# Managing Limited Resources in an Internet of Things (IoT) Operating System

A. Davis
*Dept. of Computer Science*
*Syracuse University*
adavis@syr.edu

***Abstract*—The Internet of Things(IoT) is a rapidly expanding method of connecting many of life's everyday objects in the physical world to the digital world by linking them via the internet. These connections have the power to improve and change daily life both at home and in the industrial setting. As these networks and connections grow in number and complexity, so does the demand for appropriate methods for managing these devices' resources and communications: thus, there is a need for IoT operating systems (OSs). Unfortunately, this demand cannot be met with traditional OS solutions of the past due to the restrictive nature of IoT devices. Objects in the IoT may need to be small, lightweight, wireless, and portable—which leads to tight resource budgets for computations, memory, and power. Thus, new OS solutions have, and are currently, being developed and improved for the purpose of optimally managing the limited resources available in an IoT network. This paper explores the specific challenges/needs of an IoT system. It addresses the solutions currently available for managing these constrained resources with regards to process, memory, energy, communications, and file management. Finally, following a survey of the current solutions, the recommendation is made to continue improving and supporting the integration of heterogeneous OS devices with improved simulators.**

## I. Introduction

The Internet of Things (IoT) defines the creation of networks of connections from everyday objects in the physical world to the cyber world. These connections are made useful by incorporating energy efficient micro-controllers, low-power radio transceivers, sensors, and actuators into every day objects: thus, making them *smart objects* [1].

An IoT network is made up of devices/equipment with the ability to sense, actuate, store, or process information. It has the power to allow smart communication between objects and devices—free of user facilitation—for the everyday user [2]. It is often expected for IoT smart objects to be small in size, high in quantity, and light in memory. It is also imperative that these objects use little energy as they are often wireless and deployed remotely, operating on just low-power batteries [3].

IoT is composed of three main subsets: the smart objects themselves, the networks that connect these objects, and systems to process data collected by the smart objects [4]. As these networks grow in scope and popularity, so does the complexity of their communications and computations. Thus, the need for management and synchronization of these devices, their resources, and the communications between them is a prevalent and growing concern. Consequently, these devices and networks require an appropriate operating system to manage them [5].

An operating system is software used to manage hardware and software of the device on which it is present while hiding the minor details of the device. Unfortunately, due to the scarce computational, memory, and communication resources in IoT systems, traditional OS solutions cannot be used to meet this demand. Although the essential tasks and components of all operating systems are usually very similar, there is variety amongst different OSs in terms of how resources are allocated and how its functionalities are implemented. OSs may also have application-specific or additional functionalities that others do not [6].

The basic duties that must be performed by any IoT OS include process management, memory management, energy conservation, communication management, and file management: all while minimizing usage of resources. For certain applications there may also be a need for real-time capabilities. IoT-specific operating systems have been newly developed in recent years to meet these specific needs including, but not limited to, Contiki, TinyOS, and FreeRTOS [2]. The methods employed by these OSs for meeting the IoT's unique needs are detailed and compared.

The ultimate goal and challenge of IoT network is achieving seamless communication across a variety of low-end devices. Enabling communication across these small, wireless nodes presents specific network challenges. As implied by their name, IoT systems are expected to be able to connect to the internet (i.e. support internet protocol (IP)), however traditional IPv4 and IPv6 may be too bulky for some low-end IoT devices. Thus, a summary of the unique IoT network protocol solutions that have been developed to meet this challenge are also addressed. Some or all of these protocols are then supported by IoT-specific operating systems.

This paper seeks to address the questions of how IoT OSs have been uniquely design to manage the limited resources of IoT devices and networks, which of these existing solutions is best, and how additional improvements could be made going forward.

## II. Significance

It is projected that by 2050 approximately 50 billion IoT systems will be in use worldwide [2]. The evolution of IoT is progressing rapidly towards the seamless integration of this wide array of communication devices into a global internet [1]. As the Internet of Things has grown in popularity and scope over the past decade, several IoT-specific hardware systems have been developed, and gradually, various operating systems began to develop as well. Due the the specific needs of an IoT network, IoT operating systems must be able to manage very limited resources quickly and efficiently. Three of the most popular OSs currently being used for IoT applications, according to Musaddiq et. al, include Contiki, TinyOS, and FreeRTOS. [2].

IoT has already had a large impact on modern industry in areas such intelligent transport, smart electric grids, and safety [1]. These networks of devices may make use of connections including 2G, 3G, 4G, 5G, or even other wireless technologies such as Zigbee, Wi-Fi, 6LowPAN Bluetooth, or WirelessHART [4]. The deployment of IPv6 in 2012 massively increased address space from IPv4 and has only further expanded the possibilities for IoT devices— and by extension, demand for OSs [3].

