small RAM footprint) [136]. In this scenario, Contiki provides a base for building such an abstraction to support various resource-constrained devices. The two file systems that implement CFS with full

TABLE 8. An overview of file management.

Relevant References	OS	Key Concept	Advantages	Limitations
[138]	Contiki	Coffee File System (CFS): Coffee.	Supports a file system for flash.	The size of files must be reserve beforehand; it introduces delay in real-time applications if it requires a bigger size than the reserved size.
[139]	TinyOS	Single level file system.	Reduces overall power consumption.	It cannot deal with a large number of files simultaneously.
[140]	FreeRTOS	Super Lean FAT File System	Provides a low memory footprint.	FAT file system is not suitable for the IoT, particularly because it was not designed to be used in flash memory.

functionalities are CFS-POSIX and Coffee. CFX-POSIX on the Contiki platform runs in native mode. Other file systems supported by CFS are also available, such as CFS-EEPROM, CFS-RAM, and CFS-XMEM, but these file systems are constrained to a single file only. Coffee, on the other hand, supports a file system for flash (EEPROM)-based sensor devices. Each file system uses the CFS API for reading, writing, and extracting files in a mechanism similar to the Portable Operating System Interface for Unix (POSIX) API. Coffee provides a programming interface to develop an independent storage abstraction. Coffee uses a small RAM footprint profile, and requires 5 kB ROM for the code and 0.5 kB RAM at run-time. Hence, it is suitable for resource-constrained devices. Coffee allows multiple files to coexist on the same onboard flash chip, and provides 92% of the achievable direct flash driver throughput. In order to provide better memory management, the concept of micrologs was introduced. A microlog, unlike the conventional log structure, allows configuring the logs of individual files, providing a tradeoff between space and speed. Flash memory may increase the chance of memory corruption if it erases the page every time. Coffee spreads the spectrum evenly to reduce the risk of corruption.

2) TINYOS FILE SYSTEM

TinyOS supports a single-level file system. It assumes that a node runs a single file in any given time period [134]. Therefore, a single-level file system is sufficient (e.g. TinyOS 1.x uses a microfile system called Matchbox, which provides an interface to read, write, delete, and rename files [137]). Matchbox is designed to provide reliability and low-resource consumption. The Matchbox filing system is very simple; it stores files in an unstructured way and provides only sequential reads and append-only writes. Other third-party file system implementations are also present, e.g. TinyOS FAT 16 supports SD cards aimed at reducing the overall power consumption of sensor nodes. TinyOS 2.x is based on the NesC programming language, which provides an abstraction layer that separates hardware interfaces and provides a framework for developing a portable application [138]. The portable implementation of the FAT file system allows a node to store a large amount of data. Similarly, the Efficient Log-structured Flash (ELF) file system for microsensors is implemented in TinyOS [139]. The ELF log structure provides better memory efficiency, low-power operation, and tailored support for the common types of sensor files.

3) FREERTOS FILE SYSTEM

FreeRTOS uses the Super Lean FAT File System (FreeRTOS+FAT SL). FreeRTOS+FAT (FAT12/FAT16/FAT32) is a DOS/Windows-compatible embedded file system with the main objective of minimizing both flash and RAM footprint (<4 kB and <1 kB, respectively).

IV. OPEN RESEARCH ISSUES AND RECOMMENDATIONS

To develop a practical and efficient IoT OS, many research challenges need to be addressed. The OS should mainly focus on the severe resource shortages and requirements for diverse IoT applications. In this section, we first discuss the general IoT OS research directions, and we then pinpoint some specific research directions.

A. SMALL MEMORY FOOTPRINT

For general research directions, we first argue that more research should be put into the effort to utilize a small memory footprint while providing a developer-friendly API and adding sophisticated features, which may require adding a new programming language or extensions of existing ones. An IoT device contains only a few kilobytes of memory. Hence, the fundamental characteristics of an IoT OS are to reduce the code size and utilize the minimum memory in an efficient manner.

B. ENERGY EFFICIENCY

Another general research direction is to consider a practical energy-efficiency mechanism to prolong the IoT device battery lifetime by designing more efficient network protocols. Similarly, leveraging the hardware features in a smarter manner can lead to better energy efficiency.

C. RELIABILITY OF IoT DEVICES

The reliability of IoT device operation is extremely crucial, especially if they are deployed in a remote location.

To support IoT complex deployments, OS reliability can be achieved by using a microkernel, memory protection units, static code analysis, etc.

D. REAL-TIME SUPPORT

IoT devices require diverse applications; some of them provide real-time operation. These real-time sensings tend to be time-sensitive. Thus, a timely real-time operation guarantee is another challenge faced by the IoT OS.

E. SCHEDULING MODEL

The IoT OS also faces some limitations during task executions that affect the processor and which can lead to extra burden and load on the processor. They can also affect the system's energy efficiency and real-time capabilities.

F. NETWORK BUFFER MANAGEMENT

The network buffer is the main component of the IoT operation. This area requires extensive research to efficiently allocate limited memory to the packets.

G. PROGRAMMING LANGUAGE

Choosing a standard programming language, such as C or C++, or an OS-specific language like NesC, can affect performance, safety, and portability.

H. PROGRAMMING MODEL

The IoT environment could have diverse application needs; application development is highly affected by the programming model. Therefore, the limitation with the programming structure also needs to be addressed.

I. HARDWARE ABSTRACTION LAYER

Research to reduce the amount of overhead in designing the HAL could benefit the overall OS efficiency, especially in dense and lossy networks. On the other hand, it is very important that the HAL be well-designed and portable to different platforms.

J. REAL-TIME OS ISSUES:

Managing a real-time OS is quite challenging. In a complex IoT system, a simple RTOS task may result in complex run-time behavior. Extensive research is required to provide proper task priorities and processor shared timing. FreeRTOS utilizes a multi-threading approach where each process can be interrupted, and the higher priority task can be executed in any given time period, which can delay low priority–task execution time. Similarly, the dependency between tasks may block execution of certain tasks, which may also result in unnecessary delay. Priority inversion, timing properties, and task dependencies require further investigation for RTOS systems. Predicting real-time behavior is very difficult; tasks may execute slower than predicted; they may fail during execution, or can have unexpected delays. Therefore, implementing an RTOS in the IoT environment, especially for time-critical applications, might cause a problem. Hence, dealing with these RTOS challenges is an important research direction.

K. COEXISTENCE

With the growing application scenarios of IoT networks in a limited frequency spectrum, coexistence technologies are an ongoing research problem for OS and radio designers. It is a diverse research area that requires an optimal design of the physical, MAC, and network layers. Achieving wireless coexistence can provide spectrum resource sharing, traffic offloading, and optimal connectivity for diverse IoT services.

Besides the above-mentioned challenges; we also highlight some specific issues, which are yet to be fully addressed.

