The Most Efficient Operating systems for IoT devices

Atefeh niroomand
Department of computer engineering, payam noor university,Tehran, Iran,

Abstract

Internet of Things (IoT) technology has so deeply penetrated our daily lives that IoT devices have been everywhere. Not only do IoT devices provide basic means to convenient living but they also offer intelligent services tailored to diverse user needs. Intelligent devices in the form of smart applications will get connected to make an individual's life smoother, more comfortable, faster, and accessible from anywhere at any time. IoT combines the power of IPv٦ for network connectivity, sensing, and nextgen communication technologies to meet future demands. The operating system is an important factor in memory, computation, and energy in IoT devices. By doing this survey, it is possible for other research communities to make an appropriate choice for OS and make IoT a reality. Furthermore, this article focuses on achieving user-level reliability in an IoT operating system (OS) that executes both real-time (RT) and non-real-time (NRT) tasks concurrently. this study is discussed which provides Reliable and real-time IoT OS and other security challenges; We compare several important operating systems of the Internet of Things.

Keywords: internet of things, operating systems, smart appliances, IoT integration

١. Introduction

Ever growing networks of physical object that tends to interconnect the real world with a digital concept in the form of smartness are gaining momentum. It gives emergence to the term Internet of Things (IoT) proposed by Kevin Aston [١]. According to a Gartner report Internet of things installed base will be populated by ٣٠ billion smart devices in the near future [٢]. IoT enabled devices will provide a smart application in the form of Industrial IoT, agriculture, smart home, healthcare, logistics, etc. Wireless sensor networks, actuators, and embedded systems with microcontrollers and chips acting as an integral part in designing smart and intelligent devices. Due to various challenges in terms of heterogeneity, scalability, security, and limited resource constraint environment in the form of memory and computational power, normal OS'es will not work. This distinct and flexible feature of IoT devices provides the need to survey an efficient portable, lightweight operating system.

During this growth, the IoT operating system (OS) market is also anticipated to expand exponentially at a compound.

growth rate (CAGR) of ٤١٪ during the forecast period ٢٠١٧–٢٠٢٣ [٣]. It is natural to state that the tremendous growth of smarter, more autonomous IoT devices has a dominant effect on the IoT OS market expansion.

The IoT OS market is generally divideed into two groups: ١) commercial OSs and ٢) open-source OSs. The latter group of OSs receives more interest because they typically have permissive licenses, such as Berkeley Software Distribution (BSD) or Apache ٢,٠, and have been maintained by giant information technology (IT) companies, e.g., FreeRTOS by Amazon [٤], [٥], mbed OS by ARM [٦], Zephyr by Intel [٧], and LiteOS by Huawei [٨]. They are in common real-time OSs (RTOSs) for various IoT devices. Along with commercial OSs, such as THREADX RTOS [٩] and Nucleus RTOS [١٠], these RTOSs have been promoting the popularization of low-cost IoT devices. They can run on ARM Cortex-M [١١] and Cortex-R [١٢] processors with only a few kB or MB of random access memory

(RAM). In contrast, the IoT version of Android called Android Things [١٣], which is based on the Linux kernel, requires at least ٥١٢ MB of RAM [١٤].

Recently, IoT devices have become more intelligent and more autonomous such that IoT virtually stands for the Intelligence of Things instead of IoT. For being smarter, IoT devices include not only RT tasks but also more complex tasks, which are usually non-real-time (NRT), i.e., relatively time-consuming tasks.

This review paper provides comprehensive details analysis of all operating systems which can be applied in the Internet of Things environment. Along with OS under a single basket, it discusses the IoT protocol stack perspective from the developer side community in terms of constraint devices, IoT Gateway, and Cloud service provider support. Key highlights of this review are as follows:

- Considering various challenges for the IoT environment.
- Provides design characteristics necessary for considering the operating System.
- Choosing the right candidate OS based on the comparison of the state of the art efforts and additional features.
- Case study showing IoT protocol stack implementation in terms of testbed facilities, research count of widely used protocol, and bifurcation in terms of constraint devices, gateway, and cloud service provider.
- Widely used IoT simulator for real-time analysis is also explored.
- Finally, the integration of IoT with future thrust domains along with issues is discussed at the end.[١٥]

٢. Major characteristics for designing IoT OS

In this section, we thus give an overview of crucial desirable characteristics for designing OS in the IoT context that should aim to satisfy.