## III. Background

Research on distributed systems of low-power WSNs was first performed in 1998 at the University of California in Berkeley. This research focused on providing resource-constrained node devices with the protocols and networking algorithms necessary to connect these devices. Around the same time, a variety of OSs and middleware for WSNs were developed and implemented. As early OSs and protocols developed, the hardware technologies were also rapidly advancing. Microcontroller units (MCUs) became more popular and cost effective, digital radio transceivers became easier to program and provided energy-saving features, and a variety of cheap sensors to measure a myriad of physical properties suddenly became commercially available. As WSN technologies expanded, it became evident that the easiest way to ensure interoperability of WSNs from different vendors was to have all of these WSNs support internet protocol (IP), and thus the idea of the IoT was born.

Today, the uses of IoT device networks is vast and growing. These systems are used for applications include (but are not limited to) industrial automation, building automoation/smart homes, smart metering and advanced infrastructure, and even mobile health tracking. IoT has found traction in industry, government, *and* home-life [1].

## IV. Classes of IoT Devices

There are two separate categories of IoT devices: high-end and low-end. Examples of high-end devices include smart phones and Raspberry Pis. On the other end of the spectrum, the smallest of IoT devices simply do not have the resources to handle a traditional OS such as Linux. It is said that Moore's Law is not applicable to IoT devices in terms of processing power because devices will only strive to become smaller, cheaper, and more energy efficient; increased memory and processing power are not IoT priorities [1].

IoT devices can be categorized in a variety of ways. They can be generally categorized by their communication capabilities: devices are either able to communicate to central servers or not. Devices that communicate to the server often require more power (not battery operated) and more powerful processors. Gateway devices refer to devices with powerful processors, extendable memories, and ample power supplied to them. These devices can often even support Linux operating systems with application containers. On the other end of the spectrum, constrained devices are often connected to the gateway-like devices. Constrained devices are only designed to handle a specific purpose. These are devices that are infamous for being more 'lossy' and communicate using low power wireless protocols such as RPL, 6LoWPAN, Zigbee, Thread, WirelessHART, etc [7].

IoT smart devices are formally broken down into three classes: class 0, class 1, and class 2.

**Class 0** devices have $<1$ KB of RAM, $<100$ KB of flash, and are connected to proxies, gateways, or servers for internet connection.

**Class 1** devices make use of approximately 10 KB of RAM, approximately 100 KB of flash, and use protocol stack to interact with other devices.

**Class 2** devices utilize approximately 50 KB of RAM, 250 KB of flash, and may support regular IPV4 and IPv6 protocols allowing them to function akin to ordinary network devices.

## V. IoT-Specific OS Design

With the growing momentum of IoT networks there has also been an expansion of micro-electro-mecanical systems (MEMS) which use embedded OSs in their various applications and devices. Embedded OSs tend to be more customizable, application-specific, and better suited for applications with real-time needs. Yet another type of specific OS is a network OS. Although both computer and embedded OSs have support for communication protocol stack and networking, the network OS has enhanced support for networking functions and is intended to not encompass a single device, but rather coordinate the operations and resources of several devices in a network [6].

Although there are proponents for network operating systems such as the Open Network Operating System (ONOS) for the IoT, many of the currently competing OS solutions for low-end devices in the IoT are embedded OSs and will remain the focus of this report [8].

## VI. IoT Network Solutions

Unfortunately, the wireless connections employed by the IoT are slow and frequently hindered by high packet loss. Athough WSN research and technologies led the charge into the IoT phenomenon, a WSN does not necessarily have to support IP protocol whereas and IoT network *does*. A main reason for this push for adherence to IP is a desire for universal interoperability. As the potential for the IoT became clear and use cases expanded, so did the desire for devices from different vendors to be able to communicate and cooperate seamlessly. Thus, having all of these emerging WSNs support internet protocol guarantees a platform for heterogeneous communication.

Various organizations have set out to create standards for communications among these devices. An IoT OS must be versatile enough to support the various protocols without overwhelming tiny devices with limited resources [1]. Some of the specific protocols and solutions supported by various IoT OSs for handling low power lossy networks (LLNs) include RPL, 6LoWPAN, and CoAP. The one thing that is agreed upon in the IoT OS research community is that there is *no one-fits-all* networking solution for these resource constrained domains. [8].

