A study of various aspects of the operating systems designed for the IoT (Internet of Things) environment and devices

Rohan Chaudhury
Department of Computer Science and Engineering
Texas A&M University
College Station Texas USA
rohan.chaudhury@tamu.edu

ABSTRACT

With the advent of recent technologies, it's easy to connect the real world to the digital cyber world by integrating efficient microcontrollers, radio transceivers, and different types of actuators and sensors into these putative smart-objects. These smart gadgets should be optimally connected to the Internet in accordance with the broader aim of the Internet of Things (IoT), which is defined as a network of physical devices integrated with software, sensors as well as other technologies for linking and sharing data between systems via the traditional internet. These physical devices can exchange and gather data with minimum human interaction thanks to the cloud, low-cost computers, big data, and mobile computing. Digital systems can capture, analyze, and modify each interaction between linked items in today's highly interconnected environment. In essence, these smart devices are severely limited in memory, energy, and computing resources and wireless connectivity between these aforementioned smart devices and the internet is not fast enough, suffering from considerable packet loss. These traits provide issues, both in terms of the network protocols used by the smart devices to interact and the software operating on these smart devices. These limitations should be considered when developing new operating systems, APIs, middleware, and frameworks for these smart devices. Several dedicated operating systems have been developed to address the issues described above. This paper aims to outline a brief study of these various dedicated operating systems that are currently available as well as the various elements of these operating systems which are targeted toward the efficient handling of the emerging technology of IoT devices where limitations in resources represent a significant difficulty for the execution of the general operating systems developed for standard computing hardware. I will begin by giving a brief background and history of the Internet of Things followed by an introduction to the general theme of this study. Then I will talk about the various prerequisites of the Operating Systems which are dedicated to Low-end Internet of Things devices. I will briefly discuss some available Operating Systems dedicated to the Internet of Things that more or less meet these prerequisites. I will then give a brief classification of these Operating Systems dedicated to IoT based on their architectural concepts followed by the conclusion of this study.

CCS CONCEPTS

• Software and its engineering → Operating systems. • Computer systems organization → Embedded and cyber-physical systems • Networks → Sensor networks

KEYWORDS

Operating Systems, Internet of Things, OS, IoT devices, Sensors, Low-end IoT devices

ACM Reference format:

Chaudhury, Rohan. 2022. A study of various aspects of the operating systems designed for the IoT (Internet of Things) environment and devices. In *Project 3 of 22 SPRING CSCE 611 600: OPERATING SYSTEMS. Texas A&M University, College Station, Texas, USA, 12 pages.*

1. Background

The Internet is a communication network that connects people to information, whereas the Internet of Things (IoT) is a network of uniquely addressable physical objects with varying degrees of computation, detection, and actuation functionalities that share the ability to connect and interact using the Internet as their common platform [1].

The "Internet" was originally established in 1989 and swiftly expanded over the globe. The practice of linking diverse devices to the Internet has exploded since the Internet's inception. The Coffee Pot Trojan program is the first of its type. John Romkey created the very first Internet gadget in 1990, a toaster that could be turned on and off through the Internet. Wearcam was created by Steve Mann in 1994. With a 64-processor machine, this occurred nearly instantly. Paul Saffo offered the first brief overview of censorship and

his future activities in 1997. Kevin Ashton, Managing Director of the Auto-Id Center at MIT, founded the Internet of Things in 1999. Additionally, they have created RFID-based systems for object identification all around the world. LG Electronics Giant issued a statement in 2000 about its plan to sell refrigerators that would now recognize if the stored food would be restocked or not, which was a big milestone toward IoT industrialization. As an aspect of the Savi Program in 2003, the US Army utilized RFID in large numbers. The IPSO Alliance was founded in 2008 by a collection of enterprises with the goal of increasing the usage of Internet Protocol (IP) within the smart object ecosystem and enabling the Internet of Things. The FCC approved the use of the "white space spectrum" in 2008. IPv6's release in 2011 sparked a wave of popularity and development in this field. Many academic and commercial efforts with IoT are being pursued by IT behemoths including Cisco, Ericson, IBM, etc. IoT technology may also be thought of as the most crucial link between humans, the Web, and different objects. Everything we are using in our everyday lives can now be supervised through the Internet. Sensors are used to carry out the vast majority of IoT tasks. Almost every gadget makes use of this sensor. The sensor collects physical unprocessed data, transforms it into a digital signal, and transmits it to the central controller. We may argue that the Internet of Things has made our lives much easier and more comfortable. The Internet of Things has a significant influence and plays a crucial role in people's lives [2]. The Market for IoT is depicted in Figure 1 below.

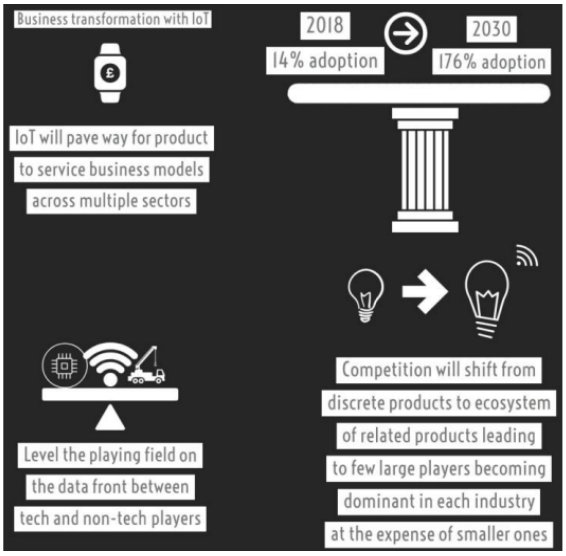

Figure 1: Market of IoT [2]

2. Introduction

The IoT is the result of the widespread availability of inexpensive, small, and energy-efficient communication devices. For the IoT network model, many standardized communication protocols have been created at various tiers. Because such protocols are available, diverse devices may be networked and accessed over the Internet.

The IoT is made up of more heterogeneous technology in the hardware domain than the traditional Internet. Based on their functionality and efficiency, IoT devices may be divided into two groups. High-end IoT devices, such as single-board computers (for example, Rasberry Pi [11]) and smartphones fall under the first group. Standard Operating Systems such as Linux may be operated on high-end IoT devices since they have ample resources and capabilities.

