Ubiquitous Sensing: Towards the future of IoT operating systems

Aniket Shenoy
Indiana University
Bloomington, Indiana
ashenoy@iu.edu

Abhinash Tummala
Indiana University
Bloomington, Indiana
abhitumm@iu.edu

ABSTRACT

Internet of Things (IoT) is projected to connect billions of devices as they get smaller and cheaper. IoT devices range from low-end units to devices having powerful processors. Traditional operating systems are not able to fulfil the diverse requirements of heterogeneous IoT devices. The requirements of an OS for IoT devices need to be revisited to operate with minimal resources and at the same time cover a wide range of devices, thereby avoiding redundant development, and maintenance of devices. This paper presents an analysis on such requirements and a comparative study of some current IoT operating systems.

KEYWORDS

Operating System (OS); Internet of Things (IoT); MANTIS OS (MOS); Contiki; RIOT

1 INTRODUCTION

IoT is a network of physical devices connected to each other enabling them to transfer data and communicate. Deployment of IoT devices provides tremendous opportunities and applications in various industries from consumer applications to large scale enterprises. There has been an exponential increase in the number of IoT devices. IoT is being scaled from smart houses to smart cities. Its applications include smart homes, connected vehicles, traffic monitoring, surveillance, garbage management, water supply, supply chain, to name few. It is reshaping entire industries and also being used for cost effective solutions. It is estimated that 30 billion devices would be part of the IoT ecosystem, taking its market value to 7 trillion dollars by 2020 [6]. IoT devices have evolved from tiny sensors and now range all the way from 8-bit microcontrollers to high-end devices with powerful processors. Advancements in technology have led to devices becoming smaller and cheaper.

Over the years, many networking protocols have been developed so that heterogeneous devices can communicate with each other. All these heterogeneous devices have diverse requirements when it comes to computing power, memory, energy consumption, etc. As compared to high-end devices, low-end devices have stringent constraints when it comes to energy consumption, processing power, and memory. At the same time, they are also supposed to fulfill reliability, efficient communication and real-time processing for their respective tasks.

In a utopian situation, the capabilities of a traditional full-fledged OS should be available for all types of IoT devices. The problem at hand is that current full-fledged operating systems have high minimum requirements, deeming them unfit for low-end devices. Similarly, lightweight operating systems cannot harness the complete potential of high-end devices. Therefore, it is required to rethink the design of operating systems so as to provide features such as traditional multi-threading, hardware abstraction, dynamic memory management, real-time support, etc. for both high-end as well as low-end devices, without compromising complexity. Also, there is a need for Application Programming Interfaces (APIs) apart from bare metal programming to cater to the development for diverse IoT use cases.

In this paper, we try to identify the key requirements and challenges for IoT operating systems and study how few of the current operating systems handle these challenges. IoT devices are application specific and are very dynamic in nature. The build equipment is very heterogeneous and the operating system best suited for these IoT devices must overcome these challenges and satisfy such diverse requirements. This paper studies three open source operating systems: Contiki, a dominant OS for low-power IoT devices; MANTIS OS (MOS), a multithreaded OS for Wireless Sensor Networks (WSNs); and RIOT, an OS which tries to match various software requirements of heterogeneous IoT devices. Each of these operating systems have their unique architecture as they were designed to cater to different device requirements. We look at the strengths and contributions of each of these operating systems and try and see whether any of them could serve as a perfect fit IoT OS. Although there are several closed source operating systems, this paper only considers open source solutions for IoT operating systems where standard APIs can be used and the OS can be ported to any device independent of hardware.

The rest of the paper is structured as follows. Section 2 gives a background on IoT. Section 2.1 talks about some of the challenges that IoT operating systems need to overcome and Section 2.2 presents some design characteristics that need to be considered for an IoT OS. Section 3 focuses on solutions provided by Contiki, MOS and RIOT and their features. Finally, the paper closes with conclusions in Section 4.

2 BACKGROUND

Among the many factors driving the growth of IoT devices, one of the primary factors is the cost. The device hardware price has come down and has made them more accessible. A lot of devices enable real time collection and analysis of data which can be used for decision making. Large data sets of such information is being used to improve upon the efficiency and predictions for organizations using machine learning. IoT have been categorized into 3 different categories based on how these devices are used, who uses them and their functionality.

