IET Wireless Sensor Systems

Special Issue: Smart Cities and Smart Sensory Platforms

IET Journals
The Institution of Engineering and Technology

Comprehensive survey of the IoT open-source OSs

ISSN 2043-6386
Received on 5th February 2018
Revised 27th September 2018
Accepted on 27th September 2018
E-First on 30th October 2018
doi: 10.1049/iet-wss.2018.5033
www.ietdl.org

Mahmoud H. Qutqut[1,2] ✉, Aya Al-Sakran[1], Fadi Almasalha[1], Hossam S. Hassanein[2]

[1]*Faculty of Information Technology, Applied Science Private University, Amman 11931, Jordan*

[2]*Telecommunications Research Laboratory, School of Computing, Queen's University, Kingston, Ontario, Canada ON K7L 2N8*

✉ *E-mail: qutqut@asu.edu.jo*

Abstract: The Internet of things (IoT) has attracted a great deal of research and industry attention recently and is envisaged to support diverse emerging domains including smart cities, health informatics, and smart sensory platforms. Operating system (OS) support for IoT plays a pivotal role in developing scalable and interoperable applications that are reliable and efficient. IoT is implemented by both high-end and low-end devices that require OSs. Recently, the authors have witnessed a diversity of OSs emerging into the IoT environment to facilitate IoT deployments and developments. In this study, they present a comprehensive overview of the common and existing open-source OSs for IoT. Each OS is described in detail based on a set of designing and developmental aspects that they established. These aspects include architecture and kernel, programming model, scheduling, memory management, networking protocols support, simulator support, security, power consumption, and support for multimedia. They present a taxonomy of the current IoT open-source OSs. The objective of this survey is to provide a well-structured guide to developers and researchers to determine the most appropriate OS for each specific IoT devices/applications based on their functional and non-functional requirements. They remark that this is the first such tutorial style paper on IoT OSs.

1 Introduction

The Internet of things (IoT) can be described as a dynamic distributed networked system that represents the most energising technological revolution nowadays [1]. IoT is a network that consists of a massive number of things (e.g. sensors, machines, or appliances) communicating, sending, and receiving data via the Internet [2]. The number of physical objects that are connected to the Internet has been exponentially growing [3]. According to Gartner Inc. report, there will be about 21 billion connected devices by the year 2020 [4]. There are many domains and fields that the IoT can play an essential role and improve the quality of our lives [5]. These domains and fields range from health informatics, smart transportation, smart sensory platforms, to emergency cases where human decision making might be difficult [6].

IoT devices have constrained functionalities and minimal footprint, which consume lower internal storage, memory, and computation power than typical devices [7]. They are also battery operated or embedded within integrated circuits. Some IoT devices can survive for a month to several years before a new battery placement is required [8]. However, now, due to the IoT revolution, the functionalities of these devices are expanding. So the device or the controller must be more intelligent in monitoring several inputs, updating events to gateways, or devices as well as receiving commands from the gateways or other devices [9]. IoT enables physical objects to see, hear, and perform several tasks by allowing them to communicate with each other to share information and to coordinate in order to make decisions [9]. IoT devices are considered heterogeneous, and they are used in different applications. Various types of applications require different architecture and hardware support that can operate on low-power sources. Hence, having an operating system (OS) that satisfies all the requirements for IoT devices in various domains is almost impossible [10]. The IoT is implemented by both high-end and low-end devices that required appropriate OSs. High-end devices can operate using traditional OSs (such as Linux), whereas low-end devices operate with limited capabilities OSs that cannot perform the same task of traditional OSs. Furthermore, OS support for IoT plays a pivotal role in deploying reliable and scalable large-scale IoT deployments. Over the years, we have witnessed diverse OSs emerging into the IoT environment to facilitate IoT deployments and developments. Additionally, several research works have been devoted to improve OS performance and capabilities in different dimensions.

To this end, we provide a comprehensive overview of the most common open-source OSs for IoT. The choice of open-source OSs is because we can have more information about their requirements and functionalities, allowing a thorough investigation of such OSs. This paper is written in a tutorial style, where each OS is described in-depth according to a group of design and development aspects that we define. We remark that this is the first such tutorial style paper on this topic. The primary objective of this paper is to provide an easy to follow and well-structured guide for researchers and developers targeting IoT platforms. First, we establish a set of design and development aspects for IoT OSs to help in evaluating and understanding each OS in-depth. Second, we present a taxonomy to provide a classification of the different kinds of open-source OSs for IoT from different perspectives. Then, we study each OS from the following aspects; architecture and kernel, programming model, scheduling, memory management, networking protocols, simulator, security, power consumption, and support for multimedia. This paper reveals the strengths and weaknesses of each OS presented in this paper. Finally, we provide several comparative analysis of the presented OSs in this paper and summarise their features.

The remainder of this paper is organised as follows. Section 2 presents a general overview of IoT and its challenges and existing OSs for IoT environment. Section 3 shows the related aspects of designing and developing an IoT OS. Section 4 illustrates a proposed taxonomy for open-source IoT OSs. In Section 5, we discuss open-source OSs for low-end IoT devices, whereas in Section 6 we present open-source OSs for high-end IoT devices. Section 7 provides three comparisons between the discussed IoT OSs followed by the conclusion of this paper in Section 8.

2 Background

In this section, we provide an overview of IoT and its challenges. We also elaborate on why we need different and modified OSs for the IoT.

2.1 Internet of things

The number of Internet-connected devices and machines is increasing exponentially [11]. These devices and machines create a dynamic network that consists of billions of things/objects communicating with each other. One of the main goals of IoT is to facilitate the collection of data from one point to another at any time and anywhere through efficient, secure, and reliable connections [5]. IoT is characterised by its exceptionally identifiable items, things, and their virtual representations in an Internet-like structure [12]. Things can interact with each other at any time and place, in any way, and from any device. They can enter and leave the network without the need to be restricted to a single physical location to exchange data [13]. In IoT, millions, if not billions, of heterogeneous devices should be connected to the Internet [14]. These devices vary in computational power, available memory, communication, and energy capacity. The IoT environment consists of protocols, network designs, and service architectures that deal with a massive number of IoT objects and devices to exchange data [15]. Thus, the IoT needs to support multiple objects based on different types of radio interfaces with various numbers of requirements regarding the available resources [16]. IoT applications apply to different domains and disciplines such as smart homes, environmental monitoring, health care, inventory and products management, smoke and fire detectors, transportation, security, and surveillance systems [17, 18].

2.2 IoT challenges

One of the crucial challenges in IoT is to manage, maintain, and deal with an enormous number of heterogeneous devices and the data generated by them. This massive number of connected things and objects lead to many technical and application challenges. The main challenges and issues in IoT are presented below:

Heterogeneity: IoT connects and manages a massive number of heterogeneous devices (e.g. full-fledged web servers and other devices) constitute a critical challenge that has different operating conditions, platforms etc. to provide advanced applications that can improve our life [18]. As a result, the complexity will be increased. Hence, developing applications that run on all platforms will become extremely difficult, and increases the need for standard interoperable architecture [18]. Also, data should be exchanged of large-scale heterogeneous network elements in dynamic local autonomy with highly efficient network convergence [18].

Security issues: Owing to the number of heterogeneous devices that are exchanging data over the Internet, there is a risk of people privacy. This is because these devices will record huge amounts of data about people daily lives that could be pieced collectively to create an in-depth portrait of their presence [19]. So, it is essential to ensure a secure data channel between the participated devices in IoT. Also, it is a must to have a secure and reliable connection between heterogeneous devices in the IoT environment [19].

Scalability: With a massive number of heterogeneous devices such as smart sensors and light bulbs which are fitted with minimal processing and storage units, scalability becomes a crucial challenge for the current growth of IoT [20]. For example, calculation of daily temperature variations around all the country may require millions of devices resulting in a substantial amount of data that cannot be easily processed and managed. That is why IoT needs data compression and data fusion to reduce this significant data volume [21].

Interoperability: One of the significant challenges in IoT applications is interoperability for crossing layers among different IoT devices and deployments [22]. Interoperability challenge appears when heterogeneous devices use different data formats and various protocols to collaborate in communication and data exchange [23]. In IoT applications, there are multiple competing application-level protocols, function, and devices exist to provide communication interoperability. Each of these protocols maintains unique characteristics and messaging architecture for different types of IoT applications. These traditionally are built with different languages and protocols. So, it is essential to design a scalable IoT architecture, called middle-ware layer, to support a large number of heterogeneous devices and work independently from messaging protocol standards [22].

Architecture: IoT includes an increasing number of heterogeneous interconnected devices and sensors that are often transparent and invisible. Owing to the number of these devices and machines, a single architecture cannot be applied to all these heterogeneous devices [24]. Heterogeneous reference architectures adapted to IoT environment should be open, and they should not restrict users to use fixed or end-to-end solutions [24]. Also, they should be flexible to deal with different deployments such as identifications (radio-frequency identification, tags), intelligent devices, and smart objects [24].

2.3 OSs for the IoT environment

Like any new software, an OS has to be integrated with existing IoT environment. To keep the complex IoT environment running, a new OS has to be customised to meet specific requirements. Owing to some constraints existence in many traditional OSs, they are impractical to be used in IoT as IoT devices are designed with limited resources [25]. The adoption process of an OS requires that the OS should be able to operate IoT devices efficiently. In addition to that each IoT device may need to be customised to make use of its components that may not be available in other devices. Nowadays, we are surrounded by many smart devices that are different in their levels of complexity depending on their purposes. All of them have a processor, a memory to store data, and other peripherals [9]. To adapt to the constraints of typical IoT devices successfully, it is necessary to have an appropriate OS that allows easy control, connectivity, and communications. We are witnessing some OSs for IoT implementations and large-scale deployments [25]. However, there is a rapid need for development tools, standardisation, easy maintenance, and porting of applications across a wide variety of hardware platforms.

3 Designing and developmental aspects of IoT OSs

The OS is the most basic system software which runs directly on hardware resources to act as an intermediate between applications/users and hardware. An OS mainly contains various components but necessary a kernel, utility software, and system shell [26]. The kernel is the essential part of an OS. It is a programme that manages all activities in the system and gives permissions to other software and users to perform any action [26]. The traditional OSs are designed for workstations and personal computers (PCs) with plenty of resources. For this reason, they are not appropriate for IoT devices with constrained resources and diverse data-centric applications. IoT devices need a customised type of OS considering their unique characteristics. Moreover, IoT devices require an entirely different architecture of OS and an extensive range of hardware support. In this section, we present the main characteristics and criteria to be considered when designing and developing an OS for IoT devices. These characteristics and criteria are discussed below.

3.1 Architecture and kernel models

Architecture is one of the most critical criteria for designing an OS. The core software component of an OS is known as kernel. The architecture of an OS has an effect on the size of the core kernel and on how to provide services to applications. There are mainly five standard architectures for OS: monolithic, microkernel, virtual machine, modular, and layered architecture [27]. We will discuss all OS architectures in the following sections.

3.1.1 Monolithic architecture: This architecture model does not follow any specific structure, where the OS architecture is working in the kernel space such as Linux and Unix. It is also called multi-tier architecture because monolithic applications are divided into four or more layers such as presentation, application, database, and business layers. It mainly consists of a set of primitives or system calls to access input/output (I/O) devices, memory, hardware interrupts, the central processing unit (CPU) stack, file systems, and network protocols [27]. The monolithic kernel has better throughput than other kernels because they handle many aspects of computer processing at the lowest level [28]. Hence, it requires to incorporate code that deals with many devices, I/O, interrupt channels, and other hardware operators [28]. The main disadvantage of this kind of architecture is that its functionality has high complexity because all its components are placed in one element [28]. So, if any programme component modified, the entire application has to be rewritten which may lead to crashes in installation [28].

3.1.2 Microkernel architecture: This architecture model is divided into a number of separated processes [28]. Some of these processes run in the kernel space and some run in the user space. The microkernel architecture provides only the main functionalities of OSs such as scheduling, inter-process communication (IPC), and synchronisation. All other OS functionalities including device drivers and system libraries operate in threads [28]. The microkernel architecture provides high implementation flexibility. So, it allows to add additional features such as plugins to the core application, and provide extensibility efficiently and easily. Moreover, it allows other OSs to be built on top of this microkernel such as Windows NT [28].

3.1.3 VM architecture: This architecture allows the user to run one OS on another OS to enable a higher degree of software portability and flexibility [29]. An OS that executes in this architecture is called guest OS, and the VM is usually called a hypervisor [29]. The hypervisor can be run on the top of an OS. The VM is often implemented as a combination of real machine and virtualisation software. The hypervisor provides access to the hardware resources for the OS through specific interfaces. Its main advantage is its portability, whereas its main disadvantage is its low system performance [29].

3.1.4 Modular architecture: This architecture allows to replace or add kernel components dynamically at run time. In a modular kernel, some components with similar functionality will be located in separate files called modules that can be configured and handled for various types of functionalities easily [28].

3.1.5 Layered architecture: This architecture consists of several layers, in which each layer is built on top of the one below [30]. The bottom layer (layer 0) is the hardware layer, and the highest layer (layer *n*) is the user interface layer. The layered architecture is manageable, easy to understand, and reliable. The main disadvantage is that they are not a very flexible architecture from an OS design perspective because it requires an appropriate definition of the different layers and precise planning of the correct placement of a layer [30].

3.2 Programming model and development environment

Programming model represents the style of programming applied to create a software which is primarily used to guide the development through programming languages. Many factors influence the choice of an appropriate programming model such as concurrency control mechanism, memory hierarchy design, and other factors [31]. There are two types of programming models: *multithreading* and *event-driven* programming [31]. *Multithreading* is the most familiar model for developers but it is not considered well suited for resource-constrained devices such as sensors. *Event-driven* programming is the most common model for writing programmes. It is useful for developing IoT devices but deemed inconvenient for traditional application developers [27].

