SvedbergOpen
DISSEMINATIONOFKNOWLEDGE

International Journal of Artificial Intelligence and Machine Learning

Publisher's Home Page: https://www.svedbergopen.com/

Research Paper Open Access

Cross-platform Web of Things framework for smart home sensors and legacy device gateways

Boobalakrishnan Sadasivam [1*], Kavitha Rajamohan[2]

[1*]Department of Computer Science, CHRIST (Deemed to be University), Bangalore, https://orcid.org/0000-0001-8171-6622

[2]Department of Computer Science, CHRIST (Deemed to be University), Bangalore, https://orcid.org/0000-0002-7803-8901

*Corresponding author: Email: boobala.s@res.christuniversity.in

Abstract

The Web of Things (WoT) aims to enable universal interoperability across a wide range of Internet of Things (IoT) platforms and devices by leveraging conventional web technologies. Still, most current WoT solutions rely on high-level programming languages like Python and JavaScript; as a result, they are generally incompatible with IoT sensors that have limited resources. To address this restriction, this paper presents WoTC++, a native, lightweight, cross-platform, modular WoT stack built in C++. The proposed system consists of a portable WoT core, libraries for binding to protocol interfaces, and device drivers, enabling real-time interactions on resource-constrained devices. The framework is built on the W3C specifications and Internet protocols and can be integrated into any Ethernet or Wi-Fi-enabled platform. The main contribution of the suggested system is a cross-platform, configurable WoT core compatible with all platforms and providing flexible integration with platform network drivers. The suggested framework has been evaluated with an ESP32 IoT sensor and a gateway for the Zigbee legacy protocol running on a Raspberry Pi. Compared with alternative runtimes, the suggested framework shows encouraging performance across all workloads in benchmarking tests against other interpreted-language implementations. Overall, the findings support the scalability, real-time capabilities, effective protocol bindings (HTTP, CoAP), and cross-device interoperability of the suggested framework in limited environments. The foundation for a future Web of Things is laid by this gateway design, which provides a smooth interface to legacy protocols such as Thread, Z-Wave, and Zigbee.

1. Introduction

The WoT is a paradigmatic extension of the Internet of Things (IoT). Web of Things (WoT) uses standard web technologies such as Hypertext Transfer Protocol (HTTP), RESTful APIs, and WebSocket to communicate at their network level [1]. The IoT market is quickly increasing at a fast pace with the ability of devices to connect, communicate and interact with one another and deliver intelligent services in different applications like a healthcare, smart home, environment monitoring, and industrial automation. [2]. Smart Home a IoT realization and that is fragmented across protocols, devices, platforms and applications. There are no universal standards yet. With the increasing deployment of connected smart-home devices, managing these heterogeneous devices with different wireless standards and protocols has become a daunting task. Moreover, many of these devices operate under severe resource constraints, including limited memory, energy availability, and computing capability [3].

The obvious advantages of WoT in streamlining IoT ecosystems are not enough to overcome the enormous hurdle of bringing WoT capabilities to devices with limited resources [4]. Smart home devices range from battery-powered sensors to always-on gateways or edge controllers, making it difficult to apply WoT principles. In particular, smart home IoT devicesand sensor nodes, which have limited processing power, are not well-suited to current WoT implementations because they rely on middleware or high-level programming languages that are computationally and memory-intensive[5].

Recently, WebAssembly (Wasm) has been offering support on resource-constrained devices, aiming for near-native execution speed, cross-platform support, and emerging as a standard for developing cross-platform web services, and is designated as the fourth language for web development along with HTML, CSS, and JavaScript. Existing WoT solutions based on Wasm[6][7]are an interesting approachfor resource-constrained devices. However, it has complex workflows, runtime overhead, and no direct hardware access, which are bottlenecks in resource-constrained platform development.C and C++ have the best execution speed, hardware control, and no runtime requirements, and produce target-platform machine code via cross compilation. These are still the preferred languages for embedded systems where resources and costs are limited.

Juan Domingo et al. [8] suggested the EmintWeb for creating embedded web applications in C++ for specific systems. Embedded web applications are created in this work. They are one-off, generated executable programs that include a web server. A compiled language (C++) is used to code these apps. They dynamically build the HTML5 code for each page at runtime and start a new session thread to establish an optionally encrypted connection with the client. Among the many benefits of this method is that it simultaneously strengthens client and server security. The executable is the only file on the server that can be modified. Eliminating client-side code execution and the need for cookies is possible. Compiling code improves performance compared to interpreted languages, which can slow down application performance.