٢,١. Architecture and modularity

The first layout choice that has to be made for any OS is the selection of the kernel type. This choice has a primary impact on the overall architecture and modularity of the operating system. A kernel is a bridge between the application and actual data processing done at the hardware level.

٢,٢. Scheduling

Another crucial parameter for designing an operating system is a scheduler. There are mainly two types of schedulers: ١) preemptive schedulers and ٢) non-preemptive or cooperative schedulers.

٢,٣. Memory allocation/footprint and management

Memory is typically a rare asset for IoT devices. The bigger question that remains while deciding on memory is how to allocate memory which eventually affects the other operating system design criteria. Typically there are two ways to manage memory allocation: ١) Static and ٢)Dynamic memory allocation. Static memory allocation is simple and provides faster access. Still, there are more chances of wastage while dynamic memory allocation is a flexible technique that efficiently utilizes memory during the run time but is slower in access. Real-time application, caching, virtual environment, and application type may play a significant role when deciding on allocation technique.

٢,٤. Energy efficiency

Most of the IoT devices are power constrained and run on batteries. Energy efficiency can be an essential factor considering billions of IoT devices expected to get deployed in the future. Several factors which may affect energy efficiency are the type of architecture used while designing, interrupts generated either by a kernel or externally, and scheduling strategy using the periodic timer for time-slicing purpose. To achieve energy efficiency periodic tasks must be reduced, eventually allowing the system to spend maximum time in sleep mode .[٢٠]

٢,٥. Network connectivity and protocol support

The key requirement for IoT Network Connectivity and Protocol support should be lightweight, modular in nature, open standard, flexible in supporting demands of wide range heterogeneous devices. applications capable of communication with low power consumption and internet-enabled. Ipv٦ addressing is compulsory for full filling uniqueness among a broader domain .[٢١]

٢,٦. Programming model and languages

The choice of programming model can impact performance, the productivity of application development, and flexibility. it can be classified into an event-driven and thread-based system.

٢,٧. Real-time support

The optional design characteristics based on application are gaining a lot of momentum since millions of smart devices are getting connected to the internet daily performing critical and real-time tasks. The real-time application can be broadly categorized into periodic, non-periodic, critical, and non-critical. Operating System should be

designed to guarantee the implementation of QoS architecture, and the kernel should be able to operate with deterministic run time.[٢٠]

٣. The Common OS for IoT

In this section, different open and closed-ended OS's for various low powered resource constraint IoT devices is discussed.

٣,١. Contiki

It is a lightweight, flexible, event-driven, portable, and open-source operating system created by ''Adam Dunkels in ٢٠٠٢ at the Swedish institute of computer science" .

٣,١,١.Architecture

Contiki uses hybrid modular architecture based on an event-driven model which reduced overall system memory size due to asynchronous events. It provides concurrency by considering a single shared stack for all processes. Core and loadable programs are triggered by events at runtime or by a polling mechanism, which helps to set priority and avoid a race condition.

٣,١,٢.Programming model and scheduling

Support pre-emptive multithreading in the form of protothreads which are lightweight stackless threads implemented as application libraries on top of the kernel, which performs CPU-multiplexing and event handling. Protothreads reduce overhead by not creating a separate stack for each thread. It provides a sequential flow of structure by allowing blocking functions. Interrupt, and real-time timer plays a vital role in preemptive nature.

٣,١,٣.Memory management and protection

Support dynamic memory allocation and dynamic linking of programs with automatic defragmentation with the help of managed memory allocator which prevents wastage of memory and guarded protection against fragmentation.