Sometimes the behaviour of devices and their algorithms may need to be modified either because of their functionality or energy-consumption properties. Hence, the OS should be able to be reprogrammed and upgraded when required [32]. However, a *software development kit* (*SDK*) for an OS provides the software framework for the programmers for interfacing with different microcontrollers, sensors, and devices to run on IoT devices. SDK consists of a set of libraries [33]. Also, a *standard programming interface* (*API*) should be provided such as portable OS interface (POSIX) or standard template library (STL) to facilitate software development and simplify the porting of existing software [34]. In addition to that, when a code is propagating, a whole OS can be in a dysfunctional state because multiple programmes will be running concurrently. This transition time between programmes is considered wasted time, and thus, draining energy. In this regard, an efficient and robust reprogramming technique must be used to propagate and maintain the new code promptly [35].

3.3 Scheduling

The selection of the scheduling strategy is tightly bound to the capabilities of a system to fulfil real-time requirements in order to support different priorities and degrees of interaction with users [34]. There are different scheduling algorithms such as *priority-based* and *non-priority-based* schedulers. *Priority-based* schedulers are classified into *preemptive* and *non-preemptive*. *Preemptive* schedulers select the highest priority task to run even if there is another running task. *Non-preemptive* schedulers will wait till the lower running task completes its execution in the processor [36]. *Preemptive* scheduling is called so because interrupting the processes during execution is possible. The processor might switch from the ready or waiting state to the running state. *Non-preemptive* scheduling takes place when a process terminates or turns from running to a waiting state. It is called non-preemptive because processes cannot be interrupted or scheduled [36].

3.4 Memory management and performance

In a traditional OS, memory management refers to the method of allocating and deallocating memory for operations. There are two conventional memory management techniques: *static* and *dynamic* memory management methods [27]. The *static* memory management method is simple and useful when dealing with limited memory resources. However, the results are inflexible due to the run-time memory allocation which cannot occur. On the other hand, the *dynamic* memory management method is more flexible because memory can be allocated and deallocated at run time. A process memory protection is also important which means protection of one process address space from another to prevent unauthorised interfering or data loss [27]. The OS should be designed with the smallest footprint to provide the fastest performance including memory operations. Memory management and performance are significant characteristics of IoT device and is the primary reason why so many sophisticated OSs cannot be easily adapted to IoT devices [32].

3.5 Communication and networking protocols support

Another important aspect of choosing or designing an OS is the communication and networking protocol supported. Protocols specify interactions among different communicating entities. They exist at different levels in a telecommunication connection [37]. Choosing an optimised communication and networking protocol for a particular application is highly essential [37]. Especially, with a wide range of communication and networking protocols such as wireless fidelity (WiFi), ZigBee, Bluetooth, and second-generation (2G)/3G/4G cellular technologies etc. [37]. There are also several new growing communication and networking protocols such as thread as an alternative for home applications, and Whitespace TV technologies that can be applied in cities. Relying on the application and its factors such as data requirements, security, power consumption will dictate the choice of the most appropriate communication and networking protocols [38]. IoT devices can be directly connected using cellular technologies such as 2G/3G/4G

cellular, or they can be connected through a gateway, making a local area network, to get a connection to the Internet [37, 38]. The mentioned technologies of ubiquitous computing such as embedded sensors, light communication, and Internet protocols (IPs) are essential for IoT [37]. However, they impose several challenges and introduce the need for specific standards and communication protocols [37]. Processes communicate with each other within the same system or with a different one or even with other processes on heterogeneous devices. Therefore, IoT OSs must provide communication and networking protocols and heterogeneity must also be taken into consideration as well [27, 37].

3.6 Simulation support

Simulator refers to the process of imitating one OS into another OS or another device. Simulations and emulations can also make programmes to run on OSs which were not intended formerly for them [27]. The OS simulator provides varying degrees of scalability and detail for understanding the behaviour of IoT devices throughout two main aspects of a computer system's resources which are memory and process management. The primary user interface for OS simulator contains a CPU, where all codes are available to the simulator to create multiple instances of the code as separate processes [27].

3.7 Security

With the tremendous progress of IoT, more and more objects/things will be connected to the Internet. So, it is essential to ensure a secure data channel between the participated devices in IoT [39]. Also, it is a must to have a secure and reliable connection between heterogeneous devices in IoT environment [19, 39]. The IoT entities will mostly neither be a single-use, nor sole-ownership solution. The devices, things, and the control policies could have a different use, policies, administration, and connectivity domains. Consequently, devices will be ordered to have open access to some data users. On the basis of the fact that the collected information may contain personal information of the users, so it is essential to ensure the security of the devices in IoT. Generally, these IoT devices are limited in the resources, memory, and computational power. Moreover, they are more susceptible to attacks than other endpoint devices such as computers, tablets, or smartphones. One of the challenges facing the security of IoT is its components spend most of the time unattended, so that they can be physically attacked. Another challenge is that most of the communications in IoT are wireless, which makes eavesdropping extraordinarily easy. Also, IoT devices cannot apply complicated security schemes due to their limited capabilities [40].

Security is a top priority and should be considered in the hardware too as similar to conventional desktop computers, severe challenges exist. IoT devices will be expanded to most aspects of our lives, so we have to overcome these difficulties and challenges [41].

To provide security, there are several traditional security techniques such as patch upgrades, security scanning, virus checking and killing, intrusion detection, and other security techniques. However, these techniques and tools can be used for a small part of the IoT system that can hardly overcome with the rapid growth of security threats and attacks. Trusted platform module (TPM) security system of IoT is a technology designed to provide hardware-based and security-related functions. TPM hardware chips can be used with any OS [42]. A TPM chip stores cryptographic keys to be used for encryption that set on computer's motherboard. When enabled, the TPM provides full disc encryption capabilities. It becomes the ‘source of trust’ for the system to provide integrity and authentication to the boot process. It holds hard drives locked until the system completes an authentication check or a system verification. The chip includes the following trusted modules: user, perception, terminal, network, and an agent. These modules are specially designed to avoid the different security threats in the applications of IoT [42]. Moreover, security problems of IoT can be described from the network layer such as sensor attacks, sensor abnormalities, radio interference, network content security, hacker intrusion, and illegal authorisation. However, IoT may face many security issues at the application layer such as database access control, privacy protection technology, information leakage tracking technology, secure computer data destruction technology, and protection technology of secure electronic products and intellectual property of software [42].

3.8 Power consumption

Power efficiency is a crucial constraint with portable devices that run on battery source [43]. Power models are developed from physical measurements on the hardware platform. In certain situations, batteries are required to operate for 10 years at least. Even though that power utilisation is mainly dependent on hardware selection, OSs which sustain power management features are capable to efficiently manage applications to enhance battery life and allow for long sleep cycles as much as possible [44]. The OS represents the primary software component and determines the total power in many modern application executions. The selection of OS can have a significant impact on the power consumption of an OS in both active and passive manners. An active manner when there is active power management as the OS can take specific actions to control, limit, or optimise the device power consumption, whereas in a passive mode, the architectural features of the OS have an indirect effect on power consumption [44].

3.9 Supporting multimedia

Some IoT applications may require that OSs support time constrained data types such as streaming media applications, distributed games, and online virtual environments. Most traditional OSs are incompatible with these timing constraints, and they are also poorly matched to the multimedia processing since user requirements changing dynamically. Moreover, supporting commercial real-time multimedia software has the additional requirement that the OS must provide for controlling and communicating resource usage among independent real-time activities. Moreover, audio and video sources generate data that needs exhaustive processing to be compressed and decoded for streaming. The data exchange occurs from these sources to other destinations such as loudspeakers and video situated on the computer or at another remote station. Multimedia data is processed on the way from the source to the sink through copying, moving, and transmitting operations. OSs manage the resources where the data processing occurs. The OS must be able to process audio and video, and the amount of data that has to be transferred can be substantial [45].

Fig. 1 summarises the main features and aspects discussed in this section. We use these design and development aspects to evaluate the open-source IoT OSs discussed in our survey.

4 Classifications of IoT OSs

OSs can be classified based on different criteria. For example, they can be classified by their source code either open or closed. Also, they can be categorised according to the purpose of their design into either specially designed IoT OS or customised version of an existing OS. Another classification for OSs is to divide them into Linux based or non-Linux based. Finally, OSs can be classified based on the targeted devices whether high-end IoT devices or low-end IoT devices.

In open sources, the source is available to anyone so that the user can use the freely distributed code, and modify it even for commercial purposes to fit a particular requirement such as Linux OS. Closed source OSs use code that is implemented by private parties and is kept unpublished to have full control over the OS and keep its proprietary [9]. An OS is usually classified into two different software ecosystems. The first is Linux based, and the second is non-Linux based. Linux OS is an open-source cross-platform based on Unix OS that can be installed on PCs, servers, and other hardware. On the other hand, non-Linux OS is not based on Linux or Unix; rather it depends on other OSs such as Windows and ReactOS.

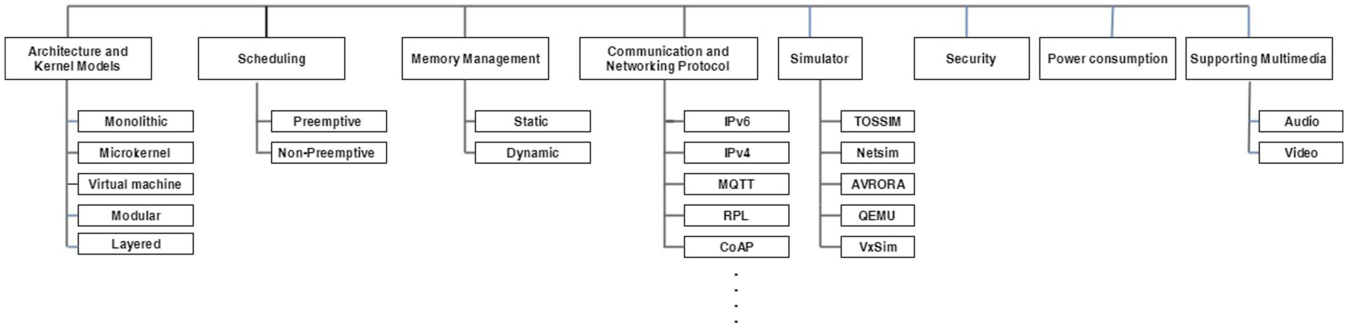

Fig. 1 *Overview of the designing and developmental aspects of IoT OSs*

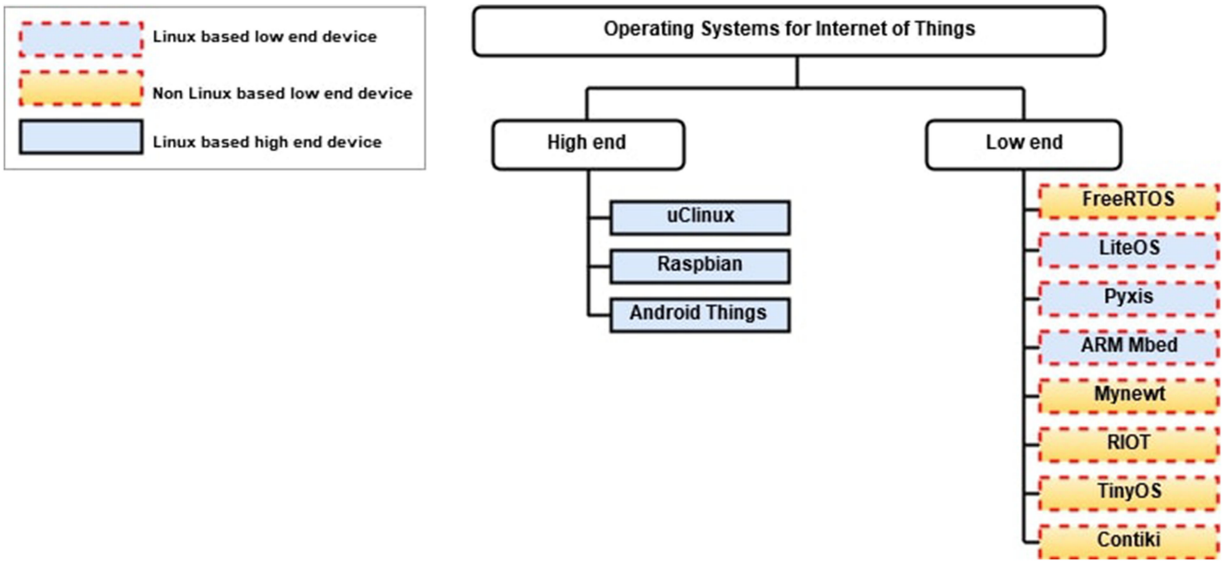

Fig. 2 *Proposed taxonomy of open-source IoT OSs*

The last classification of IoT OSs is based on the capability and performance of IoT devices; they can be classified into two categories. The first category is for high-end IoT devices which include single-board computers such as the Raspberry Pi (RPi). High-end IoT devices can run traditional OSs such as Linux. The second category is for low-end IoT devices, which have limited resources and cannot run by traditional OSs. An example of low-end IoT devices is Arduino.

In this paper, we target the most used and state-of-the-art open-source IoT OSs due to the limit on the number of pages. For the closed source IoT OSs, we recently published a paper on this topic [46]. Fig. 2 shows a proposed taxonomy of open-source IoT OSs based on low-end or high-end IoT devices, and on Linux based and non-Linux based from a high-level perspective either.

5 OSs for low-end IoT devices

In this section, we will describe the most widely used low-end OSs for IoT devices from the aspects and criteria presented in Section 3. The main objective of this section is to provide an exhaustiveness understanding of each low-end IoT OS.

5.1 TinyOS

TinyOS is an *open-source non-Linux*-based OS designed explicitly for low-end IoT devices, embedded and wireless devices such as sensor node networks, smart buildings, and smart sensory platforms [47]. TinyOS is built based on a set reusable software component [47]. It is written using NesC programming language, which has a similar syntax to C language [47]. Each TinyOS component has a frame and a structure of private variables. These components have three computational abstractions: commands, events, and tasks [47, 48]. Commands are used to call a component to do a specific task. Events are mechanisms for entering component communication, while tasks are used to represent component concurrency [49].