L. CONTIKI PROTOTHREADS

Contiki provides multi-threading using protothreads, which are stackless and lightweight. Each process runs to completion, and does not allow interrupt handlers to post new events. If an IoT application requires priority for processes and threads, Contiki would not be an ideal OS choice. Hence, extensive research could be done to provide proper process synchronization in Contiki.

M. TinyOS SCHEDULING

The TinyOS scheduling mechanism is not suitable for all application scenarios. For example, the application of encryption security task execution time is very long. If the execution time of some tasks is longer, compared to others, it can affect the baud rate. Similarly, if the local task frequency is high, the OS may lose the other tasks, which affects the overall IoT communications system. TinyOS also fails to handle execution of abnormal tasks, which can lead to a system crash.

N. FreeRTOS SCHEDULING

FreeRTOS is a small real-time OS and provides a very basic scheduling procedure, i.e., highest priority first. Therefore, it does not offer the possibility of hard real-time scheduling. FreeRTOS is well suitable to a small embedded system that has limited, predefined tasks. Research should be done on designing FreeRTOS to handle large IoT system tasks with more advanced scheduling mechanisms.

O. MEMORY PROTECTION IN CONTIKI

Contiki supports a dynamic memory management mechanism. It does not provide a memory protection unit (MPU). Hence, this area still needs to be explored.

P. X-MAC

X-MAC uses a stream of strobes for broadcasting. The strobes do not contain a destination address, which forces each node to wake up for a specific time period. It also does not provide a collision avoidance mechanism, i.e., no clear channel assessment is provided.

Q. ContikiMAC

ContikiMAC is an asynchronous radio duty-cycling procedure. The nodes perform clear channel assignment to determine the presence of ongoing transmissions. It is based on radio signal strength. If this signal strength threshold is not calibrated properly, ContikiMAC can face false and unnecessary wake-ups. This could be an interesting research area in order to provide proper CCA threshold calibration. Similarly, the phase-lock mechanism in ContikiMAC allows a node to learn neighboring nodes' wake up phases. More research efforts can be devoted to improving the phase-lock accuracy.

R. TinyOS LPL

TinyOS LPL provides low-power listening implementations to radios with predefined intervals. A basic problem with LPL is that it may suffer from false alarms, including false positives (waking up when there is no activity on the channel) and false negatives (falling asleep when there is traffic on the channel). LPL performance can degrade dramatically in the presence of external interference and hidden terminal collisions. An open research area is to study the complex relationships between TinyOS LPL duty cycling false positives, false negatives, and latency in the presence of external interference and the hidden terminal problem.

S. ContikiRPL AND TinyRPL INTEROPERABILITY

Interoperability between different protocols is of utmost importance, especially in commercial IoT networks. ContikiRPL performs better in an independent implementation but can affect system performance in a mixed setup. Similarly, ContikiRPL interaction with different MAC protocols has not been investigated yet. Therefore, there is still a need to carry out extensive research to make the ContikiRPL protocol interoperable [140]. Similarly, the current RPL protocols do not support mobility. Hence, a lot of research issues need to be explored, for example, neighbor discovery, link quality estimation, and mobility pattern identification.

T. NETWORK AND LINK LAYER INTEROPERABILITY

The interoperability on the network and link layers is highly important. ETSI has developed plugtest events to solve this kind of problem. The shortcomings of the standards can be tested to develop a new standard or update an existing one.

U. MULTIMEDIA IoT DEVICE TRANSPORT PROTOCOL SUPPORT

Contiki and FreeRTOS provide a simple implementation of the TCP and UPD protocol stack. The IoT contains wide application scenarios for multimedia applications where the battery is either provided by solar or green energy. In these cases, using SCTP is more suitable, which provides continuous and event-driven data flow support. Hence, research is required to implement an efficient protocol at the transport layer in the Contiki and FreeRTOS to fit the requirements of multi-streaming and multi-homing features.

V. CONGESTION CONTROL MECHANISM

The IoT contains a huge number of devices, for example, IEEE 802.11ah access points can support approximately 8000 devices. With this huge number of devices, there is a need to provide an optimal congestion control mechanism. One possible solution is to implement TCP-friendly rate control (TFRC) and a datagram congestion control protocol (DCCP). Therefore, implementing a transport layer protocol with the congestion issue simplified by the TFRC or DCCP should be considered in future research directions.

W. CONTIKI FILE SYSTEM

The Coffee file system provides an excellent storage abstraction in a limited-resource environment. The size of the files requires them to reserve sizes beforehand. If the size of the file is more than the reserved size, Coffee will make a new file with the bigger size. Thus, it will copy all the data from an old file to a new file. This introduces some delay in real-time application scenarios. Similarly, Coffee supports random access semantics, which adds some complexity in delay for log and storage read/write operations.

X. TinyOS FILE SYSTEM

The TinyOS single-level file system provides easy access to files in the directory. However, the main limitation is that the system cannot deal with a large number of files simultaneously. It is also inconvenient to name a large number of files.

Y. FreeRTOS FILE SYSTEM

The FreeRTOS super lean FAT file system is a DOS-compatible basic embedded file system. This file system has yet to mature. It needs further research to enhance storage abstraction capabilities of an RTOS system.

Z. NEW OS CHALLENGES

There are some new OSs, like RIOT [141], Mbed OS [142], embedded Linux, and Zephyr [143]. There is a need to study these OSs in detail to explore future research challenges.

V. CONCLUSION

In this paper, an effort is made to provide insight into various proposed approaches in the IoT OS resource management research area. This paper provides the characteristics of different IoT OS protocols, their design strategies, along with their relevant advantages and limitations. The contributions are multi-fold. First, the IoT concept, various standardization efforts, and motivations to study the management of IoT resources through an IoT OS are provided. Second, various previous surveyed papers are discussed. Third, each resource management aspect of Contiki, TinyOS and FreeRTOS is elucidated. Their resource management mechanisms are classified into various sections, including process management, memory management, energy management, communication management, and file management. These approaches are further classified according to their problem formulations.

Each OS's main approach is discussed with a summary of its underlying idea followed by its advantages and limitations. Finally, open research issues are split into two categories; i.e., general and specific directions for future research, with recommendations given. We believe this survey will stimulate the research community, and pave the way towards more-efficient and robust OSs for low-end devices.