Low-end IoT devices belong to the other category which is too limited in resources to operate these standard operating systems. When it comes to addressing the issue of huge restrictions on hardware resources, these low-end IoT devices provide fresh difficulties to solve for Operating System designers [10] and so these types of IoT devices would be the main point of discussion in this paper. Real-time functionality, security awareness, low power utilization, connectivity, and inter-operability are all general requirements for such a dedicated Operating System for resource-constrained IoT devices [12]. Zolertia ReMote [6], Arduino [5], OpenMote nodes [8], IoT-LAB M3 nodes [7], and TelosB motes [9] are some examples of this type of device, some of which are depicted in Figure 2 below [4].

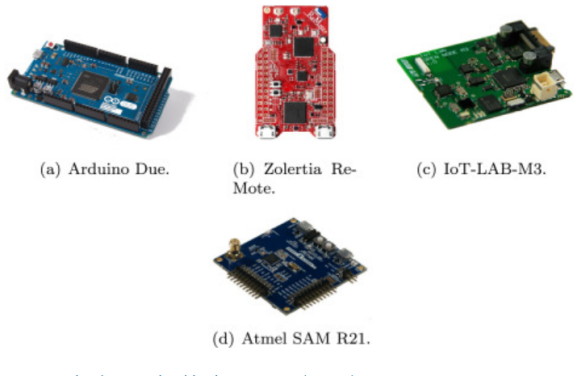

Figure 2: Some low-end Internet of Things devices [10]

The Internet Engineering Task Force has established a categorization system [13] for low-end IoT devices, dividing them into three classifications depending on their capacity of memory.

• Class 0 devices (for example a dedicated mote in a Wireless Sensor Network) possess the fewest resources

(less than 100 KB Flash and 10 KB RAM). High customization and resource limits make the usage of a standard operating system inappropriate in most cases on Class 0 devices. As a result, software designed for these kinds of devices is generally plain and extremely device-specific.

• Class 1 devices comprise moderate resources (around 100 KB Flash and 10 KB RAM), enabling bigger applications and sophisticated functionality such as secure communication and routing protocols, which are not available on simple motes.

• Class 2 devices possess greater capabilities than Class 0 or Class 1 devices but they are still more limited than regular Internet hosts and high-end IoT devices.

Class 1 and Class 2 IoT devices, on the other hand, are often less specific. With a common network stack and reconfigurable programs running on top of this stack, the software may convert such a unit into a server, host, or Internet router [14]. As a result, new technologies based on transportable, hardware-independent apps operating on IoT devices of these classes are now arising. Many large firms, including Google [17], ARM [21], etc., have recently introduced new Operating Systems developed exclusively for such classes of IoT devices. Likewise, it is frequently beneficial to be given software primitives that allow for straightforward device-agnostic code generation on such hardware. Besides raw programming, there exists a requirement for Application Programming Interfaces (APIs) that really can accommodate a diverse variety of IoT scenarios and make enormous application development, distribution, and support easier. An operating system usually provides such functionalities. Moore's law is unlikely to apply to these devices since IoT devices will get more compact, more energy-efficient, and inexpensive rather than enabling much more memory or computing power [3] [4]. As a result, IoT devices of type Class 1 & Class 2 are expected to prevail in the IoT domain in the coming future.

3. **Prerequisites of dedicated Operating Systems for Low-end Internet of Things devices**

Conventional operating systems such as Linux or BSD, as aforementioned, are incompatible with low-end IoT devices due to the restricted resources available. In this paper, I will provide a brief outline of dedicated Operating systems for IoT devices that are mostly Open-Source and that (i) meet moderate memory requirements of around 10 kB RAM and 100 kB Flash (Class 1 and above devices) (ii) are consistent with IP protocols from a network perspective, and (iii) are consistent with normal developer tools, designs, and languages used on the Internet. In this part, I will go through the many criteria that a standard Operating system for low-end IoT devices should strive to accommodate.

3.1 *Network Connectivity*

The fundamental benefit of IoT devices is how they can link and interact with each other as well as with the Internet. As a result, most IoT devices include one (or even more) network interfaces. In the Internet of Things, communication strategies include a wide range of low-power radio technologies (for example, Bluetooth/BLE, IEEE 802.15.4, EnOcean, DASH7, etc.), but also a number of tethered technologies (for example Ethernet, PLC, or other bus systems). In comparison to WSN wireless sensor network instances [15] [18], IoT devices are anticipated to smoothly connect with the Internet, that is, to be able to interact end-to-end with the other systems on the Internet [28]. Because of the need to handle several link-layer protocols while also communicating with other Internet hosts, networking frameworks relying upon IP standards have been used explicitly on IoT devices [22]. Support for diverse link layer protocols and networking frameworks relying upon IP protocols pertinent to the IoT is therefore a major prerequisite for a generalized Operating system for the IoT [22]. The OS should also be able to support numerous network stacks as well as ongoing network stack improvement. The standard network stack of IoT is shown in Figure 3 below.

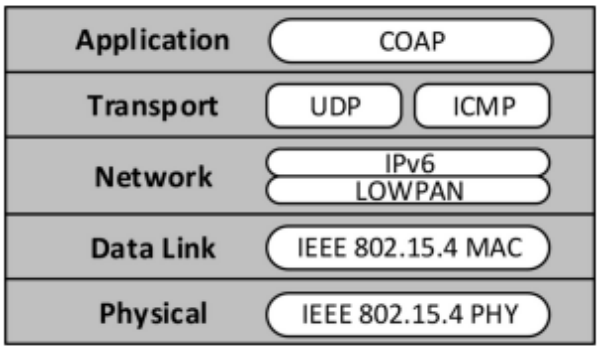

Figure 3: Standard network stack of IoT [12]

3.2 *Diverse Hardware Compatibility*

The degree of diversity in the IoT expands, despite the fact that the variety of hardware and standards utilized in recent Internet technology is rather limited from an architectural standpoint. Because of the wide range of IoT application

cases, a wide range of technologies has been developed. IoT devices use a variety of microcontroller designs and families. Furthermore, important platform attributes differ greatly: some IoT equipment, for instance, has hundreds of KBs of RAM but no permanent memory to hold executables resulting in the requirement to put both application code and data into RAM. Other IoT devices have a low amount of RAM but a substantial quantity of storage. [10] As a result, supporting diversity in hardware designs and networking protocols is among the key necessities a fundamental challenge for a generalized Operating system for the IoT.