- Consumer IoT: These constitute the mainstream IoT devices such as wearables gadgets, smart cars, appliances, security systems.
- Commercial IoT: The medical devices, tracking systems, logistics control all such fall under commercial IoT devices.
- Industrial IoT: These are used at scale as part of smart city ecosystem or large scale industrial devices. Water systems, Electric grids, wind turbines, manufacturing robots are could be considered as Industrial IoT.

All these added utilities come with their own set of challenges and requirements. To deploy and get them to obtain the desired results, the challenges have to overcome.

2.1 Challenges

2.1.1 Resource Constraints. IoT devices have limited computing and memory capabilities. The amount of computing that can be done IoT devices is constrained because of the limited configuration of these devices. They can't store or compute large information, and this could be a challenge in certain situations where computing and analysis need to be done in real time to make decisions. Most of the typical low-end IoT devices have very limited memory ranging from kilobytes to a few hundred megabytes. Likewise, the processors in these devices are not very powerful and often have low clock cycles.

2.1.2 Connectivity. The very name of IoT suggests that IoT devices need to communicate with each other over the internet. Current models could be used for thousands of nodes or devices but when billions of devices are being added to the IoT ecosystem, no existing protocols could be extended to match the scale of IoT. Also, very often IoT devices are deployed at remote locations and traditional communication protocols and mechanisms may not work under such circumstances [7]. Devices connected using heterogeneous network technologies also need to be able to communicate with each other.

2.1.3. Security and Privacy. IoT devices raise serious security concerns. Very often these devices have limited to no security mechanisms in place which leaves them vulnerable to malicious attacks. Because of the wide range of applications of IoT devices and their deployment everywhere, IoT hacks would be a threat to our privacy and wellbeing. Some IoT devices collect sensitive data and could fall into wrong hands. The reason for this is because these devices often lack the computational, storage capabilities and a powerful operating system to deploy any security solutions [8].

2.1.4. Energy constraints. Most IoT devices are essentially sensors which are low power devices and are energy efficient. Most IoT devices run on batteries and many of these might have to run on a single battery source for long periods of time [1]. Also, given the fact that IoT devices are going to be almost everywhere in the near future, global power efficiency is essential. Thus, operating systems for IoT must be able to adapt to the power saving features of the devices and provide features such as power saving mode. IoT operating systems must provide energy saving functions at the application level.

2.1.5. Programmability. Many of the current IoT operating systems are not developer friendly and lack in providing a standard API (e.g. Portable Operating System Interface (POSIX), Standard Template Library (STL)). Although IoT hardware is heterogeneous, operating systems must bridge the gap between the heterogeneity and the programmability for developers. Developers must have an interface to harness the capabilities that each device has to offer to solve various complex problems. This would eliminate development redundancy and reduce maintenance costs.

Despite these challenges, IOT devices are expected to satisfy certain fundamental requirements such as:

- Reliability
- Real-time behavior
- Adaptive communication stack

Probably the biggest challenge is to obtain a trade-off between performance, an API and memory constraints.

2.2 Characteristics

This subsection talks about some of the key OS design characteristics that have to be considered as they impact the potential of the OS. Fig. 1 below shows the typical architecture of an IoT OS. It consists of a hardware abstraction layer above the hardware which allows communication between the operating system software and the hardware, the device drivers layer, libraries, the communication stack, the kernel itself and finally the application layer that sits on top of these layers.

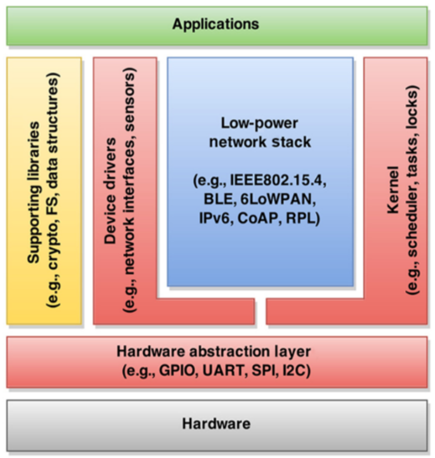

Applications
Supporting libraries (e.g., crypto, FS, data structures)
Device drivers (e.g., network interfaces, sensors)
Low-power network stack (e.g., IEEE802.15.4, BLE, 6LoWPAN, IPv6, CoAP, RPL)
Kernel (e.g., scheduler, tasks, locks)
Hardware abstraction layer (e.g., GPIO, UART, SPI, I2C)
Hardware

Figure 1: Typical architecture stack of an IoT OS [5]

2.2.1. Kernel. When it comes to OS design, the primary aspect is the kernel architecture. OS kernel architectures can be monolithic, layered or microkernel. Monolithic kernel architecture is the simplest of the lot, however it is a complex structure and lacks modularity. The other two architectures provide modularity. The advantage of the microkernel architecture is that only a minimal number of functions run in the kernel modem, thereby increasing reliability as bugs in other components won't result in system failure. Most of the low-end IoT devices do not have a Memory Management Unit (MMU). This results in stack and buffer overflows. Therefore, while choosing a kernel architecture, one has to identify the tradeoff between a more robust and flexible microkernel or a less complex and efficient monolithic kernel, or go for a hybrid approach altogether.