The nodes of an IoT network can be what is known as *constrained nodes*. These are nodes where some features that are usually characteristic of Internet nodes are lacking due to cost and/or physical constraints on power, memory, and processing capabilities. Furthermore, a *constrained network* is a network that lacks the traditional level of throughput (including limits on duty cycle), experiences high packet loss for larger packets due to link-layer fragmentation, and has limitations on when the device is reachable due to sleep-mode cycles. Low-power and lossy networks (LLNs) are typically composed of various constrained nodes—operating with limited processing, energy, and memory—and interconnected by a network adhering to the IEEE 802.15.4 standard. IEEE 802.15.4 is a standard released in early 2003 for the purpose of meeting the needs of low-cost, low-power, and low-speed wireless personal area networks (WPANs). The standard defines the physical and data-link layers of the SI model in such networks [9]. LLNs cannot implement the traditional, more resource-intensive protocols (open shortest path first (OSPF), optimized link state routing protocol (OLSR), etc). Thus, new protocols have emerged to meet these needs including routing protocols for LLNs (RPLs) [6]. Some of the common network protocols supported by most of the current leading IoT OSs include RPL, 6LoWPAN, and CoAP [10].

## *A. RPL - Routing Protocol for LLNs*

RPL is a routing protocol specialized for low power and lossy networks that was designed specifically for the purposes of connecting billions of IoT devices in the future [5]. It creates a network similar to a tree. This tree-like network is known as a Directed Acrylic Graph (DAG) or Destination-Oriented Directed Acryclic Graph (DODAG) in which all leaf nodes have one route to the root node. All nodes in the model are given a *rank*, which increases with greater distance from the *root node*. These ranks are used in calculations that determine routing decisions. Thus, routing decisions are ultimately made based on distance vectors (DV) between nodes.

RPL is specialized to run in power-constrained nodes because it is *reactive* to current traffic instead of being proactive, although proactive implementations have been explored [10]. Routes are determined as needed (ad hoc) instead of being maintained in a routing table over time. In this way, RPL minimizes the overhead and energy costs by focusing on links actively being used. RPL is highly scalable and runs on top of link layers such as IEEE 802.15.4.

RPL usually makes use of what is known as the objective function (OF). The OF sets the rules for how routes between nodes are selected and optimized. Further optimizations of the OF and reducing the latency it incurs is still an unsaturated research topic [6], [10].

Another nuance of RPL is what is known as the *Trickle timer algorithm*. This is used in the construction and maintaining of the DODAG. The Trickle algorithm specifies the amount of time a node spends listening for new information, and how often it sends new information to its neighbors [6].

## *B. 6LoWPAN - IPv6 Over Low-Power WPAN*

6LoWPAN protocol is an IP-Based mesh protocol, allowing WSNs to route packets over IPv6 in accordance with IEEE 802.15.4, where each node connects directly to the Internet. This lightweight protocol is also considered *ad hoc*, as it uses multi-hop routing between devices to establish and maintain on-demand routes [6]. 6LoWPAN is special in that is allows the datagram to be restructured to combat the challenge of the limited payload size in an IoT network. This restructuring process involves the fragmentation and reassembly of IPv6 packets and the compression of IPv6 and UDP/ICMP headers.

6LoWPAN protocol is required to perform this refragmentation because the minimum size of IPv6 is 1280 bytes, yet nodes in low class smart objects may not have the capability to pass this many bytes. Thus, IEEE 802.15.4 instead mandates a frame size of 127 bytes. This extra step in the network protocol also opens the network up to new security concerns. Mitigation methods for these security concerns are open research topics [11].

## *C. CoAP - Constrained Application Protocol*

CoAP is a constrained machine-to-machine web protocol. It is good for LLNs due to its low header overhead and limited parsing complexity. CoAP can provide reliable UDP messages but is slower than 6LoWPAN and RPL [9].

## *D. Opportunistic Routing (OR)*

Another method of specializing operating systems to minimize resources is *opportunistic routing (OR)*. This involves using opportunistic path planning algorithms. This method increases efficiency throughput, and reliability of the sensor network and ultimately leads to longer WSN lifetimes. A key feature for the routing is route selection: what is the best route between two nodes. Traditional routing uses a multihop method for forwarding data where the sender proactively chooses the node to route to. Conversely, opportunistic routing is different in that is capitalizes on the broadcasting nature of WSNs for packet forwarding. Instead of forwarding to a proactively selected node as is done in traditional networks, OR allows the transmission of one node to be overheard by several nodes. The decision of what node is selected is made dynamically. This increases efficiency and reliability of the network [6].