REFERENCES

[1] M. Weiser, "The computer for the 21st century," *ACM SIGMOBILE Comput. Commun. Rev.*, vol. 3, no. 3, pp. 3–11, Jul. 1999.
[2] A. Gubbi, R. Buyya, S. Marusic, and M. Palaniswami, "Internet of Things (IoT): A vision, architectural elements, and future directions," *Future Generat. Comput. Syst.*, vol. 29, no. 7, pp. 1645–1660, Sep. 2013.
[3] D. Miorandi, S. Sicari, F. De Pellegrini, and I. Chlamtac, "Internet of Things: Vision, applications and research challenges," *Ad Hoc Netw.*, vol. 10, no. 7, pp. 1497–1516, Sep. 2012.
[4] L. D. Xu, W. He, and S. Li, "Internet of Things in industries: A survey," *IEEE Trans. Ind. Informat.*, vol. 10, no. 4, pp. 2233–2243, Nov. 2014.
[5] I. Yaqoob *et al.*, "Internet of Things architecture: Recent advances, taxonomy, requirements, and open challenges," *IEEE Wireless Commun.*, vol. 24, no. 3, pp. 10–16, Jun. 2017.
[6] D. Lake, A. Rayes, and M. Morrow, "The Internet of Things," *Internet Protocol J.*, vol. 15, no. 3, pp. 10–19, Sep. 2012.
[7] O. Hahm, E. Baccelli, H. Petersen, and N. Tsiftes, "Operating systems for low-end devices in the Internet of Things: A survey," *IEEE Internet Things J.*, vol. 3, no. 5, pp. 720–734, Oct. 2016.
[8] The Internet Engineering Task Force. *Internet Standards*. Accessed: Jan. 5, 2018. [Online]. Available: https://www.ietf.org/
[9] ITU Telecommunication Standardization Sector. *Standardization*. Accessed: Jan. 5, 2018. [Online]. Available: http://www.itu.int/en/ITU-T/Pages/default.aspx
[10] European Telecommunications Standards Institute. *Standards*. Accessed: Jan. 5, 2018. [Online]. Available: http://www.etsi.org/
[11] *ISO/IEC JTC 1: Internet of Things (IoT)*. Accessed: Jan. 5, 2018. [Online]. Available: https://www.iso.org/files/live/sites/isoorg/files/developing_standards/docs/en/internet_of_things_report-jtc1.pdf
[12] *One M2M: Standards for M2M and the Internet of Things*. Accessed: Jan. 5, 2018. [Online]. Available: http://www.onem2m.org/
[13] *3GPP Standards for the Internet-of-Things*. Accessed: Jan. 5, 2018. [Online]. Available: http://www.3gpp.org/images/presentations/2016_11_3gpp_Standards_for_IoT.pdf
[14] *RPL: IPv6 Routing Protocol for Low-Power and Lossy Networks*. Accessed: Jan. 5, 2018. [Online]. Available: https://tools.ietf.org/html/rfc6550
[15] *Transmission of IPv6 Packets Over IEEE 802.15.4 Networks*. Accessed: Jan. 5, 2018. [Online]. Available: https://tools.ietf.org/html/rfc4944
[16] *Transmission of IPv6 Over MS/TP Networks*. Accessed: Jan. 5, 2018. [Online]. Available: https://datatracker.ietf.org/doc/draft-ietf-6lo-6lobac/
[17] *6TiSCH Operation Sublayer (6TOP) Interface*. Accessed: Jan. 5, 2018. [Online]. Available: https://tools.ietf.org/html/draft-ietf-6tisch-6top-interface-04
[18] *Transport Layer Security (TLS): Datagram Transport Layer Security (DTLS) Profiles for the Internet of Things*. Accessed: Jan. 5, 2018. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc7925
[19] *The Constrained Application Protocol (CoAP)*. Accessed: Jan. 5, 2018. [Online]. Available: https://tools.ietf.org/html/rfc7252
[20] *Concise Binary Object Representation (CBOR)*. Accessed: Jan. 5, 2018. [Online]. Available: https://tools.ietf.org/html/rfc7049
[21] *CBOR Object Signing and Encryption (COSE)*. Accessed: Jan. 5, 2018. [Online]. Available: https://datatracker.ietf.org/wg/cose/documents/
[22] *Y.2060: Overview of the Internet of Things*. Accessed: Jan. 5, 2018. [Online]. Available: https://www.itu.int/rec/T-REC-Y.2060-201206-I/en
[23] G. M. Lee and D. H. Su, "Standarization of smart grid in ITU-T," *IEEE Commun. Mag.*, vol. 51, no. 1, pp. 90–97, Jan. 2013.
[24] *SG17: Security*. Accessed: Jan. 5, 2018. [Online]. Available: http://www.itu.int/en/ITU-T/studygroups/2017-2020/17/Pages/default.aspx
[25] *ISO/IEC JTC 1: Information Technology*. Accessed: Jan. 5, 2018. [Online]. Available: https://www.iso.org/isoiec-jtc-1.html
[26] *IEEE SA P2431: Standard for an Architectural Framework for the Internet of Things (IoT)*. Accessed: Jan. 5, 2018. [Online]. Available: https://standards.ieee.org/develop/project/2413.html
[27] *IEEE SA P2431: Standard for an Architectural Framework for the Internet of Things (IoT)*. Accessed: Jan. 5, 2018. [Online]. Available: http://grouper.ieee.org/groups/2413/Intro-to-IEEE-P2413.pdf
[28] ETSI. *GS LTN 003: Low Throughput Networks (LTN)—Protocols and Interfaces*. Accessed: Jan. 5, 2018. [Online]. Available: http://www.etsi.org/
[29] A. Díaz-Zayas, C. A. García-Pérez, A. M. Recio-Pérez, and P. Merino, "3GPP standards to deliver LTE connectivity for IoT," in *Proc. IEEE 1st Int. Conf. Internet-Things Design Implement. (IoTDI)*, Berlin, Germany, Apr. 2016, pp. 283–288.
[30] *3GPP: Release 12*. Accessed: Jan. 5, 2018. [Online]. Available: http://www.3gpp.org/specifications/releases/68-release-12
[31] D. Astely, E. Dahlman, G. Fodor, S. Parkvall, and J. Sachs, "LTE release 12 and beyond [accepted from open call]," *IEEE Commun. Mag.*, vol. 51, no. 7, pp. 154–160, Jul. 2013.
[32] W. Dong, C. Chen, X. Liu, and J. Bu, "Providing OS support for wireless sensor networks: Challenges and approaches," *IEEE Commun. Surveys Tuts.*, vol. 12, no. 4, pp. 519–530, 4th Quart., 2010.
[33] E. Baccelli, O. Hahm, M. Günes, M. Wählisch, and T. C. Schmidt, "OS for the IoT—Goals, challenges, and solutions," in *Proc. Workshop Interdisciplinaire Sécurité Globale (WISG)*, Troyes, France, Jan. 2013, pp. 1–6.
[34] G. E. Moore, "Cramming more components onto integrated circuits," *Electronics*, vol. 38, no. 8, pp. 114–117, Apr. 1965.
[35] A. Sehgal, V. Perelman, S. Kuryla, and J. Schonwalder, "Management of resource constrained devices in the Internet of Things," *IEEE Commun. Mag.*, vol. 50, no. 12, pp. 144–149, Dec. 2012.
[36] I. Ishaq *et al.*, "IETF standardization in the field of the Internet of Things (IoT): A survey," *J. Sensor Actuator Netw.*, vol. 2, no. 2, pp. 235–287, Apr. 2013.
[37] P. Semasinghe, S. Maghsudi, and E. Hossain, "Game theoretic mechanisms for resource management in massive wireless IoT systems," *IEEE Commun. Mag.*, vol. 55, no. 2, pp. 121–127, Feb. 2017.
[38] *Contiki: The Open Source Operating System for the Internet of Things*. Accessed: Jan. 5, 2018. [Online]. Available: http://www.contiki-os.org/
[39] P. A. Levis, "TinyOS: An open operating system for wireless sensor networks (invited seminar)," in *Proc. 7th Int. Conf. Mobile Data Manage. (MDM)*, May 2006, p. 63.
[40] *FreeRTOS: Quality RTOS & Embedded Software*. Accessed: Jan. 5, 2018. [Online]. Available: http://www.freertos.org/
[41] M. Watfa, M. Moubarak, and A. Kashani, "Operating system designs in future wireless sensor networks," *J. Netw.*, vol. 10, no. 1, pp. 1201–1214, Oct. 2010.
[42] A. Rajandekar and B. Sikdar, "A survey of MAC layer issues and protocols for machine-to-machine communications," *IEEE Internet Things J.*, vol. 2, no. 2, pp. 175–186, Apr. 2015.
[43] N. A. Pantazis, S. A. Nikolidakis, and D. D. Vergados, "Energy-efficient routing protocols in wireless sensor networks: A survey," *IEEE Commun. Surveys Tuts.*, vol. 15, no. 2, pp. 551–591, 2nd Quart., 2013.
[44] B. Sharma and T. C. Aseri, "A hybrid and dynamic reliable transport protocol for wireless sensor networks," *Comput. Elect. Eng.*, vol. 48, pp. 298–311, Nov. 2015.
[45] M. Amjad, M. Sharif, M. K. Afzal, and S. W. Kim, "TinyOS-new trends, comparative views, and supported sensing applications: A review," *IEEE Sensors J.*, vol. 16, no. 9, pp. 2865–2889, May 2016.
[46] Z. D. Purvis and A. G. Dean, "TOSSTI: Saving time and energy in TinyOS with software thread integration," in *Proc. IEEE Real-Time Embedded Technol. Appl. Symp. (RTAS)*, St. Louis, MO, USA, Apr. 2008, pp. 354–363.
[47] S. Abbate, M. Avvenuti, A. Biondi, and A. Vecchio, "Estimation of energy consumption in wireless sensor networks using TinyOS 2.x," in *Proc. IEEE Consum. Commun. Netw. Conf. (CCNC)*, Las Vegas, NV, USA, May 2011, pp. 842–843.
[48] G. Strazdins, A. Elsts, K. Nesenbergs, and L. Selavo, "Wireless sensor network operating system design rules based on real-world deployment survey," *J. Sens. Actuator Netw.*, vol. 2, no. 3, pp. 509–556, 2013.
[49] T. Reusing, "Comparison of operating systems TinyOS and Contiki," *Sensor-Nodes Oper. Netw. Appl.*, vol. 7, pp. 7–13, Aug. 2012.