٣,١,٤.Network protocol communication

Contiki provides uIP TCP/IP communication stack which uses an ٨–١٦ bit microcontroller that supports IPv٤ networking, uIPv٦ stack to support both IPv٤ and IPv٦ networking.

٣,١,٥.Real-time support

Contiki lacks native real-time support but provides using Ring File System (RFS).
and may not contain subheadings. Figure legends are limited to ٣٥٠ words each. References are limited to ٧٠. Footnotes are not used.[١٧]

٣,٢. Tiny OS

It is ''open-source, event-driven, flexible, application-specific, component-based, providing the concurrent environment with low memory footprint, i.e. < ١ kb of RAM and < ٤ kb of ROM and sophisticated design prominent operating system developed and managed by TinyOS Alliance licensed under BSD" .

٣,٢. ١.Architecture

The tiny OS uses monolithic kernel architecture based on a component model which enables fast innovation and helps to reduce code. Component intercommunication can be carried out by commands and preemptive events while non-preemptive tasks are used to provide component concurrency. It uses a single shared stack with no differentiation between kernel space and user space which further helps to achieve component concurrency.

٣,٢,٢.Programming model and scheduling

The tiny OS uses an event-driven mode, which helps to utilize the resource of the CPU effectively. Tiny OS ٢,١ supports TOS threads, which allows achieving more significant concurrency concept. Additional benefits of using the TOS thread are that it offers both NesC and C APIs as well as fully support preemptive nature at application level threads. The scheduler executes high priority threads and events using FIFO preemptive scheduling. Later on, EDF (Earliest Deadline First) scheduling was supported, which overcome the problem of real-time task scheduling.

٣,٢,٣.Memory management and protection

Supports and uses static memory allocation at compile time which helps in preventing fragments as well as runtime allocation failures. Tiny OS ٢,١ version provides memory protection service called Safe TinyOS, which is robust in memory safety checks at runtime through Deputy Compiler.

٣,٢,٤.Network protocol communication

Tiny OS uses BLIP (Berkeley low power Internet Stack)IPv٦ communication stack and supports .

٣,٢,٥.Real-time support

The tiny OS doesn't support real-time applications, but the task can be run in real-time with the help of the earliest deadline first (EDF) scheduling.[١٨]

٣,٣. RIoT

It is ‘‘open-source, modular, lightweight, energy-efficient, multithreaded, uniform API access, smaller memory footprints.٥ kb of RAM to ~ ٥ kb ROM and developer-friendly operation system licensed under LGPLv٢,١٠٠. It was developed originally by the National Institute for Research in Computer Science and Automation (INRIA), HAW Hamburg, Free University of Berlin, and other open community around ٢٠١٣ which supports MCU hardware platforms in the form of ٨(Arduino Mega٢٥٦٠), ١٦(MSP٤٣٠) and ٣٢ bits.

٣,٣,١.Architecture

RIOT uses microkernel architecture inherited from FireKernel which provides stability, and security, i.e. if one module fails no need to load all modules, and extensibility through message passing. The typical architecture is written in C Language and also supports other powerful libraries of C++ like wiselib.

٣,٣,٢.Programming model and scheduling

RIOT uses multithreading model, which helps to optimize CPU resource utilization. It provides a preemptive, priority-based tickles (without fixed periodic timer) scheduler. Due to its dynamic tick property, it allows the system to go into the deepest possible sleep mode during idle time eventually leading to minimum energy/power consumption of the whole system.

٣,٣,٣.Memory management and protection

For memory, it doesn’t require MMU (Memory Management Unit) nor FPU (Floating Point Unit). It supports both static and dynamic memory allocation schemes with no memory protection. Static allocation helps to achieve real-time capabilities by providing deterministic requirements.