## VII. IoT Resource Management Considerations

An IoT system is often expected to transfer data collected by a sensor or network of sensors back to a base station for further processing. This process may involve communications design, signal processing, data reception, data transmission, sleeping/waking of radio: all of which need to be handled efficiently with regards to energy. The fundamental functionality of an IoT OS is resource management which can be broken down into five main subcategories: process management, memory management, energy management, communication management, and file management [2]. Additional consideration is also given to support for heterogeneous hardware, real-time capabilities, and security [6].

### A. Process Management

An IoT operating system takes on the responsibilities involving the architecture, programming model, and scheduling model used by the network. The kernel manages processes and threads. Sharing resources in a fair manner amongst simultaneously occurring processes is the responsibility of an IoT OS [2].

The scheduling model may vary widely in different IoT OS designs depending on whether or not it supports real-time scheduling. Real-time scheduling demands a preemptive capability to be able to perform tasks within a given timeframe. Conversely, non-real-time scheduling cannot control the exact time frame of task execution but is often simpler and leads to less energy consumption [3], [5].

### B. Memory Management

The amount of memory available in a smart device is dependent on its class as is discussed in Section IV. These limits on available memory are constricting, especially for Class 0 and Class 1 devices. Thus, fundamental goals for an IoT OS is to reduce code size, minimize memory usage—and when memory usage is absolutely necessary—ensure it is allocated as efficiently as possible [2].

Memory management involves providing a means for allocating and deallocating memory of the various processes and threads. Memory allocation can be broken into two main categories: static or dynamic. Static allocation provisions memory at compile-time, which cannot change at run-time. Static allocation includes the program code itself. On the other hand, dynamic memory allocation *is* performed at run-time which therefore makes it flexible, however it opens up the vulnerability of causing a memory leak if allocating memory is never deallocated. It also is difficult to use with a real-time operating system because dynamic allocation typically has deterministic run-times, needs extract code space, and leads to increased memory fragmentation. Dynamic memory contains run-time variables, the buffer, and the stack [2].

### C. Energy Management

Some of the primary energy-consuming processes for IoT devices include sensing, data processing, and data transfer. Increasing longevity of smart devices is one of the primary challenges for an IoT OS. Smart devices may often operate solely on battery power in remote environments, independent of human interactions.

Both hardware and software techniques can be employed to tackle the task of energy conservation, however hardware techniques often incur additional costs for the creation of the device. Software techniques are often more practical, but introduce additional overhead. Energy management methods include network protocol designs and OS scheduling methods such as implementing sleep/wake duty-cycle settings [2], [6].

### D. Communication Management

The overarching goal of any IoT network is ultimately increased connectivity which expects seamless communication. The communication demands for an IoT system include the need to communicate over the Medium Access Control (MAC) layer, the transport layer, and the network layer. As a rule of thumb, IoT communications protocols should value energy efficiency over higher throughput [2]. As these systems grow in size and scope, it becomes increasingly important to provide efficient network connectivity and appropriate routing protocols that can handle the diversity of these heterogeneous, and often mobile, networks [6]. Unlike routing protocols for traditional networks, IoT routing protocols have been designed specifically for devices with limited power and constrained CPUs as was discussed in detail in Section VI. Not all IoT OSs support every network protocol, thus what protocols an OS supports can be a driving decision behind choice of OS for a specific application.

### E. File Management

There are instances in which data being processed by an IoT network must be stored. The previous method of handling this situation would be for all low-end devices involved in the network to transfer their data to a centralized base station for storage. However, more recently onboard flash storage has made it possible for the sensor network itself to store data. A sensor node's memory is still extremely constrained, and a larger, more efficient file system is still required for many scenarios. The file system is tasked with storing and retrieving data efficiently. Accomplishing sufficient storage abstraction is also even more challenging in a resource-constrained device because of the need to keep code small and RAM usage to a minimum [2].

## VIII. Current OS Solutions

While there is still lots of room for research and improvement, several OSs have pulled ahead in the race to meet the resource management needs of IoT networks. Three of the most popular according to Mussaddiq et. al. are Contiki, TinyOS, and FreeRTOS. The methods used by each of these popular operating systems to meet the unique demands of an IoT OS are outlined as follows:

### *A. Contiki*

Contiki is an operating system designed specifically for the internet of things by Adam Dunkels et al. at the Swedish Institute of Computer Sciences. It is written in C language, as are all programs for it and is highly portable. Contiki only requires 10 kB of RAM and 30 kB of ROM. It supports microcontroller devices and is open source under BSD license [11]. According to Javed et al., Contiki is the most frequently used OS for IoT applications [5].