3.3 *Minimal Memory Consumption*

IoT devices have far fewer resources than traditional linked equipment, particularly in terms of memory. Fitting under such memory limits is thus one of the prerequisites for a generalized IoT Operating system. While standard computing devices like desktops or smartphones have GBs or TBs of RAM or ROM, IoT devices only have a few KBs of both. IoT program developers should be equipped with a collection of efficient packages that offer typical IoT features and optimized data structures in order to conform to memory footprint limits. In many circumstances, the OS designer must find the optimal balance of RAM and ROM utilization. In addition, a compromise should be found between rational programming rules and programming standards and the great level of flexibility and customizability that is sought to suit a broad variety of usage scenarios. [10]

3.4 *Capabilities for Real-Time Operation*

In various IoT use-cases, such as in robotic systems in factory automation settings or in intelligent healthcare apps (for example, IoT networks integrated with pacemakers delivering mobile supervision and management [19]), high precision in timing and prompt operation are critical. A Real-Time Operating System (RTOS) "is an operating system (OS) for real-time applications that process data and events that have critically defined time constraints. An RTOS is distinct from a time-sharing operating system, such as Unix, which manages the sharing of system resources with a scheduler, data buffers, or fixed task prioritization in a multitasking or multiprogramming environment. Processing time requirements need to be fully understood and bound rather than just kept to a minimum. All processing must occur within the defined constraints. Real-time operating systems are event-driven and preemptive, meaning the OS is capable of monitoring the relevant priority of competing tasks and making changes to the task priority. Event-driven systems switch between tasks based on their priorities, while time-sharing systems switch the task based on clock interrupts." [30] It is meant to ensure worst-case operation durations and worst-case interrupt latencies. As a result, another need for a general IoT Operating system is that it should be an RTOS.

3.5 *Energy Efficiency [10]*

Most types of IoT equipment are powered by battery packs or other sources of energy that are limited. Intelligent meters as well as other household automation systems, for example, are needed to run for a long time on a given charge [16]. Microcontrollers, transceivers, and sensors are all instances of IoT devices that include energy-saving characteristics. However, energy efficiency cannot be accomplished until IoT applications make use of such characteristics. As a result, some of the most important requirements for IoT Operating Systems are to give electricity-saving alternatives to higher levels and to employ these capabilities as often as feasible, for instance, through utilizing methods like radio duty cycling or reducing the number of periodical activities that must be performed.

3.6 *Safety and Security*

Vulnerabilities in software are flaws in the source code. Vulnerabilities arise from instabilities in the language used, as well as developers' disregard for safe programming techniques, the stress of timelines, or a dearth of managerial emphasis on the issue [32]. Vulnerabilities in IoT systems can have serious effects since they incorporate various equipment, sensors, and actuators that interact closely with individuals in several of our operations. Ponder the devastating possibilities of exploiting a pacemaker or a self-driving automobile. The reality that the majority of IoT Operating Systems are developed in C/C++ because of their extremely robust low-level programming capability greatly worsens the matter since they are, nevertheless, amongst the least secure coding languages available. According to certain research, C programs were responsible for 50% of vulnerabilities reported in open-source projects between 2009 and 2019 [33] [31].

IoT devices are required to fulfill stringent privacy and security requirements. IoT security concerns include integrity of data, identification, and security systems in various aspects of the IoT architecture, in addition to the overall trust management difficulty. As a result, providing the appropriate technologies (security standards and strong encryption tools) while maintaining adaptability and

accessibility is a prerequisite and a difficulty for an Operating System dedicated to the IoT.

4. **Available Operating Systems dedicated to the Internet of Things**

In this section, I'll give a quick overview of several of the current operating systems targeted for low-end IoT devices. These Operating Systems mostly meet the requirements outlined in the 3rd section of this paper, and they are currently used widely and receiving ongoing support in the scope of low-end IoT applications.

4.1 *RIOT [20], [23], [34]*

RIOT production commenced in 2013, and it is built on a microkernel architecture derived from FireKernel [35], a microkernel originally designed for WSN situations requiring real-time operating abilities. RIOT progress so far has centered on expanding IoT hardware compatibility (8, 16, 32-bit microcontrollers), writing optimized cross-platform programs, and creating and maintaining multiple network stacks. The microkernel design of RIOT enables full multi-threading. Because multi-threading often adds run-time and memory cost, special emphasis was given to efficient context switching designs, inter-process communication (blocking and non-blocking), and a compact thread control block. As a result, context switching in RIOT takes only a few CPU cycles (for example, fewer than 100 on an ARM platform when initiated from an interrupt) and the thread control block is lowered to 46 bytes on 32-bit devices.

RIOT has a tickless scheduler that doesn't require any periodic triggers to operate. If there are no outstanding tasks, RIOT will transition to the idle thread, that, depending on the peripherals in use, can employ the deepest feasible sleep state. Just kernel-generated or external interrupts can cause the system to wake up from its idle state. Either static or dynamic memory allocation is allowed by RIOT. Within the kernel, however, only static methods are utilized, allowing RIOT to meet deterministic criteria by ensuring consistent times for kernel tasks such as timer operations, inter-process communication, or scheduler run.

To facilitate network communication, RIOT provides a number of stacks. The gnrc network stack is built on industry-standard IP protocols, including RPL (non-storing and storing mode), 6LoWPAN, IPv6, CoAP, and UDP, and is developed in a modular manner using general, well-defined interfaces and inter-process communication. CCNlite, which implements the Information Centric Networking paradigm, and OpenWSN, which implements the entire 6TiSCH [39] protocol stack, are two more network stacks accessible as BSD-like packages. The gnrc is RIOT's default network stack, which stores packets, headers, and other routing metadata in a central network buffer structure, with only pointers passing between levels.

RIOT offers a broad range of features, including a terminal, multiple crypto packages, and complex data structures, in addition to many network stacks. In RIOT, the programming architecture is based on traditional multi-threading, including memory-passing inter-process communication among threads. Its kernel is developed in the C programming language with very minimal portions being developed in assembler. For applications and libraries, meanwhile, both C and C++ are supported as programming languages. For peripheral interfaces, networking, sensors, and actuator devices, RIOT includes a very well-defined hardware abstraction layer. Because RIOT is coded in ANSI C, popular and well-established debugging tools like GDB and Valgrind may be utilized.

RIOT also allows users to execute instances of the Operating system as processes on Linux or Mac OS, allowing for simple debugging of programs as well as virtual network emulation utilizing nativenet to imitate a single ethernet connection or the desvirt framework [41] for more sophisticated architectures. Cooja may potentially be utilized to emulate platforms that this emulator supports. For smoke and regression testing, RIOT provides a collection of unit tests and apps. Travis, a web-based service platform, is used for continuous integration testing. In addition, a distributed test framework was created in order to run the tests on all systems supported [40]. Tests could also be run on a variety of RIOT-supported open testbeds, such as IoT-LAB [24][7]. RIOT emphasizes establishing standardized interfaces such as POSIX on the system side. In terms of networking, RIOT focuses on open standard protocols defined by organizations such as the OMA, IETF, W3C, IRTF, and others.

