Chapter 5

An Architecture and Reference Implementation for WSN-Based IoT Systems

Burak Karaduman
https://orcid.org/0000-0002-7262-992X
University of Antwerp and Flanders Make, Belgium

Bentley James Oakes
University of Antwerp and Flanders Make, Belgium

Raheleh Eslampanah
https://orcid.org/0000-0001-8188-7464
University of Antwerp and Flanders Make, Belgium

Joachim Denil
https://orcid.org/0000-0002-4926-6737
University of Antwerp and Flanders Make, Belgium

Hans Vangheluwe
University of Antwerp and Flanders Make, Belgium

Moharram Challenger
University of Antwerp and Flanders Make, Belgium

ABSTRACT

The Internet of Things and its technologies have evolved quickly in recent years. It became an umbrella term for various technologies, embedded devices, smart objects, and web services. Although it has gained maturity, there is still no clear or common definition of references for creating WSN-based IoT systems. In the awareness that creating an omniscient and ideal architecture that can suit all design requirements is not feasible, modular and scalable architecture that supports adding or subtracting components to fit a lot of requirements of various use cases should be provided as a starting point. This chapter discusses such an architecture and reference implementation. The architecture should cover multiple layers, including the cloud, the gateway, and the edges of the target system, which allows monitoring the environment, managing the data, programming the edge nodes and networking model to establish communication between horizontal and vertical embedded devices. In order to exemplify the proposed architecture and reference implementation, a smart irrigation case study is used.

DOI: 10.4018/978-1-7998-4186-9.ch005

Copyright © 2022, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

1. INTRODUCTION

The Internet of Things (IoT) is a paradigm that aims to connect physical objects, intelligent devices, vehicles, machines, buildings and/or sensors to the Internet using communication protocols, wired/wireless hardware and embedded software (Karimpour, et al., 2019). The background technology of IoT includes radio-frequency identification (RFID), near-field communication (NFC), Wireless Sensor Networks (WSN), and other wired or wireless communication. Generally, IoT is based on establishing a bridge between the digital and the physical world by sensors and actuators. According to a study (Sharma et al., 2019), over 70 billion devices will be connected to the Internet by 2025, and the world will become more digitized through smart, distributed and power-efficient nodes.

In order to increase the network coverage of an area, these IoT devices can create ad-hoc networks with their neighbor nodes, termed a Wireless Sensor Network (WSN). The WSN paradigm is well-suited for distributed data acquisition using low-power antennas and embedded devices for various applications (Arslan, et al., 2017). Generally, wireless sensor networks use routing protocols to send a packet from a source node to the sink node. If a gateway transmits this data from a sink node and sends it to a computer or log manager system, that WSN system can be considered part of an IoT ecosystem. Inside the IoT ecosystem, various platforms can be included, such as IoT nodes, WSN nodes, Long Range Wide Area Network (LoRaWAN) and Bluetooth Low Energy (BLE) devices. The common basis of these systems is embedded computing systems designed to perform tasks such as measuring environmental changes and converting them into a human-readable format or digital data. These systems can perform the tasks in an event-based or real-time manner. Additionally, the embedded systems may have an operating system to manage the system resources and have an antenna to establish wireless communication. Therefore, hardware and an accompanying protocol are required to create a network between embedded systems to wirelessly collect data in a wide area.

1.1 Motivation

IoT systems should be designed considering both environmental and user-oriented requirements. They are inherently connection-based systems, and as expected, there will be billions of these devices where scalability becomes an essential feature in the future. Moreover, these devices may not have any user interface or maybe abstracted from human intervention. Therefore, they need a log and event management system that handles changes in the environment by remotely controlling IoT devices. However, the design constraints of the system should be aligned considering the conditions of the environment. In particular, the lifetime of the IoT node is essential when these nodes are deployed in vast rural areas. It may not be possible to find a power source to provide continuous energy to these nodes. Therefore, the dependency on the power source should be reduced to increase nodes' lifetime. The necessity of creating a network without requiring a direct Internet connection has emerged.

For these reasons, IEEE 802.15.4-based WSN nodes are suitable since they are designed for low-power and long life-time dependent applications. When this low-power antenna technology is merged with energy-efficient micro-controllers (Chéour et al., 2020), it can create a mesh network without requiring a direct Internet connection and any power source. WSN nodes create their dynamic network, and new nodes can be easily added/removed. The network can organize itself if the topology changes. The WSN can be opened to the Internet. When the sink is connected to the gateway, data can reach to Internet level via a gateway. Internet level may also have IoT nodes. Suitable communication protocols should be selected considering user requirements and environmental conditions. In addition, variants of operating systems and devices may create indecision for practitioners. For these reasons, we provide a reference architecture of physical components, operating systems, embedded hardware and software covering IoT layers while integrating IoT and WSN systems. The WSN empowers the communication of IoT devices in a wide area (using IEEE 802.15.4) where direct Internet connection (IEEE 802.11) is not available. In this study, we present our architecture integrating the WSN paradigm into the IoT ecosystem. The architecture acts as a reference for how a WSN-based IoT system design could be defined.

1.2 Case Study: Smart Irrigation System

In order to assess our architecture's benefits and evaluate the proposed architecture, a case study of a smart irrigation system is analyzed, designed, and implemented. The irrigation system includes both WSN and Wi-Fi modules to detect the soil's moisture level in an area such as large fields, vineyards, and gardens.

The distributed nodes continuously sample the moisture level of a field, while each IoT node controls a solenoid valve to irrigate a part of a field. The distributed data is collected by the sink node and delivered to the Log Manager system. Log Manager logs the data and monitors the state changes in target fields while comparing the desired moisture level with collected moisture values. When the system detects any moisture drop in the field, the log manager requests the corresponding IoT node to irrigate that part by controlling the solenoid valve. This case study demonstrates how the WSN paradigm can enhance an IoT system addressing irrigation. It shows the distribution of WSN nodes across large fields with many IoT components attached to water pipes to control specific zones of those fields. In addition, the irrigation case study also contains a complex, layered IoT system starting from sensor sampling, routing protocol creation, gateway implementation, to log/event management.

The primary motivation for this architecture is that series of case studies are established based on the provided reference architecture. Provided architecture is applied (Asici et al., 2019; Marah et al,. 2020; Karaduman et al,. 2018a; Karaduman et al,. 2020a; Karaduman et al,. 2020b) where fire detection, lighting automation and emergency control systems are implemented using reference operating systems and hardware.

1.3 Contributions

This study contributes a reference architecture and implementation for WSN-based IoT systems. The WSN paradigm is integrated into the IoT ecosystem such that IoT systems can benefit from the advantages of the settled WSN technologies. In this way, IoT systems are empowered with low-power, low-cost and well-established WSN technology as well as providing distributed topology to increase the coverage for IoT. Moreover, our architecture also eases the problems of resource-constrained environments where the energy source is limited or not continuous.

With the integration of the WSN, we also provide a self-sustainable network where the nodes can act as a subsystem as centralized and decentralized operation modes. We present an architecture that can be scale-able via dynamic network behaviour where new WSN nodes can be added or removed easily. Considering the environmental conditions of operation area the coverage may be extended, so WSN should also adopt these physical extensions by adding new nodes to their network. We show how IoT and WSN systems can be implemented by showing the implementation details and engineering methodologies starting from edge nodes to the cloud log manager. In the fog level, we present how a gateway can be created to read incoming data and transfer them to the Internet level. Lastly, we provide a comprehensive case study to show the applicability of our architecture. Specifically, this study provides assistance in resolving hardware selection confusion while also eliminating the operating system selection problem.

When two paradigms must be merged to solve a complex problem, then architectural requirements can be complex. Because of the variety of the firmware and hardware, the research and development phase can be delayed until a suitable architecture, development environment, and development boards are found. Then, developers start to seek references that will take them to the next step in the software life-cycle.

In this chapter, we have addressed these architectural challenges and included these requirements. The architectural requirements are the gateway and log manager of our architecture. In this way, developers are not only going to know about only operating systems and supported devices. They are also going to have an idea of how and why these requirements are filled and what are the alternatives. Using this reference architecture, an implementer can have a starting point for how to design and implement a WSN empowered IoT system.

This chapter is organized as follows: Section 2 provides an overview of the proposed architecture. The reference implementation is demonstrated in Section 3. Section 4 describes the implementation workflow for a case study. The chapter is concluded in Section 5, along with a discussion of possible future work.