Philip Branch and Phillip Weinstock [9] recommended the Employment of a LoRa-MQTT Gateway Written in Elixir and C++ for functional programming in the IoT. This article details the experiences of creating a bridge between LoRa networks and MQTT brokers in C++ and the more traditional functional language, Elixir. Using Elixir to understand this development technique, the author built the prototypes on single-board computers. The author replicated the Process in C++ and compared the two systems' functionality via experiments. The author built the same system in C++ and Elixir on a cheap ESP32 microcontroller so that readers can comprehend how these systems work on low-end IoT devices. The author discovered that the ESP32 microcontroller could run the Elixir-based system yet performed far worse than the same system built in C++.

On the other hand, C++ implementation was more efficient in speed, memory usage and overall responsiveness. These results highlight that C++, as a natively compiled language, is better suited for embedded systems due to its low-level hardware access, optimized resource management, and widespread compatibility. Therefore, C++ remains the preferred choice for building lightweight, high-performance IoT frameworks such as WoTC++, ensuring seamless integration and reliable operation in the WoT ecosystem.

2. Motivation

With the emergence of WoT, physical objects can be connected to the online environment in a standardized, interoperable way, enabling automation, remote monitoring, and intelligent decision-making [10]. Hence, it is important to apply the WoT architecture from battery-powered devices to the central controller of a smart home to achieve scalable, interoperable smart home systems. Currently, most WoT implementations use JavaScript or other high-level interpreted languages,which can be used only on unconstrained platforms or necessitate complex workflows[6, 7] to support constrained platforms.Though some runtimes are available in the open community, these languages and their implementations place a heavy memory and processing burden on low-power processors, which are often used in smart home and IoT applications, such as the ESP8266 and the ESP32 [11], where transactions are limited, and the devices are low-power.

3. Problem Statement

Existing WoT implementations are not well-suited for direct deployment on embedded microcontrollers or devices with limited resources, even though WoT can provide easy, portable access to IoT devices[12][13]. In addition, complex device-specific or aggregated gateway solutions are sometimes required to provide WoT access to IoT devices, since traditional frameworks lack native execution capabilities optimized for resource-constrained IoTsystems [14].An overview of a traditional WoT system is shown in Figure 1. Scalability and deployment simplicity across multiple embedded systems are further hindered by the complexity and lack of portability of current WoT development environments [15]. Addressing these challenges requires a

lightweight, natively implemented WoT framework that delivers efficient, standards-compliant, and portable integration for resource-constrained IoT devices [8].