٣,٣,٤.Network protocol communication

RIOT supports several network stack in a modular fashion which allow easy exchange of every protocol at any layer. CCN-lite stack implements ICN (Information Centric Networking) paradigm and OpenWSN stack, which implements full ٦TiSCH (Time synchronized channel hopping) protocol suite on top of IEEE٨٠٢,١٥,٤e standard.

٣,٣,٥.Real-time capabilities

RIOT fully supports real-time capabilities with the help of static memory allocation, deterministic runtime at the kernel level, and tickless scheduler, which eventually minimizes interrupts.

٣,٣,٦.Additional features

RIOT provides comprehensive and extensive documentation both at API and architectural level along with good standards applied using Doxygen.

RIOT uses gdb and Valgrind as debugger tools. Wireshark can be used as a packet analyzer tool, and the cooja simulator can be used to support the MSP٤٣٠ hardware platform. For visualization purposes, inbuilt RIOT-TV is used.[١٦]

٣,٤.FreeRTOS

It is ‘‘open-source, portable, royalty-free, scalable, real-time, energy-efficient (using tick-less mode), simple in design, with smaller memory footprints, i.e. ٤–٩ kb of RAM to ٥–١٠ kb ROM, multithreaded and easy to use operating system licensed under modified GPL”. Initially developed by Richard Barry in ٢٠٠٢ is now maintained and distributed by Real Time Engineers Ltd. As per the recent survey, FreeRTOS on average is downloaded every ١٢٠sec due to its real-time characteristics and also supports more than ٣٥ architectures.

٣,٤,١.Architecture

FreeRTOS mainly used relatively simple microkernel features consisting of mainly three C files and a handful of header files totaling ٩٠٠٠ lines of code including comments and blank lines. It implements the core functionality of the operating system, such as queues, semaphores (binary and counting), software timers, and mutexes.

٣,٤,٢.Programming model and scheduling

It supports a multithreading programming model based on task (separate stack) and co-routines (shared stack). Priority-based Round Robin scheduling is applied on tasks while co-operative scheduling is applied on co-routines.

٣,٤,٣.Memory management and protection

Due to different RAM and timing requirements for various applications, FreeRTOS supports dynamic memory allocation with five heaps. Heaps provide simplicity, fragmentation protection, and thread safety. FreeRTOS-MPU (Memory Protection Unit) secure data from corruption, modification, and stack overflows.

٣,٤,٤.Network protocol communication

By combining FreeRTOS with Nabto, communication becomes easy between IoT peer devices. Nanostack supports ٦lowpan implementation by considering mesh-under routing.

٣,٤,٥.Real-time capabilities

FreeRTOS versions after ٢٠١٢ support tickless mode in scheduling which reduced interrupts. Deterministic operations at kernel level help to achieve real-time capability.[٢١]

٣,٥. Mantis OS

The Multimodal system for networks of in-situ wireless sensors (MANTIS) is ‘‘open-source, multithreaded, lightweight, portable with cross-platform support, the footprint of ٥٠٠ bytes (kernel, scheduler, and network stack) and energy-efficient developed and released in ٢٠٠٥ licensed under BSD”.

۳,۵,۱. Architecture

Mantis OS adheres layered architecture which resembles UNIX style schedulers. Top layers provide API for I/O and system interaction for various applications while lower levels help in achieving a smaller memory footprint.

۳,۵,۲. Programming model and scheduling

Mantis OS supports time-sliced multithreading where each program can be written without thread parting in C language with the support of portability and reusability. The default number for
thread creation is ۱۲ with ۱۲ ms as default time slice for each thread.MantisOS uses pre-emptive Round Robin scheduling with multiple priority classes. Whenever there are no threads for processing in the queue system goes to sleep/idle mode, which eventually saves power and resources. Race conditions can be avoided by
using mutexes and semaphore, which is of ٥-byte structure, and it is declared as and when needed.

۳,۵,۳. Memory management and protection

Mantis OS provides dynamic memory allocation, which prevents the wastage of memory. It manages threads memory by dividing RAM into two portions, i.e. variable space and second managed by a heap. It does not support memory protection.