RIOT is powered by an open-source community mostly originating mostly from academics that attempts to offer extensive documentation on both the Application programming interface and design levels. While the code's syntax and documentation have already met high quality, the community is presently revamping sample code and high-level explanations. The core components of RIOT have been utilized by a network of people and programmers for a long time mostly by people in academia, but lately, it is also used by industry, primarily for more easy prototyping. The kernel, for example, can be called robust because only

small flaws have been discovered in recent years. Some components of RIOT, such as the network stack, are more recent and are potentially susceptible to modification, such as, due to co-evolving with current IoT network protocols as they arise.

The core components of RIOT have been utilized by a network of people and programmers for a long time mostly by people in academia, but lately, it is also used by industry, primarily for more easy prototyping. The kernel, for example, can be called robust because only small flaws have been discovered in recent years. Some components of RIOT, such as the network stack, are more recent and are potentially susceptible to modification, such as, due to co-evolving with current IoT network protocols as they arise. The adoption of standardized and general interfaces (such as netapi or POSIX sockets) is, however, stabilizing the code base's usage. Several developers manage RIOT's master branch on GitHub (RIOT is open-sourced) and are responsible for evaluating and merging outside modifications submitted via pull requests. RIOT makes it possible to create secure IoT apps. IEEE 802.15.4 encryption, DTLS transport layer security, Secure Firmware Updates (SUIT), various cryptographic packages, and crypto secure components are all supported by RIOT.
Table 1 outlines some of the major attributes of RIOT and how it differs from Contiki, Tiny OS, and Linux.

OS	Min RAM	Min ROM	C Support	C++ Support	Multi-Threading	MCU w/o MMU	Modularity	Real-Time
Contiki	<2kB	<30kB	◦	✘	◦	✓	◦	◦
Tiny OS	<1kB	<4kB	✘	✘	◦	✓	✘	✘
Linux	~1MB	~1MB	✓	✓	✓	✘	◦	◦
RIOT	~1.5kB	~5kB	✓	✓	✓	✓	✓	✓

Table 1 [20]: Major Attributes of CONTIKI, TINYOS, LINUX, and RIOT. Full support is denoted by (✓), Partial support is denoted by (◦), No support is denoted by (✘)

4.2 *CONTIKI [37], [36], [27]*

Adam Dunkels created Contiki in 2003 as an operating system based on the uIP stack for resource-limited embedded computer systems. Contiki has grown into a more broad operating system that is utilized in fields including the Internet of Things, WSNs, & also in retro computing. It works with 8-bit AVR systems, 16- and 20-bit MSP430 platforms, and 32-bit ARM Cortex M3 platforms, among other resource-limited platforms. Contiki does have a monolithic architecture, which means that it consists of a core system and a collection of programs that are compiled into a unified system image.

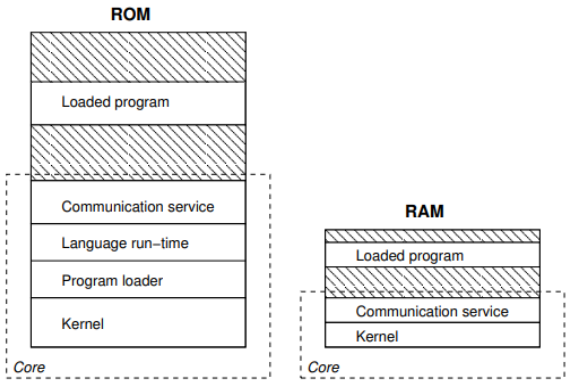

Figure 4 [37]**:** A Contiki system is divided into two parts: the core and the loaded programs.

All programs share the very same memory and permissions as the core system at runtime. As illustrated in Figure 4, a Contiki system is divided into two parts: the core and the loaded programs. Contiki's scheduling model is cooperative thread scheduling, which necessitates Contiki processes to explicitly relinquish control to the scheduler. Contiki is mainly structured for static allocation when it comes to memory allocation. Memb and mmem are two libraries that make memory management easier. Third-party dynamic allocation packages for Contiki that implement the conventional C malloc API are also available. The uIP stack, which was initially designed as an independent stack and was included in Contiki after version 0.9, as well as the lighter Rime stack, which is focused on sensor network applications, are the two network stacks that may be used to equip Contiki products with network access. IPv4, IPv6, 6LoWPAN, UDP, IPv6 neighbor discovery, IPv6 multicasting, RPL, and TCP are among the protocols supported by uIP. Queuebuf, a dedicated component that distributes packet buffers from a static reservoir of memory, handles network buffer governance.

Protothreads, a lightweight collaborative threading concept related to continuations [26], is used in the programming approach. Contiki's primary programming language is C, however, there are runtime environments that allow for development in languages like Python [25] and Java [42]. Contiki offers a hardware layer of abstraction wherein functionality specific to particular hardware is separated into independent components and a standard application programming interface for utilizing that hardware is implemented across all supported Contiki platforms. Clocks, radio drivers, and sensors, for example, each have their own Application programming interface, which varies depending on the platform.

The Cooja/MSPsim emulator, which blends network simulations and cycle-accurate simulation of the hardware, is the main debugging utility made available to Contiki developers. Reading from and writing to particular memory locations, setting breakpoints, and single-stepping through commands are all included in this emulator. Contiki also supports hardware utilized on a number of open testbeds, including IoT-LAB [7] and Indriya [43].

Contiki features a large feature set, in addition to networking and fundamental system operations. It has a terminal, a database management system, a file system, cryptography libraries, a fine-grained power tracing tool, and runtime linking dynamically, among other things. Contiki provides specific testing tools, including regression and unit testing, and complete system integration testing, to improve the efficiency of all of these functionalities. Travis CI [44] is used to autonomously verify Contiki code additions with a test suite. Contiki supports a variety of standards, most of which are primarily linked to networking. Contiki, for example, implements 6LoWPAN and RPL, two IETF specifications for low-power IPv6 networking. Contiki also received a silver certification in the IPv6 Ready Logo Program for its fundamental IPv6 capabilities.