[50] A. A. Fröhlich and L. F. Wanner, "Operating system support for wireless sensor networks," *J. Comput. Sci.*, vol. 4, no. 4, pp. 272–281, Apr. 2008.
[51] M. O. Farooq and T. Kunz, "Operating systems for wireless sensor networks: A survey," *Sensors*, vol. 11, no. 6, pp. 5900–5930, May 2011.
[52] L. Dariz, M. Selvatici, and M. Ruggeri, "Evaluation of operating system requirements for safe wireless sensor networks," in *Proc. 42nd Annu. Conf. IEEE Ind. Electron. Soc.*, Florence, Italy, Dec. 2016, pp. 5671–5676.
[53] X. Liu *et al.*, "Memory and energy optimization strategies for multithreaded operating system on the resource-constrained wireless sensor node," *Sensors*, vol. 15, no. 1, pp. 22–48, Dec. 2014.
[54] P. Corcoran, "The Internet of Things: Why now, and what's next?" *IEEE Commun. Mag.*, vol. 5, no. 1, pp. 63–68, Jan. 2016.
[55] R. S. Oliver, I. Shcherbakov, and G. Fohler, "An operating system abstraction layer for portable applications in wireless sensor networks," in *Proc. ACM Symp. Appl. Comput.*, Sierre, Switzerland, Mar. 2010, pp. 742–748.
[56] J. Hill and D. Culler, "A wireless embedded sensor architecture for system-level optimization," UC Berkeley, Berkeley, CA, USA, Tech. Rep., 2002. [Online]. Available: http://www.cs.uml.edu/~fredm/courses/91.548-spr05/papers/MICA_ARCH.pdf
[57] L. Wang and Y. Xiao, "A survey of energy-efficient scheduling mechanisms in sensor networks," *Mobile Netw. Appl.*, vol. 11, no. 5, pp. 723–740, Oct. 2006.
[58] J. G. Ko, N. Tsiftes, A. Dunkels, and A. Terzis, "Pragmatic low-power interoperability: ContikiMAC vs TinyOS LPL," in *Proc. 9th Annu. IEEE Commun. Soc. Conf. Sensor, Mesh Ad Hoc Commun. Netw. (SECON)*, Seoul, South Korea, Jun. 2012, pp. 94–96.
[59] N. Tsiftes and A. Dunkels, "A database in every sensor," in *Proc. Conf. Embedded Netw. Sensor Syst. (SenSys)*, Seattle, WA, USA, Nov. 2011, pp. 316–332.
[60] L. Luo, Q. Cao, C. Huang, T. Abdelzaher, J. A. Stankovic, and M. Ward, "EnviroMic: Towards cooperative storage and retrieval in audio sensor networks," in *Proc. 27th Int. Conf. Distrib. Comput. Syst. (ICDCS)*, Toronto, ON, Canada, Jun. 2007, p. 34.
[61] A. Dunkels, F. Osterlind, and Z. He, "An adaptive communication architecture for wireless sensor networks," in *Proc. 5th ACM Conf. Netw. Embedded Sensor Syst. (SenSys)*, Sydney, NSW, Australia, Nov. 2007, pp. 335–349.
[62] P. Dutta and A. Dunkels, "Operating systems and network protocols for wireless sensor networks," *Phil. Trans. Roy. Soc. A, Math., Phys. Eng. Sci.*, vol. 370, no. 1958, pp. 68–84, Jan. 2012.
[63] A. Dunkels, B. Gronvall, and T. Voigt, "Contiki—A lightweight and flexible operating system for tiny networked sensors," in *Proc. 29th Annu. IEEE Int. Conf. Local Comput. Netw.*, Tampa, FL, USA, Nov. 2004, pp. 455–462.
[64] P. Levis *et al.*, "TinyOS: An operating system for sensor networks," in *Ambient Intelligence*, W. Weber, J. M. Rabaey, and E. Aarts, Eds. Berlin, Germany: Springer-Verlag, 2005, ch. 7, pp. 115–148.
[65] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali, "Protothreads: Simplifying event-driven programming of memory-constrained embedded systems," in *Proc. 4th Int. Conf. Embedded Netw. Sensor Syst.*, New York, NY, USA, Oct. 2006, pp. 29–42.
[66] T. Alliance, "TinyOS 2.1 adding threads and memory protection to TinyOS," in *Proc. 6th ACM Conf. Embedded Netw. Sensor Syst.*, Berkeley, CA, USA, Nov. 2008, pp. 413–414.
[67] P. Lindgren, H. Mäkitaavola, J. Eriksson, and J. Eliasson, "Leveraging TinyOS for integration in process automation and control systems," in *Proc. 38th Annu. Conf. IEEE Ind. Electron. Soc. (IECON)*, Montreal, QC, Canada, Oct. 2012, pp. 5779–5785.
[68] R. Goyette, "An analysis and description of the inner workings of the FreeRTOS kernel," Dept. Syst. Comput. Eng., Carleton Univ., Tech. Rep., Apr. 2007.
[69] D. Déharbe, S. Galvão, and A. M. Moreira, "Formalizing FreeRTOS: First steps," in *Formal Methods: Foundations and Applications* (Lecture Notes in Computer Science), vol. 5902, M. V. M. Oliveira and J. Woodcock, Eds. Berlin, Germany: Springer, Aug. 2009, pp. 101–117.
[70] J. F. Ferreira, C. Gherghina, G. He, S. Qin, and W.-N. Chin, "Automated verification of the FreeRTOS scheduler in HIP/SLEEK," *Int. J. Softw. Tools Technol. Transf.*, vol. 16, no. 4, pp. 381–397, Aug. 2014.
[71] *Contiki 2.6: Memory Block Management Functions*. Accessed: Jan. 5, 2018. [Online]. Available: http://contiki.sourceforge.net/docs/2.6/a01684.html