۳,۵,٤. Network protocol communication

Mantis OS implements network stacks by dividing it in two parts.۱) User space that implements layer ۳ and above protocols by providing flexibility. If a user wants to add its own data-driven routing protocol, it can add it in userspace.۲) Comm. layer which implements MAC and PHY layer protocols. It takes care of performance by providing a unified interface for communication with device drivers and hardware. It also performs packet buffering if a packet arrives for the thread but yet not scheduled.

۳,۵,٥. Real-Time capabilities

Provides very less support to a real-time application using preemptive scheduling with multiple priority classes but cannot be considered as real-time OS since it does not provide support to deadlines.[۱۸]

۳,٦. LiteOS

It is ‘‘open-source, interactive, zero-configuration, smaller memory footprints i.e. ~ ٤ kb of RAM to ۱۲۸ kb flash ROM, online debugging and lightweight Unix-like operating system initially designed at the University of Illinois, Urbana Campaign licensed under GPLv۳۰۰.

۳,٦,۱. Architecture

LiteOS uses modular architecture divided into three subsystems. Lite Shell residing at the base station having enough resources provides Unix-like command interface that supports the process, debugging, environment, and device commands. LiteFS provides the facility of wireless node mounting where nodes within range mount themselves as file and network as a directory. . The third component is a kernel that provides multithreading as well as dynamic loading.

۳,٦,۳. Memory management and protection

LiteOS uses dynamic memory allocation with the help of malloc and free functions. The size of memory is adjusted since it is used from an unused area between kernel variables and application memory
blocks, thus providing protection and security.

۳,٦,٤. Network protocol communication

LiteOS provides communication through radio files. It includes a plug, and play/file assisted routing stack. Mintroute which provides reliability is used at the network layer.BMAC which provides scalability, energy efficiency, and collision avoidance is used at the MAC layer.

۳,٦,٥. Real-Time capabilities

LiteOS does not provide real-time support since it uses priority base scheduling that runs tasks till completion.[۲۰]

۳,۷. NanoRK

It is ‘‘open-source, lightweight, energy-aware, real-time, resource-centric, smaller memory footprint, i.e. ۲ kb of RAM to ۱۸ kb of ROM, multitasking and multi-hop networking support operating system developed by Alexei Colin, Christopher Palmer, and Artur Balanuta at Carnegie Mellon University licensed under GPL.

۳,۷,۱. Architecture

NanoRK applies monolithic kernel architecture having similarities with Tiny OS. To fulfill task priorities and deadlines, the static approach is used so that the admission control procedure can be applied efficiently. Task parameters can be changed at run time through various APIs by the application programmer, but it is not encouraged when the task performed is a real-time job.

۳,۷,۲. Programming model and scheduling

Each task will have a different priority, running, and operative frequency. NanoRk applies multitasking with the help of multiple threads and Task Control Block (TCB). Synchronization and Concurrent Control among tasks will be handled with the help of mutexes and semaphores. Each individual task will have its own TCB initialized during creation. TCB contains stack and register information of tasks, priority, port identifiers, and reservation size. NanoRK provides fully pre-emptive priority-based scheduling at the process and network level. To fulfill real-time capabilities, it uses a rate monotonic scheduling algorithm where task priorities are assigned statically. To efficiently

utilize energy during CPU idle time rate, harmonized scheduling technique is used. Priority inversion where the lower priority process tends to use resources of higher priority is solved by the priority ceiling technique by binding blocking time.

٣,٧,٣. Memory management and protection

NanoRK provides static memory management wherein application, and OS both resides in a single memory space. It does not provide any memory protection to safeguard co-locatedOS.