Contiki's documentation varies in level of information for different aspects of the program. The open-source Doxygen tool is used to document the code, and a project Wiki is used to provide supplementary resources like lessons and technical explanations. Contiki's main components have matured to a high level of code maturity, however, there are still less utilized experimental elements of the operating system. These experimental elements consist mostly of programs or libraries that were created as a result of research efforts.

Contiki has been utilized in a variety of real-world implementations and is extensively employed in market IoT solutions and also in academia on wireless sensor networks and other restricted wireless multi-hop networks. Contiki, along with TinyOS, is now amongst the most popular and commonly used wireless sensor network operating systems.

Contiki is created by a broad group of professional programmers, researchers, and enthusiasts. The development is centered on a GitHub repository, where anybody may offer code contributions in the form of pull requests. A merge team manages the source code, reviewing new code submissions from the Contiki community and making broader choices like major architectural modifications and deployment cycles. There are several source code branches wherein separate organizations create fresh functionalities or corporations manage their respective implementations of Contiki with possible compatibility for their own hardware. Additionally, because Contiki has a large number of users from academia, research initiatives for particular versions of Contiki are routinely established.

4.3 *FreeRTOS [46], [29]*

FreeRTOS was created in 2002 by Richard Barry and is presently managed and provided via Real Time Engineers Ltd. FreeRTOS is being utilized in a range of commercial and professional settings and provides the foundation for a number of research initiatives. FreeRTOS is meant to be compact, simple, mobile, and straightforward to use, in comparison to several other Real-Time Operating Systems. As a result, it has a significant community behind it and it has been adapted to a wide variety of Microcontrollers, which also includes open testbed hardware such as IoT-LAB [7].

FreeRTOS has a basic design, consisting of just 4 C files and being more of a threading library rather than a complete OS. Mutexes, thread handling, software timers, and semaphores are the only features available. A preemptive, round-robin scheduler based on priority is used by default in FreeRTOS, and it is prompted by a periodical timer interrupt. The scheduler currently provides a tickless mode from version 7.3.0. To assure that real-time requirements are met, FreeRTOS only performs deterministic actions from within an interrupt or critical section. Queues are utilized for Interprocess Communication in FreeRTOS, and they allow both blocking and nonblocking insert via deep copy, as well as support removal functions.

There are no networking features in FreeRTOS. In the FreeRTOS ecosystem, however, there are several other packages and tools which are primarily via 3rd parties. Real-Time Engineers Ltd., for example, provides an authorized FreeRTOS+TCP add-on that enables an Ethernet-based IPv4 framework with TCP, UDP, and other protocols support. There are also implementations of 3rd-party integrated network stacks such as lwIP [38] available. The stack utilized determines how network buffers are managed. For example, the official FreeRTOS+TCP may be set to utilize a dynamically allocated buffer space when requested or a static pre-allocated buffer. With statically created tasks, FreeRTOS offers a multi-threading computing approach. The operating system's scripting language is C, allowing users to simply incorporate it into any C++ program.

FreeRTOS defines five memory allocation schemes:
(a) only allocate,
(b) allocating & freeing with a straightforward and rapid algorithm,
(c) a more complex but rapid free and allocate algorithm with memory coalescence,
(d) a somewhat more advanced version of (d) that enables memory coalescence, which allows a heap to be split over many memory locations
(e) along with mutual exclusion protection, and the C library free and allocate [46]

Microcontroller peripheral abstraction interfaces or a portable driver model are not specified in FreeRTOS. Instead, it collaborates with board support modules provided by vendors. The platform also relies on 3rd-party tools for testing and troubleshooting, despite the Operating systems' architecture allowing it to be incorporated into many current development workflows. FreeRTOS stresses tight code standards, certification, and quality management in order to conform to the standards and criteria of commercial usage scenarios. As a result, FreeRTOS has been included in a number of formal verification initiatives [45]. FreeRTOS has a plethora of documentation available online.

There are several variations of the FreeRTOS available, such as Amazon FreeRTOS, OpenRTOS, SafeRTOS.

Amazon FreeRTOS [46]:

Amazon supplies a:FreeRTOS, which is a FreeRTOS extension. This is FreeRTOS with Amazon Web Services-specific packages enabling IoT compatibility. Amazon has been in charge of the FreeRTOS code from release 10.0.0 in 2017, together with any changes to the base kernel.

OPENRTOS [46]:

WITTENSTEIN High Integrity Systems sells OPENRTOS, a commercially licensed variation of Amazon FreeRTOS. This package provides support and enables businesses to utilize the Amazon FreeRTOS kernel and packages without the need for MIT license of a:FreeRTOS.

SAFERTOS [46]:

SAFERTOS was created as a counterpart to FreeRTOS, with similar functionality but a focus on security applications. FreeRTOS was put through a hazard and operability assessment (HAZOP), and flaws were found and fixed. The final product underwent the complete IEC 61508 SIL 3 development cycle, which is the maximum level for a component that is software-only.

SAFERTOS was created in collaboration with Real Time Engineers Ltd, the major creator of the FreeRTOS project, by Wittenstein High Integrity Systems. Both SAFERTOS and FreeRTOS utilize identical scheduling algorithms, offer identical APIs, and are practically identical, but their goals are different. SAFERTOS was written entirely in C to comply with IEC61508 certification standards.

For standards compliance, SAFERTOS can be stored entirely in a microcontroller's on-chip ROM. SAFERTOS code could only be utilized in its initial, authorized configuration when it is implemented in hardware memory. This means that there is no requirement to retest the kernel element of a system while validating a system. SAFERTOS is integrated into the read-only memory of various Texas Instruments Stellaris Microcontrollers. The source code for SAFERTOS does not require to be obtained explicitly. A C header file is utilized to link SAFERTOS application programming interface functions to their ROM locations in this use case.

4.4 *mbedOS [21], [47], [10]*

Mbed is an internet-connected device platform and OS that is based on the 32-bit ARM Cortex-M microcontrollers and supports a limited selection of platforms. Arm and its technology partners are collaborating together on the project. Mbed OS includes a C/C++ software platform for the Mbed platform as well as utilities for developing microcontroller programs for Internet of Things (IoT) devices. It comprises the RTOS and runtime environment, microcontroller peripheral drivers, development tools, networking, and test and debug scripts provided by the core libraries. Compatible SSL/TLS libraries, such as Mbed TLS or wolfSSL, which enable mbed-rtos, can secure these connections. ARM has recently published a technical preview version of its mbed OS Operating system for low-end IoT devices which is designated as 15.11. A closed source 6LoWPAN application which professes to fulfill the Thread 1.0 protocol, a PolarSSL port, numerous interface definitions, and Bluetooth Low Energy compatibility are among the experimental capabilities shown in the preview.