2.2.2. Scheduler. An integral function of IoT devices is the real-time processing most of these devices are sensors which respond to information from the environment. The real-time functionality and energy efficiency of an OS is heavily dependent on the scheduler design. The scheduler must be able to handle tasks of various priority levels in real-time. Typically, there are two types of scheduler, them being either preemptive or non-preemptive. An optimal scheduler design needs to be selected to satisfy the power saving needs of IoT devices.

2.2.3. Programming Model. The processing power of an OS depends on the programming model. The programming language offered for application development and whether the operating system supports multi-threading are key aspects. Event-driven models seem to be more memory efficient as compared to multithreaded models. When it comes to programming languages, high level programming languages have a lot of established libraries and debugging tools to offer to aid in application development.

2.2.4. Memory. As already mentioned several times above, IoT devices are low memory. Thus, memory management is integral in OS design for IoT devices. Both static allocation and dynamic allocation have their own pros and cons. Static allocation introduces some memory overhead and is less flexible whereas dynamic allocation leads to a more complex system.

2.2.5. Network. As in can be seen in Fig. 1, the network stack is the central component of a typical IoT OS architecture. This is where the information that is communicated between devices is stored in the form of packets is handled. The network stack is responsible for transmitting the packets between the various layers.

Operating systems can be broadly divided into two categories:

- Event-driven: Most of WSN operating systems use the event driven approach. In a nutshell, in the event-driven model, the kernel consists of an infinite loop handling all events in the same context and these events run to completion. This approach is memory efficient and simple to implement, however it imposes a lot of restrictions when it comes to application development.
- Multithreaded: Traditional operating systems like Linux follow the multithreaded approach where each thread runs in its own context and manages its own stack. This model requires a scheduler and a scheduling policy to carry out context switches, thereby contributing to runtime and memory overhead.

3 SOLUTIONS

In this section, we have a look at three open source operating systems for IoT devices and see what each of them bring to the table to tackle the challenges in the IoT ecosystem.

3.1 MANTIS OS

Multimodal system for Networks of In-situ wireless Sensors (MANTIS) is a multithreaded cross-platform embedded OS for wireless sensor networks. The key challenges for designing this multithreaded OS for sensors is similar to what we have earlier discussed, most important of them being severe resource constraints and limited energy lifetime [3].

Mantis OS is based on multithreaded preemption, executed using standard I/O synchronization and a network protocol stack. What stands out in MOS is that It manages to do all this in less than 500 bytes of RAM. It also provides cross platform support for sensors networks and personal computers as well. It provides tools for easily deploying and managing these sensor networks. Mantis OS also enables sensor network application developers to design and test applications before distribution and deployment on a live environment. It supports features such as multimodal prototyping, dynamic reprogramming and remote shells.

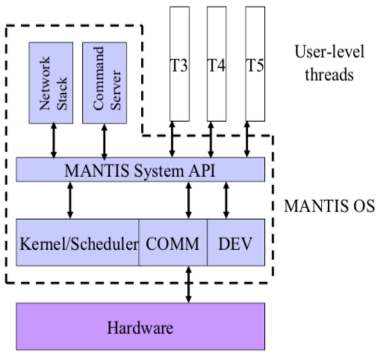


Figure 2: MOS architecture which fits in < 500 bytes of RAM [3]

MOS addresses the unique demands of sensor networks. Following are some key feature of Mantis OS:

3.1.1. Memory efficiency. MOS is remarkably memory efficient. The kernel code size, scheduler and the network stack all together occupies less than 500 bytes of RAM. This permits sufficient space for multiple application threads to execute on severe memory constrained platforms.

3.1.2. Power management. Mantis OS achieves energy efficiency through a sleep function, the semantics of which is identical to the UNIX sleep function. The parameter for this function is the duration for sleep and he thread system is shut down whenever there is no meaningful work to do, thereby saving energy.