*1) Process Management:* Contiki is event-driven, meaning event handlers continuously wait for internal or external events. When an event (such as an interrupt) occurs, a memory stack is allocated by the kernel to a new event handler process which runs to completion. This allows the use of a shared stack for all processes which helps save memory. On the other hand, it has the downside of offering very poor real-time performance. The task running in an event-driven scheme cannot be blocked and must wait for completion. Thus, multiple event handlers are necessary [2]. Contiki supports preemptive multithreading that can be applied to individual processes but is primarily an event driven OS [2], [4], [5].

To minimize stress on the mere 1-10 KB of RAM for class 0 and class 1 IoT devices, it is not desirable to have a stack of memory for every thread. As a compromise, Contiki uses protothreads: as lightweight and stackless alternative to traditional multi-threading. This model has a mechanism to perform what is known as a "blocking wait." Each protothread encompasses a conditional wait statement that has the ability to block the program in the event of an interruption by an explicit blocking wait statement. This method is motivated by a desire to allow multi-threading while forgoing the large memory footprint of creating a stack for every process. As a result, each protothread incurs a mere *two bytes* of memory overhead. However, the downside of this approach is that there is no means for synchronization amongst protothreads [2].

*2) Memory Management:* As previously mentioned, the stackless protothreads help save memory resources. A key feature of Contiki is the fact that all of the processes ultimately share the same stack. Additional memory management methods in Contiki include functions such as the managed memory allocator function, *mmem()*. This method increases the usability of fragmented memory by compacting it when blocks are unused. It also has methods for allocation and deallocation of the heap at runtime which offer the benefits of dynamic memory allocation. However, one of the negatives of Contiki is that it does not have a memory protection unit (MPU) [2].

*3) Energy Conservation:* The Contiki OS kernel does not have a specialized power-saving procedure, however it does offer a sort of power-saving mode by monitoring its event queue. In order to save power the OS application has the capability to put the CPU to sleep when the queue is empty [2].

*4) Communication Management:* Contiki has four fixed network layers: radio, radio duty cycle (RDC), medium access control (MAC), and network layers [6]. Contiki supports CoAP, 6LoWPAN, and RPL networking protocols [11].

The network layer of Contiki is split into an upper IPv6 layer and a lower adaption layer. The adaption layer provides IPv6 and UDP header compression and fragmentation using 6LoWPAN as discussed in Section VI. The network routing layer is also where ContikiRPL is implemented.

RDC is essential to achieve energy efficiency while maintaining network communication. It saves energy by allowing a node to keep the radio transceiver off most of the time. The radio layer receives bytes or full packets via interrupt handlers. The incoming data is buffered and the process is polled. This polling mechanism causes the process to pass on to the special event and then finally it is passed to the upper layers.

For the MAC Layer, contiki supports ContikiMAC, X-MAC, and CSMA-MAC. This layer takes on the responsibility of coordinating when each node transmits/receives packets. ContikiMAC provides a sleep/wake mechanism. The Contiki implementation of the MAC layer strives to minimize the time any given node spends listening when no packet is present, also known as idle listening. X-MAC in Contiki has nodes send a preamble to their data. Then the receiver does not have be listening all of the time, but rather may go back to sleep if it does not detect the preamble. In the event that upon wake-up, the receiver *does* detect the preamble, the node will remain in the idle state waiting to receive data [2].

The RDC method allows nodes to keep the radio tranceiver off for the majority of the time; thus, using minimal energy while maintaining network communication. A node's radio layer must then receive packets via interrupt handlers. It has been seen that asynchronous duty cycling led to decreased packet delivery ratio (PDR) due to increased packet collisions and network contention. This loss in reliability is a trade-off accepted by Contiki for the sake of power conservation, however future research could be put into minimizing this performance loss [6].

*5) File System:* Contiki handles data storage via what is known as its Contiki File System (CFS). The CFS has the capability to interface with different file systems: thus, providing a much needed degree of storage abstraction by creating a base from which to support various devices with more limited resources. CFS has the capabilities to fully function in cooperation with CFS-POSIX and Coffee [2].

### *B. TinyOS*

TinyOS is written in NesC and is known for having good support for a wide variety of devices. It has a monolithic architecture and is event-driven. Potential uses include smart buildings and smart meters [5].

*1) Process Management:* TinyOS implements a very similar multi-threading approach to that used in Contiki, except TinyOS Thread (TOSThread) offers a fully-implemented preemptive application-level thread library [2]. TinyOS has a two-level scheduler which is split into events and tasks. The task queue is managed as FIFO, meanwhile an event handler may perform preemption for a kernel-level task (an event). Tasks cannot preempt, but events can. Later versions of TinyOS began to incorporate priority scheduling routines instead of just FIFO for the task queue. Although TinyOS utilizes a real time scheduling algorithms, it is not actually a RTOS [5].