4.5 *TinyOS [48], [49]*

TinyOS is a component-based embedded operating system and architecture for low-power wireless systems utilized in

ubiquitous computing, wireless sensor networks, personal area networks, smart meters, smartdust, and smart buildings. TinyOS, along with Contiki, is the most popular Operating system for wireless sensor network applications, focusing on 8-bit and 16-bit architectures, and is noted for its complex design. It is implemented as a series of collaborating tasks and procedures utilizing the programming language nesC, a C language variant adapted for the memory constraints of sensor networks.

It started as a collaborative project between Intel Research, the University of California at Berkeley, and the Crossbow Technology, and was distributed as open-source software under the BSD license. It has subsequently established into the TinyOS Alliance, an international consortium. The majority of TinyOS' additional utilities are Java & shell scripting front-ends. The nesC compiler & Atmel AVR binutils toolchains are primarily developed in C, and so are the related packages and utilities. TinyOS applications are made up of software elements, including some that use hardware abstractions. Interfaces are used to link elements to one another. TinyOS offers standard abstractions like routing, storage, packet communication, sensing, and actuation with interfaces and components.

TinyOS features a single call stack and is completely non-blocking. As a result, every I/O action spanning more than a mere hundred microseconds is asynchronous and requires a callback. TinyOS employs nesC's capabilities to statically link these callbacks, termed events, to allow the compiler to efficiently optimize over call boundaries. TinyOS' non-blocking nature allows it to retain great concurrency with a single stack but also compels programmers to build sophisticated algorithms by combining numerous tiny event handlers. TinyOS includes tasks, which are comparable to Deferred Procedure Calls and interrupt handler bottom halves, to facilitate bigger calculations.

TinyOS tasks operate in a FIFO sequence and are not preemptive. This basic concurrency paradigm is usually suitable for I/O-centric programs, but its inability to handle CPU-intensive workloads resulted in the development of TOSThreads, a thread library for the Operating system. TOSThreads has been deprecated and is no longer supported.

4.6 *Nano-RK [50]*

Nano-RK is a wireless sensor networking RTOS created by Alexei Colin, Christopher Palmer, and Artur Balanuta at Carnegie Mellon University. It is meant to operate on microcontrollers and be used in sensor networks. In order to enable real-time task sets, Nano-RK includes a fixed-priority completely preemptive scheduler with fine-grained scheduling primitives. The term "nano" denotes a tiny RTOS, with 2 KB of RAM and 18 KB of flash memory, whereas RK stands for resource kernel. A resource kernel specifies how frequently available resources of the system can be used. These restrictions serve as a simulated energy limit, ensuring that a node's battery lifespan is met and that a failing node does not generate additional networking traffic. Nano-RK is developed in C, is open-source, and operates on the MSP430 CPU, MicaZ motes, and the Atmel-based FireFly sensor networking platform.

NanoRK uses priority-based preemptive scheduling to ensure task punctuality and coordination while respecting the real-time component of being consistent. With the aid of a lightweight wireless networking stack, it enables packet forwarding, routing, and other network scheduling protocols.

For energy consumption control, Nano-RK offers a static design-time technique that prohibits task creation dynamically, forcing application developers to define both work and allocation quotas and priorities inside a static testbed design. This design enables the creation of an energy budget for every activity, ensuring that the system's application needs and energy efficiency are met throughout its lifespan. All of the runtime configurations, as well as the power needs, are established and confirmed by the developer using a static configuration technique prior to the system deployment and execution. In comparison to traditional Real-Time Operating Systems, this approach also aids in ensuring consistency and compact size.

Watchdog Timer is supported by Nano-RK. If the system lingers on critical faults for a lengthy period of time, the Watchdog software timer initiates a system reset. By waiting until the timer runs out and then restarting the device, this mechanism can return the system back to regular functioning from its nonresponsive condition. The watchdog timer in Nano-RK is directly connected to the processor's "REBOOT ON ERROR" reset signal. It is activated by default when the system starts up and is reset every time the scheduler runs. If the system does not reply within the predetermined time, it will reboot and restart the startup instructions in the hopes of regaining control.

If there are no suitable tasks to execute, the machine can be turned down and given the choice to undergo deep sleep mode for energy-saving purposes. Just the deep sleep timer can awaken the machine with a predetermined delay interval when it is in this state. The subsequent context

swap time is configured after awakening from sleep mode to ensure the CPU wakes up on time. If a sensor node does not want to fall into a deep sleep, it has the option of switching to a low-energy mode while still controlling its peripherals.

4.6 *eCos [51], [52]*

eCos (Embedded Configurable Operating System) is an open-source RTOS designed for embedded systems and programs that only require a single process with many threads. eCos is built to be adaptable to specific run-time efficiency and hardware constraints. It contains compatible layers and APIs for POSIX and the Real-time Operating System Nucleus variation ITRON, and it is written in C and C++. Popular SSL/TLS libraries like wolfSSL support eCos, ensuring that it complies with all embedded security standards.

eCos was created for systems with memory capacities ranging from a few 10s to several 100s kBs, as well as real-time applications. ARM, Hitachi H8, MIPS, Motorola 68000, FR-V, CalmRISC, IA-32, Nios II, PowerPC, Matsushita AM3x, SuperH, NEC V850, and SPARC are among the hardware platforms that eCos supports. RedBoot, an open-source program that leverages the hardware abstraction layer of the eCos to create bootstrap firmware for embedded systems, is included in the eCos release.

Cygnus Solutions, which was eventually purchased by Red Hat, created eCos in 1997. Red Hat stopped developing eCos in early 2002, and the project's personnel was let off. Many of the laid-off employees kept working on eCos, and some even started their own firms to provide software services for eCos. Red Hat decided to pass the eCos rights to the Free Software Foundation, at the petition of the eCos programmers and the process was finished in May 2008.

5. **Classification of IoT-dedicated Operating Systems**

The operating systems examined in the preceding section may be divided into three groups based on their architectural concepts: (a) pure Real-Time Operating Systems, (b) multithreading Operating Systems, and (iii) event-driven Operating Systems. The features of each group will be briefly described in this part [10].