5.1.1 Architecture and kernel models: TinyOS has a *monolithic* architecture and uses a component-based architecture that depends on the application requirements. This reduces the size of the code needed to setup hardware [47]. Different components are grouped with the scheduler to run on the mote platform. The mote platform has very insufficient physical resources depending on which components are active. Typical TinyOS motes consist of a 1 microprocessor without interlocked pipeline stages (MIPS) processor and tens of kilobytes of storage. A component is an independent computational element that shows one or more interfaces. Components have three computational abstractions: commands, events, and tasks. Mechanisms for inter-component communication are commands and events, whereas tasks are used to express intra-component concurrency. A command is a request to perform some service while the event signals represent the completion of service [27, 47]. Fig. 3 shows the architecture of TinyOS. The scheduler schedules operation of those components. Each component consists of four parts: command handlers, event handlers, an encapsulated fixed-size frame, and a group of tasks. Commands and tasks are performed in the context of the frame and operate on its state. Each component declares its commands and events to allow the modularity and easy interaction with other components [32].

5.1.2 Programming model and development environment: TinyOS supports an *event-driven* concurrency model which consists of split-phase interfaces, deferred computation, and asynchronous events [31]. TinyOS is programmed in NesC for memory limitations of sensor networks which are similar but not compatible with the C language. It allows writing pieces of reusable code which explicitly indicates their

dependencies [47]. Also, TinyOS uses a mechanism called Trickle. Trickle is an algorithm used for propagating and maintaining code updates when needed. Trickle applies a polite gossip policy, where nodes occasionally broadcast code to all neighbour nodes, and remain silent. When a node hears an older summary of its own, it broadcasts an update rather than sending a network signal with packets. Then, the algorithm manages the process of sending, so each node only hears a small trickle of packets which is just enough to stay up to date. Trickle propagates new code within seconds and makes the maintenance cost less in terms of time (propagation of new code to all neighbours' nodes) [35]. The primary challenge in TinyOS development is the creation of flexible and reusable components [49].

5.1.3 Scheduling: The task scheduler in TinyOS is a simple *non-preemptive first-in–first-out* (*FIFO*) scheduler using a bounded size scheduling data structure. The TinyOS scheduler sets the processor to sleep when the tasks are completed; to maximise CPU utilisation as well as the OS performance [50].

5.1.4 Memory management and performance: TinyOS uses *static memory* allocation with memory protection. There are no concepts of dynamic memory allocation such as hidden heaps, dynamic memory, or function pointers because TinyOS programmes are organised in components and are written in NesC language [47]. TinyOS has a small footprint as it uses a *non-preemptive* FIFO task scheduling. It applies synchronisation clock in software, which increases the number of entries in the task queue at compile time when the system begins with 1, 32 kHz, or 1 MHz.

5.1.5 Communication and networking protocols support: TinyOS has built-in support for common network protocols such as transmission control protocol (TCP), user datagram protocol (UDP), ICMPv6, IPv6, IPv6 over low-power WPAN (6LoWPAN), IPv6 routing protocol for low-power and lossy networks (RPL), and constrained application protocol (CoAP), in addition to hydrogen routing protocol that is used for reliable communication [51].

5.1.6 Simulation support: To validate the analysis model of TinyOS applications, a TinyOS Simulation (TOSSIM) simulation environment has been developed. TinyOS simulation (TOSSIM) provides a high flexibility simulation of TinyOS applications which work by replacing components with simulation implementations [47]. Moreover, TOSSIM provides developers an integrated environment of the network and troubleshooting capabilities. Server-side applications can be connected to a TOSSIM proxy only if it is a real sensor network. Hence, facilitating the transition between the simulation and real deployments [47]. TOSSIM also provides support integration for troubleshooting and debugging applications directly on the mote. Unfortunately, TOSSIM does not support gathering power measurements [47].

5.1.7 Security: TinyOS uses TinySec library which was developed using the NesC programming language and implemented by the link layer to provide confidentiality, message authentication, integrity, and semantic security [52]. The default block cipher encryption in TinySec is Skipjack algorithm that is used with cipher block chaining (CBC-CS) method. Skipjack has an 80 bit key length that provides immunity to brute force attacks [52]. Skipjack generates message authentication code (MAC) method which utilises CBC-MAC [52]. However, CBC-MAC has security lacks since it furnishes semantic security with an 8 B introduction vector which includes only a 2 B counter overhead per packet [52]. TinySec holds <10% energy, inactivity, and transfer speed overhead [52].

5.1.8 Power consumption: TinyOS provides efficient low-power consumption operation and limited storage using a simple execution model [44]. TinyOS execution model is based on split-phase operations and interrupts handlers. It allows the scheduler to decrease its random access memory (RAM) utilisation and easily maintains sync code. This avoids the need for threads and allowing all programmes to execute on a single stack. However, it implies that if one sync code runs for a long term, then it prevents other sync code from running; which can negatively influence system responsiveness [44].

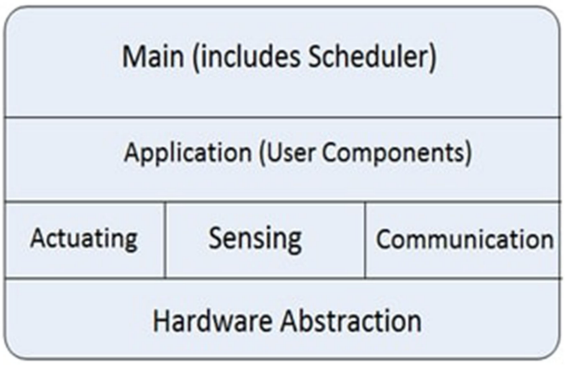

Fig. 3 *Architecture of TinyOS (reproduced from [27])*

5.1.9 Supporting multimedia: TinyOS supports full IP network stack, with standard IP protocols such as UDP, TCP, and hypertext transfer protocol (HTTP) that are used to stream multimedia content. The frameworks available for video codecs and multimedia streaming are limited in this OS and have no extended support. Moreover, real-time transport protocol (RTP) is not found in the base of TinyOS [53].

5.1.10 More about TinyOS: TinyOS applies fully non-blocking split-phase operations that enable developers to re-define the kernel API by choosing an existing set of operations or by implementing one system call stack. In this method, all I/O operations that last longer than a few hundred microseconds are asynchronous and have a callback known as deferred procedure calls [47].

5.2 Contiki OS

Contiki is an *open-source non-Linux*-based OS for *low-end* IoT devices designed especially for IoT. It is lightweight, highly portable, and multitasking OS that runs on tiny low-power microcontrollers with minimal memory. Contiki OS is written in C programming language. It uses 2 kB of RAM and 40 kB of read-only memory (ROM). Nowadays, Contiki can be run on various hardware platforms such as Alf and Vegard RISC processor (AVR), MSP430, and Z80 [31, 52, 54].

5.2.1 Architecture and kernel models: In contrast to TinyOS. Contiki OS has a *modular* architecture [25]. The core of Contiki OS mainly consists of multiple lightweight event schedulers and a polling mechanism. The event schedule is responsible for dispatching events to run processes and periodically calls processes' polling handlers, which identifies the action of the polled process [31]. On the other hand, the polling mechanism identifies high priority events. Polling mechanism is used by processes that operate near the hardware to check the status updates of hardware devices. All processes that implement a poll handler are requested in order of their priority [54]. Fig. 4 shows the architecture of Contiki OS. Contiki OS contains sensor data handling, communication protocols, and device drivers as services. Each service has its interface and implementation.

5.2.2 Programming model and development environment: Unlike TinyOS, the programming models in Contiki support both *multithreading* and *event-driven* using protothreads. The main advantage of protothreads is their very minimal memory overhead with no extra stack for a thread. Since events run to completion, Contiki does not allow interruption of handlers to post new events, and it does not allow process synchronisation [27]. Programming models with Contiki are defined by events in a way

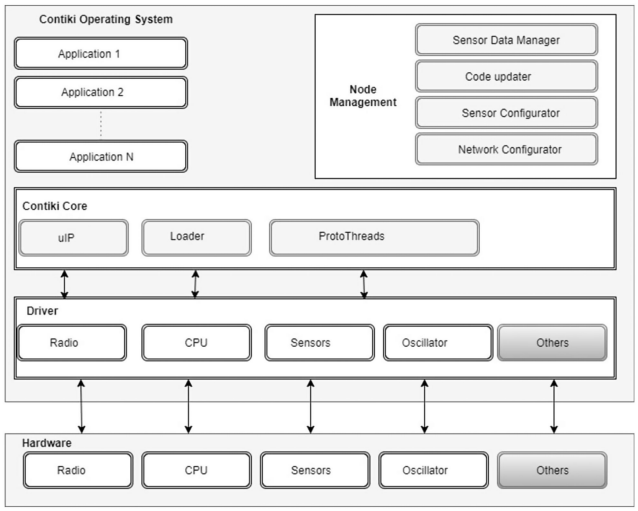

Fig. 4 *Architecture of Contiki OS (reproduced from [27])*

that all tasks are executed in the same context [52]. Protothreads mechanism runs on top of the event-driven kernel. A protothread process is invoked whenever a process receives an event, and the protothreads mechanism decides which memory should be allocated [55]. Contiki is implemented by system libraries which are connected with programmes. Programmes can be connected with libraries in three ways. The first way, the programmes can be statically connected with libraries that are part of Contiki core. Second, programmes can be statically linked with libraries that are part of the loadable programme. Third, programmes can call services using a specific library. Libraries that are applied as services can be replaced dynamically at the running time. Consider a programme that uses the *memcpy()* and *atoi()* functions to copy memory and to convert strings to integers, respectively. The *memcpy()* function is a frequently used C library function; whereas *atoi()* is used less often. Therefore, in this example, *memcpy()* has been included in the Contiki core but not *atoi()*. The *memcpy()* function will be linked against its static address in the core when the programme is linked to produce a binary. The object code for the part of the C library that implements the *atoi()* function must, however, be included in the binary programme [31]. Moreover, Contiki uses loadable modules to perform dynamic code reprogramming and upgrading. With loadable modules, only specific parts of the codes need to be modified when a single programme is changed [56]. Besides, Contiki provides a command-line shell which is useful during development and debugging of Contiki systems [57].

5.2.3 Scheduling: The scheduling used in Contiki OS is similar to TinyOS, in which both use FIFO scheduling strategy. In Contiki, all *event-driven* scheduling is done at a single level and events (*preemptive multitasking*); events are executed as they arrive [31].

5.2.4 Memory management and performance: Unlike TinyOS, Contiki supports *dynamic allocation* or *deallocation* of memory through *mmeb()* and *mmem()* as well as *malloc()*. The *memb()* memory block allocator is the most frequently used. The *mmem()* managed memory allocator is used infrequently and it uses the standard C library *malloc()* heap memory allocator [58]. In addition, Contiki uses Contiki coffee file system technique for data storage inside the sensor network. It allows multiple files to exist on the same physical onboard flash memory [58].

5.2.5 Communication and networking protocols support: Contiki OS supports many protocols such as CoAP and the message queue telemetry transport (MQTT) [59]. In addition to that, the two main communication stacks are uIP and Rime stack that consists of a set of custom lightweight protocols for power constrained wireless networks [59]. Contiki supports a full IP network stack with standard IP protocols such as UDP, TCP, and HTTP [60]. Also, it has support for 6LoWPAN adaptation layer, the RPL IPv6 multi-hop routing protocol, and the CoAP RESTful application-layer protocol [61].

5.2.6 Simulation support: Cooja simulator supports Contiki, which is a useful tool for Contiki OS application development. Cooja makes simulation colossally less demanding by providing a simulation environment to allow testing of code before running it on the target hardware devices [62].

5.2.7 Security: Contiki OS uses ContikiSec transport layer security (TLS)/datagram transport layer security (DTLS), which is a secure network layer, and contains three modes: authentication, confidentiality, and integrity in communication. ContikiSec uses low-energy utilisation and security while complying with a little memory footprint [52].

5.2.8 Power consumption: Contiki is intended to run on low-power devices that may need to keep running for quite long time on batteries [63]. To help the improvement of low-power devices power consumption, Contiki provides software-based power profiling mechanism for estimating the system power utilisation and for knowing where the power was consumed giving power awareness [62].

5.2.9 Supporting multimedia: Contiki supports full IP network stack protocols such as UDP, TCP, and HTTP that are used to stream multimedia contents. The frameworks available for video codecs and multimedia streaming are limited in this OS and has no extended support. Moreover, RTP protocol is not found in the base of Contiki.

5.2.10 More about contiki OS: One of the essential features of Contiki is *dynamic loading*; which is the ability to link modules at run time [64]. Contiki transferred nodes can be battery-operated because of the ContikiMAC radio duty cycling mechanism which allows nodes to sleep between each relayed message [65]. Unlike TinyOS that has no blocking operations, Contiki provides some conditional blocking of functions in a sequential instruction block.

5.3 Real-time OS for IoT (RIOT) OS

The RIOT is known as 'the friendly OS for the IoT'. RIOT is an *open-source non-Linux*-based OS specialised for *low-end* IoT devices with a minimum of 1.5 kB of RAM and 5 kB of ROM [66]. RIOT provides a uniform abstraction over the details of different IoT hardwares. It was developed by a grassroots community using C programming language. RIOT can run on various platforms including embedded systems, and it is easy to use. It supports many functionalities such as interruption handling, memory management, IPC, and synchronisation. Moreover, RIOT has many advantages such as reliability, predictability, performance, and scalability [67].