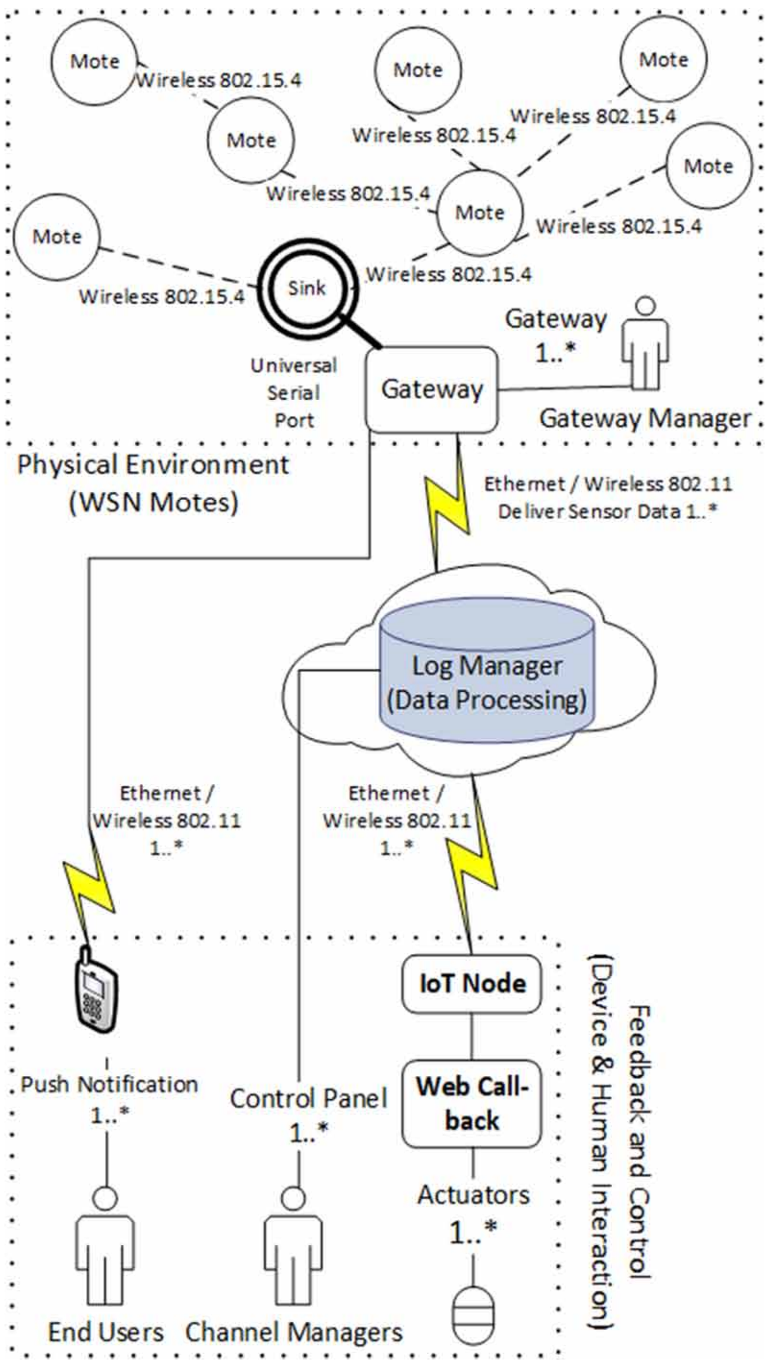
2. PROPOSED ARCHITECTURE

The proposed architecture is discussed in this section. It includes high-level design, including required components to create a WSN-based IoT system.

2.1 High-Level Design of the System

The architecture proposed in this study covers a high-level design of a WSN based IoT system. As illustrated in Figure 1, the high-level design includes the different groups of components such as WSN nodes, IoT devices, a gateway and cloud level log manager. WSN nodes can cover a large area using low-power motes, while IoT devices provide computation power and control for resource and response demanding tasks by directly accessing the Internet. These tasks can be related to M2M communication or human interaction. Specifically, IoT nodes may listen to a port for incoming requests to actuate a physical and electronic component like a servo motor to control a door lock mechanism. Cloud level log management systems can be implemented from scratch or pre-ready technologies such as ThingSpeak (Maureira et al., 2011), Google Cloud (Korobeinikova et al., 2020), Azure IoT (Klein, 2017) and AWS IoT (Kurniawan, 2018) are also preferable. Alternatively, graphical-based programmable solutions like Node-RED are also a choice to customize the cloud level. Specifically, these cloud technologies should be convenient for data-visualisation, logical comparison, rule definition and storing data.

Figure 1. The general view of the proposed architecture.



WSN-based Sensors

WSN is a suitable technology while there is no possibility to establish a direct Internet connection in the environment (Akyildiz et al., 2002). Using the multi-hopping feature of WSNs, they can have their coverage widened. These devices can create their network using IP-based protocols. They are also adaptable for low-power transmission (Toscano and Bello, 2012). Moreover, most of the WSN devices' microprocessors are designed considering low power consumption. They can yield the processor to switch to ultra-low-power when there is not an ongoing process. In this way, they provide additional energy-saving by going into the deep-sleep mode to minimize power consumption.

As the WSN nodes use the IEEE 802.15.4 protocol, the collected data from the environment is delivered to the sink node using a routing algorithm between source nodes. WSN nodes can convey their sensor data node-to-hand to reach the sink node. The sink node starts the routing algorithm as the root device to establish a network tree. This node acquires the environment information. That information is forwarded by distributed source nodes. In WSNs, intermediate nodes between the source node and the sink node cooperate with each other by forwarding each other's packets to achieve distributed data collection. This enables multi-hopping where a node cannot reach the sink node directly. In this way, wide areas can be covered, and data can be sampled and distributed.

Finally, all data received by the sink node must be stored, processed and probably forwarded to the Internet. This is done by a computer playing the role of a gateway and sending the data using its direct Internet connection. The gateway reads the data from the sink node using a Universal Serial Port (USB). Therefore, the data can be sent to a cloud or a Log Manager system, which is discussed in the following subsections.

IoT-based Nodes

Wi-Fi-based IoT nodes are useful when there is direct Internet connectivity. They can communicate using IEEE 802.11 protocol and can be utilized with socket programming techniques. Low-end devices such as ESP32/8266, MSP430, Particle Photon can be preferred for low-power and long-term tasks, while high-end single board computers such as RaspberryPi, Jetson Nano and BeagleBoard devices are suitable for real-time and high-computation demanding tasks.

Compared with WSN Motes, IoT-based nodes might consume more power. To ease this problem, some operations such as enabling deep-sleep mode, constraining computation by switching ultra-low power operation and implementing energy harvesting solutions can be preferred.

As illustrated in Figure 1, Web-based sensors can be used in connection with Wi-Fi connectivity modules. Also, in addition to using physical sensors, these devices can request data from a web resource such as a weather forecast website by sending a simple web request or using an API since they are directly connected to the Internet. As a result, if Internet connectivity is available in the area, the user should prefer to use Wi-Fi connectivity modules while using WSN motes to create a distributed network through a wide area.

Feedback and Control

In the proposed architecture, the feedback and control section includes the user system notifications, e.g., mobile notifications for the user. These notifications also provide two-way confirmation as they send a fail or success message to the sender when a notification or actuation is realised. Generally, these control messages are sent from the cloud to the edge level as feedback messages are created by edge devices and delivered to the cloud level. Specifically, if an event is triggered, like a rise in moisture level inside a farm, then the log manager in cloud level can send an action request to an actuator to open the irrigation valve to raise the soil's moisture. Then the farmer confirms whether irrigation is completed as planned and presses the Irrigation is completed button to notify the log manager. The log manager receives this feedback manager and marks that area as irrigated. Thus, human-machine interaction can be achieved using two-way confirmation through feedback and control messages.

Gateway

WSN devices benefit from gateways to reach cloud services. A gateway has an application that acts as an intermediate component. At some point, distributed data should be delivered to the cloud level by connecting the sink node of the WSN to itself and being connected to an access point wireless or cable. In this way, the gateway creates interoperability between the WSN and IoT.