5.1 *Pure Real-Time Operating Systems*

In an industrial environment, a Real-Time Operating System (RTOS) is primarily concerned with achieving real-time assurances. Standardization, formal verification, and certification are frequently critical in this setting. The programming architecture employed in such Operating Systems often puts stringent requirements on developers to facilitate model testing and formal verification. Because of these limitations, the Operating system is frequently rigid, making migration to different hardware architectures challenging. FreeRTOS, RTEMS, ThreadX, and a number of other industrial offerings are among the OS's for Internet of things devices that fit into this category. Due to its widespread use in diverse scenarios, FreeRTOS is quite a popular open-source Real-Time Operating System for Internet of Things devices.

5.2 *Operating Systems that are event-driven*

The essential concept of this paradigm is that every system processing is initiated by an event that is external in nature and is generally communicated by an interrupt. As a result, the kernel is approximately equal to an endless loop, which handles all events that occur in the very same context. Usually, an event handler of this type executes until it completes. Although this technique is economical in terms of memory use and sophistication, it places significant restrictions on the developer, for example, not every program can be represented as a state machine. The extensive calculation required for cryptographic procedures is one instance. On CPU-constrained devices, such activities generally take many seconds for completion. A long calculation in a fully event-driven operating system completely monopolizes the CPU, rendering the system incapable to react to exterior events. That wouldn't be an issue if the OS was built on preemptive multi-threading, as a long operation could be preempted.[37].

Contiki, TinyOS, and OpenWSN are examples of this type of operating system.

5.3 *Operating Systems that support multi-threading*

Most current operating systems (for example, Linux) use multi-threading, in which every thread operates in its separate context and maintains its separate stack. With this strategy, considerable scheduling is required to conduct thread-to-thread context switching. Every process is maintained on its separate thread, which may be halted at any time. Stack memory is typically not shared amongst threads. As a result, runtime overhead from context

switching and memory waste from stack oversupplying are common in multi-threading operating systems. RIOT, eCos, nuttX, and other operating systems come within this group.

6. **Conclusion**

In this paper, I have provided a brief study of the various dedicated operating systems that are currently available for low-end IoT devices which are too resource-limited to run general operating systems (such as Linux) developed for standard computing hardware. I have outlined the various key prerequisites for such a dedicated operating system which are targeted toward the efficient handling of the emerging technology of IoT devices as well as briefly discussed some of the available Operating Systems dedicated to the Internet of Things that more or less meet these prerequisites. Additionally, I have also provided a brief classification of these Operating Systems based on their architectural concepts and provided some example operating systems for each of these classifications. There are several distinct operating systems to choose from for the domain of the Internet of Things. The different Operating Systems have their own pros and cons which I have briefly outlined in this paper and based on these details the users can make the best choice of Operating Systems to use based on their needs. The Internet of Things technology is rapidly evolving as it is highly beneficial for humankind and so further research in the domain of efficient and secure Operating Systems for the IoT is essential.

7. **Acknowledgements**

I would like to extend my sincere thanks to several individuals without whom this study would not have been feasible. Firstly, I would like to thank Dr. John Hamilton for providing this wonderful opportunity to explore a recent topic of research in Operating systems and also for helping me to choose the topic and answering any questions that I had. Then I would like to thank our Teaching Assistant, Reza Dehkordi for clearing any doubts that I had regarding the project. I would also like to thank my peers, Rohit Sah, Abhishek Sinha, Sherine Davis Kozhikadan, and Shubham Gupta, for their many suggestions for the topic of this paper and their continuous help and support in clearing my concerns about some of the crucial concepts of the topic. Lastly, I would like to thank my Father, Grandmother, Sister, and friends for their continuous support in my journey.

REFERENCES

[1] A. H. Hussein, “Internet of Things (IoT): Research Challenges and Future Applications”, in International Journal of Advanced Computer Science and Applications (IJACSA), Vol.10, No.6, pp 77-82, 2019.

[2] Malik, A.; Magar, A.T.; Verma, H.; Singh, M.; Sagar, P. A Detailed Study of an Internet of Things (IoT). Int. J. Sci. Technol. Res. 2019, 8, 2989–2994.

[3] L. Mirani, Chip-makers are betting that Moore’s law won’t matter in the Internet of Things, 2014. http://qz.com/218514.

[4] Y.B. Zikria, H. Yu, M.K. Afzal, M.H. Rehmani, O. Hahm, Internet of Things (IoT): Operating System, Applications and Protocols Design, and Validation Techniques, Future Generation Computer Systems, 2018, pp. 699–706.

[5] Arduino Due. URL http://arduino.cc/en/Main/arduinoBoardDue.

[6] Zolertia, Z1 Datasheet. URL http://www.zolertia.com/.

[7] IoT-LAB: Very large scale open wireless sensor network testbed, 2016. https://www.iot-lab.info/

[8] OpenMote,OpenMote-CC2538.URL http://www.openmote.com/hardware/openmote-cc2538-en.html

[9] MoteIV Corporation, Telos — Ultra Low Power IEEE 802.15.4 Compliant Wire-less Sensor Module, Datasheet. URL http://www.willow.co.uk/html/telosb_mote_platform.php.

[10] Oliver Hahm, Emmanuel Baccelli, Hauke Petersen, Nicolas Tsiftes. Operating Systems for Low-End Devices in the Internet of Things: a Survey. IEEE internet of things journal, IEEE, 2016, 3 (5), pp.720-734. 10.1109/JIOT.2015.2505901 . hal-01245551

[11] E. Upton and G. Halfacree, Meet the Raspberry Pi. John Wiley & Sons, 2012.

[12] M. Silva, D. Cerdeira, S. Pinto and T. Gomes, "Operating Systems for Internet of Things Low-End Devices: Analysis and Benchmarking," in IEEE Internet of Things Journal, vol. 6, no. 6, pp. 10375-10383, Dec. 2019, doi: 10.1109/JIOT.2019.2939008.

[13] C. Bormann, M. Ersue, and A. Keranen, “Terminology for constrained node networks,” RFC 7228 (Informational), Internet Engineering Task Force, May 2014. [Online]. Available: http://www.ietf.org/rfc/rfc7228.txt

[14] M. Durvy, J. Abeille, P. Wetterwald, C. O’Flynn, B. Leverett, ´ E. Gnoske, M. Vidales, G. Mulligan, N. Tsiftes, N. Finne et al., “Making sensor networks ipv6 ready,” in Proceedings of the 6th ACM conference on Embedded network sensor systems. ACM, 2008, pp. 421–422

[15] W. Dong, C. Chen, X. Liu, and J. Bu, “Providing os support for wireless sensor networks: challenges and approaches,” Communications Surveys & Tutorials, IEEE, vol. 12, no. 4, pp. 519–530, 2010.