5.3.1 Architecture and kernel models: In contrast to the other OSs such as TinyOS or Contiki. RIOT has a *microkernel* architecture, which has been designed to work on several IoT platforms with different CPU architectures (32 bit, 16 bit, 8 bit) such as ARMv7, ARM Cortex-M0 + , MSP430, and some recent AVR microcontrollers. The microkernel architecture of RIOT OS was developed using C + +, and it supports full *multithreading* that provides a developer-friendly API and allows C + + and ANSI C application programming. RIOT kernel will never crash because it supports error device drivers. The architecture design of RIOT also contains POSIX compliance [42, 68]. Fig. 5 shows the structure of RIOT, which is divided into four layers. The first layer is the kernel; which consists of the scheduler, inter-process communication, threading, thread synchronisation, supporting data structures and type definitions. The second layer is platform specific code (CPU boards), which contains the configuration for that particular CPU. The third layer is device drivers, which consist of the drivers for external devices such as network interfaces, sensors, and actuators. The fourth layer comprises of libraries,

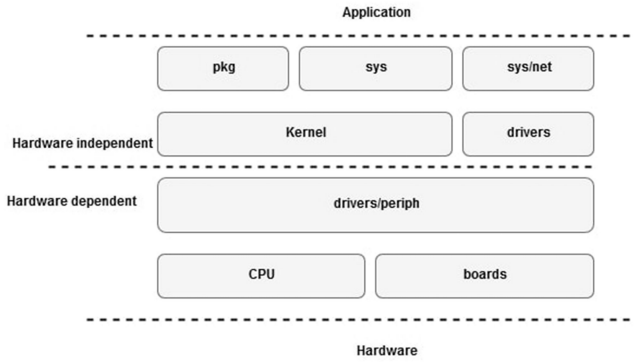

Fig. 5 *Architecture of RIOT OS (reproduced from [66])*

network code, and applications for demonstrating features and testing. Moreover, this layer includes a collection of scripts for various tasks as well as predefined environment documentation (doc) [67].

5.3.2 Programming model and development environment: RIOT is similar to Contiki that it also supports *preemptive multithreading*. RIOT is developed using standard programming languages such as ANSI C and C++ [69]. With RIOT, developers can code the application once and run it on various IoT hardware devices. Moreover, RIOT provides common programmer APIs such as Berkeley software distribution (BSD) sockets or POSIX thread (pthread) functionalities [70]. Besides, RIOT can run and debug the same as in Linux and MacOS using a set of popular debugging tools such as GNU Debugger (GDB) and Valgrind [67]. The C++ programming capabilities used in RIOT allow RIOT to use powerful libraries such as the Wiselib, which contains algorithms for routing, clustering, time sync, localisation, and security. RIOT has other programming features such as *dynamic linking* support, Python interpreter, and energy profiler [71]. Also, RIOT provides virtualisation, where the code and application can run as a simple Unix process. RIOT uses Wireshark for packet analysing [67].

5.3.3 Scheduling: Together with Contiki. RIOT implements *preemptive* priority-based and tickless scheduling, where each task has a priority in execution that helps the scheduler to select the highest priority task to run on CPU. RIOT tasks with the highest priority are executed first, and if there are more than one high priority tasks, a round-robin (RR) mechanism will be used [71].

5.3.4 Memory management and performance: In RIOT OS, both *dynamic* and *static memory* allocations are provided for applications [72]. RIOT OS does not have a memory management unit (MMU) or floating point unit. However, it has a low memory footprint in the order of a few kBs [66, 68].

5.3.5 Communication and networking protocols support: RIOT OS supports several networking protocols including TCP/IP v4 and v6 and the latest standards for connecting constrained systems to the internet engineering taskforce (IETF) 6LoWPAN [72]. In addition to that, RIOT has built-in support for other IoT-related network protocols such as CoAP and RPL [73].

5.3.6 Simulation support: At the time of writing this paper, RIOT OS does not have a simulator. Rather, we can have a full-scale simulation for RIOT applications from Contiki–Cooja simulator for IoT [67].

5.3.7 Security: RIOT supports powerful attack detection capabilities called secure cyber-physical ecosystem (CPS). CPS is a system that interacts, monitors, and controls smart objects through complicated processes. When an attack is detected, then the reaction to it occurs [69].

5.3.8 Power consumption: The simplicity of microkernel architecture of RIOT is the main characteristic to enable maximum energy efficiency [67]. RIOT context switching can happen in two situations. The first situation is when a corresponding kernel operation gets called by itself such as a mutex locking. The second situation is when an interruption happens in a thread switch. Fortunately, the first situation will happen once in a while. Then, when RIOT's kernel gets called again, a task switch can be performed in very few clock cycles [71]. Moreover, RIOT OS is an exceptionally suited software platform to optimise energy consumption on battery-powered microcontroller (MCU)-based devices and consumes less energy [68].

5.3.9 Supporting multimedia: RIOT supports full TCP/IP network stack protocols such as UDP, TCP, and HTTP that are used to stream multimedia content. It has many onboard modules which are essential for developing multimedia applications [71].

5.4 LiteOS

LiteOS is an *open-source Linux*-based lightweight OS designed to run on low-power devices. This makes LiteOS suitable for a wide range of areas including wearable, smart homes, connected vehicles, and microcontrollers. LiteOS can be installed on devices that run by Google Android OS, and it can connect with other third-party devices. It is developed purposely to provide a Unix-like OS for IoT developers and to provide programmers with familiar programming paradigms such as a hierarchical file system developed using LiteC programming language and a Unix-like shell [27, 74].

5.4.1 Architecture and kernel models: In contrast to RIOT, LiteOS has a *modular architecture* divided into three subsystems; LiteShell, LiteFS, and the kernel [32] as shown in Fig. 6. LiteShell is a Unix-like shell that provides support for shell commands such as file management, process management, and debugging. LiteShell resides on a base station or a PC. This leverage allows more complex commands as the base station, or PC has abundant resources. The LiteShell can only be used with user intervention. Some local processing is done on the user command by the shell and then transmitted wirelessly to the intended IoT node. The IoT node does the required processing of the command and sends a response back which is then displayed to the user. When a mote does not carry out the commands, an error code is returned. The second architectural component of LiteOS is its file system, LiteFS, which consists of sensor nodes as a file and it mounts a sensor network as a directory and then lists all one hop sensor nodes as a file. A user on the base station can use this directory structure just as the traditional Unix directory structure and can also use legitimate commands. The third subsystem of LiteOS is the kernel which resides on the IoT node. The kernel supports concurrency *multithreading*, *dynamic loading*, and uses RR and *priority scheduling*, which allows developers to register event handlers through callback functions [27].

5.4.2 Programming model and development environment: LiteOS is a *multitasking* OS that supports multithreading. In LiteOS, processes run applications in separate threads. Each thread has its allocated memory which helps in protecting the memory. LiteOS also provides support for event handling [27]. Also, it supports *dynamic reprogramming* and replacement mechanism through the user application. Reprogramming can be performed either if the source code of the OS is available or not. If it is available, it will be easily recompiled with new memory settings, and all pointers of the old version will be redirected, whereas if the source code is not available, it uses a differential patching mechanism to upgrade the older version. Also, LiteOS supports online debugging including variable watches and a vast number of breakpoints. Additionally, it contains extensive development libraries [76].

5.4.3 Scheduling: LiteOS implements both *priority-based* and RR scheduling in the kernel. The task to be executed is chosen

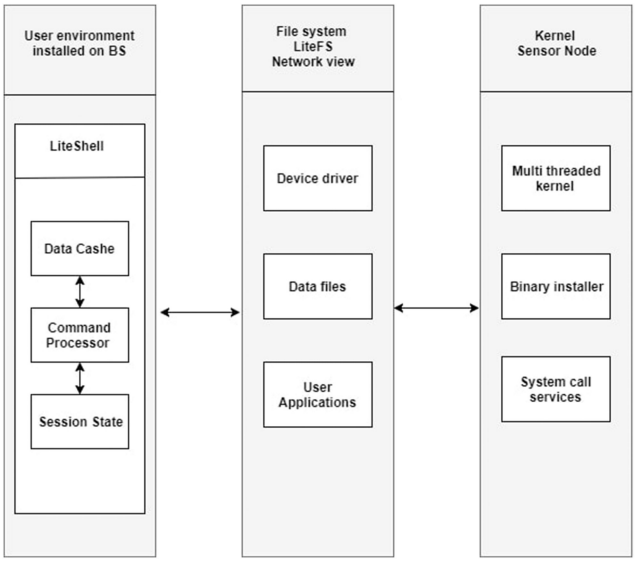

Fig. 6 *Architecture of LiteOS (reproduced from [75])*

from the ready queue using *priority-based* scheduling. When a task requires a resource that is not available currently, the task allows interrupts and goes to sleep mode [32].

5.4.4 Memory management and performance: LiteOS implements *dynamic memory* allocation with an almost zero overhead through system calls using *malloc()* and *free()* API functions. The *malloc()* function allocates memory using a pointer. If the size of memory is zero, then *malloc()* returns to either NULL or a unique pointer value that can be passed to *free()* function. The *free()* function frees the memory space, which must have been returned by a previous call to *malloc()* function. This enables adapting the size of dynamic memory as required by an application [77].

5.4.5 Communication and networking protocols support: LiteOS does not have any built-in networking protocols that support real-time applications [77]. LiteOS provides support for long-distance connection based on technologies such as long-term evolution (LTE) and NodeB (NB)-IoT, and short-distance connection based on communication protocols such as ZigBee and 6LoWPAN [74].

5.4.6 Simulation support: AVRORA simulator can be used to emulate LiteOS on physical IoT devices. AVRORA is a set of simulation tools for programmes written for the AVR microcontroller. AVRORA contains an adaptable system for simulating and prototyping programmes, which allows Java API experimentation, profiling, and investigation [78].

5.4.7 Security: In terms of security, LiteOS provides independent user space and application separation through a set of system calls. The authentication mechanism is needed between the base station and mounted motes, especially low-cost authentication mechanisms. To ensure the security of communications between the sensors and systems, LiteOS has a security component by embedding Hi3519 chip Huawei Lite OS that can be implemented to security cameras and portable high-definition camera [74].

5.4.8 Power consumption: LiteOS supports ultra-low-power consumption; it can be used to run MicaZ motes having 128 B of flash memory, 4 kB of RAM, and 8 MHz CPU. LiteOS battery can power a device for five years or more [74, 79].

5.4.9 Supporting multimedia: LiteOS does not support any implementation of networking protocols that support multimedia applications [27].

5.4.10 More about LiteOS: LiteOS offers many novel features including zero configuration, auto-discovery, auto-networking, fast boot, a hierarchical file system, a wireless shell interface for user interaction and real-time operations. It also provides extensive wireless support including LTE and mesh networking [60]. LiteOS supports Windows XP, Windows Vista, and Linux in addition to both MicaZ and IRIS nodes. Moreover, it supports both plug-and-play routing stack. Other advantages of LiteOS are that it has an extremely lightweight event logging and it has a built-in hierarchical file system. Finally, LiteOS snapshots a thread state and can restore it to a previous state [74].

5.5 FreeRTOS

FreeRTOS is an *open-source non-Linux*, *multitasking*, *preemptive*, and *mini real-time* OS for *low-end* IoT devices mainly developed using C and some assembly functions [80]. It is very scalable, simple, easy to use, and highly portable. Its main strength is the small kernel size, which makes it possible to run different small applications. Another strength is that FreeRTOS supports an extensive variety of hardware architectures, which make it a good choice to be used with different IoT applications [81, 82].

5.5.1 Architecture and kernel models: FreeRTOS has a *microkernel* real-time operating system (RTOS) architecture, which is designed to be small, simple, and easy to use [28]. FreeRTOS kernel is specifically designed for embedded processors. FreeRTOS consists of hardware layer, device driver, FreeRTOS kernel, and the tasks to be performed. Fig. 5 shows the architecture of FreeRTOS.

5.5.2 Programming model and development environment: FreeRTOS is professionally developed. It is free open source and fully supported, even when used in commercial applications. However, FreeRTOS decreased the debugging efforts due to the lack of hardware abstraction layer (HAL) by using the STM32Cube MCU firmware [83]. In addition, FreeRTOS supports multiple threads, mutexes, semaphores, and software timers [77]. All FreeRTOS kernels contain three or four C files (depends on the usage of coroutines) which are; (i) list.c, (ii) queue.c, and (iii) tasks.c, and one microcontroller source file are needed. The tasks.c performs most of the scheduler assembler functionalities. The list.c defines the structures and functions to be used by file tasks.c. The queue.c performs thread-safe queues that are used for synchronisation and intertask communication. These files are included in the .zip file related only to the applications which make the code more readable and maintainable. FreeRTOS API is designed to be simple and easy to use [84].

5.5.3 Scheduling: The scheduler used in FreeRTOS is either a *fixed prioritised preemptive* or *cooperative scheduler* strategies depending on the configuration of the scheduler [77]. A *fixed-priority* scheduling ensures that the processor executes the highest priority task that is ready to be executed at any given time by using the RR scheduling policy [77].

5.5.4 Memory management and performance: FreeRTOS supports *dynamic memory* allocation, and its memory footprint is small [84].

5.5.5 Communication and networking protocols support: FreeRTOS uses 6LoWPAN and CoAP networking protocols [85] in addition to the open source, and thread-safe TCP/IP stack for FreeRTOS called FreeRTOS + TCP [85].

5.5.6 Simulation support: FreeRTOS can be simulated in both Windows and Linux OSs using the following simulators. Win32 simulator using visual studio 2015 for Windows OS and developed by Dushara Jayasinghe [84], whereas POSIX/Linux simulator is used for Linux OS using GNU compiler collection (GCC) and Eclipse provided by William Davy [84].