*2) Memory Management:* TinyOS achieves some of it's memory conservation by not supporting dynamic memory allocation: strictly static. This not only reduces memory fragmentation but also eliminates the danger of stack overflow and the need for an additional stack to manage the dynamic heap. Memory efficiency is also increased by using source-to-source transformation. This allows TinyOS multithreads to be transformed into stackless threads. Condensing all memory down to a single stack greatly reduces the burden on memory [2], [5]. Finally, additional memory is made available through TinyPaging—a paging mechanism that can perform conversions between virtual and physical addresses in memory [2].

*3) Energy Conservation:* All of the tasks of smart devices (sensing, processing, and trasmitting) involve both idle and busy time, however the idle time is not always substantial enough to justify conventional context switching. TinyOS uses this concept to conserve energy by only allowing context switching when it is absolutely necessary to meet deadlines and by not performing the context switch until the latest possible time in an effort to avoid it at all costs [2], [5]. TinyOS also tracks the energy-usage of various components in the system network. Energy tracking allows for optimization of energy distribution within the system. Unfortunately, the tracking method introduces increased memory-needs and thus is not suitable to use with mobile devices [2].

*4) Communication Management:* TinyOS supports Broadcast based routing, Multi-Path Routing, Geographical Routing, Routing Reliability based, TDMA base Routing. Similar to Contiki, TinyOS supports the low power listening approach, X-MAC. This mechanism uses the same radio duty cycling where nodes go to sleep and wake up periodically to see if there is a preamble to be heard. The preamble serves as an interrupt for the node to wake up and continue listening. A short, strobed preamble is an energy saving technique because it allows pauses in which a node can interrupt the preamble as soon as it wakes up so it can stop hearing the preamble and move on to immediately receiving packets [2].

As discussed in Section VI, TinyRPL forwards packets by constructing a DODAG. TinyOS makes use of Berkeley lw-power IP stack (blip), which allows the use of an IPv6 stack based on 6LoWPAN specifications. blip makes use of 6LoWPAN's header compression and neighbor discovery to support IPv6 on limited resources. Also like ContikiRPL, TinyRPL utilizes OF to carry out route selection [2].

Finally, a unique feature of TinyOS's transport layer imlementation is its use of Sensor Transmission Control Protocol (STCP) which has most of its functionalities implemented at the base station. In this way, STCP conserves a considerable amount of energy in sensor nodes [2].

*5) File System:* TinyOS supports a single-level file system. A node cannot run more than a single file at a time. While this is a restrictive assumption, it reduces overall power consumption of the network. TinyOS implements a microfile system called *Matchbox* which has the capabilities to read, write, and delete files. Matchbox stores files in an unstructured way that can only support sequential reads and append-writes. Other supported thrid-party file systems include FAT which supports SD cards and reduces power consumption in sensor nodes while allowing them to store a large amount of data, and Efficient Log-Structured Flash (ELF) which is a system for microsensors that is energy and memory efficient and supports many common sensor files [2].

### C. FreeRTOS

FreeRTOS is a C-based IoT OS that was developed in 2002 with the goals of being simple, portable, and easy for developers and user. It's primary purpose is to serve as a real-time operating system for constrained resource networks [5].

*1) Process Management:* FreeRTOS uses a microkernel approach to also implement a multithreaded system and includes methods for mutexes and semaphores for process synchronization purposes [5]. FreeRTOS has the option for process interruptions and context switching between threads [2]. In order to support the real-time needs, its scheduler must support preemption [5]. FreeRTOS is an option for real-time multitasking for low-end devices. It has a priority-based scheduler that is round robin in the event of processes of equal priority. FreeRTOS uses just four C files for its scheduling: *task.c, list.c, queue.c, and croutine.c*.

*2) Memory Management:* Memory is allocated dynamically by the kernel for every event. It offers three different heap management schemes that vary depending on application requirements [2].

*3) Energy Conservation:* In FreeRTOS, since tasks are event-driven, some time and energy is wasted waiting for an interrupt, or the end of a time period. When a thread is stuck waiting on this it is referred to as being in a blocked state. In the event that all tasks are blocked, an "idle task" is created and run. Thus, when the processor is idle, it can simply enter power saving mode. The actual implementation of this functionality is accomplished by creating a task with the lowest priority which will only be run when no higher priority task exists and is ready [2].

*4) Communication Management:* As a minimalist OS, FreeRTOS does not have its own native networking techniques and rather has to implement those of third parties [5].

*5) File Management:* FreeRTOS also supports FAT file system like TinyOS but uses super Lean FAT File System. The main objective of this set-up is to minimize both flash and RAM footprint (taking up a mere $<4$ kB flash and $<1$kB RAM) [2].