3.1.3. Flexible. Different layers can be flexibly implemented in different threads, or all layers in the stack can be implemented in one thread. The performance tradeoff is made in favor for flexibility. Also, the kernel is programmed in C, making it more portable and execute on variety of platforms. Through it achieves code reusability, the entry barriers in terms of programming is low.

3.1.4. Remote management. MOS provides the framework for prototyping sensor network applications and bridging the live deployed sensor networks and the internet. As discussed earlier, since MOS can test applications before deployment, it can save a lot of resources, thereby not resulting in any severe losses. This framework goes beyond just simulation. Its helps in development, application management and visualization of applications. Since sensors could be deployed in remote areas and the network could comprise of many nodes, it would be very convenient to reprogram these sensors or change calibrations remotely.

3.1.5. Dynamic Reprogramming. Sensor networking can be dynamically reprogrammed by Mantis OS. Through this, we would be able to re-flash the entire OS, programming single threads, and change variables within these threads. Also, one can remotely debug running threads. It provides a remote login through which one can login to the network and inspect the memory of sensor nodes. These capabilities are implemented as system call libraries and is built into Mantis OS kernel.

3.2 Contiki OS

Initially designed as on OS for WSNs, Contiki is now also a popular OS for IoT devices. Contiki works well in heavily constrained devices. In general, operating systems require the complete binary image of the system to be present on a device. The binary image consists of the OS, system libraries, and the applications that run on top. One of the main features of Contiki is that of dynamic loading and unloading. Contiki allows the loading and unloading of applications at runtime. The advantage of this is twofold. Applications are much smaller than the entire binary image of the system and also the time taken to transfer only an application image is significantly less than the time taken to load the entire system binary [2].

Being event-driven, Contiki is able to provide concurrency without locks. This is because two event handlers will never run concurrently with respect to each other. Processes run to completion and all processes occupy same stack. Contiki also provides optional preemptive multithreading at application level. In some situations, event-driven models can lead to the CPU to be unresponsive (for example, in cryptographic calculations). To tackle such cases, Contiki provides a hybrid model where preemptive multithreading is provided as an application library which can be used by developers if and when required.

Contiki OS consists of the kernel, libraries, the program loader and processes. All processes communicate through the kernel. There is no hardware abstraction layer. Rather, device drivers and applications communicate directly with hardware. Although Contiki does not contain a native function for power saving, applications can implement such a function when there are no events in the event queue.

3.3 RIOT OS

RIOT tries to bridge the gap between operating systems for WSNs and traditional full-fledged operating systems. It supports IoT devices a wide range of IoT devices ranging from 8-bit microcontrollers to powerful 32-bit processors. It aims to implement all open standards supporting IoT in a connected, secure and effective way [4].

RIOT is based on a microkernel architecture and supports multi-threading. It provides a TCP/IP network stack and adds support for C++ which helps as one can make use of powerful libraries. It uses static memory allocation in the kernel which helps it have a constant runtime. This constant runtime of the scheduler is achieved by using a fixed-sized circular linked list of threads. Fig. 3 gives an overview of the RIOT OS structure.

RIOT solves important challenges for IoT and its main design objectives include things such as energy efficiency and low memory footprint. It supports modularity and API access. It is also independent of the underlying hardware its running on, making it flexible. All these factors along with a developer friendly API, make RIOT a very reliable OS. As it supports modularity, it makes it more robust against certain bugs limiting its effect to one component and not affecting the whole system.

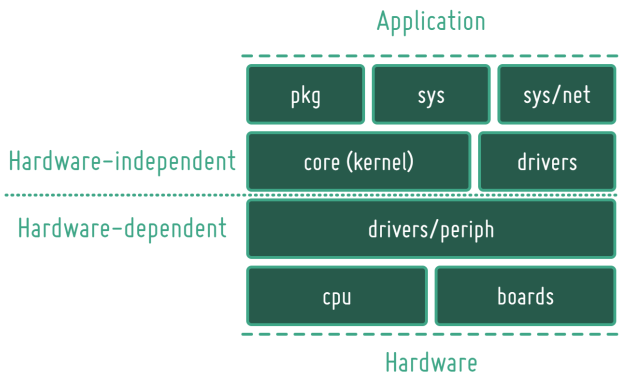
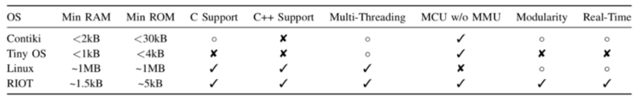