[72] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler, "The *nesC* language: A holistic approach to networked embedded systems," in *Proc. Program. Lang. Design Implement. (PLDI)*, San Diego, CA, USA, Jun. 2003, pp. 1–11.
[73] R. Züger, "Paging in TinyOS," Swiss Fed. Inst. Technol., Zürich, Switzerland, Tech. Rep., Aug. 2006.
[74] J. L. Hill, "Electronic access control, tracking and paging system," U.S. Patent 7 367 497, May 6, 2008.
[75] N. Cooprider, W. Archer, E. Eide, D. Gay, and J. Regehr, "Efficient memory safety for TinyOS," in *Proc. 5th Int. Conf. Embedded Netw. Sensor Syst.*, Sydney, NSW, Australia, Nov. 2007, pp. 205–218.
[76] J. Regehr, N. Cooprider, W. Archer, and E. Eide, "Memory safety and untrusted extensions for TinyOS," School Comput., Univ. Utah, Salt Lake City, UT, USA, Tech. Rep. UUCS-06-007, 2006.
[77] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer, "CCured: Type-safe retrofitting of legacy software," *ACM Trans. Program. Lang. Syst.*, vol. 27, no. 3, pp. 477–526, May 2005.
[78] J. Mistry, M. Naylor, and J. Woodcock, "Adapting FreeRTOS for multicores: An experience report," *Softw., Pract. Exper.*, vol. 44, no. 9, pp. 1129–1154, Feb. 2014.
[79] *Using the FreeRTOS Real Time Kernel*. Accessed: Jan. 5, 2018. [Online]. Available: http://www.freertos.org/Documentation/FreeRTOS-tutorial-book-PIC32-edition-TOC.pdf
[80] R. Lajara, J. Pelegrí-Sebastiá, and J. J. P. Solano, "Power consumption analysis of operating systems for wireless sensor networks," *Sensors*, vol. 10, no. 6, pp. 5809–5826, Jun. 2010.
[81] S. Abbate, M. Avvenuti, D. Cesarini, and A. Vecchio, "Estimation of energy consumption for TinyOS 2.x-based applications," *Proc. Comput. Sci.*, vol. 10, pp. 1166–1171, Dec. 2012.
[82] O. Landsiedel, K. Wehrle, and S. Götz, "Accurate prediction of power consumption in sensor networks," in *Proc. 2nd Workshop Embedded Netw. Sensors*, Washington, DC, USA, May 2005, pp. 37–44.
[83] S. K. Sarna and M. Zaveri, "EATT: Energy aware target tracking for wireless sensor networks using TinyOS," in *Proc. 3rd IEEE Int. Conf. Comput. Sci. Inf. Technol. (ICCSIT)*, Chengdu, China, Jul. 2010, pp. 187–191.
[84] D. Liu, Z. Cao, Y. Zhang, and M. Hou, "Achieving accurate and real-time link estimation for low power wireless sensor networks," *IEEE/ACM Trans. Netw.*, vol. 25, no. 4, pp. 2096–2109, Apr. 2017.
[85] *Study of An Operating System: Freertos*. Accessed: Jan. 5, 2018. [Online]. Available: http://wiki.csie.ncku.edu.tw/embedded/FreeRTOS_Melot.pdf
[86] M.-Y. Zhu, "Understanding FreeRTOS: A requirement analysis," CoreTek Syst., Inc., Beijing, China, Tech. Rep., Sep. 2016.
[87] *Hook Functions*. Accessed: Jan. 5, 2018. [Online]. Available: http://www.freertos.org/a00016.html
[88] *Low Power Support: Tickless Idle Mode*. Accessed: Jan. 5, 2018. [Online]. Available: http://www.freertos.org/low-power-tickless-rtos.html
[89] O. Bello and S. Zeadally, "Intelligent device-to-device communication in the Internet of Things," *IEEE Syst. J.*, vol. 10, no. 3, pp. 1172–1182, Sep. 2016.
[90] A. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari, and M. Ayyash, "Internet of Things: A survey on enabling technologies, protocols, and applications," *IEEE Commun. Surveys Tuts.*, vol. 17, no. 4, pp. 2347–2376, 4th Quart., 2015.
[91] K. Sohraby, D. Minoli, and T. Znati, *Wireless Sensor Networks: Technology, Protocols, and Applications*. Hoboken, NJ, USA: Wiley, Mar. 2007.
[92] I. Glaropoulos, V. Vukadinovic, and S. Mangold, "Contiki80211: An IEEE 802.11 radio link layer for the Contiki OS," in *Proc. IEEE 6th Int. Symp. Cyberspace Safety Secur.*, Paris, France, Aug. 2014, pp. 621–624.
[93] *The uIP TCP/IP Stack*. Accessed: Jan. 5, 2018. [Online]. Available: http://contiki.sourceforge.net/docs/2.6/a01793.html#_details
[94] *Contiki Netstack*. Accessed: Jan. 5, 2018. [Online]. Available: http://anrg.usc.edu/contiki/index.php/Network_Stack
[95] M. Michel and B. Quoitin. (Apr. 2014). "Technical report: ContikiMAC vs X-MAC performance analysis." [Online]. Available: https://arxiv.org/abs/1404.3589
[96] H. Wang, X. Zhang, F. Naït-Abdesselam, and A. Khokhar, "An asynchronous low-power medium access control protocol for wireless sensor networks," *Wireless Commun. Mobile Comput.*, vol. 13, no. 6, pp. 604–618, Apr. 2013.
[97] A. Dunkels, "The ContikiMAC radio duty cycling protocol," Swedish Inst. Comput. Sci., Stockholm, Sweden, Tech. Rep., Dec. 2011.