٣,٧,٤. Network protocol communication

It uses a lightweight networking stack wherein communication is carried out with the help of port numbers and sockets. NanoRK applies TDMA based communication protocol RT-Link at the link layer, which provides functionality like collision-free energy efficient real-time transmission and supports in-band and out of band hardware/time-based synchronization. WiDOM dominance collision-free wire-less protocol which assigns priority in a static manner where each node fight as a tournament to have access
to the channel. B-MAC carrier sense multiple access protocols are also used at the MAC layer, which provides efficient channel utilization and scalability. A U-Connect protocol which helps to achieve synchronization for neighbor discovery challenges by providing energy efficiency and latency.

٣,٧,٥. Real-time capabilities

NanoRK fully supports real-time features through a rate monotonic priority-based pre-emptive scheduling algorithm where task priority is assigned offline and statically by full filing deadlines.[١٦]

٤. Evolution Toward IoT OS

MPU-based memory protection is generally supported by open-source RTOSs, such as Mbed OS, FreeRTOS, and Zephyr. Mbed OS provides only two basic types of protection: ١) preventing execution from RAM and ٢) preventing write to flash memory. These features are automatically enabled or disabled according to device situations, such as starting a new application or flash programming. FreeRTOS protects the kernel from invalid execution by user tasks and detects stack overflow with up to three MPU regions per task (or thread). This MPU port, however, is seldom used and is not well maintained. Zephyr also provides stack-overflow protection and thread-level memory protection by dynamically updating at least two MPU regions, whereas all threads share one memory region, where global variables are placed. This shared memory among all threads is against per-thread memory isolation and likely becomes vulnerable to a single point failure.

TizenRT was kicked off with the TinyAra project, which was based on a Linux-like kernel architecture inherited from the NuttX kernel [٤٩], in ٢٠١٥ and opened as a public open-source project under Apache ٢,٠ on GitHub in ٢٠١٦. While maintaining the kernel architecture, TizenRT has grown toward an IoT OS by building up the Wi-Fi management modules, IPv٤/IPv٦ network stacks, and IoT protocols, such as Open Connectivity Foundation (OCF), SmartThings with message queuing telemetry transport (MQTT), and lightweight machine to machine (LWM٢M). These features have been developed to control and monitor IoT devices easily and securely over the Internet, which is the most important feature of IoT devices. In addition, IoT devices are utilized to collect sensor data, which may be uploaded to cloud servers via wireless connectivity or consumed by on-device AI. To this end, TizenRT provides a fail-safe filesystem and lightweight database, which enable create, read, update, and delete (CRUD) functions to process data reliably and easily.

With these features since ٢٠١٧, TizenRT has been applied mostly to smart appliances, such as refrigerators, air conditioners, air purifiers, etc. To the best of our knowledge, TizenRT is the first full-fledged and open-source RTOS that is applied to almost all types of appliances. Beyond these features, IoT devices should have user-friendly interfaces, such as touch screen (i.e., for wearable devices) and voice recognition (i.e., for headless devices).

TizenRT has already supported not only a user-interface framework but also a voice service called Bixby. The latter is particularly necessary for going forward to the Intelligence of Things. We already demonstrated Bixby on top of TizenRT at Samsung Developer Conference (SDC) ٢٠١٨, where a presenter commanded a Bixby-enabled device in a voice not only to turn ON and OFF a light-emitting diode (LED) ceiling light but also to play and stop a song.[١٩]

٤,١.Memory Protection and Its Overhead

Resource sharing among tasks is logically prohibited except for intertask communication (ITC) methods, such as message queues and pipes. However, a task can access all resources of other tasks intentionally or by mistake if there is no MPU-based memory protection. ARM Cortex-M/R processors support an MPU, which specifies per-region access control rules for physical memory. The number of MPU regions varies from ٨ to ١٦, depending on vendor configurations of microcontrollers, among which NXP i.MX RT١٠٢٠ has ١٦ MPU regions. Each MPU region is defined by a start address, a size, and attributes. The attributes determine the way in which the memory in a region can be accessed by a processor.