Log Manager

The number of IoT devices like smart houses to intelligent transport systems and production systems to public services is increasing day by day. This technological advancement necessitates the storing, visualisation and management of the information generated by these devices. Moreover, change in the context of data can trigger various events to take necessary actions. Therefore, a log manager has to store this data, visualise it for the system administrator and take actions according to these changes.

In the scope of this study, the record management system is called the Log Manager, which can transfer the data coming from the devices located in different locations quickly and produce the desired events reports. To support the system's heterogeneity, the cloud-based log manager proposed in this architecture accepts the requests and sensor data via standard web-based requests, and the feedback can be a form of web-based callback, mobile notification, SMS, or email. These features can be extended depending on the requirements.

Log manager should operate independently from any platform or device. Therefore, the sensor data is received as a key-value with the key as a predefined tag in the log manager. So, each sensor has a tag in the log manager, and the sensor sends its data using this tag to the log manager. Unlike the available log managers such as Think Speak, the proposed log manager has no limitation to submit data at specific periods. Also, the log manager proposed in this architecture is much easier to use and configure compared to the available fully-fledged log manager such as AWS. Finally, it can also analyze the data based on the user's queries and report the results in the user's predefined forms such as charts and graphs.

2.2 Lifecycle of WSN-based IoT Systems

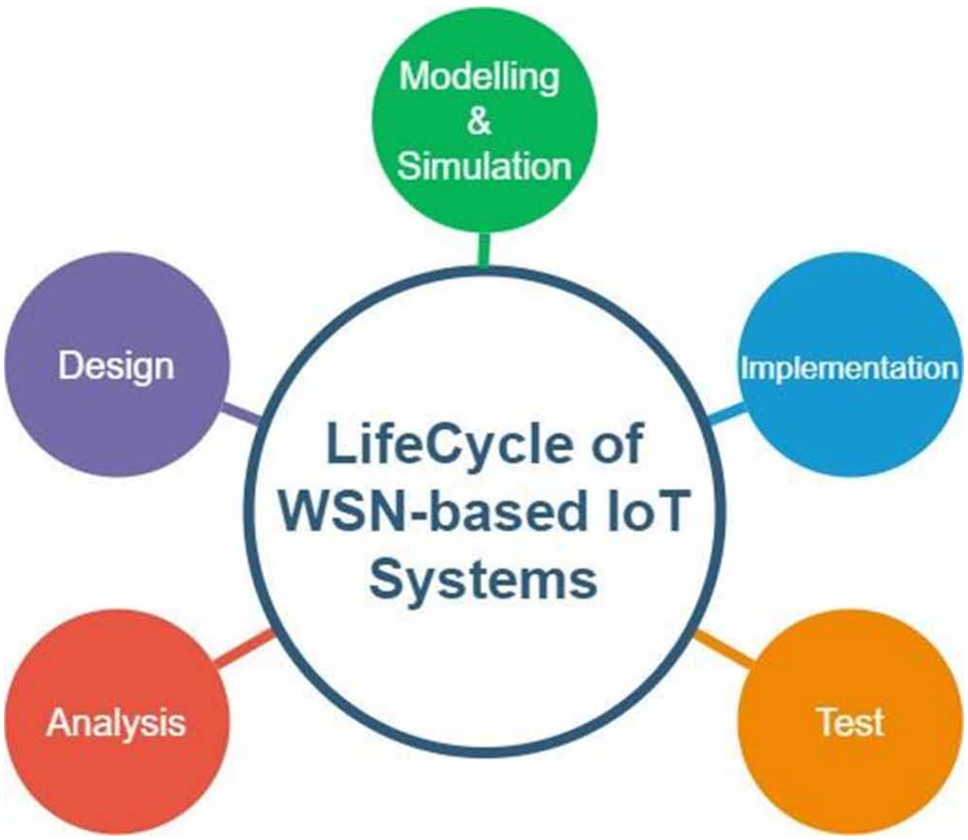
In addition to reference architectures, the lifecycle of WSN-based IoT systems guides developers to implement their systems, starting from bare-metal programming to fully functioning systems. It begins with System Analysis, System Design, System Modelling and Simulation, Implementation and ends with System Test. SysML diagrams (Holt and Perry, 2008) might be preferred to describe different phases of the development life-cycle of these systems, see Figure 2.

System Analysis: In this phase, the functional and non-functional requirements of the system are defined. Moreover, Use Case Diagrams are provided to describe user-system interactions. Specifically, use case specifications elaborate on the events and specifications of the system.

System Design: In order to model the system components, Block Definition Diagrams (BDD) are preferred to describe the types of hardware components, hardware connections of components, and hierarchies of software. A detailed design can be realized by providing Internal Block Diagrams (IBD). The BDD and IBD can provide the required structural modeling of the system. State machine diagrams and Activity diagrams are useful to model the system behaviour. These diagrams can be developed using the use case scenarios provided in the analysis phase.

In addition to this diagram, the other behavioural diagrams of SysML can be used, such as sequence diagrams, depending on the needs of the system under development (SUD). Early verification of system design can be made using Petri-Net models. This can help to check the important system properties, such as reach-ability, live-ness, traffic/load, and so on, to check whether the development of the system under design is feasible and efficient (Karaduman et al,. 2020a).

Figure 2. Lifecycle of WSN-based IoT Systems.



System Modeling and Simulation: WSN-based IoT systems are composed of various hardware, network and software components, a bottom-up approach can be a valuable way to develop the system. In order to realize the system modeling and simulation, these steps can be applied:

First, the system modeling can start with the physical model of the system where the hardware has interaction with mechanical or physical components. For example, a Cyber-physical system (CPS) with a servo motor component used for lifting weight can be modelled by physical rules to be used later in the implementation phase. Second, the development of a setup model for the configuration of sensors and actuators with micro-controllers or development boards. This can be realized in different hardware modeling environments depending on the level of modeling, such as Proteus. Third, the WSN program and topology can be simulated in application-specific or general-purpose WSN or network simulators. Then, the software can be modelled using UML diagrams such as Class Diagram, Object diagram and Interaction Diagram to model the structural and behavioural aspects of the system software. These diagrams are useful to model software components and the hierarchy of the WSN-based IoT systems. Domain-specific modelling languages are preferable technologies to generate code for target systems. In case of using DSMLs in the development of WSN-based IoT systems, they can automatically synthesize code artefacts for the target system.

Implementation: In this phase, hardware setup is realized using sensors and actuators. Networking setup is configured to establish a network both horizontally (between edges, gateways and clouds) and vertically (between edge, gateway and cloud). Implementation of embedded software to program WSN Motes and IoT nodes is realized using domain-specific embedded programming languages such as Embedded C, nesC, Embedded Java and microPython. Gateway program is developed using web based or socket-based applications such as NodeJS and Perl.

System Test: Hardware, network, and software components can be tested separately and integration test can be done via test cases.

3. REFERENCE IMPLEMENTATION

This section will describe the important components of our reference implementation: edge level operating systems, edge level devices, the fog level, and the cloud level. Information will be provided on different solutions for each component such that an implementer can efficiently select the one best suited for their task.

3.1 Target Development Platforms

In order to implement WSN-based IoT systems, Table 1 is provided as a summary of proposed platforms and their supported devices. As mentioned before, these systems consist of WSN, IoT, Gateway and Log Manager components. It describes platforms that can be used to develop these applications, the target paradigm of these platforms (IoT and/or WSN), the language they support, the model of the operating system, supported devices, simulation support, and the on-board sensors of their supported development devices.

Overall, the WSN part can be implemented using either TinyOS or ContikiOS. These operating systems are one of the most preferred technologies to create distributed and ad-hoc wireless sensor networks. They use common hardware devices so developed software can be deployed using the same hardware technology. In this way, the developer does not have to deal with emerged heterogeneity because of hardware complexity. Both operating systems have simulation support.

Target devices have onboard sensors to help developers by abstracting away from hardware connections. They have an event-driven operating system model which works using event definitions such as network event when a message packet is received or generation of timer event when the duration of a timer is expired.

On the WSN side, Contiki operating system is programmed using C language while TinyOS applications are developed using nesC language, a dialect of C. On the IoT side, RIOT and Arduino IDE are some of the commonly preferable development environments. They are suitable for real-time applications. To eliminate hardware heterogeneity, we also present common supported hardware for RIOT and Arduino IDE. Both firmware applications can be developed using C/C++. However, they do not have specific simulation support currently.