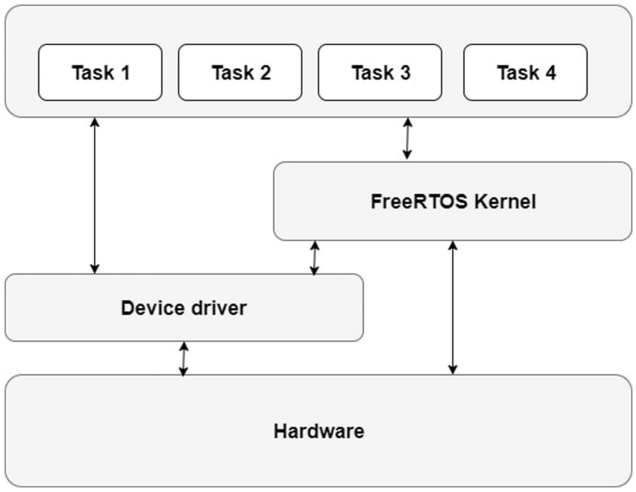

Fig. 7 *Architecture of FreeRTOS (reproduced from [80])*

5.5.7 Security: FreeRTOS uses WolfSSL, which is a lightweight TLS/secure sockets layer (SSL) library. WolfSSL is used to provide security, authentication, integrity, and confidentiality of communications over the FreeRTOS. WolfSSL is best suited for the embedded system since it is 20 times smaller in footprint compared with the OpenSSL library [84, 86].

5.5.8 Power consumption: In order for the microcontroller to function in the low-power state, FreeRITOS uses idle task hook. Using this method, it decreases power consumption through process tick interruptions. However, if too many tick interruptions occur, the power consumption will increase more than the power saving limit. The tickless of FreeRTOS is an idle mode that stops the periodic tick interrupt during idle periods. This makes a correcting adjustment to the RTOS tick count value when the tick interrupt is restarted [84] (Fig. 7).

5.5.9 Supporting multimedia: FreeRTOS is designed for low-powered applications, which leaves the expensive encoding/decoding process to external devices. However, the TCP and UDP stack support in FreeRTOS can be used to implement some applications limited to image contents using Javascript and HTML5.

5.5.10 More about FreeRTOS: This OS is recommended to handle time-critical tasks that require user control and sensor monitoring as it can be handled through multiple threads, software timers, and semaphores along with tickless mode for low consumption of resources while running various applications.

5.6 Apache Mynewt OS

Apache Mynewt is an *open source*, *Linux-based*, and *real-time* OS originally developed by Runtime Inc. and hosted by Apache. Mynewt OS targets *low-end* devices that have limited memory and storage capabilities that need to operate for a long time under power constraints. Additionally, Mynewt OS has many powerful features such as precise reconfigurability of concurrent connections and granular power controls [87].

5.6.1 Architecture and kernel models: Mynewt OS has a *modular architecture*. Mynewt targets ARM Cortex M0–M4 and RISC-V architectures with a plan to extend the hardware support to MIPS architecture [88]. Also, Mynewt OS is supported on the Arduino Zero, Arduino Zero Pro, and Arduino M0 Pro processors [88, 89].

5.6.2 Programming model and development environment: Mynewt OS supports *multithreading* tasks [89]. Developers can debug code by setting breakpoints, avoiding stack smashes, and eliminating stolen interrupts [89].

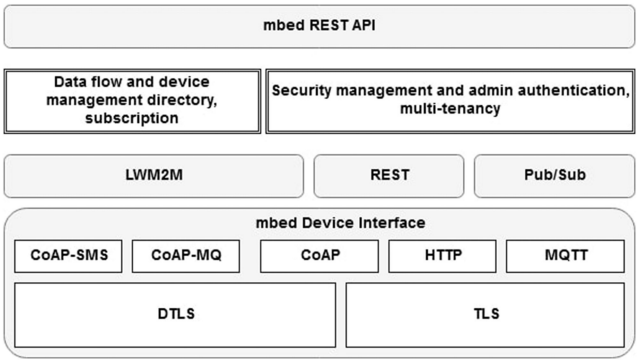

Fig. 8 *Architecture of ARM Mbed OS (reproduced from [91])*

5.6.3 Scheduling: Mynewt OS supports tickless *preemptive priority-based* scheduling [89].

5.6.4 Memory management and performance: Mynewt OS supports memory heap and memory pool allocation [87].

5.6.5 Communication and networking protocols support: Mynewt OS provides full support for the TCP/IP suite. It also supports protocols for constrained networks such as CoAP and 6LoWPAN [87]. Also, Mynewt OS has full implementation of the Bluetooth low-energy 4.2 stack, Bluetooth mesh, LoRa PHY, and LoRaWAN [87].

5.6.6 Simulation support: Mynewt can be simulated using the Quick Emulator (QEMU).

5.6.7 Security: Apache Mynewt OS enables secure remote updates to maintain ongoing security. Also, Mynewt OS provides safe bootloader to verify firmware integrity and authenticity [89], and uses security manager protocol for pairing and transport specific key distribution for securing radio communication [89].

5.6.8 Power consumption: Mynewt has low-power consumption; devices operate in sleeping mode to conserve battery power and maximise power usage [87].

5.6.9 Supporting multimedia: At the time of writing this paper, there is no support for multimedia contents or devices in Apache Mynewt OS.

5.6.10 More about Apache Mynewt: HAL is used in Mynewt to provide a uniform interface for peripherals across different microcontrollers, which allow direct access to peripherals for granular power control [89].

5.6.11 ARM Mbed OS: ARM Mbed is an *open-source Linux-based* OS for *low-end* IoT devices licenced under Apache Licence 2. ARM cores introduce it for 32 bit ARM Cortex M microcontrollers. It supports all essential open standards for connectivity and device management. ARM Mbed OS provides a platform that includes the following features: connectivity, security, cloud management services, and device management functionalities that are required by IoT devices [33, 90].

5.6.12 Architecture and kernel models: ARM Mbed OS supports *monolithic* architecture [25]. ARM Mbed OS runs on 32 bit ARM embedded architecture and supports a few other platforms [91]. Fig. 8 shows the architecture of ARM Mbed OS and its different layers.

5.6.13 Programming model and development environment: ARM Mbed OS uses a *single thread* and adopts an *event-driven* programming model [91]. ARM Mbed OS is developed using C and C + + programming languages. Moreover, it has many programming features that allow developers to select

from different types of microcontrollers. Also, Mbed API can keep application code readable, simple, and portable [91].

5.6.14 Scheduling: Mbed OS includes a basic *non-preemptive* scheduler with limited synchronisation and communication primitives to support its communication and cloud protocols [91].

5.6.15 Memory management and performance: ARM Mbed OS supports *dynamic* memory allocation [91].

5.6.16 Communication and networking protocols support: Mbed OS provides support for many communication protocols such as WiFi, Bluetooth low energy, Lightweight M2M (LwM2M), and Ethernet [92]. Also, it supports several networking protocols including HTTP/CoAP stack with TLS and DTLS for end-to-end IP (v4 and v6) security with 6LoWPAN, SSL, and MQTT [10, 93].

5.6.17 Simulation support: Mbed OS can be simulated using QEMU, which is a generic and open-source machine emulator and virtualiser [94].

5.6.18 Security: Untrusted and malicious codes are blocked in Mbed OS for IoT platforms as the communications between device and cloud, and the life cycle of the system itself occurs through uVisor which separates security domains on Arm Cortex-M3, M4, and M7 microcontrollers with a memory protection unit [91]. There are two mechanisms to guarantee security in Mbed by integration with the application development. The first is Mbed TLS for cryptographic and SSL/TLS capabilities. The second is Mbed OS uVisor for hardware-enforced secure domains [94].

5.6.19 Power consumption: The Mbed OS has support for an advanced power management technique that increases power efficiency and improves throughput [94]. This technique allows to turn off some external devices and processor to power down unused devices inside the processor chip and enter low-power sleep modes. It is also possible to modify the clock rate on Mbed OS from 128 to 48 MHz [94].

5.6.20 Supporting multimedia: Imaging modules can be integrated easily to Mbed OS through a set of imaging sensors available as an extension. Streaming media contents can be achieved using a simple TCP connection or any other high-level communication protocols [94].

5.6.21 More about ARM Mbed OS: The Arm Mbed device connector can connect IoT devices to the cloud without needing any additional infrastructure [91].

6 OSs for high-end IoT devices

In this section, we will describe the most widely used OSs for *high-end* IoT devices from the aspects and criteria presented in Section 3.

6.1 uClinux OS

uClinux is an *open-source Linux-based* OS for *high-end* devices. It is an extension to the Linux kernel. uClinux kernel includes a collection of Linux 2.x kernel releases intended for single microcontrollers without an MMU. Also, it has a set of user applications, libraries, and toolchains. This OS needs special support for inter-processor communication [77, 78].

6.1.1 Architecture and kernel models: uClinux OS follows a *monolithic* architecture [95]. uClinux kernel supports various CPU platforms such as ColdFire, Axis ETRAX, and others. The primary difference with Linux OS is it has MMU-less. However, uClinux OS additionally supports different file systems including files designed especially for the embedded solutions similar to Linux OS. There have been previous attempts to achieve compatibility between uClinux and Linux to make uClinux as similar as possible to Linux. These attempts proposed several applications that developed under general public licence. These applications should support MMU-less version of Linux with slightly few changes [95]. Fig. 9 shows the architecture of uClinux OS.

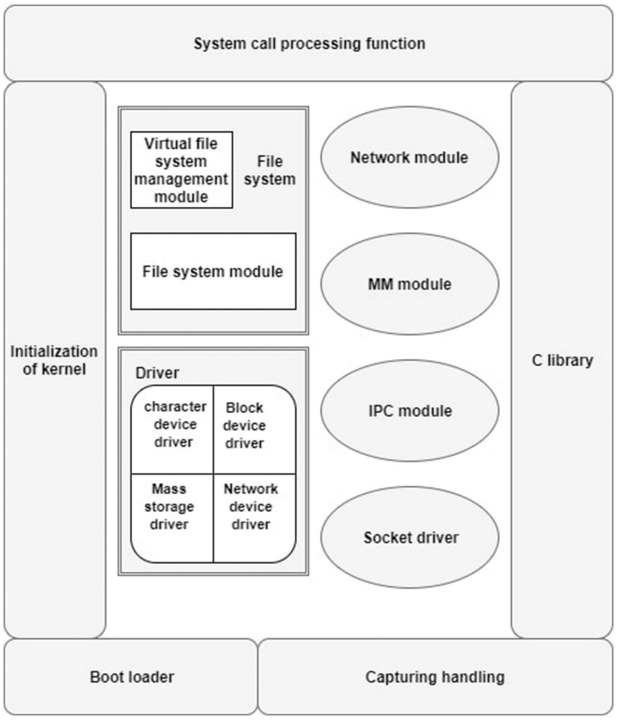

Fig. 9 *Architecture of uClinux OS (reproduced from [96])*

6.1.2 Programming model and development environment: uClinux OS supports *multithreading* programming model [97]. uClinux includes a cross-compiler platform that is built from the GNU compiler tools (GCC). Its architecture is x86 which is often built without the ability to be modified on any uClinux target. Debugging can be performed using GNU debugger (gdb) [98].

6.1.3 Scheduling: uClinux implements *priority-based preemptive* scheduling [77].

6.1.4 Memory management and performance: uClinux is derived based on Linux 2.0 kernel and is designed for microcontrollers without MMUs, so any process can read and write other process memory. It provides *dynamic* and *static* memory allocation [99]. However, uClinux OS has the downside in memory management hardware that does not fit low-end IoT device [25].

6.1.5 Communication and networking protocols support: uClinux OS uses uIP and lwIP networking protocols [95]. uClinux also has a wide networking and communication protocols support including a full TCP/IP stack, IPv6, WiFi, and other networking protocols [95].

6.1.6 Simulation support: uClinux can be simulated using *GDB/ARMulator* and *SWARM-Software ARM-arm7* emulator. ARMulator is developed using C programming language and provides more than just an instruction set simulator; it provides a virtual platform for system emulations. It can emulate an ARM processor and other ARM co-processors [98].

6.1.7 Security: uClinux OS applies a shepherd process to manage security issues [100]. A shepherd process is responsible for accessing security association, and later dropping that access when a root trip happens. uClinux consists of three main primitives; register, start, and finish. Each of these primitives communicates

with the kernel to execute their particular functions. Moreover, uClinux applies the encrypted storage security technique which forces cryptographic overheads. uClinux supports run-time interception of the different system calls. For example, running and open files can make such calls take longer time. When a root trip starts, shepherd process uses a register to perform any security actions through communication to the kernel and must be allowed to drop its security association before that root trip finishes. When a register finishes its function, the shepherd process can perform its task to achieve the security. Such as, the mount-efs reads the UCLinux file system (UCFS)'s secret key from/etc/crypt.key unseals it and pipes it into cryptsetup. Then, sets up a cryptographic loopback device, and mounts the file system. Next, its security association becomes accessible; a shepherd process uses start action to put the shepherd process to sleep and only wakes it up when a root trip happens. When it wakes up, it must drop its security association. Finally, when the shepherd process is sure that its security association is protected, it uses finish preemptive through communication to the kernel to complete the root trip. Hence, allow untrusted programmes to run with a privileged effective user. In this way, any number of security associations can be fully protected [100].

6.1.8 Power consumption: uClinux supports power management techniques which can be modified, so that the idle process can be called when no other processes are running. This makes the kernel enter into a sleep mode until further processing is required. The kernel uses a kernel ticker to wake up the system 100 times per second which causes a problem. However, this problem can be solved by building a tickless kernel that only calculates when the process needs to wake up. This solution reduces the number of interrupts occurred [98].

6.1.9 Supporting multimedia: uClinux provides support for multimedia through a software project for the record, convert, and processing audio and video streams called FFmpeg. FFmpeg uses HTTP server for live broadcasts processing and gst-real time streaming protocol (RTSP) streaming server which is a library on top of GStreamer. The gst-RTP server is designed to enable many sources to connect, rebroadcasting, and transmit video and audio streams over a network to single or multiple users [98]. GStreamer supports multimedia processing, encoding, and streaming libraries [98].

6.1.10 More about uClinux: uClinux OS supports a vast number of devices, filesystems, networking protocols, and applications (such as GNU software). The source code of uClinux is available to end users and developers. It is tested and refined by many programmers and users. Moreover, systems running uClinux OS may be configured in different ways other than that of the familiar Unix-like Linux distribution.