### D. Additional OS Options

The fundamental features of a few more IoT OSs are outlined. Even still, not nearly all of the currently vailable IoT OSs are detailed in this report. Many of the OSs have different combinations of similar features to those already outlined for Contiki, TinyOS, and FreeRTOS:

*1) RIOT:* RIOT is another IoT OS, released in 2013. RIOT incorporates a multi-threaded model. It also supports 6LoWPAN network protocol and implements real-time scheduling like FreeRTOS. It has a micro-kernel architecture and is developed in C/C++. It takes up a mere 1.5 KB RAM and 5 KB ROM [5].

*2) Nano-RK:* Nano-RK follows an entirely preemptive process management scheme taking up <2KB of RAM and <18KB of ROM. It is unique in its highly efficient energy and memory management. It is similar to TinyOS in its monolithic architecture and event-driven process management, however is also like FreeRTOS in that is can support real-time applications. It is written in C making it developer-friendly [5].

*3) SOS OS:* SOS is a modular, event-based OS written in C. It is different from many of the other popular IoT OSs in its dynamic memory management. While this comes at a cost, SOS would be a viable option if it was necessary to make run-time changes in an application [5].

## IX. Interoperability

As commercial, industrial, and government-related use-cases for IoT systems have expanded, so have efforts for standardization. It is critical to the interconnectivity of the IoT for devices from different vendors to be interoperable, despite their heterogeneity. In an IoT context, interoperability shall be defined as the ability of two systems to be able to communicate and share services with one another [7]. This need was one of the driving factors that led to pushing emerging WSN technologies to be connected to the worldwide network infrastructure back at the dawn of the IoT. Standardizing WSNs to be able to adhere to a standards with the broad reach of the Internet Protocol suite makes them far more versatile, extendable, and interoperable [1]. As the number of IoT platforms continues to grow, more companies are introducing their own IoT infrastructure, protocols, interfaces, standards, formats, and semantics which still hinders the cooperation of cross-platform devices. If devices from different vendors are not able to communicate, only a fraction of the potential benefits that the IoT is projected to provide will ever be realized.

Interoperability of IoT devices can be extended to the sub-categories of devices, networks (like those discussed in Section VI), syntactics, semantics, and platforms. Platform interoperability includes different OSs, programming languages, data structures, architectures, and mechanisms for accessing data. While industry has tried to enforce standards to circumvent this problem, it is believed that any universally agreed upon standards are still a ways off (if they ever even come to fruition). Thus, it is critical that researchers tackle the issue of increasing interoperability across IoT systems [7].

Unfortunately, a lot of IoT OS research focuses on “homogeneous nodes” which operate all on the same OS. In reality, heterogeneous operation is common and it is desirable that IoT nodes with different OSs be able to cooperate cleanly. Research has been done on the interoperability of Contiki OS and TinyOS RPL implementations. Subtle differences in the underlying layers of the two OSs led to diminished performance. There are still many unexplored topics in this area of IoT OS research [6].

## X. Simulation Support

An important and valued feature for an IoT OS is a simulator. Simulators can save users a great deal of time and money by allowing the testing of various factors prior to real-world deployment of an IoT network. Simulators may model power consumption, coverage area, and layout of nodes. These simulators can model the various network protocols being supported by the IoT OS and aid in testing and debugging [3].

Some simulators supported by the OSs discussed in Section VIII include Cooja for Contiki and TOSSIM for TinyOS [5]. Cooja is a Java-based simulator that can simulate networks of Contiki motes. It also allows real hardware devices to be simulated prior to implementation [4]. TOSSIM is the built-in simulator for TinyOS. It is a discrete event simulator and can simulate large networks of motes. FreeRTOS, RIOT, Nano-RK, and SOS OS do not have built-in simulators like Contiki and TinyOS [5].

## XI. Opinion and Analysis

Following an overview of IoT netwoking methods and some existing OS solutions, it is clear that there is no singular solution to the question: *what is the best method for managing the limited resources of IoT devices and networks?* Thus, I am of the opinion that the wide variety of IoT OSs being developed is beneficial to the future of the IoT. Different use-cases will inevitably have different OSs that are best for them and having a variety to choose from expands the overall possibilities of the IoT. Furthermore, one singular IoT OS would never be practical because of the goal of minimization. If an IoT OS is trying to minimize code size and the resources it consumes, it would not make sense for it to have more functionalities than are necessary— for example, having to have code and capabilities for an RTOS when that is not needed. However, in the event of a real-time application, this feature cannot be ommitted: thus, there should never be a single IoT OS.