Gateway programs can be implemented using general-purpose programming languages. To this end, serial port communication and socket programming libraries are required. According to project requirements, API's can also be added. For example, PushBullet API can be added into a Java program that runs in a RaspberryPi. Then, according to incoming data, notifications can be sent to the user via this API.

A log management system is required at the cloud level to visualize data, handle data-based events, and store data. This cloud level software can be a custom log manager implemented using a server side programming language, such as NodeJS. Alternatively, Node-RED, ThingSpeak, Google Cloud, Azure IoT and AWS IoT can also be preferred. From our point of view, Node-RED comes forward among them as a free tool to visualize data, customize/define events and establish network operations.

The operating system is important for the requirements of the tasks. Generally, it depends on the environmental conditions. Since environmental changes occur based on events and periods between events, a real-time model cannot be preferred where these events occur rarely. Moreover, when hardware capabilities and environmental changes are considered, it is not convenient to use battery-powered devices using a real-time operating system since real-time operation consumes battery. On the other hand, an event-driven operating system processes tasks when there is an event. Then, once that event is completed, it stops the CPU and goes to sleep mode to save energy.

Table 1. Summary table of proposed architecture's components.

Platform	IoT	WSN	Language	OS Model	Supported Devices	Simulation Support	On Board Sensors
RIOT	X	X	C	Real-time	ESP32, ESP8266, ARM	-	Specific Verion
Arduino IDE	X		C/C++	Real-time	ESP32, ESP8266, ARM	-	Specific Verion
ContikiOS		X	C	Event-driven	TmoteSky, IRIS, Z1	CooJA	Yes
TinyOS		X	nesC	Event-driven	TmoteSky, IRIS, Z1	ToSSiM	Yes
Gateway	X	X	Java, Python	Real-time	RaspberryPi	Not Required	Not Required
Log Manager	X	X	NodeJS, Perl	Real-time & Event-driven	PC, RaspberryPi	Not Required	Not Required

As proposed in section 2, WSN creates distributed and multi-hop topology of the architecture using event-based models. At the same time, the IoT side provides single-hop topology to support real-time activities. In case of multiple arbitrary or parallel requests sent to an IoT device, a real-time operating system should collect and handle these requests.

3.2 Edge Level Operating Systems

As mentioned, WSN and IoT have received immense attention in the research community that leads to the emergence of various operating systems and development environments. Because of the variety of operating systems and embedded technologies, development boards are also gaining popularity, creating extra decision challenges for the developers. This subsection discusses edge-level operating systems, and in Section 3.3, supported and practised devices are given.

WSN and IoT are highly dynamic networks, and their embedded devices are mostly battery powered. Those nodes may become deactive quickly because of frequent antenna usage. Moreover, these devices are equipped with constrained resources such as memory and computation power. In many cases, it is impossible to replace the sensor device after deployment. Therefore the first objective may be to optimize the sensor devices lifetime.

These challenges of WSN and IoT bring additional solutions like creating a specialized OS. According to the needs, various operating systems are designed for these devices. Therefore, operating systems for WSN and IoT have been studied to provide a reference. These features are considered: Architecture, Programming Approach, Programming Architecture, Programming Model, Threading, Scheduling, Real-time, and Memory Management.

RIOT OS

Table 2. Features of RIOT

Architecture	Microkernel
Programming Approach	Thread-base
Programming Architecture	Event/Threads
Programming Model	Multithreaded
Threading	POSIX-like API
Scheduling	Tickless, Priority-based
Real-time	Yes
Memory Management	Dynamic/Static/File System/Device Driver Protection

Table 2 describes the features of the RIOT operating system. RIOT is an open-source microkernel-based operating system designed to match the requirements of IoT and WSN devices. RIOT requires a very low memory footprint and high energy efficiency while has real-time capabilities, support for a wide range of hardware, and communication stacks for wireless and wired networks. It provides an abstract API that eliminates embedded level configurations and enables ANSI C and C++ application programming, including multi-threading, IPC, system timers and mutexes. It uses a tickless scheduler (Baccelli et al., 2013) to reduce power consumption. The system enters sleep mode by switching to idle thread whenever there are no pending tasks, and it remains in sleep mode until an interrupt wakes it up. RIOT has a modular Internet stack which is more flexible than a layered stack as modules can be separately maintained. Its communication protocols 6LoWPAN and RPL are provided with support to TCP, UDP and IPv6 protocols.

Arduino IDE

Arduino is an open-source embedded programming platform for creating IoT applications using various development boards. Recently, Arduino IDE and its libraries created a legacy to program different development boards using wrappers that consist of Arduino-like programming. By adapting its functions and libraries, it enlarged the scope of its supported boards. Arduino programming language is similar to C++, with some additional predefined functions and constants. Minimal Arduino code consists of two functions which are setup and loop. Function setup is called once when the program starts after power-up or board reset. It is usually used to initialize variables, libraries, or I/O pins. It uses freeRTOS for threading, and it has platform-independent libraries to create a network and establish a connection. It has various communication protocol libraries such as TCP, UDP, CoAP, MQTT etc., including communication stacks such as lwIP and uIP. Table 3 shows the features of the Arduino development environment.

Table 3. Features of Arduino.

Architecture	Microkernel
Programming Approach	Setup/Loop functions, FreeRTOS
Programming Architecture	Task and Function Creation
Programming Model	Functional and/or Multithreaded
Threading	POSIX-like API
Scheduling	Tickless, Priority-based, Round-Robin
Real-time	Yes
Memory Management	Memory Coalescence

ContikiOS

Contiki (Dunkels et al., 2004) is an operating system that implements the Protothreads threading (Dunkels et al., 2006) model that supports both event-driven and multi-threading. Protothreads provide lightweight and stack-less multi-threading to save ROM and RAM requirements. Multiple threads share a common stack. There is no explicit mechanism in Contiki to switch to deep sleep mode. The applications in execution are self-responsible to save battery power by observing event-queue size. When there are no events scheduled on the queue, the processor can go to sleep mode until an interrupt caused by an event wakes it up.

ContikiOS supports dynamic memory allocation, but there is not a Memory Protection Unit (MPU). For establishing network and M2M communication, ContikiOS has microIP (uIP) (Dunkels, 2003) and Rime stacks (Dunkels et al., 2007). Rime stack contains a set of custom lightweight protocols and application layer functions, while uIP stack is also a lightweight stack that supports IPv6. Specifically, uIP stack is developed for low-power and wireless memory-constrained sensor devices. Table 4 summarizes the features of the Contiki operating system.

Table 4. Features of ContikiOS.

Architecture	Modular
Programming Approach	Thread-based
Programming Architecture	Event/Threads
Programming Model	Events and Protothreads
Threading	Protothreads, Multi-Threads (Alternatively)
Scheduling	Earliest Deadline First
Real-time	Partially with Protothreads, but no guarantee
Memory Management	No Support.

TinyOS

Table 5. Features of TinyOS.

Architecture	Monolithic
Programming Approach	Component-based
Programming Architecture	Commands/Events/Tasks
Programming Model	Event-Driven
Threading	TOS Threads
Scheduling	Non-preemptive, Earliest Deadline First
Real-time	No Support.
Memory Management	Memory Recovery, Pointer and Array Error Tracking