6.2 Raspbian OS

Raspbian is an *open-source Linux-based* OS for *high-end* devices based on Debian (Linux) and optimised for RPi hardware [101]. Raspbian provides more than 35,000 packages that can be installed from the terminal. It has a pre-compiled software in a simple format for easy installation [102]. RPi is a credit-card-sized and inexpensive single-board computer that can to be run by Linux and other lightweight OSs [103]. It can be run on multiple low-performance ARM processors [103].

6.2.1 Architecture and kernel models: Similar to uClinux OS, Raspbian OS follows a *monolithic* architecture [104]. Fig. 10 shows the architecture of Raspbian OS.

6.2.2 Programming model and development environment: Like uClinux OS, Raspbian OS supports *multithreading* [105]. It is written using Python programming language, and the code can be modified according to the requirements of an application [105]. A considerable number of

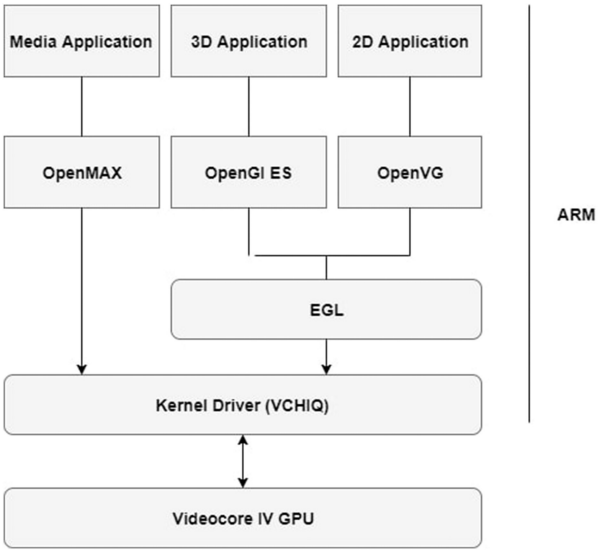

Fig. 10 *Architecture of Raspbian OS (reproduced from [105])*

programming languages have been adapted for Raspbian OS such as Python, C, C++, Java, Scratch, and Ruby; all installed by default on the Raspbian OS [105]. Also, scripting languages such as HTML5, Javascript, and JQuery are supported [105].

6.2.3 Scheduling: Raspbian utilises *real-time preemptive* scheduling [106].

6.2.4 Memory management and performance: Raspbian OS supports a virtual memory technique that is performed by hardware through MMU. Raspbian also offers virtual memory swapping which divides the hard disc into parts to exchange fractions of main memory. Hence, allowing occupied regions that did not take sufficient time to become available for allocation [104].

6.2.5 Communication and networking protocols support: Raspbian OS supports a wide range of communications through serial peripheral interface (SPI), UART, I2c, and universal serial bus (USB). It also implements the full stack of TCP/IP and Bluetooth. Similar to other Linux distributions, Raspbian OS has support for almost all networking protocols that are imported from Debian distribution. In addition, a free open-source library for LTE and other wireless protocols are available to be used easily with Raspbian [105].

6.2.6 Simulation support: Raspbian OS can be simulated using *QEMU ARM* which is a CPU architecture emulator and virtualisation tool. QEMU is capable of emulating an ARM architecture which is very similar to the RPi boards. This allows booting an RPi image directly to x86 or x86–64 systems. However, to deal with the QEMU virtualised hardware which is not a core component of the RPi, Raspbian OS must have a RPi kernel for QEMU to control the QEMU kernel image [103].

6.2.7 Security: Raspbian OS supports many encryption, authentication, and authorisation techniques that suit most IoT applications. The open-source community provides support for this OS and for most encryption algorithms that are available from Raspbian repositories. Customised security requirements can be easily integrated into Raspbian OS even if it requires modification in core libraries. Encryption algorithms such as AES 128, AES 256, data encryption standard (DES), and Blowfish are available in all framework libraries in addition to HTTP secure and virtual private network protocols [105]. Selective encryption algorithms for multimedia streaming are also supported in Raspbian OS [107].

6.2.8 Power consumption: One of the great things about creating a cluster with ARM-based processors is low-power consumption; each RPi uses about 2 W of power (when running at 700 MHz)

Table 1 Power consumption of different RPi boards [105]

Pi mode	Pi state	Power consumption
a +	idle, high-definition multimedia interface (HDMI) disabled, light-emitting diode (LED) disabled	80 mA (0.4 W)
a +	idle, HDMI disabled, LED disabled, USB WiFi adapter	160 mA (0.8 W)
b +	idle, HDMI disabled, LED disabled	180 mA (0.9 W)
b +	idle, HDMI disabled, LED disabled, USB WiFi adapter	220 mA (1.1 W)
model 2 B	idle, HDMI disabled, LED disabled	200 mA (1.0 W)
model 2 B	idle, HDMI disabled, LED disabled, USB WiFi adapter	240 mA (1.2 W)
zero	idle, HDMI disabled, LED disabled	80 mA (0.4 W)
zero	idle, HDMI disabled, LED disabled, USB WiFi adapter	120 mA (0.7 W)

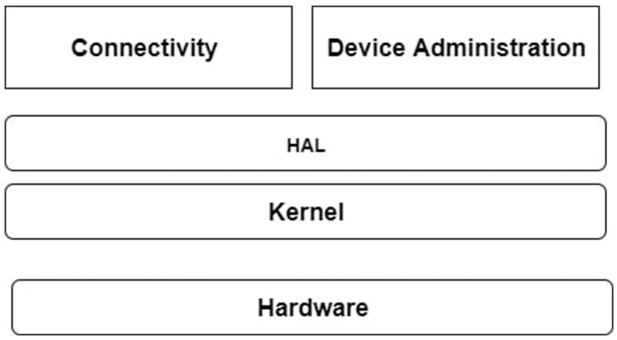

Fig. 11 *Architecture of Android Things OS (reproduced from [111])*

[103]. Raspbian OS power consumption is based on the RPi boards used and type of applications running on the device itself. Table 1 summarised the power consumption using different RPi boards [105].

6.2.9 Supporting multimedia: Raspbian OS can perform live audio and video streaming by using session initiation protocol (SIP) and RTP protocols [105]. Raspbian OS employs code-excited linear prediction compression algorithms to ensure low latency and high-quality communication [102, 108]. Moreover, it supports HD videos and music [105]. Also, GStreamer is supported on Raspbian OS; which supports streaming, encoding and packing of various multimedia formats such as flv and H264 [105]. A full list of supported plugins can be found in [101]. In addition, a special camera module is available for RPi boards and Raspbian OS [105]. The module supports 1080p30, 720p60, and VGA90 video modes. The camera is connected through camera serial interface port available on RPi, and there are several third-party libraries built for it including the Python Picamera library [105].

6.2.10 More about Raspbian OS: Raspbian OS has a desktop environment called Lightweight graphical X11 desktop environment. It is very similar to Windows and Mac desktops and provide an attractive user interface [101]. Moreover, it includes Wayland display server protocol which allows efficient use of the graphics processing unit for hardware accelerated graphical user interface drawing functions for Robots [102].

6.3 Android Things OS

Android Things is an *open-source Linux-based* OS for *high-end* IoT devices. It is developed by Google and derived from Android OS. Android Things OS is coded using a specialised programming language called Weave, which is a common cross-platform language. Android Things OS can run on high-end IoT devices offering a few tens of megabytes of memory because it depends on the lower levels of Android which can keep running on insignificant system requirements such as light bulbs. It has a user-friendly interface which makes it easier to set up hardware. Also, Android Things may run wearable devices due to both environment common characteristics, and the energy-consumption restriction [33, 90, 109, 110]. Android Things supports many hardware platforms such as Intel (X86), Edison (Dual-core Atom 500M), minnowboard, Qualcomm (Arm), dragonboard (MSM8916, QCore A53), Marvell (Arm), ABox Edge (IAP140, QCore A53), Freescale (Arm), and Rockchip (Arm) [111]. The structure of Android Things OS is shown in Fig. 11.

6.3.1 Architecture and kernel models: In contrast to Raspbian OS, Android Things OS has a *modular* architecture [112].

6.3.2 Programming model and development environment: Similar to Raspbian OS, Android Things OS supports *multithreading* and a number of Android SDKs such as APIs and AdMob, where authentication is required for user input [112]. Moreover, Android Things OS developers can develop their programmes using C and C++ programming languages in addition to Java [111].

6.3.3 Scheduling: Android Things OS scheduler supports either *prioritised preemptive* or *cooperative* depending on the configuration of the scheduler [113].

6.3.4 Memory management and performance: Android Things OS implements *dynamic* memory allocation through system calls [77] and it has a small memory footprint [111].

6.3.5 Communication and networking protocols support: Android Things OS supports WiFi, bluetooth low energy, ZigBee, IPV6, and other networking protocols [112].

6.3.6 Simulation support: Android Things OS is not yet supported on Android Studio Android Virtual Device as other Android platforms. However, it can be employed on RPi 3, Intel Edison, and NXP Pico in order to be used for the initial simulation [111].

6.3.7 Security: As Android Things OS is an open software based on Android, it allows building applications smoothly and quickly. Hence, there will be more security loopholes such as virus attacks and hacking [111]. Android Things OS uses verified secure boot and signed over-the-air updates which make it more secure [111]. Android Things OS provides full disc encryption so all data will be protected. Everything on the drive will be encrypted including the files which keep exact copies of the data that the user has been working on such as temporary files [111].

Table 2 Programming features summary for open-source IoT OSs

OS	Kernel	Scheduler	Programming model	Language support	Real time
TinyOS	monolithic	non-preemptive FIFO	event driven	NesC	not supported
Contiki	modular	preemptive FIFO	event driven, protothreads	C	partially supported
RIOT	microkernel	preemptive priority, tickless	multithreading	C, C + +	supported
LiteOS	modular	preemptive priority (RR)	multithreading	LiteC + +	not supported
FreeRTOS	microkernel	preemptive, optional tickles	multithreading	C	supported
Mynewt	modular	preemptive	multithreading	Go (golang), C	supported
Mbed	monolithic	non-preemptive	single thread	C, C + +	supported
uClinux	monolithic	preemptive	multithreading	C	partially supported
Raspbian	modular, monolithic	preemptive	multithreading	Python, C, Ruby, Java, PHP, C + +, Node.js	supported
Android Things	modular	preemptive	multithreading	Weave using C, C + +	supported

Table 3 Recommended hardware requirements for IoT OSs

OS	Min-RAM, kB	Min-ROM, kB	Processor/CPU, MHz
TinyOS	1	4	7.4, 8 bit
Contiki	2	30	8 bit
RIOT	1.5	5	16–32 bit
LiteOS	4	128	8 MHz
FreeRTOS	10	12	32 bit
Mynewt	8	64	12–200 MHz
Mbed	16	32-bit	32
uClinux	512	512	200
Raspbian	512–256	—	ARM Cortex-A53
Android Things	128	32 [111]	ARM, Intelx86, MIPS [102]

6.3.8 Power consumption: Android Things OS is running on WiFi and Bluetooth low energy using the minimum system requirements such as low memory and small processors. Hence, it needs less power requirement to run [111].

6.3.9 Supporting multimedia: Android Things OS can fully support high-performance multimedia streaming and processing [111]. It supports the same stack of Android OS media contents such as H264, MP3, and VP9 [111]. This processing includes several tasks such as image and video analytics and data processing that can be processed inside the device instead of processing on the cloud [111].

6.3.10 More about android things: Google Cloud Platform components (such as Firebase) can be easily integrated with Android Things OS [112]. Developers will be able to use different cloud services for storage, state management, and messaging [112]. When it comes to Weave, SDK will be embedded in the devices for local and remote communications. Moreover, Weave is an independent protocol that can be as Zigbee, Z-Wave, and Bluetooth Smart. What makes Android Things OS unique is that it is compatible with all Android Source Packages [111].

Other open-source OSs: There are a few open-source OSs that are still growing and not popular or deprecated that are not covered in this paper such as Pyxis, Ubuntu Snappy Core, and Ostro. For example, through many searches, we revealed very little about Pyxis OS which has been deprecated and replaced by Pyxis 2. It does neither seem to be a PPC64, nor it is CentOS/Linux based. It looks such as a custom OS written in C# for Arduino microcontrollers. It also appears to be a small project developed around 2010 with no updates afterward. Other OSs still work very hard to shine in the field but the competition is high as this area is growing dramatically.

7 Comparisons of IoT OSs

In this section, we provide several summarised tabular overviews of all OSs listed in this survey. Table 2 summarises the programming features of all OSs discussed in this paper. This table lists the programmability features for each OS in terms of the kernel and architectural usage for a basic application, scheduling type, programming model, supported programming language, the ability of reprogramming, and supporting of real time. We noted that the common programming languages used in IoT OSs include C, C + +, and Java. Java always runs on top of an IoT OS. So, the choice is not between C/C + + or Java; it is whether C or C + + and Java.

Table 3 provides a summary of the hardware requirements for the surveyed IoT OSs. This table defines the OS hardware configuration requirements in terms of RAM and ROM usage for a basic application, processor, and main supported hardware platform. The purpose of hardware specifications is to give appropriate design decisions for devices that will run by IoT OSs.

Finally, we briefly summarise the primary technical aspects of IoT OSs in Table 4. It highlights implementation aspects in terms of the remote scriptable wireless shell, remote file system interface for networked nodes, file system, online debugging, dynamic memory, simulation support, and list of supported network technologies and protocols.