[98] M. Buettner, G. V. Yee, E. Anderson, and R. Han, "X-MAC: A short preamble MAC protocol for duty-cycled wireless sensor networks," in *Proc. 4th Int. Conf. Embedded Netw. Sensor Syst.*, Boulder, CO, USA, Nov. 2006, pp. 307–320.
[99] A. El-Hoiydi and J.-D. Decotignie, "WiseMAC: An ultra low power MAC protocol for the downlink of infrastructure wireless sensor networks," in *Proc. 9th Int. Symp. Comput. Commun.*, Alexandria, Egypt, Jul. 2004, pp. 244–251.
[100] P. Gonizzi, P. Medagliani, G. Ferrari, and J. Leguay, "RAWMAC: A routing aware wave-based MAC protocol for WSNs," in *Proc. IEEE 10th Int. Conf. Wireless Mobile Comput., Netw. Commun. (WiMob)*, Larnaca, Cyprus, Oct. 2014, pp. 205–212.
[101] N. Tsiftes, J. Eriksson, and A. Dunkels, "Low-power wireless IPv6 routing with ContikiRPL," in *Proc. 9th ACM/IEEE Int. Conf. Inf. Process. Sensor Netw. (ISPN)*, Stockholm, Sweden, Apr. 2010, pp. 406–407.
[102] J. Brown and U. Roedig, "Demo abstract: GinLITE—A MAC protocol for real-time sensor networks," in *Proc. 9th Eur. Conf. Wireless Sensor Netw. (EWSN)*, Trento, Italy, Feb. 2012, pp. 1–2.
[103] *Routing Metrics Used for Path Calculation in Low-Power and Lossy Networks*. Accessed: Jan. 5, 2018. [Online]. Available: https://tools.ietf.org/html/rfc6551.
[104] D. Carels, E. De Poorter, I. Moerman, and P. Demeester, "RPL mobility support for point-to-point traffic flows towards mobile nodes," *Int. J. Distrib. Sensor Netw.*, vol. 2015, Jan. 2015, Art. no. 111.
[105] J. Ko *et al.*, "ContikiRPL and TinyRPL: Happy together," in *Proc. Workshop Extending Internet Low Power Lossy Netw. (IPSN)*, Chicago, IL, USA, Apr. 2011, pp. 1–6.
[106] Y. Tahir, S. Yang, and J. McCann, "BRPL: Backpressure RPL for high-throughput and mobile IoTs," *IEEE Trans. Mobile Comput.*, vol. 17, no. 1, pp. 29–43, Jan. 2017.
[107] H. Al-Kashoash, M. Hafeez, and A. Kemp, "Congestion control for 6LoWPAN networks: A game theoretic framework," *IEEE Internet Things J.*, vol. 4, no. 3, pp. 760–771, Jun. 2017.
[108] Y. Al-Nidawi, N. Salman, and A. H. Kemp, "Mesh-under cluster-based routing protocol for IEEE 802.15.4 sensor network," in *Proc. 20th Eur. Wireless Conf.*, Barcelona, Spain, Jun. 2014, pp. 1–7.
[109] A. Hassan, S. Alshomrani, A. Altalhi, and S. Ahsan, "Improved routing metrics for energy constrained interconnected devices in low-power and lossy networks," *J. Commun. Netw.*, vol. 18, no. 3, pp. 327–332, Jun. 2016.
[110] *The uIP TCP/IP Stack*. Accessed: Jan. 5, 2018. [Online]. Available: http://contiki.sourceforge.net/docs/2.6/a01793.html
[111] *Packet Protocols, TEP Core Working Group*. Accessed: Jul. 30, 2017. [Online]. Available: http://www.tinyos.net/tinyos-2.x/doc/html/tep116.html
[112] *TinyOS Tutorials*. Accessed: Jul. 30, 2017. [Online]. Available: http://tinyos.stanford.edu/tinyos-wiki/index.php/TinyOS_Tutorials#Network_Protocols
[113] *TinyOS Network Protocols*. Accessed: Jul. 30, 2017. [Online]. Available: http://tinyos.stanford.edu/tinyos-wiki/index.php/Network_Protocols
[114] J. Polastre, J. Hill, and D. Culler, "Versatile low power media access for wireless sensor networks," in *Proc. 2nd Int. Conf. Embedded Netw. Sensor Syst. (ACM SenSys)*, Baltimore, MD, USA, Nov. 2004, pp. 95–107.
[115] D. van den Akker and C. Blondia, "MultiMAC: A multiple MAC network stack architecture for TinyOS," in *Proc. 2nd Int. Conf. Comput. Commun. Netw. (ICCCN)*, Munich, Germany Aug. 2012, pp. 1–5.
[116] *BLIP Tutorial*. Accessed: Jun. 30, 2017. [Online]. Available: http://tinyos.stanford.edu/tinyos-wiki/index.php/BLIP_Tutorial
[117] H.-S. Kim, H. Kim, J. Paek, and S. Bahk, "Load balancing under heavy traffic in RPL routing protocol for low power and lossy networks," *IEEE Trans. Mobile Comput.*, vol. 16, no. 4, pp. 964–979, Apr. 2017.
[118] J. Carnley, B. Sun, and S. K. Makki, "TORP: TinyOS opportunistic routing protocol for wireless sensor networks," in *Proc. IEEE Consum. Commun. Netw. Conf. (CCNC)*, Las Vegas, NV, USA, Jan. 2011, pp. 111–115.
[119] C. Hou, K. W. Tang, and E. Noel, "Implementation and analysis of the LEACH protocol on the TinyOS platform," in *Proc. IEEE Int. Conf. ICT Converg. (ICTC)*, Jeju, South Korea, Oct. 2013, pp. 918–923.
[120] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister, "System architecture directions for networked sensors," in *Proc. 9th Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, Cambridge, MA, USA, Nov. 2000, pp. 93–104.
[121] A. Hauck and P. Sollberger, "Babel multi-hop routing for TinyOS low-power devices," in *Proc. 5th Int. Conf. Mobility Ubiquitous Comput., Syst., Services Technol.*, Lisbon, Portugal, Nov. 2011, pp. 111–114.
[122] S. Khan, S. Basharat, M. S. H. Khiyal, and S. A. Khan, "Investigating energy consumption of localized and non localized ad hoc routing protocols in TinyOS," in *Proc. IEEE Multitopic Conf. (INMIC)*, Islamabad, Pakistan, Dec. 2006, pp. 355–358.
[123] A. Ludovici, P. Moreno, and A. Calveras, "TinyCoAP: A novel constrained application protocol (CoAP) implementation for embedding RESTful Web services in wireless sensor networks based on TinyOS," *J. Sens. Actuator Netw.*, vol. 2, no. 2, pp. 288–315, May 2013.
[124] Y. G. Iyer, S. Gandham, and S. Venkatesan, "STCP: A generic transport layer protocol for wireless sensor networks," in *Proc. IEEE Int. Conf. Comput. Commun. Netw. (ICCCN)*, San Diego, CA, USA, Oct. 2005, pp. 17–19.
[125] S. Shekhar, R. Mishra, R. K. Ghosh, and R. K. Shyamasundar, "Post-order based routing & transport protocol for wireless sensor networks," *Pervas. Mobile Comput.*, vol. 11, pp. 229–243, Apr. 2014.
[126] T. Le, W. Hu, P. Corke, and S. Jha, "ERTP: Energy-efficient and reliable transport protocol for data streaming in wireless sensor networks," *Comput. Commun.*, vol. 32, nos. 7–10, pp. 1154–1171, May 2009.
[127] J. Paek and R. Govindan, "RCRT: Rate-controlled reliable transport protocol for wireless sensor networks," *ACM Trans. Sensor Netw.*, vol. 7, no. 3, pp. 1–20, Oct. 2010.
[128] *FreeRTOS+TCP*. Accessed: Jul. 30, 2017. [Online]. Available: http://www.freertos.org/FreeRTOS-Plus/FreeRTOS_Plus_TCP/index.html
[129] *lwIP—A Lightweight TCP/IP Stack*. Accessed: Jan. 5, 2018. [Online]. Available: https://savannah.nongnu.org/projects/lwip/
[130] *IoT-Lab Libraries*. Accessed: Jan. 5, 2018. [Online]. Available: https://github.com/iot-lab/iot-lab/wiki/Libraries
[131] A. Schoofs, M. Aoun, P. van der Stok, J. Catalano, R. S. Oliver, and G. Fohler, "A framework for time-controlled and portable WSN applications," in *Sensor Applications, Experimentation, and Logistics* (Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering). Berlin, Germany: Springer, 2010, pp. 126–144.
[132] *NanoStack Manual*. Accessed: Jan. 5, 2018. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.129.1353&rep=rep1&type=pdf
[133] N. Tsiftes, A. Dunkels, Z. He, and T. Voigt, "Enabling large-scale storage in sensor networks with the Coffee file system," in *Proc. Int. Conf. Inf. Process. Sensor Netw.*, San Francisco, CA, USA, Aug. 2009, pp. 349–360.
[134] *Interoperable Sensor Networks: Contiki and TinyOS*, Edosoft Factory, Las Palmas, Spain, Aug. 2012.
[135] *FreeRTOS+FAT: DOS Compatible Embedded FAT File System*. Accessed: Jan. 5, 2018. [Online]. Available: http://www.freertos.org/FreeRTOS-Plus/FreeRTOS_Plus_FAT/index.html
[136] L. A. Jenß, "Design and implementation of a data storage abstraction layer for the Internet of Things," M.S. thesis, Faculty Eng. Comput. Sci., Dept. Comput. Sci., Hamburg Univ. Appl. Sci., Hamburg, Germany, Jan. 2017.
[137] P. Levis and D. Gay, *TinyOS Programming*. Cambridge, U.K.: Cambridge Univ. Press, Jun. 2009.
[138] *A Tutorial for Programming in TinyOS*. Accessed: Jan. 5, 2018. [Online]. Available: http://www.ece.rochester.edu/projects/wcng/code/Tutorial/TinyOs_Tutorial.pdf
[139] H. Dai, M. Neufeld, and R. Han, "ELF: An efficient log-structured flash file system for micro sensor nodes," in *Proc. 2nd Int. Conf. Embedded Netw. Sensor Syst.*, Baltimore, MD, USA, Nov. 2004, pp. 176–187.
[140] Y. B. Zikria, F. Ishmanov, M. K. Afzal, S. W. Kim, S. Y. Nam, and H. J. Yu, "Opportunistic channel selection MAC protocol for cognitive radio ad hoc sensor networks in the Internet of Things," *Sustain. Comput., Inform. Syst.*, Jan. 2018. [Online]. Available: https://doi.org/10.1016/j.suscom.2017.07.003
[141] *RIOT: The Friendly Operating System for the Internet of Things*. Accessed: Feb. 15, 2018. [Online]. Available: https://riot-os.org/
[142] *Mbed*. Accessed: Feb. 15, 2018. [Online]. Available: https://www.mbed.com/en/
[143] *Zephyr Project*. Accessed: Feb. 15, 2018. [Online]. Available: https://www.zephyrproject.org/