Looking at the amount of research and applications for each of the different IoT OSs, it is my opinion that no singular OS will ever monopolize the market. With that said, it becomes all the more important to focus research efforts on how best to maximize the interoperability of these different OSs. Moving forward I would encourage more research into the cross-over of OS systems. In overviews of Contiki, one feature that is always listed in its favor is its thorough and helpful simulator tools (i.e. Cooja). I would promote the creation of a similar software for some of the other IoT OSs that do not have their own simulators such as Nano-RK or RIOT. Beyond even than that, I would advocate for research and work into the design of a simulator to demo heterogeneous IoT networks with different OS platforms. This paper’s companion demo will show capabilities of Contiki’s Cooja simulator and discuss the predicted value of creating a similar cross-over tool for modeling devices of different OSs.

## XII. Conclusion

IoT devices and networks are designed to create connectivity between many of life's everyday objects the world wide web. However, connecting smaller, simpler devices means finding ways to manage them despite their limited processing, memory, and power capabilities. This is done through the development of IoT-specific OSs such as (but not limited to) Contiki, TinyOS, and FreeRTOS. These OSs each tackle the basic challenges of process, memory, energy, communication, and file management in ways that are uniquely designed to minimize resource consumption. Some methods employed to do this are by using lightweight, stackless threading techniques, making the most of RDC methods and protocols for LLNs, minimizing context switching, and supporting simple file systems. Since the ultimate goal of IoT devices is increasing connectivity, one of the most important features of an IoT OS is the network protocols it can support. OS support for standard network protocols for LLNs also helps support the goal of interoperability across heterogeneous platforms of IoT devices which is a growing concern as the variety of IoT platforms continues to expand. In conclusion, I support the diversity of IoT OSs developed and acknowledge that there are use-cases for a wide variety of IoT OSs. Future work should look to support and the interoperability this broad spectrum of IoT OSs and their heterogeneous simulation.

## References

[1] 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 Gener. Comput. Syst.* 2018, no. 88, pp. 699–706.

[2] A. Musaddiq, Y. B. Zikria, O. Hahm, H. Yu, A. K. Bashir and S. W. Kim, "A Survey on Resource Management in IoT Operating Systems," in *IEEE Access*, vol. 6, pp. 8459-8482, 2018.

[3] Zikria, Y.B.; Kim, S.W.; Hahm, O.; Afzal, M.K.; Aalsalem, M.Y. "Internet of Things (IoT) Operating Systems Management: Opportunities, Challenges, and Solution." Sensors 2019, 19, 1793.

[4] Velinov, A.; Mileva, A. "Running and Testing Applications for Contiki OS Using Cooja Simulator." In *Proceedings of the 7th International Conference on Information Technologies and Education Development–ITRO 2016*, 2016 pp. 279–285.

[5] F. Javed, M. K. Afzal, M. Sharif and B. Kim, "Internet of Things (IoT) Operating Systems Support, Networking Technologies, Applications, and Challenges: A Comparative Review," in IEEE Communications Surveys & Tutorials, vol. 20, no. 3, pp. 2062-2100.

[6] Zikria YB, Afzal MK, Ishmanov F, Kim SW, Yu H. A survey on routing protocols supported by the Contiki internet of things operating system. Future Gener Comput Syst. 2018; 82: 200- 219.

[7] Noura, M., Atiquzzaman, M. & Gaedke, M. Interoperability in Internet of Things: Taxonomies and Open Challenges. textitMobile Netw Appl, 2019, 24, pp. 796–809.

[8] A. G. Anadiotis, L. Galluccio, S. Milardo, G. Morabito and S. Palazzo, "Towards a software-defined Network Operating System for the IoT," *2015 IEEE 2nd World Forum on Internet of Things (WF-IoT)*, Milan, 2015, pp. 579-584.8.

[9] J. Schänwälder, "Internet of Things: 802.15.4, 6LoWPAN, RPL, COAP," Presented to Jacobs Unviersity, Bremen, Germany, October 14, 2010. [PowerPointslides]. Available: https://dig.watch/sites/default/files/2010-utwente-6lowpan-rpl-coap.pdf.

[10] N. Khelifi, S. Oteafy, H. Hassanein and H. Youssef, "Proactive maintenance in RPL for 6LowPAN," *2015 International Wireless Communications and Mobile Computing Conference (IWCMC)*, Dubrovnik, 2015, pp. 993-999.

[11] M. Asim and W. Iqbal, "IoT operating systems and security challenges," Int. J. Comput. Sci. Inf. Security, vol. 14, no. 7, pp. 314–318, 2016.