Figure 3: Overview of the RIOT OS structure [8]

RIOT introduces a scheduler that works without any periodic events. When there are no pending or scheduled tasks, RIOT switches to the idle thread. Maximizing this time spent in sleep state improves its energy efficiency. Interrupts wake the system from the idle state. The kernel functions are not complex, thereby minimizing context switching, which also helps in making it more energy efficient. RIOT manages to build this efficient scheduler with a sophisticated architecture with a very low memory footprint requiring less than 5 Kbytes of ROM and less than 1.5 Kbytes of RAM. It uses the multi-threaded programming model with C code and common POSIX like API for all supporting hardware. So, one could use it to build software systems for heterogeneous IoT devices. It has partial POSIX compliance and is working towards full compliance. It is developer and user friendly with standard programming in C or C++.

Fig. 4 below compares the minimum memory requirements, programming language support, multi-threading support, modularity, real-time behavior, etc. RIOT has relatively low minimum requirements but at the same time, it manages to provide features such as C programming support, multi-threading, modularity and real-time support. Contiki being an OS for low-end devices, has lower memory requirements. However, it only provides partial support when it comes to C programming, multi-threading, modularity and real-time capabilities.



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	✓	✓	✓	✓	✓	✓

Figure 4: Comparison of the key characteristics of Contiki, Tiny OS, Linux and RIOT [4]

4 CONCLUSIONS

In summary, we have studied the typical challenges that need to be overcome by an ideal IoT OS and the key design criteria that need to be kept in mind while designing an OS for IoT devices. We also performed a comparative study of three open source operating systems for IoT devices and saw the unique features that each of them have to offer.

Because of its modular kernel design, RIOT provides better protection against bugs. Also, it's the only one out of the lot to provide complete preemptive multithreading support. The dynamic loading and unloading of modules through remote deployment is one key advantage of Contiki. The ability of MOS to fit the entire OS < 500 bytes of memory and the sleep function are what stand out. So, each of these operating systems have certain key features as their strong sites but they only succeed partly in meeting all the requirements of IoT ecosystem. To conclude, it can be said that RIOT is the most promising among the three but more research is required in the field to come up with a kind of "unifying" OS that meets the wide spectrum of requirements of the IoT ecosystem and can serve as a "go-to" OS for IoT.

REFERENCES

[1] E. Baccelli, O. Hahm, M. Gunes, M. Wahlisch, T. C. Schmidt, "OS for the IoT—Goals challenges and solutions", *Proc. WISG*, pp. 1-6, 2013.

[2] A. Dunkels, B. Gronvall, and T. Voigt, "Contiki—A lightweight and flexible operating system for tiny networked sensors," in *Proc. 29th Annu. IEEE Int. Conf. Local Comput. Netw.*, 2004, pp. 455–462.

[3] S.Bhatti, J.Carlson, H.Dai, J.Deng, J.Rose, A.Sheth, B. Shucker, C. Gruenwald, A. Torgerson, and R. Han. Mantis os: An embedded multithreaded operating sys- tem for wireless micro sensor platforms. *ACM/Kluwer Mobile Networks & Applications (MONET), Special Issue on Wireless Sensor Networks*, 10(4):563–579, August 2005.

[4] E. Baccelli, O. Hahm, M. Günes, M. Wählisch, and T. C. Schmidt, "RIOT OS: Towards an OS for the Internet of Things," in *Proc. IEEE Conf. INFOCOM WKSHPS*, 2013, pp. 79–80.

[5] O. Hahm, E. Baccelli, H. Petersen, and N. Tsiftes. Operating Systems for Low-End Devices in the Internet of Things: a Survey. *IEEE Internet of Things Journal*, PP(99):1–1, 2015.

[6] Internet of things. 2017. WikipediA: the Free Encyclopedia. Retrieved from https://en.wikipedia.org/wiki/Internet_of_things

[7] A. Benafa, 'Three Major Challenges Facing IoT', 2017. [Online]. Available: https://iot.ieee.org/newsletter/march-2017/three-major-challenges-facing-iot

[8] B. Dickson, '4 Major Technical Challenges Facing IoT Developers', 2016. [Online]. Available: https://www.sitepoint.com/4-major-technical-challenges-facing-iot-developers/

[8] *RIOT OS - An OS for the IoT*, 2012. [Online]. Available: http://www.riot-os.org