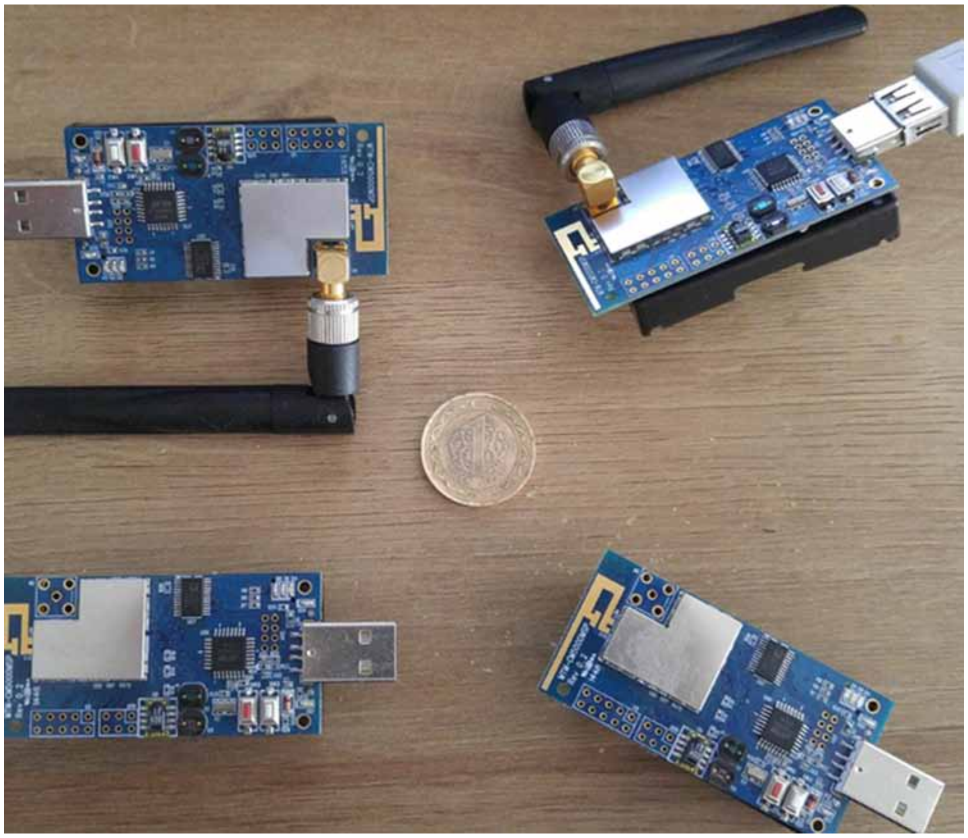
TinyOS (Levis et al., 2005) is a component-based operating system where components stick together to form statically linked programs. Specifically, these components are software modules and wrappers around hardware. It has non-preemptive tasks management and preemptive events mechanisms. Tasks cannot preempt each other, but events can preempt tasks and other events. Events represent hardware interrupts. TinyOS has cooperative TOS threads. In other words, the user is responsible for yielding the CPU explicitly when it is not in use. Task scheduler executes as a high priority thread and schedules threads using FIFO preemptive scheduling model. TinyOS is written in nesC, a dialect of C (Gay et al., 2003). nesC is a component-based and event-driven programming language to develop WSN-based applications. TinyOS has a low-power Internet Stack that consists of TCP, UDP, ICMPv6 and IPv6, including support for 6LoWPAN, RPL and CoAP. Features of TinyOS are shown in Table 5.

3.3 Edge Level Devices

In this section, edge level components that are used with respect to the proposed architecture are mentioned. Although both WSN and IoT sides have various Edge level nodes, we have investigated two pieces of hardware that are commonly preferred and low-cost. Moreover, we also test these devices considering power consumption, support level (partial or full), and availability on the market. We concluded that TmoteSky motes are suitable for implementing the WSN side while ESP32's can be preferred to develop IoT applications.

Tmote-Sky Mote

Figure 3. Tmotesky Mote.

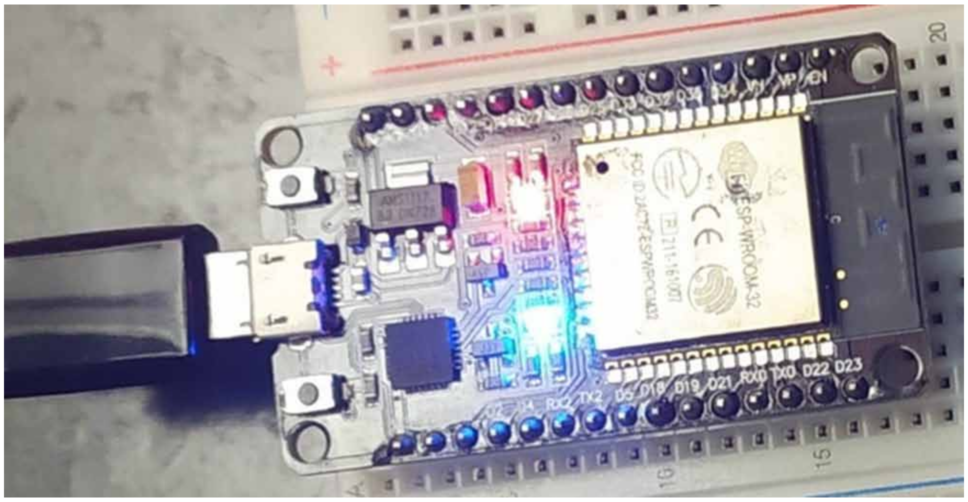


TmoteSky is a low power wireless sensor device. It offers a high data rate sensor network for applications requiring ultra-low power. It also provides high reliability and ease of development. It has multiple integrated peripherals including a 12-bit ADC, 12-bit DAC, Timer, and UART bus protocols. It has an antenna chip that is IEEE 802.115.4 compliant and sensors for measurements such as humidity, temperature, and light. Figure 3 displays the TmoteSky motes used to implement the smart fire detection and the smart lighting system in our previous work (Karaduman et al., 2018a; Karaduman et al., 2018b; Karaduman et al., 2020b).

ESP32-WROOM-32

ESP32-WROOM-32 is a 32-bit, low-cost and dual-core microcontroller, seen in Figure 4. It has integrated Wi-Fi and Bluetooth modules that target a wide variety of applications ranging from low-power sensor networks to the most demanding tasks. It supports 802.11 b/g/n Wi-Fi connectivity with speeds up to 150 Mbps. It has 520 KB of SRAM, 448 KB of ROM and 16 KB of RTC SRAM. In addition, it is equipped with 34 Programmable GPIOs. 18 GPIO's can operate as 12-bit ADC and two of them can work as 8-bit DAC.

Figure 4. ESP32 Node.



3.4 Fog Level System

The Fog layer resides on top of the edge layer and forms the network backbone between Edge and Cloud. In the Fog layer, we have placed a gateway to creating a bridge between the cloud and edge layers. The gateway plays an intermediate role in the architecture. Its basic function is moving data between Edge nodes and Cloud devices. Since they are more advanced high-end computational devices, they are usually powered by stable power sources and can also perform data processing.

WSN devices can be connected to the Internet using a single gateway. As mentioned, each source node samples environmental data and sends it to the sink node. Then, the sink node aggregates distributed data and delivers it to the gateway. The sink node is connected to the gateway physically via USB and the gateway is connected to the Internet using either Wi-Fi or Ethernet cable. Therefore, gateway devices have to have embedded features such as USB, Wi-Fi, Ethernet port, and advanced operating systems like Linux. We refer to RaspberryPi 3. It uses the Raspbian operating system, including the aforementioned hardware components.

Gateway

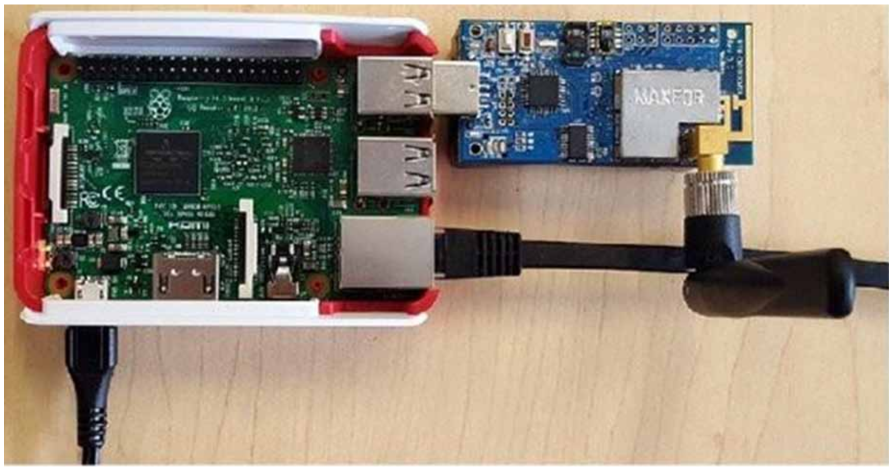
A gateway is a bridge that connects between the Internet network and the wireless sensor network. To this end, it should have enough capabilities to combine two different networks. Customization of the gateway depends on the system requirements, e.g., network protocols and data parsing, and hardware features of the components such as port types and power requirements. Moreover, hardware selection for the gateway is important because it should handle requests that come from nearly 50-60 WSN nodes and deliver them to the cloud level without any significant delay. The operating system of the gateway level should be able to run applications that are written for high-level and object-oriented languages such as Java, Python and C/C++. Therefore, data operations can also be applied at the fog level.