ARSLAN MUSADDIQ received the B.S. degree in electrical engineering (telecommunication) from Bahria University, Islamabad, Pakistan, in 2011, and the M.S. degree in communication and network engineering from University Putra Malaysia in 2015. He is currently pursuing the Ph.D. degree with the Department of Information and Communication Engineering, College of Engineering, Yeungnam University, Gyeongsan, South Korea. His research interests include wireless networking, Internet of Things, wireless resource management, routing protocols and ad hoc networks. He was a recipient of the Outstanding Dissertation (M.S. level) Award at the IEEE Malaysia Communication Society and the Vehicular Technology Society Joint Chapter in 2015.

YOUSAF BIN ZIKRIA received the B.S. degree in computer engineering from the University of Arid Agriculture Rawalpindi, Rawalpindi, Pakistan, in 2005, the M.S. degree in computer engineering from the Comsats Institute of Information Technology, Islamabad, Pakistan, in 2007, and the Ph.D. degree from the Department of Information and Communication Engineering, Yeungnam University, Gyeongsan, South Korea, in 2016. He was a Research Officer with Horizon Technology Pvt. Ltd., Pakistan, from 2007 to 2011. He was with King Khalid University, Saudi Arabia, as a Lecturer, from 2011 to 2012. He is currently a Post-Doctoral Fellow with the Department of Information and Communication Engineering, College of Engineering, Yeungnam University. He has authored over 10 years of experience in research, academia, and industry in the field of information and communication engineering, and computer science. His research interests include IoT, 5G, wireless communications and networks, opportunistic communications, wireless sensor networks, routing protocols, cognitive radio ad hoc networks, cognitive radio ad hoc sensor networks, transport protocols, VANETS, embedded system, and information security. He was a recipient of the Excellent Paper Award at the ICIDB 2016 Conference and a fully funded scholarship for Masters and Ph.D. He held the prestigious CISA, JNCIS-SEC, JNCIS-ER, JNCIA-ER, JNCIA-EX, and Advance Routing Switching and WAN Technologies certifications. He is the Editor FT/SI on Unlocking 5G Spectrum Potential for Intelligent IoT: Opportunities, Challenges and Solutions for *IEEE Communications Magazine*, Internet of Things(IoT): Operating System, Applications and Protocols Design, and Validation Techniques for *Future Generation Computer Systems* (FGCS) (Elsevier), and 5G Mobile Services and Scenarios: Challenges and Solutions for MDPI *Sustainability*. He is also serving as a Reviewer of the IEEE Communications, Surveys and Tutorials, the IEEE Sensors Letters, the IEEE Access, the IEEE IT Professional, FGCS (Elsevier), *Computer Standards and Interfaces* (Elsevier), *The Journal of Supercomputing* (Springer), the *International Journal of Distributed Sensor Networks* (Sage), and *KSII Transactions on Internet and Information Systems*.