[16] R. Min et al., "Energy-centric enabling technologies for wireless sensor networks," in IEEE Wireless Communications, vol. 9, no. 4, pp. 28-39, Aug. 2002, doi: 10.1109/MWC.2002.1028875.

[17] Anand Karwa, Trak.In. Google Brillo – An Internet Of Things OS That Runs on 32 MB RAM. [Online]. Available: http://trak.in/tags/business/2015/05/23/google-brillo-internet-of-things-operating-system/

[18] L. Saraswat and P. S. Yadav, “A comparative analysis of wireless sensor network operating systems,” International Journal of Engineering and Technoscience, vol. 1, no. 1, pp. 41–47, 2010.

[19] A. Milenkovic, C. Otto, and E. Jovanov, “Wireless sensor networks for personal health monitoring: Issues and an implementation,” Computer communications, vol. 29, no. 13, pp. 2521–2533, 2006.

[20] E. Baccelli, O. Hahm, M. Gunes, M. W ¨ ahlisch, and T. C. Schmidt, “RIOT OS: Towards an OS for the Internet of Things,” in 32nd IEEE INFOCOM, IEEE. Turin, Italy: IEEE, 2013.

[21] mbed OS: https://mbed.org/technology/os/

[22] M. R. Palattella, N. Accettura, X. Vilajosana, T. Watteyne, L. A. Grieco, G. Boggia, and M. Dohler, “Standardized protocol stack for the internet of (important) things,” Communications Surveys & Tutorials, IEEE, vol. 15, no. 3, pp. 1389–1406, 2013.

[23] E. Baccelli, O. Hahm, H. Petersen, and K. Schleiser, “RIOT and the Evolution of IoT Operating Systems and Applications,” ERCIM News, vol. 2015, no. 101, 2015. [Online]. Available: http://ercim-news.ercim.eu/en101/special/riot-and-the-evolution-of-iot-operating-systems-and-applications

[24] C. Adjih, E. Baccelli, E. Fleury, G. Harter, N. Mitton, T. Noel, R. Pissard-Gibollet, F. Saint-Marcel, G. Schreiner, J. Vandaele, and T. Watteyne, “FIT IoT-LAB: A Large Scale Open Experimental IoT Testbed,” in Proceedings of the 2nd IEEE World Forum on Internet of Things (WF-IoT), December 2015.

[25] S. Bocchino, S. Fedor, and M. Petracca, “PyFUNS: A Python Framework for Ubiquitous Networked Sensors,” in Proceedings of the European Conference on Wireless Sensor Networks (EWSN), Porto, Portugal, Feb. 2015.

[26] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali, “Protothreads: Simplifying event-driven programming of memory-constrained embedded systems,”

in Proceedings of the International Conference on Embedded Networked Sensor Systems (ACM SenSys), Boulder, Colorado, USA, Nov. 2006
[27] https://en.wikipedia.org/wiki/Contiki
[28] R. Jedermann, T. Potsch, and C. Lloyd, “Communication techniques ¨ and challenges for wireless food quality monitoring,” Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences, vol. 372, no. 2017, 2014.
[29] https://www.freertos.org/
[30] https://en.wikipedia.org/wiki/Real-time_operating_system
[31] Al-Boghdady, A.; Wassif, K.; El-Ramly, M. The Presence, Trends, and Causes of Security Vulnerabilities in Operating Systems of IoT’s Low-End Devices. Sensors 2021, 21, 2329. https://doi.org/10.3390/s21072329
[32] Ibrahim, A.; El-Ramly, M.; Badr, A. Beware of the Vulnerability! How Vulnerable are GitHub’s Most Popular PHP Applications? In Proceedings of the IEEE/ACS 16th International Conference on Computer Systems and Applications (AICCSA), Abu Dhabi, United Arab Emirates, 3–7 November 2019.
[33] WhiteSource. What Are the Most Secure Programming Languages. WhiteSource Software, 17 March 2019. Online: https://resources.whitesourcesoftware.com/research-reports/what-are-the-most-secure-programming-languages
[34] RIOT Operating System. http://www.riot-os.org.
[35] H. Will, K. Schleiser, and J. H. Schiller, “A real-time kernel for wireless sensor networks employed in rescue scenarios,” in IEEE LCN, 2009.
[36] https://www.contiki-ng.org/
[37] A. Dunkels, B. Gronvall and T. Voigt, "Contiki - a lightweight and flexible operating system for tiny networked sensors," 29th Annual IEEE International Conference on Local Computer Networks, 2004, pp. 455-462, doi: 10.1109/LCN.2004.38.
[38] http://savannah.nongnu.org/projects/lwip/
[39] M. R. Palattella, N. Accettura, X. Vilajosana, T. Watteyne, L. A. Grieco, G. Boggia, and M. Dohler, “Standardized protocol stack for the internet of (important) things,” Communications Surveys & Tutorials, IEEE, vol. 15, no. 3, pp. 1389–1406, 2013.
[40] P. Rosenkranz, M. Wahlisch, E. Baccelli, and L. Ortmann, “A Distributed Test System Architecture for Open-source IoT Software,” in ACM MobiSys Workshop on IoT Challenges in Mobile and Industrial Systems (IoT-Sys), May 2015.
[41] The DES Testbed virtualization framework. [Online]. Available: https://github.com/des-testbed/desvirt
[42] N. Brouwers, K. Langendoen, and P. Corke, “Darjeeling, a featurerich vm for the resource poor,” in Proceedings of the International Conference on Embedded Networked Sensor Systems (ACM SenSys), Berkeley, CA, USA, 2009.
[43] M. Doddavenkatappa, M. C. Chan, and A. Ananda, “Indriya: A Low-Cost, 3D Wireless Sensor Network Testbed,” in Proceedings of the Conference on Testbeds and Research Infrastructures for the Development of Networks & Communities (TridentCom), 2011.
[44] https://travis-ci.org
[45] C. Pronk, “Verifying FreeRTOS; a feasibility study,” Delft University of Technology, Software Engineering Research Group, Tech. Rep., 2010.
[46] https://en.wikipedia.org/wiki/FreeRTOS
[47] https://en.wikipedia.org/wiki/Mbed
[48] http://www.tinyos.net/
[49] https://en.wikipedia.org/wiki/TinyOS
[50] https://en.wikipedia.org/wiki/Nano-RK
[51] https://ecos.sourceware.org/
[52] https://en.wikipedia.org/wiki/ECos