Suppose that an MPU region is configured to be executable and read-only (RO). When a processor tries to access this area of memory to write, the MPU checks the access type against the attributes and generates a permission fault. TizenRT as a reliable IoT OS must use at least three MPU regions in RAM, each of which has different MPU attributes, to protect a single-user binary. Its text and RO data are loaded into an RO and executable memory region

and an RO memory region, respectively. The RW data are copied into an RW memory region and bss (i.e, uninitialized global variables), stack, and heap are also created in the same region. With this configuration, text modification and data access for execution, which would happen due to internal errors or security attacks, are not allowed.

٤,٢.RELIABLE IOT OS

For reliability, the life cycles of all user binaries should be managed. For example, a user binary is loaded at a boot time, reloaded when a fault happens or updated if necessary. The binary manager is in charge of these roles with the help of the fault handler and a loader. When a permission fault is generated, the fault handler is invoked with the faulting instruction, the fault type, and the data address in the case of data access violation.

Instead of halting an entire system, it isolates the faulty binary and notifies the binary manager which binary has faulted. This is possible because memory protection assures that the fault is confined to the binary. The binary manager handles requests not only coming from the fault handler but also coming from an OS initialization task and update client. Then, it creates a loader to load a requested binary. If it is already loaded on memory either running or corrupted, the loader releases all memory resources used by the binary. Finally, the loader loads and executes the binary without a system reboot.

The binary manager can also adjust user binaries' heap memory at runtime. For example, a user binary itself requests the binary manager to increase its heap memory when it encounters *malloc*() failures. Then, the manager reloads it with increased heap memory. Furthermore, if a user binary reserves larger heap memory but underutilizes it, the binary manager can forcibly cut its heap.

The rest of this section is organized as follows. First, fault isolation and fast recovery will be addressed to achieve the two goals with the overall workflow of binary management.

Second, we will devise a fast interrupt notification method to offset delays caused by fault isolation and fast recovery, and a binary compression feature to reduce storage memory and a binary transmission time over the air at the expense of a loading time. The important features of this operating system include:

A. Memory Fault Isolation
B. Fast Recovery
C. Fast Interrupt Notification
D. Binary Compression.[١٧]

٥. CONCLUSION

Integrating IoT with other domain play a vital role in achieving quality of service parameter, smartness, scalability, and heterogeneity among diversified devices. The reliable IoT OS has been designed mostly for smart appliances, which are considered as richer and faster devices. In this review paper, we have started with analysing various design criteria challenges required for meeting the IoT constraint environment. Different current and future IoT operating systems are then presented with core and additional features in detail.

Contiki and TinyOS is currently the most popular open-source operating system among the research community, while FreeRTOS and RIOT are gaining momentum due to their real-time capabilities.

We experimentally proved that TizenRT OS can successfully protect all user binaries from each other. The most challenging goal is to guarantee RT threads to achieve their missions within a required time, e.g., ٥٠ μs, even while a faulty binary is being recovered. Moreover, the corrupted binary can be recovered (i.e., unloaded, reloaded, and re-executed) within a few milliseconds. The proposed reliable and effcient IoT OS has been designed mostly for smart appliances, which are considered as richer devices.

As future work, we will study architecture for kernel level reliability like the microkernel shared libraries in user space can reduce memory footprint, but requires more MPU regions and likely degrades user-level reliability.

References:

[١] Ashton, Kevin, ٢٠٠٩. RFID Journal ٧, ٩٧–١١٤.
[٢] "Newsroom", Gartner. [Online]. Available at: https://www.gartner.com/newsroom/ id/٢٦٣٦٠٧٣. [Accessed: ٠١-Nov- ٢٠١٧].
[٣] Zhang, D., Ning, H., Xu, K.S., Lin, F., Yang, .T., ٢٠١٢. Internet of things j. ucs special issue. J. nivers. Comput. Sci. ١٨ (٩), ١٠٦٩–١٠٧١.
[٤] *Amazon-FreeRTOS*, Amazon Web Serv. Inc., Seattle, WA, USA, ٢٠٢٠. [Online]. Available: https://aws.amazon.com/freertos/
[٥] *FreeRTOS*, Amazon Web Serv., Inc., Seattle, WA, USA, ٢٠٢٠. [Online].Available: https://www.freertos.org
[٦] *Mbed OS*, Arm Ltd., Cambridge, U.K., ٢٠٢٠. [Online]. Available: https://www.mbed.com/en/platform/mbed-os/