OLIVER HAHM received the Diploma degree in computer science from the Freie Universität Berlin in 2007 and the Ph.D. degree from the École Polytechnique, Paris, France, in 2016. He was a Software Engineer with ScatterWeb GmbH, a startup for wireless sensor network applications, from 2007 to 2009. He was with the Faculty of Mathematics and Computer Science, Freie Universität Berlin, as a Research Assistant until 2012, where he was responsible for the G-LAB Project. From 2012 to 2017, he was with Inria, as a Researcher. He is currently an Embedded Software Engineer with Zühlke Engineering GmbH, Eschborn, Germany. His research is focused on operating systems for the Internet of Things, embedded network stacks, information-centric networking, and standardization efforts in the area of low-power and lossy networks. He is a Co-Founder and one of the core developers and maintainers of the RIOT operating system. He served as a reviewer for various IEEE and ACM conferences and journal and has been member for multiple IEEE, ACM, and EAI conferences. He currently serves as a Guest Editor for a Special Issue of the *Future Generation Computer Systems* (Elsevier) journal on Internet of Things (IoT): Operating System, Applications and Protocols Design, and Validation Techniques.

HEEJUNG YU received the B.S. degree in radio science and engineering from Korea University, Seoul, South Korea, in 1999, and the M.S. and Ph.D. degrees in electrical engineering from the Korea Advanced Institute of Science and Technology, Daejeon, South Korea, in 2001 and 2011, respectively. From 2001 to 2012, he was with the Electronics and Telecommunications Research Institute, Daejeon. Since 2012, he has been with the Department of Information and Communication Engineering, Yeungnam University, Gyeongsan, South Korea. He has participated in the IEEE 802.11 standardization, where he has made technical contributions since 2003. He have actively participated in the IEEE 802.11ah standardization, which is a potential wireless communication standard for Internet of Things (IoT). He was the Vice Chair of the 5G convergent service working group in 5G Forum Korea from 2016 to 2017, where he is currently involved in developing the service scenarios and their function requirement for 5G wireless technology and networks. His research interests include statistical signal processing, communication theory and 5G networks including IoT. He was a recipient of the Bronze Prize at the 17th Humantech Paper Contest and the Best Paper Award in the 21st Joint Conference on Communications and Information and the 2017 Winter Conference of the Korean Institute of Communications and Information Science (KICS) in 2011 and 2017, respectively. He was also a recipient of the Contribution Award in 5G Forum, South Korea, in 2017. He has been a Guest Editor of Special issue on Internet of Things (IoT): Operating System, Applications and Protocols Design, and Validation Techniques of Future Generation Computer Systems in 2017 and a main Guest Editor of Special issue on Automotive functional safety of *Information & Communications Magazine* published by KICS in 2017.

ALI KASHIF BASHIR (M'15–SM'16) received the Ph.D. degree in computer science and engineering from Korea University, South Korea. He held appointments with Osaka University, Japan, the Nara National College of Technology, Japan, the National Fusion Research Institute, South Korea, Southern Power Co., Ltd., South Korea, and the Seoul Metropolitan Government, South Korea. He is currently an Associate Professor with the Faculty of Science and Technology, University of the Faroe Islands, Faroe Islands, Denmark. He is also with the Advanced Network Architecture Laboratory as a Joint Researcher. He is also supervising/co-supervising several graduate (M.S. and Ph.D.) students. His research interests include cloud computing, NFV/SDN, network virtualization, network security, IoT, computer networks, RFID, sensor networks, wireless networks, and distributed computing. He is an Editorial Board Member of journals, such as the IEEE Access, the *Journal of Sensor Networks*, and the *Data Communications*. He is serving as the Editor-in-Chief of the IEEE Internet Technology Policy Newsletter and the IEEE Future Directions Newsletter. He has also served/serving as a guest editor on several special issues in journals of the IEEE, Elsevier, and Springer. He is actively involved in organizing workshops and conferences. He has chaired several conference sessions, gave several invited and keynote talks, and reviewed the technology leading articles for journals, such as the IEEE Transactions on Industrial Informatics, *IEEE Communication Magazine*, the IEEE Communication Letters, IEEE Internet of Things, and the IEICE Journals, and conferences, such as the IEEE Infocom, the IEEE ICC, the IEEE Globecom, and the IEEE Cloud of Things.

SUNG WON KIM received the B.S. and M.S. degrees from the Department of Control and Instrumentation Engineering, Seoul National University, South Korea, in 1990 and 1992, respectively, and the Ph.D. degree from the School of Electrical Engineering and Computer Sciences, Seoul National University, Korea, in 2002. From 1992 to 2001, he was a Researcher with the Research and Development Center, LG Electronics, South Korea. From 2001 to 2003, he was a Researcher with the Research and Development Center, AL Tech, South Korea. From 2003 to 2005, he was a Post-Doctoral Researcher with the Department of Electrical and Computer Engineering, University of Florida, Gainesville, USA. In 2005, he joined the Department of Information and Communication Engineering, Yeungnam University, Gyeongsan, South Korea, where he is currently a Professor. His research interests include resource management, wireless networks, mobile networks, performance evaluation, and embedded systems.