Fog Device: RaspberryPi 3

The RaspberryPi 3 seen in Figure 5 is the successor of the RaspberryPi. It has a BCM2837 microprocessor with a 1.2 GHz 32/64-bit quad-core processor. In addition, the Broadcom BCM43438 microchip has been added that provides the RPi 3 with a Wi-Fi 802.11n 2.4 GHz and a Bluetooth 4.1 Low Energy (BLE) connection. It has a camera interface, Ethernet port and four USB ports. It can be booted from USB or microSD. It uses Rasbpian operating system. It is a special Linux/Debian distribution. It has 40-pin GPIO pins which can be used for PWM, digital or analogue input/output. It has an HDMI port that can be connected to a monitor to be visually controlled by an operator.

Raspbian operating system lets a RaspberryPi be programmed using Python, Java, and C/C++. It can also run web-based applications and it is possible to create a web server. Compared to the edge level devices, RaspberryPi has better capabilities but higher power consumption since it requires 5V and 1A to operate. Therefore these kinds of devices have to be supplied by a limitless power source instead of battery-powered approaches.

Figure 5. RaspberryPi 3 with the TmoteSky sink mote (Kamgueu et al., 2017).



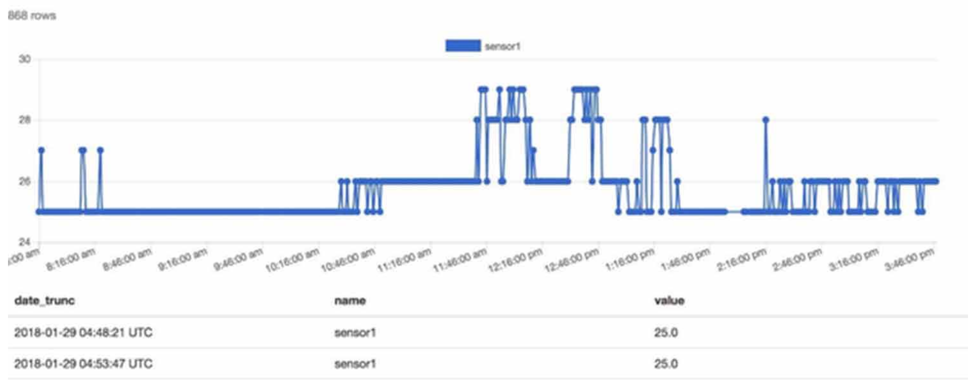
In the proposed architecture, we used RaspberryPi 3 to create a gateway between the edge and cloud. We used a USB port to attach the sink node. The USB of the RaspberryPi 3 also feeds the sink node, which means the sink node is plugged into a limitless power source. Then, a serial port reader application is written using Java to receive aggregated data from the sink node. We programmed the gateway in the manner that once it receives data, it immediately sends it to the cloud with the information of source nodes' IP and measurement value. Figure 5 represents a RaspberryPi 3 (left side) gateway with TmoteSky (right side) mote connected as a sink node. To connect those two hardware USB port is preferred. Tmotesky is roled as the sink node and deliveres incoming information to the gateway.

Cloud Level: Log Manager

Generally, IoT Systems require cloud level logging systems. In order to visualize the collected data, the Log Manager system should reflect this information graphically. This feature enables the user to track the changes in the environment and to analyse the system statistically. Figure 6 represents environmental data that are collected from a sensor. This figure shows 17 samples taken from moisture sensor. Values represent decimal value ranging between 0-255. Sampling time starts from 8:16 am (morning) and continues until 3:46 pm (afternoon). The figures can be interpreted that the moisture level did not change considerably for 7 hours, maybe due to a rainy or cloudy day.

Since WSN systems are written in an event-based application manner, these events should be handled by a log manager and turned into action by IoT nodes. A notification is sent to the log manager by an IoT node (e.g. irrigation valve is opened). Then the log manager should send a callback to the IoT node automatically to close the valve. The same behavior can also be applied for e-mailing aggregated data or sending notifications to users' mobile phones. Figure 7 illustrates a Node-RED diagram that visualizes moisture level and provides an interface to set threshold. In the Node-RED model shown in this figure, the threshold can be set by the user. It is set to decimal value of 47. Periodically, system receives sensor values. Model elements describe capabilities of Node-RED. Logic flow is established using model elements that are colored differently.

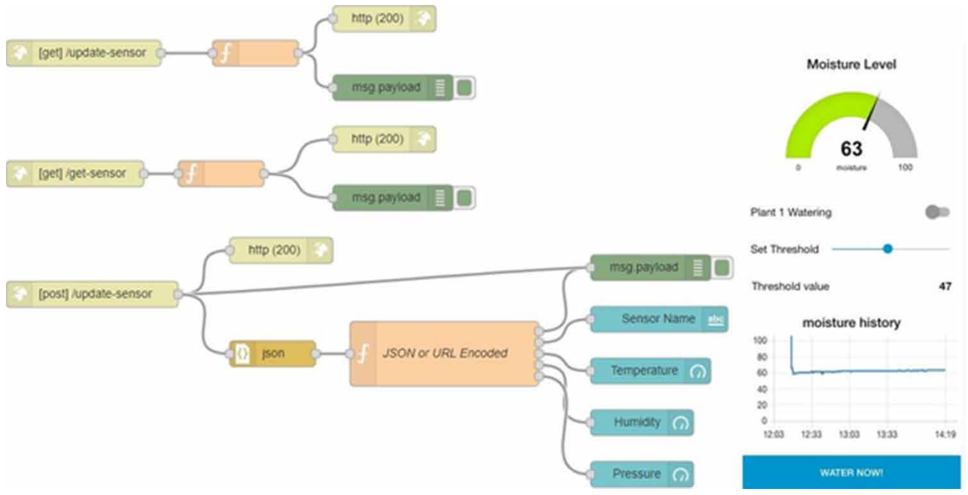
Figure 6. Measured sensor values are represented by a graph.



date_trunc	name	value
2018-01-29 04:48:21 UTC	sensor1	25.0
2018-01-29 04:53:47 UTC	sensor1	25.0

Moreover, Node-RED can also be connected to a database to log moisture data. It can send GET and SET requests to nodes for collecting moisture data or updating threshold values. Further, different sensor types can also be used, such as temperature, pressure and humidity. It has a future to create user-defined functions. In this way, users can customize functionalities of the Node-RED, and users can also add JSON elements to parse JSON coded information.

Figure 7. Node-RED diagram logs and visualizes data and sends requests when a certain threshold is exceeded.



4. IMPLEMENTATION WORKFLOW FOR A CASE STUDY

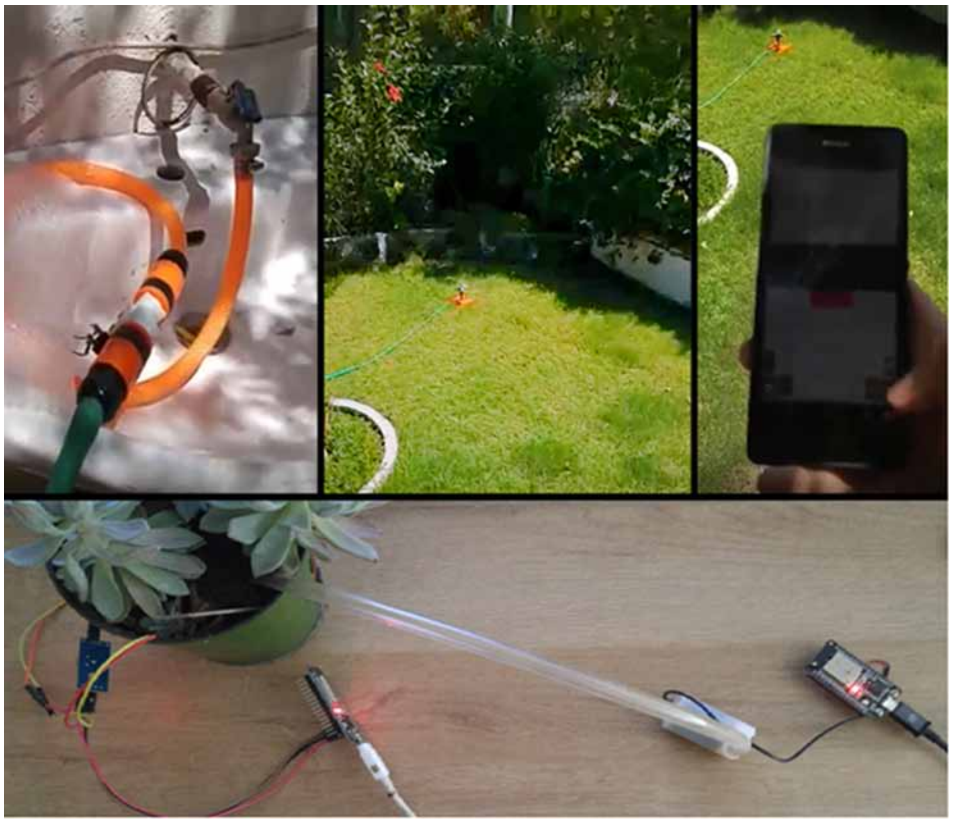
This section will present the workflow for the implementation of our architecture for the irrigation case study.