[۷] *Zephyr*, Linux Found. Project, San Francisco, CA, USA, ۲۰۲۰. [Online]. Available: https://www.zephyrproject.org/
[۸] *LiteOS*, Huawei Technol., Shenzhen, China, ۲۰۲۰. [Online]. Available: https://www.huawei.com/minisite/liteos/en/
[۹] *THREADX*, Express Logic, San Diego, CA, USA, ۲۰۲۰. [Online]. Available: https://rtos.com/solutions/threadx/real-time-operating-system/
[۱۰] *Nucleus RTOS*, Mentor Graph. Corp., Wilsonville, OR, USA,۲۰۲۰. [Online]. Available: https://www.mentor.com/embedded-software/nucleus/
[۱۱] *Cortex-M4 Processor Technical Reference Manual, Rev. r0p1*, ARM Ltd., Cambridge, U.K., ۲۰۲۰.
[۱۲] *Cortex-R4 and Cortex-R4F Technical Reference Manual, Rev. r1p4*, ARM Ltd., Cambridge, U.K., ۲۰۱۱.
[۱۳] *Android Things*, Google, Mountain View, CA, USA, ۲۰۲۰. [Online].Available: https://developer.android.com/things
[۱٤] Android Things, *Get Started With Kits*, Google, Mountain View, CA,USA, ۲۰۲۰. [Online]. Available: https://developer.android.com/things/get-tarted/kits/
[۱٥] Seong-Il Hahm , Jeongchan Kim, Ahreum Jeong, Hyunjin Yi, Sunghan Chang, Shobha Nanda Kishore,Amandeep Chauhan, and Siju Punnoose Cherian," Reliable Real-Time Operating System for IoT Devices", IEEE INTERNET OF THINGS JOURNAL, VOL. ۸, NO. ٥, MARCH ۱, ۲۰۲۱.
[۱٦] Bimal Patel, Parth Shah," Operating system support, protocol stack with key concerns and testbed facilities for IoT: A case study perspective", Production and hosting by Elsevier B.V. on behalf of King Saud University, https://doi.org/۱۰,۱۰۱٦/j.jksuci.۲۰۲۱,۰۱,۰۰۲.
[۱۷] Dunkels, A., Gronvall, B. and Voigt, T., ۲۰۰٤. Contiki- a lightweight and flexible operating system for tiny networked sensors. In ۲۹th Annual IEEE International Conference In Local Computer Networks. pp. ٤٥٥-٤٦۲.
[۱۸] Farooq, M., Kunz, T., ۲۰۱۱. Operating systems for wireless sensor networks: a survey. Sensors ۱۱ (٦), ٥۹۰۰–٥۹۳۰.
[۱۹] V. D. Silva, J. Roche, X. Shi, and A. Kondoz, "IoT driven ambient intelligence architecture for indoor intelligent mobility," in *Proc.IEEE 16th Int. Conf. Depend. Autonom. Secure Comput. 16th Int. Conf Pervasive Intell. Comput. 4th Int. Conf. Big Data Intell. Comput. Cyber Sci. Technol. Congr. (DASC/PiCom/DataCom/CyberSciTech)*, Athens,
[۲۰] Hahm, O., Baccelli, E., Petersen, H., Tsiftes, N., ۲۰۱٦. Operating systems for lowend devices in the internet of things: a survey. IEEE Internet Things J. ۳ (٥), ۷۲۰–۷۳٤.
[۲۱] Gaur, P. and Tahiliani, M., ۲۰۱٥. Operating systems for IoT devices: A critical survey. In Region ۱۰ Symposium (TENSYMP). pp. ۳۳-۳٦.