8 Conclusion

The proliferation of the IoT is dramatically increasing and already covers many prominent domains and disciplines such as smart cities, smart sensory platforms, and intelligent transit systems. OS support is vital in facilitating the development and subsistence of IoT. In this paper, we provide a comprehensive study of the most used and state-of-the-art open-source OSs for IoT. We first investigate the design and development aspects of IoT OSs. Then, we propose a taxonomy to classify and categorise the state-of-the-art and most used IoT OSs. We provide an extensive overview of open-source IoT OSs, where each OS explained in details based on the established designing and developmental aspects. These aspects are; architecture and kernel models, programming model, and development environment, scheduling, memory management and performance, communication and networking protocols, simulator, security, power consumption, and multimedia support. We survey

Table 4 Technical aspects comparison of IoT OSs

OS	Remote scriptable shell	Remote file system	File system	Online debugging	Dynamic memory	Simulator supported	Network stack and protocol supported
TinyOS	no[a]	no	single level (ELF[b], matchbox)	yes[c]	No	TOSSIM, PowerTossim	BLIP[d], TinyRPL, CoAP, WiFi, LTE, MQTT, LoWPAN, TCP, BBR[e], IPv6, Bluetooth, multi-path routing
Contiki	no[f]	no	single level	no	yes	Netsim, Cooja, MSPSim, Java nodes	RPL, uIP, uIPv6, MQTT, 6LoWPAN, CoAP, WiFi, Bluetooth
Riot OS	yes	yes	hierarchical	yes	yes	Cooja simulator	TCP, UDP, IPv6, 6LoWPAN, RPL, CoAP, CBOR[g], UBJSON[h], OpenWSN, WiFi, Bluetooth, LTE, MQTT
LiteOS	yes[i]	yes	hierarchical Unix-like	yes	yes	AVRORA simulator	NB-IoT, 6LoWPAN, ZigBee, LTE, Bluetooth
FreeRTOS	yes	yes	hierarchical	yes	yes	QEMU	uIP, lwIP, TCP, LoWPAN, CoAP, MQTT
Mynewt	yes	yes	hierarchical	yes	yes	QEMU	BLE, IPv6, 6LoWPAN, HTTP, TCP, CoAP, MQTT, UDP, LoRa PHY, LoRaWAN
ARM Mbed	yes	yes	hierarchical	yes	yes	QEMU	6LoWPAN, Ethernet, Zigbee LAN, HTTP, Zigbee IP, WiFi, BLE, IPv6, CoAP, MQTT
uClinux	yes	yes	hierarchical	yes	yes	GDB/ARMulator	uIP, lwIP, WiFi, Bluetooth, LTE, IPv6, MQTT
Raspbian	yes	yes	hierarchical	yes	yes	QEMU	RTP, LTE, HTTP, TCP, MQTT
Android Things	yes	yes	hierarchical	yes	yes	RPi 3, Intel Edison, and NXP Pico	IPv6, Zigbee, Z-Wave, Bluetooth Smart

[a]Through the application specific shell such as Simple cmd exists.
[b]Executable and Linkable Format.
[c]Through the Clairvoyant.
[d]Berkeley Low-power IP.
[e]Bottleneck Bandwidth and Round-trip.
[f]Through the mote shell.
[g]Concise Binary Object Representation.
[h]Universal Binary JSON.
[i]Through the base PC, Unix commands.

each OS's characteristics, advantages, and disadvantages. Also, several comparisons concentrating on the similarities and differences between the discussed OSs are presented. We remark that this is the first such tutorial style paper on IoT OSs.

Finally, we argue that each IoT OS has some limitations depending on the targeted deployment scenario. For that reason, it is challenging to have an OS that satisfies all requirements. Moreover, choosing an appropriate OS for IoT applications is critical to the success of IoT deployments and implementations. The developer must carefully study the strengths and weaknesses of the candidate OSs to make the best choice. As each IoT OS has its pros and cons that can be used to identify the appropriate OS based on the functional, non-functional, power consumption, sensors connectivity, communication methods, and many other requirements. Thus, this research is designed to bridge the existing gap in knowledge of the adoption and implementation of OSs for IoT from different aspects. This survey provides an easy to follow and well-structured guide in a tutorial style for researchers and developers targeting IoT OSs.

9 Acknowledgment

This work made possible by a financial support from the Applied Science Private University in Amman, Jordan.

References

[1] Ma, H.: 'Internet of things: objectives and scientific challenges', *J. Comput. Sci. Technol.*, 2011, **26**, (6), pp. 919–924
[2] Mattern, F., Floerkemeier, C.: 'From the internet of computers to the Internet of things', in Sachs, K., Petrov, I., Guerrero, P. (Eds.): '*From active data management to event-based systems and more*' (Springer, Berlin, Germany, 2010), vol. **33**, (2), pp. 242–259
[3] Zhu, Q., Ruicong, W., Chen, Q., *et al.*: 'IoT gateway: bridging wireless sensor networks into internet of things'. Proc. IEEE/IFIP Eighth Int. Conf. Embedded Ubiquitous Comput. (EUC), Hong Kong, China, December 2010, pp. 347–352
[4] Gartner Inc.: 'Gartner says 8.4 billion connected 'Things' will be in use in 2017, up 31 percent from 2016'. Available at https://www.gartner.com/newsroom/id/3598917, accessed June 2017
[5] Sundmaeker, H., Guillemin, P., Friess, P., *et al.*: '*Vision and challenges for realizing the Internet of things*' (European Commission, Brussels, 2010)
[6] Islam, S., Kwak, D., Kabir, M., *et al.*: 'The Internet of things for health care: a comprehensive survey', *IEEE Access.*, 2015, **3**, pp. 678–708
[7] Keoh, S., Kumar, S., Tschofenig, H.: 'Securing the Internet of things: a standardization perspective', *IEEE Internet Things J.*, 2014, **1**, (3), pp. 265–275
[8] Mainwaring, A., Polastre, J., Szewczyk, R., *et al.*: 'Wireless sensor networks for habitat monitoring'. Proc. ACM Int. Works Wireless Sensor Networks and Applications, Atlanta, GA, USA, September 2002, pp. 88–97
[9] Al-Fuqaha, A., Guizani, M., Mohammadi, M., *et al.*: 'Internet of things: a survey on enabling technologies, protocols, and applications', *IEEE Commun. Surv. Tutor.*, 2015, **17**, (4), pp. 2347–2376
[10] Why the Need for Special Operating Systems for IoT and Wearable Devices? Available at https://dzone.com/, accessed June 2017
[11] Cisco Visual Networking Index.: Global Mobile Data Traffic Forecast Update, 2016–2021. Available at https://www.cisco.com/
[12] Razzaque, M., Milojevic-Jevric, M., Palade, A., *et al.*: 'Middleware for Internet of things: a survey', *IEEE Internet Things J.*, 2016, **3**, (1), pp. 70–95
[13] Roman, R., Najera, P., Lopez, J.: 'Securing the Internet of things', *IEEE Comput. Netw.*, 2011, **44**, (9), pp. 51–58

[14] Internet of Things (IoT) Connected Devices Installed Base Worldwide from 2015 to 2025 (in billions). Available at https://www.statista.com/, accessed August 2017

[15] Lee, J., Sung, Y., Park, J.: 'Lightweight sensor authentication scheme for energy efficiency in ubiquitous computing environments', *Sensors*, 2016, **16**, (12), pp. 2044–2059

[16] Tarkoma, S., Ailisto, H.: 'The Internet of things program: the Finnish perspective', *IEEE Commun. Mag.*, 2013, **51**, (3), pp. 10–11

[17] Al-Turjman, F., Alturjman, S.: 'Context-sensitive access in industrial Internet of things (IIoT) healthcare applications', *IEEE Trans. Ind. Inf.*, 2018, **14**, (6), pp. 2736–2744

[18] Miorandi, D., Sicari, S., De Pellegrini, F., *et al.*: 'Internet of things: vision, applications and research challenges', *Ad Hoc Netw.*, 2012, **10**, (7), pp. 1497–1516

[19] Roman, R., Zhou, J., Lopez, J.: 'On the features and challenges of security and privacy in distributed Internet of things', *Comput. Netw.*, 2013, **57**, (10), pp. 2266–2279

[20] Balakrishna, C.: 'Enabling technologies for smart city services and applications'. Proc. IEEE Int. Conf. Next Generation Mobile Applications, Services and Technologies (NGMAST), Paris, France, September 2012, pp. 223–227

[21] Wang, L., Alexander, C.: 'Big data analytics and cloud computing in Internet of things', *Amer. J. Inf. Sci. Comput. Eng.*, 2016, **2**, (6), pp. 70–78

[22] Sarkar, C., Nambi, A., Prasad, R., *et al.*: 'DIAT: a scalable distributed architecture for IoT', *IEEE Internet Things J.*, 2015, **2**, (3), pp. 230–239

[23] Qin, Z., Denker, G., Giannelli, C., *et al.*: 'A software defined networking architecture for the Internet-of-things'. Proc. IEEE Network Operations and Management Symp. (NOMS), Krakow, Poland, June 2014, pp. 1–9

[24] Chen, S., Xu, H., Liu, D., *et al.*: 'A vision of IoT: applications, challenges, and opportunities with China perspective', *IEEE Internet Things J.*, 2014, **1**, (4), pp. 349–359

[25] Hahm, O., Baccelli, E., Petersen, H., *et al.*: 'Operating systems for low-end devices in the Internet of things: a survey', *IEEE Internet Things J.*, 2016, **3**, (5), pp. 720–734

[26] Silberschatz, A., Galvin, P., Gagne, G.: '*Operating system concepts*' (Wiley, New York, USA, 2013, 9th edn.)

[27] Farooq, M., Kunz, T.: 'Operating systems for wireless sensor networks: a survey', *Sens. J.*, 2011, **11**, (6), pp. 5900–5930

[28] Will, H., Schleiser, K., Schiller, J.: 'A real-time kernel for wireless sensor networks employed in rescue scenarios'. Proc. IEEE Conf. Local Computer Networks (LCN), Zurich, Switzerland, October 2009, pp. 834–841

[29] Dunlap, G., King, S., Cinar, S., *et al.*: 'Revirt: enabling intrusion analysis through virtual-machine logging and replay'. Proc. Symp Operating Syst. Design and Implementation (OSDI), Seattle, USA, December 2002, pp. 211–224

[30] Bandyopadhyay, D., Sen, J.: 'Internet of things: applications and challenges in technology and standardization', *Wirel. Pers. Commun.*, 2011, **58**, (1), pp. 49–59

[31] Dunkels, A., Groonvall, B., Voigt, T.: 'ContikiVA lightweight and flexible operating system for tiny networked sensors'. Proc. Annual IEEE Int. Conf. Local Computer Network (LCN), Tampa, FL, USA, November 2004, pp. 455–462

[32] Chien, T., Chan, H., Huu, T.: 'A comparative study on operating system for wireless sensor networks'. Proc. Int. Conf. Advanced Computer Science and Information Syst., Jakarta, Indonesia, December 2011, pp. 73–78

[33] Azure IoT device SDK for C. Available at https://docs.microsoft.com/en-us/azure/, accessed August 2017

[34] Baccelli, E., Hahm, O., Günes, M., *et al.*: 'OS for the IoT – goals, challenges, and solutions, OS for the IoT – goals, challenges, and solutions'. Wkshps Interdisciplinaire sur la SÃl'curitÃl' Globale (WISG), Troyes, France, January 2013, pp. 1–6

[35] Levis, P., Culler, D., Shenker, S., *et al.*: 'Trickle: a self-regulating algorithm for code propagation and maintenance in wireless sensor networks'. Proc. USENIX/ACM Symp. Networked System Design and Implementation (NSDI), San Francisco, CA, March 2004, pp. 15–28

[36] Abdelsamea, M., Zorkany, M., Abdelkader, N.: 'Real time operating systems for the Internet of things, vision, architecture and research directions'. Proc. World Symp. Computer Applications and Research, Cairo, Egypt, March 2016, pp. 72–77

[37] Deniz, U., Al-Turjman, F., Celik, G.: 'An overview of Internet of things and wireless communications'. Proc. Int. Conf. Computer Science and Engineering (UBMK), Antalya, Turkey, October 2017, pp. 506–509

[38] Sulyman, A., Oteafy, S., Hassanein, H.: 'Expanding the cellular-IoT umbrella: an architectural approach', *IEEE Wirel. Commun.*, 2017, **24**, (3), pp. 66–71

[39] AlTurjman, F., Alturjman, S.: 'Confidential smart-sensing framework in the IoT era', *J. Supercomput.*, 2018, **74**, (10), pp. 5187–5198

[40] Atzori, L., Iera, A., Morabito, G.: 'The Internet of things: a survey', *Comput. Netw.*, 2010, **54**, (15), pp. 2787–2805

[41] Perrig, A., Szewczyk, R., Wen, V.: 'SPINS: security protocols for sensor networks', *Wirel. Netw.*, 2002, **5**, (8), pp. 521–534

[42] Xiong, L., Zhou, X., Liu, W.: 'Research on the architecture of trusted security system based on the Internet of things'. Proc. Int. Conf. Intell. Computer Technology and Automation, Shenzhen, Guangdong, China, March 2011, pp. 1172–1175

[43] Demir, S., Al-Turjman, F.: 'Energy scavenging methods for WBAN applications: a review', *IEEE Sens. J.*, 2018, **18**, (16), pp. 6477–6488

[44] Lajara, R., Pelegri-Sebastia, J., Solano, J.: 'Power consumption analysis of operating systems for wireless sensor networks', *Sensors*, 2010, **10**, (6), pp. 5809–5826

[45] Hamoudy, M., Qutqut, M., Almasalha, F.: 'Video security in Internet of things: an overview', *Int. J. Comput. Sci. Netw. Secur. (IJCSNS)*, 2017, **17**, (8), pp. 199–255

[46] Al-Sakran, A., Qutqut, M., Almasalha, F., *et al.*: 'An overview of the Internet of things closed source operating systems'. Int. Wireless Communications and Mobile Computing Conf. (IWCMC), Limassol, Cyprus, June 2018

[47] Levis, P., Madden, S., Polastre, J., *et al.*: 'TinyOS: an operating system for sensor networks', in Weber, W., Rabaey, J., Aarts, E. (Eds.): '*Ambient intelligence*', vol. 1 (Springer, New York, 2005), pp. 115–148