4.1 Case Study Implementation

The irrigation case study is implemented based on the reference architecture and components. It is represented by Figure 8. The system is firstly tested in a wide garden then adapted to a large area. Since farm fields are large areas where it is hard to establish an Internet connection or is not possible due to green farming, lack of power source and lack of infrastructure. Therefore, distributed WSN motes should create a network between each other using IEEE 802.15.4 technology. Moreover, they should be battery-powered and have a deep-sleep operation. Therefore, proposed operating systems can automatically switch to a deep-sleep mode where there is no pending task or event.

We distributed TmoteSky motes to these fields and measured the moisture level of the soil. Because it is not an area that receives sunlight for a long time in a day, it was not possible to use solar batteries. Therefore, we used standard 2xAA batteries for each WSN mote. When a sensor node has detected dryness, it sends its IP address and measurement data to the neighbour node through the sink node. The gateway reads this data from the sink node via a USB port and has delivered it to the cloud. Lastly, the cloud level logs this data, including date and time, then sends a request to an ESP32 IoT node (IEEE 802.11) that controls a solenoid valve. We selected the TmoteSky node since it has built-in solutions such as in-board sensors, battery packs, and it is a well-supported device by Contiki operating system and its simulation CooJa. Moreover, we selected ESP32 since it has well-designed analogue I/O pins and a high-frequency CPU. Therefore, it can easily control a solenoid valve that draws many currents and requires a fast and smooth operation to arrange its water consumption. We selected the RIOT operating system because we wanted to use a real-time operating system that controls more than one solenoid valve attached to one ESP32 mote. Moreover, the software complexity which is derived from hardware connections is abstracted by the RIOT operating system. We used its threads and assigned one thread to each I/O and solenoid valve. Each solenoid valve is used to irrigate a part of a field.

Figure 8. Components of the irrigation case study (solenoid valve, springer, mobile control and prototype).

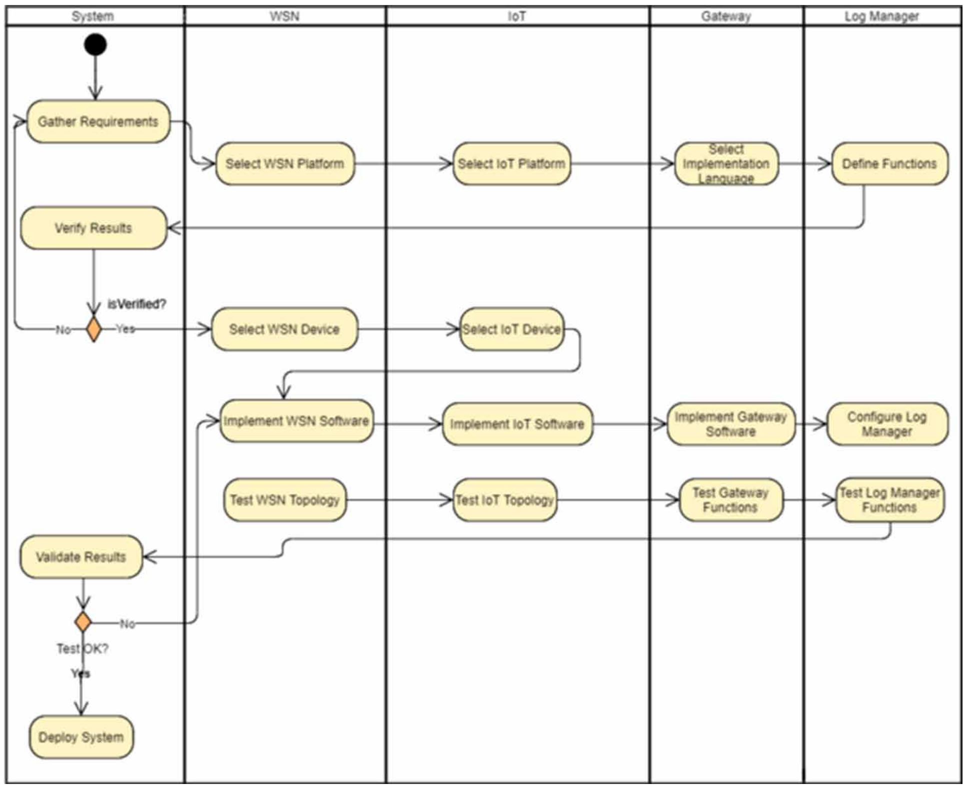


As a gateway, we preferred a RaspberryPi since we can feed four sink nodes using its USB ports and even read incoming data from these USB ports and transmit them to the cloud simultaneously. As a log management system, we preferred to use Node-RED because it has a graphical and user-friendly interface/data visualization.

4.2 Workflow

Figure 9 illustrates the workflow of the irrigation case study. The columns list the different components within the architecture, while the steps within the workflow specify how we implemented each component. Broadly, there are four phases to our workflow. Requirements are about gathering the requirements and deciding on the specification of each component. Then, implementation is about implementing and configuring the component software. Testing concerns the unit and integration testing of each component, and then finally, deployment deploys the components to the field.

Figure 9. Workflow of the irrigation case study using proposed architecture.



5. CONCLUSION AND FUTURE WORK

In this chapter, an architecture and reference implementation are proposed to eliminate unpredictable technical challenges that system designers will encounter in the development of WSN-based IoT systems. Moreover, this chapter discusses a gateway and log manager, which are architectural requirements for creating WSN-based IoT systems. Considering the device and operating system variety, we shared our experience to provide a reference architecture and implementation. In this way, designers can use the provided information to eliminate the decision problem and immediately begin developing their systems. An irrigation system is given as a case study. There is sufficient complexity since it has many components interacting with each other to control the moisture level of farm fields. Operating systems handle these components' power consumption while aggregated data is delivered to the Log manager via a gateway.

This architecture and reference implementation are helpful for designers who want to benefit from distributed data aggregation using WSN motes while empowering the system IoT nodes to control actuator components. WSN motes can be distributed along with the large fields and IoT nodes can actuate electronic components to influence the environment.

However, because of device constraints on the WSN side, they are not suitable for real-time applications. Moreover, there is no simulator for IoT side OS and platforms to simulate the WSN-based IoT and application software. However, the RIOT operating system follows a hybrid model to support WSN and IoT devices. In this way, it may be possible to create distributed topology using power-efficient IoT nodes since embedded technology is evolving.

As a recent technology, LoRa is a popular antenna technology for IoT, it brings long-range and low-power solutions for connectivity problems. Here, we discuss LoRa as a new technology with which the proposed approach can be implementation as a future work. For application areas such as agriculture, home automation, and smart cities, LoRa has some convenient features. Although, in military applications the transmission range has to be short due to safety concerns and a short-range mesh-topology is preferred. Therefore, preference of the technology depends on the application domain and task to be done. Moreover, LoRa is compatible with IoT-based hardware such as Arduino, RaspberryPi and STM boards. Although LoRa is a low-power transportation and antenna layer solution, the dependency on traditional development boards makes LoRa's advantageous features less visible. Today's IoT boards' power consumption is not very efficient for battery-powered applications. They considerably become efficient if they operate in deep-sleep mode by sacrificing real-time features and short sampling periods. They require an additional solution like solar energy to tackle this problem, yet it creates extra cost. However, as technology evolves, micro-processors will be smaller and more power-efficient. RIOT operating system has support for LoRa-based IoT applications. In this regard, RIOT may become a dominating hybrid operating system to implement various development boards that use various technologies such as IEEE 802.15.4, IEEE 802.11, LoRa and Bluetooth Low Energy.