[48] Levis, P., Lee, N., Welsh, M., *et al.*: 'TOSSIM: accurate and scalable simulation of entire TinyOS applications'. Proc. Int. Conf. Embedded Networked Sensor Systems, CA, USA, November 2003, pp. 126–137

[49] Gay, D., Levis, P., Culler, D.: 'Software design patterns for TinyOS'. Proc. ACM Conf. Languages, Compilers, and Tools for Embedded Syst., IL, USA, June 2005, vol. **40**, pp. 40–49

[50] Hill, J., Culler, D., Horton, M., *et al.*: 'Mica: the commercialization of microsensor motes', *Sens. Mag.*, 2004, **19**, pp. 40–48

[51] Sruthi, M., Rajkumar, R.: 'A study on development issues over IOT platforms, protocols and operating system'. Int. Conf. Innovations in Information Embedded and Communication Systems, Coimbatore, India, March 2016

[52] Casado, L., Tsigas, P.: 'Contiki Sec: a secure network layer for wireless sensor networks under the Contiki operating system'. Proc. Nordic Conf. Secure IT Systems, New York, NY, USA, October 2009, pp. 133–147

[53] Ma, H.: 'Experimental evaluation of a video streaming system for wireless multimedia sensor networks'. Proc. Tenth IEEE IFIP Annual Mediterranean Ad Hoc Network Workshops, Sicily, Italy, August 2011, pp. 165–170

[54] Farooq, M., Aziz, S., Dogar, A.: 'State of the art in wireless sensor networks operating systems: a survey'. Proc. Int. Conf. Future Generation Information Technology, Berlin, Heidelberg, December 2010, pp. 616–631

[55] Dunkels, A., Schmidt, O., Voigt, T., *et al.*: 'Protothreads: simplifying event-driven programming of memory-constrained embedded systems'. Proc. Int. Conf. Embedded Networked Sensor Systems, Boulder, CO, USA, October 2006, pp. 29–42

[56] Dunkels, A., Finne, N., Eriksson, J.: 'Run-time dynamic linking for reprogramming wireless sensor networks'. Proc. Fourth ACM Conf. Embedded Networked Sensor Systems (Sensys), Boulder, CO, USA, October 2006, pp. 15–28

[57] Dunkels, A., Mottola., L, Tsiftes, N., *et al.*: 'The announcement layer: beacon coordination for the sensornet stack'. Proc. Wireless Sensor Networks Conf. (EWSN), Bonn, Germany, February 2011, pp. 211–226

[58] Tsiftes, N., Dunkels, A., He, Z., *et al.*: 'Enabling large-scale storage in sensor networks with the coffee file system'. Proc. Int. Conf. Information Processing Sensor Networks, San Francisco, CA, USA, August 2009, pp. 349–360

[59] Klauck, R., Kirsche, M.: 'Bonjour Contiki: a case study of a DNS based discovery service for the Internet of things'. Proc. Int. Conf. Ad hoc, Mobile, and Wireless Networks (ADHOC-NOW), Berlin, Germany, June 2012, pp. 316–329

[60] Kuladinithi, K., Bergmann, O., Pötsch, T., *et al.*: 'Implementation of CoAP and its application in transport logistics'. Proc. Wkshps on Extending the Internet to Low power and Lossy Networks, Chicago, IL, USA, April 2011, pp. 1–7

[61] Kovatsch, M., Duquennoy, S., Dunkels, A.: 'BA low-power CoAP for contiki'. Proc. IEEE Eighth Int. Conf. Mobile Ad hoc and Sensor Syst. (MASS), Valencia, Spain, October 2011, pp. 855–860

[62] Contiki.: The open source OS for the Internet of things. Available at http://www.contiki-os.org/, accessed on August 2017

[63] Tsiftes, N., Eriksson, J., Dunkels, A.: 'Poster abstract: low-power wireless IPv6 routing with ContikiRPL'. Proc. Ninth ACM/IEEE Int. Conf. Information Processing in Sensor Networks, Stockholm, Sweden, April 2010, pp. 406–407

[64] Munawar, W., Alizai, M., Landsiedel, O., *et al.*: 'Dynamic TinyOS: modular and transparent incremental code-updates for sensor networks'. Proc. IEEE Int. Conf. Commun. (ICC), Cape Town, South Africa, May 2010, pp. 1–6

[65] Kalyoncu, S.: 'Wireless solutions and authentication mechanisms for Contiki based Internet of things networks', PhD Thesis, Halmstad University, 2013

[66] RIOT Documentation. Available at https://riot-os.org/api/, accessed on September 2017

[67] Emmanuel, B., Gündoğan, C., Hahm, O., *et al.*: 'RIOT: an open source operating system for low-end embedded devices in the IoT', *IEEE Internet Things J.*, 2018, doi: 10.1109/JIOT.2018.2815038

[68] Roussel, K., Song, Y., Zendra, O.: 'RIOT OS paves the Way for implementation of high-performance MAC protocols'. Proc. Fourth Int. Conf. Sensor Networks (SENSORNETS), Angers, France, April 2015, pp. 5–14

[69] Baccelli, E., Hahm, O., Petersen, H., *et al.*: 'RIOT and the evolution of IoT operating systems and applications', ERCIM News, April 2015, 101

[70] Petersen, H., Adjih, C., Hahm, O., *et al.*: 'IoT meets robotics-first steps, RIOT car, and perspectives'. Proc. ACM Int. Conf. Embedded Wireless Systems and Networks (EWSN), Graz, Austria, February 2016, pp. 269–270

[71] Milinkovi, A., Milinković, S., Lazic, L., *et al.*: 'Choosing the right RTOS for IoT platform', *NFOTEH-JAHORINA*, 2015, **14**, (3), pp. 504–509

[72] Baccelli, E., Hahm, O., Gunes, M., *et al.*: 'RIOT OS: towards an OS for the Internet of things'. Proc. 32nd IEEE Conf. Computing Communications (INFOCOM), Turin, Italy, April 2013, pp. 79–80

[73] Shang, W., Afanasyev, A., Zhang, L.: 'The design and implementation of the NDN protocol stack for RIOT-OS'. Proc. IEEE Globecom Wkshps (GC Wkshps), Washington, DC, USA, December 2016, pp. 1–6

[74] Huawei LiteOS. Available at http://www.huawei.com/, accessed on September 2017

[75] Vanitha, V., Palanisamy, V., Johnson, N., *et al.*: 'LiteOS based extended service oriented architecture for wireless sensor networks', *Int. J. Comput. Electr. Eng.*, 2010, **2**, (3), pp. 432–436

[76] Cao, Q., Abdelzaher, T., Stankovic, J., *et al.*: 'The LiteOS operating system: towards Unix-like abstractions for wireless sensor networks'. Proc. Int. Conf.

Information Processing in Sensor Networks (IPSN), USA, April 2008, pp. 233–244
[77] Gaur, P., Tahiliani, M.: 'Operating systems for IoT devices: a critical survey'. Region 10 Symp. (TENSYMP), Ahmedabad, Pakistan, May 2015, pp. 33–36
[78] Hammad, M., Cook, J.: 'Lightweight deployable software monitoring for sensor networks'. Proc. IEEE 18th Int. Conf. Computing Communications and Networks, Washington, D.C., USA, August 2009, pp. 1–6
[79] Ranjan, A., Sahu, H., Misra, P.: 'A survey report on operating systems for tiny networked sensors', arXiv preprint arXiv:1505.05269, May 2015
[80] Deharbe, D., Galv'ao, S., Moreira, A.: 'Formalizing FreeRTOS: first steps in formal methods: foundations and applications'. Proc. 12th Brazilian Symp. Formal Methods (SBMF), Gramado, Brazil, August 2009, pp. 101–117
[81] Andersson, K., Andersson, R.: 'A comparison between FreeRTOS and RTLinux in embedded real-time systems', Linkoping University, 2005
[82] Hos'ek, P.: 'Supporting real-time features in a hierarchical component system', MSc thesis, Charles University, 2010
[83] Yang, C., Chih, H.: 'An open source audio effect unit'. Proc. IEEE Int. Conf. Systems, Man, and Cybernetics (SMC), Budapest, Hungary, October 2016, pp. 638–643
[84] FreeRTOS. Available at http://www.freertos.org/, accessed on September 2017
[85] Kruger, C., Hancke, G.: 'Implementing the Internet of things vision in industrial wireless sensor networks'. Proc. IEEE Int. Conf. Industrial Informatics, Budapest, Hungary, July 2014, pp. 627–632
[86] Johny, A., Jayasudha, J., Anurag, R.: 'Security in automotive domain using secure socket layer', *Int. J. Eng. Innov. Technol.*, 2013, **3**, (4), pp. 214–219
[87] Apache Mynewt OS. Available at https://mynewt.apache.org/, accessed on October 2017
[88] Kordestani, M., Bourdoucen, H.: 'A survey on embedded open source system software for the Internet of things'. Proc. Free and Open Source Software Conf., Muscat, Oman, February 2017, pp. 27–32
[89] An Operating System for Arduino. Available at https://www.arduino.cc/, accessed on September 2017
[90] Malche, T., Maheshwary, P.: 'Harnessing the Internet of things (IoT): a review', *Int. J. Adv. Res. Comput. Sci. Softw. Eng.*, 2015, **5**, (8), pp. 320–323
[91] Mbed IoT Platform. Available at https://www.mbed.com/en/platform/, accessed on September 2017
[92] Persson, P., Angelsmark, O.: 'Calvinâ merging cloud and IoT'. Proc. Int. Conf. Ambient Systems Networks and Technology (ANT), London, UK, June 2015, pp. 210–217
[93] Balsamo, D., Elboreini, A., Al-Hashimi, B., *et al.*: 'Exploring ARM Mbed support for transient computing in energy harvesting IoT systems'. Proc. Seventh IEEE Int. Wkshps Advances in Sensors and Interfaces, Vieste, Italy, June 2017, pp. 115–120
[94] arm MBED. Available at https://www.mbed.com/, accessed on September 2017
[95] Nikkanen, K.: 'Uclinux as an embedded solution', Bachelor's thesis, Turku Polytechnic Institute, 2003
[96] Lu, Z., Zhang, X., Sun, C.: 'An embedded system with uClinux based on FPGA'. Proc. Pacific-Asia Wkshps on Computational Intelligence and Industrial Application (PACIIA), Wuhan, China, December 2008, pp. 691–694
[97] Wang, M., Liu, F.: 'Research and implementation of uCLinux-based embedded browser'. Proc. Second IEEE Asia-Pacific Service Computing Conf., Tsukuba Science City, Japan, December 2007, pp. 504–508
[98] uClinux in the GDB/ARMulator. Available at http://www.uclinux.org/pub/uClinux/utilities/armulator/, accessed on September 2017
[99] Teng, J., Tseng, C., Chen, Y., *et al.*: 'Integration of networked embedded systems into power equipment remote control and monitoring'. Proc. TENCON IEEE Region Conf., Chiang Mai, Thailand, November 2004, vol. **100**, (3), pp. 566–569
[100] Kyle, D., Brustoloni, J.: 'Uclinux: a Linux security module for trusted-computing-based usage controls enforcement'. Proc. ACM Wkshps on Scalable Trusted Computing, New York, NY, USA, November 2007, pp. 63–70
[101] Vujovic, V., Maksimovic, M.: 'Raspberry Pi as a wireless sensor node: performances and constraints'. Proc. Int. Convention Information and Communication Technology Electronics and Microelectronics (MIPRO), Opatia, Croatia, May 2014, pp. 1013–1018
[102] Prasad, S., Mahalakshmi, P., Sunder, A., *et al.*: 'Smart surveillance monitoring system using Raspberry Pi and PIR sensor', *Int. J. Comput. Sci. Inf. Technol.*, 2014, **5**, (6), pp. 7107–7109
[103] Kiepert, J.: 'Creating a Rraspberry Pi-based Beowulf cluster', Boise State University, May 2013, pp. 1–17
[104] Silva, S.: 'A Linux microkernel based architecture for OPENCV in the Raspberry Pi device', *Int. J. Sci. Knowl. (IJSK)*, 2014, **5**, (2), pp. 44–52
[105] The MagPi Magazine. Available at https://www.raspberrypi.org/magpi/tutorials/, accessed on September 2017
[106] Murikipudi, A., Prakash, V., Vigneswaran, T.: 'Performance analysis of real time operating system with general purpose operating system for mobile robotic system', *Indian J. Sci. Technol.*, 2015, **8**, (19), pp. 1–6
[107] Almasalha, F., Khokhar, A., Hasimoto-Beltran, R.: 'Scalable encryption of variable length coded video bit streams'. Proc. IEEE 35th Conf. Local Computer Networks (LCN), Denver, CO, USA, October 2010, pp. 192–195
[108] Bagal, N., Pandita, S.: 'A review: real-time wireless audio–video transmission', *Int. J. Emerg. Technol. Adv. Eng.*, 2015, **5**, (4), pp. 168–170
[109] MartinFerna'ndez, F., CaballeroGil, P., CaballeroGil, C.: 'Authentication based on non-interactive zero knowledge proofs for the Internet of things', *Sensors*, 2016, **16**, (1), p. 75
[110] Amorim, V., Delabrida, S., Oliveira, R.: 'A constraint-driven assessment of operating systems for wearable devices'. Proc. Computing Systems Engineering (SBESC), Joao Pessoa, Brazil, November 2016, pp. 150–155
[111] Android Things. Available at https://developer.android.com/things/, accessed on October 2017
[112] Android Platform Architecture. Available at https://developer.android.com/guide/platform/index.html, accessed on September 2017
[113] Akula, P., Yamuna, V., Ananda, C., *et al.*: 'Development of data logger for MAV using FREERTOS ON PIC32', *Int. J. Eng. Sci. Res. Technol.*, 2015, **4**, (9), pp. 2277–9655