In the future, Log Management approaches can be compared to an increased variety of selections in this respect. Moreover, multiple case studies can be implemented using the proposed architecture to generalize the results. In addition, model-driven approaches can also be applied considering the proposed architecture to provide code generation using model-based design (Marah, et al., 2018). To this end, a domain-specific modelling language can be developed to provide a platform-independent modelling environment for WSN-based IoT systems. The user can design a model representing any target WSN-based IoT system and select a specific platform to transform the platform-independent model to a platform-specific model. The user can add platform-specific elements to the model and automatically generate codes for TinyOS, Arduino, Contiki, RIOT OS and/or Java platforms. In addition, an agent-oriented paradigm can be applied for programming WSN-based IoT systems (Tezel, et al., 2016). As software agents provide a higher-level of abstraction to develop software, they can be used to ease the control of distributed WSN topology, as agents are collaborative software entities. Agents can be deployed at any level of the provided architecture to solve complex problems.

REFERENCES

Akyildiz, I. F., Su, W., Sankarasubramaniam, Y., & Cayirci, E. (2002). Wireless sensor networks: A survey. *Computer Networks*, *38*(4), 393–422. doi:10.1016/S1389-1286(01)00302-4

Arslan, S., Challenger, M., & Dagdeviren, O. (2017). Wireless sensor network based fire detection system for libraries. In *2017 International Conference on Computer Science and Engineering (UBMK 2017)*, (pp. 271-276). IEEE 10.1109/UBMK.2017.8093388

Asici, T. Z., Karaduman, B., Eslampanah, R., Challenger, M., Denil, J., & Vangheluwe, H. (2019). Applying model driven engineering techniques to the development of contiki-based IoT systems. In *2019 IEEE/ACM 1st International Workshop on Software Engineering Research & Practices for the Internet of Things (SERP4IoT)* (pp. 25-32). IEEE. 10.1109/SERP4IoT.2019.00012

Baccelli, E., Hahm, O., Günes, M., Wählisch, M., & Schmidt, T. C. (2013). RIOT OS: Towards an OS for the Internet of Things. In 2013 IEEE conference on computer communications workshops (INFOCOM WKSHPS) (pp. 79-80). IEEE.

Chéour, R., Khriji, S., & Kanoun, O. (2020). Microcontrollers for IoT: Optimizations, Computing Paradigms, and Future Directions. In *2020 IEEE 6th World Forum on Internet of Things (WF-IoT)* (pp. 1-7). IEEE.

Dunkels, A. (2003, May). Full TCP/IP for 8-bit architectures. In *Proceedings of the 1st international conference on Mobile systems, applications and services* (pp. 85-98). Academic Press.

Dunkels, A. (2007). Rime-a lightweight layered communication stack for sensor networks. In *Proceedings of the European Conference on Wireless Sensor Networks (EWSN), Poster/Demo session, Delft, The Netherlands* (Vol. 44). Academic Press.

Dunkels, A., Gronvall, B., & Voigt, T. (2004). Contiki-a lightweight and flexible operating system for tiny networked sensors. In *29th annual IEEE international conference on local computer networks* (pp. 455-462). IEEE. 10.1109/LCN.2004.38

Dunkels, A., Schmidt, O., Voigt, T., & Ali, M. (2006). Protothreads: Simplifying event-driven programming of memory-constrained embedded systems. In *Proceedings of the 4th international conference on Embedded networked sensor systems* (pp. 29-42). 10.1145/1182807.1182811

Gay, D., Levis, P., Von Behren, R., Welsh, M., Brewer, E., & Culler, D. (2003). The nesC language: A holistic approach to networked embedded systems. *ACM SIGPLAN Notices*, *38*(5), 1–11. doi:10.1145/780822.781133

Holt, J., & Perry, S. (2008). *SysML for systems engineering* (Vol. 7). IET. doi:10.1049/PBPC007E

Kamgueu, P. O., Nataf, E., & Djotio, T. (2017). Architecture for an efficient integration of wireless sensor networks to the Internet through Internet of Things gateways. *International Journal of Distributed Sensor Networks*, *13*(11), 1550147717744735. doi:10.1177/1550147717744735

Karaduman, B., Aşıcı, T., Challenger, M., & Eslampanah, R. (2018). A cloud and Contiki based fire detection system using multi-hop wireless sensor networks. In *Proceedings of the Fourth International Conference on Engineering & MIS 2018* (pp. 1-5). 10.1145/3234698.3234764

Karaduman, B., Challenger, M., & Eslampanah, R. (2018). ContikiOS based library fire detection system. In *2018 5th International Conference on Electrical and Electronic Engineering (ICEEE)* (pp. 247-251). IEEE. 10.1109/ICEEE2.2018.8391340

Karaduman, B., Challenger, M., Eslampanah, R., Denil, J., & Vangheluwe, H. (2020). Analyzing WSN-based IoT Systems using MDE Techniques and Petri-net Models. Academic Press.

Karaduman, B., Challenger, M., Eslampanah, R., Denil, J., & Vangheluwe, H. (2020). Platform-specific Modeling for RIOT based IoT Systems. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops* (pp. 639-646). 10.1145/3387940.3392194

Karimpour, N., Karaduman, B., Ural, A., Challenger, M., & Dagdeviren, O. (2019). IoT based Hand Hygiene Compliance Monitoring. In *2019 International Symposium on Networks, Computers and Communications (ISNCC)*, (pp. 1-6). IEEE.

Klein, S. (2017). *IoT Solutions in Microsoft's Azure IoT Suite*. Apress. doi:10.1007/978-1-4842-2143-3

Korobeinikova, T. I., Volkova, N. P., Kozhushko, S. P., Holub, D. O., Zinukova, N. V., Kozhushkina, T. L., & Vakarchuk, S. B. (2020). Google cloud services as a way to enhance learning and teaching at university. *Proceedings of the 7th Workshop on Cloud Technologies in Education (CTE 2019).*

Kurniawan, A. (2018). *Learning AWS IoT: Effectively manage connected devices on the AWS cloud using services such as AWS Greengrass, AWS button, predictive analytics and machine learning*. Packt Publishing Ltd.

Levis, P., Madden, S., Polastre, J., Szewczyk, R., Whitehouse, K., Woo, A., & Culler, D. (2005). TinyOS: An operating system for sensor networks. In *Ambient intelligence* (pp. 115–148). Springer. doi:10.1007/3-540-27139-2_7

Marah, H. M., Challenger, M., & Kardas, G. (2020). RE4TinyOS: A Reverse Engineering Methodology for the MDE of TinyOS Applications. In *2020 15th Conference on Computer Science and Information Systems (FedCSIS)* (pp. 741-750). IEEE. 10.15439/2020F133

Marah, H. M., Eslampanah, R., & Challenger, M. (2018). DSML4TinyOS: Code Generation for Wireless Devices. *2nd International Workshop on Model-Driven Engineering for the Internet-of-Things (MDE4IoT), 21st International Conference on Model Driven Engineering Languages and Systems (MODELS2018).*

Maureira, M. A. G., Oldenhof, D., & Teernstra, L. (2011). ThingSpeak–an API and Web Service for the Internet of Things. *World Wide Web (Bussum).*

Sharma, N., Shamkuwar, M., & Singh, I. (2019). The history, present and future with IoT. In *Internet of Things and Big Data Analytics for Smart Generation* (pp. 27–51). Springer. doi:10.1007/978-3-030-04203-5_3

Tezel, B. T., Challenger, M., & Kardas, G. (2016). A metamodel for Jason BDI agents. In *5th Symposium on Languages, Applications and Technologies (SLATE'16)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

Toscano, E., & Bello, L. L. (2012). Comparative assessments of IEEE 802.15. 4/ZigBee and 6LoWPAN for low-power industrial WSNs in realistic scenarios. In *2012 9th IEEE International Workshop on Factory Communication Systems* (pp. 115-124). IEEE.