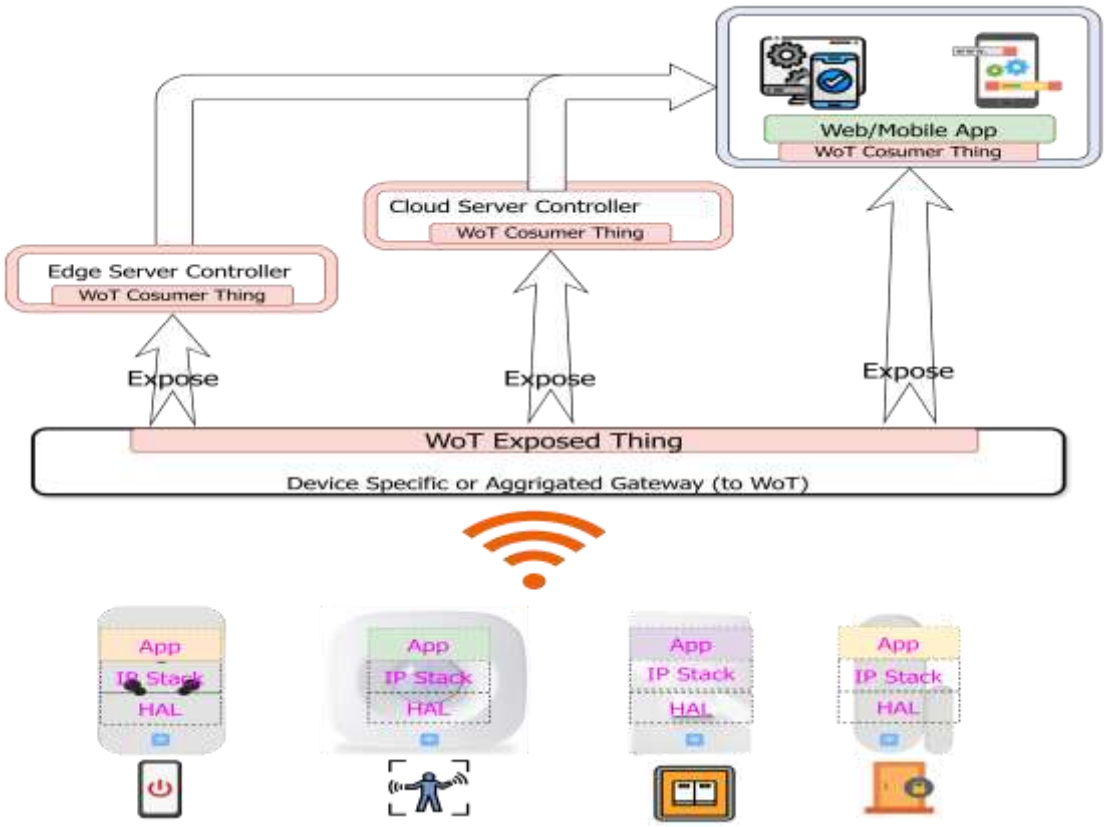


Figure 1:Overview of a traditional WoT system

The key contributions of this work are as follows:

- A native C++ WoTcorethat can be tailored for resource-constrained devices or high-end systems, enabling developers to build WoT-enabled applications with low overhead and high performance.
- Flexible integration with platform-dependent network drivers through well-defined APIs
- Evaluation on a resource-constrained ESP32 device and Raspberry Pi with a corresponding real-time application of sensor and gateway.
- Quantitative comparison with existing solutions and comprehensive benchmarking on the Raspberry Pi platform.

The remainder of the paper is organized as follows: Section 4 delves into related work, Section 5 proposes the WoT framework, Section 6 describes the implementation details, Section 7 discusses the evaluation setup, Section 8 presents the evaluation and findings, Section 9 compares with state-of-the-art frameworks, and Section 10 concludes the study.

4. Related Works

This section explains the existing work on the WoT and its limitations. The summary is listed in Table 1.

Guadalupe Ortiz et al. [16] propose a microservice model for real-time IoT data processing andareusableWoTmethod for smart ports. In this work, the author presents a reusable, standardized microservice architecture. It is designed to handle real-time data efficiently and is backed by complicated event-processing algorithms. The model is based on concepts of the WoT. As an example of the idea in action,

the author provides a smart port air quality monitoring and alerting architecture deployment with completely reusable microservices. This design has been shown to deliver outstanding performance in evaluations.

Nazmi Ekren et al. [17] discussed Smart Buildings Using WoT with .NET Core for Inter-Device Connectivity and Secure Data Transfer. To meet modern IoT demands, this research introduces a WoT-based modular framework compatible with existing protocols and devices, offering secure, adaptable, and scalable solutions. This system's hardware and communication protocols allow for the smooth integration of subsystems. The study and execution of the proposed automation system demonstrated its viability and effectiveness in handling unique and complex building environments.

The Eclipse ThingwebNode-wot is a framework[18] for developing WoT servers and clients using NodeJS. It is developed from the ground up in TypeScript, aiming to provide a rapid, adaptable platform for IoT applications. Node-wot aims to empower developers to build intricate business logic without concern for protocols or low-level details by using a standardized metadata format, the Thing Description (TD). TD abstraction enables developers to use a suite of satellite tools to rapidly and easily create their apps.

Luca Sciullo et al. [19] proposed the WoT Micro-Servient (WMS) for Resource-Constrained Edge Devices. Even with limited hardware resources, developers may use WMS to create, build, and deploy WoT applications on embedded systems and micro-controllers. Here, the author details the tool's design and features and demonstrates that it outperforms the official W3C WoT implementation, Node-wot, and a proxy-based solution in terms of latency and energy usage. At last, a practical smart home application that utilizes WMS to enable microcontrollers and mobile devices to collaborate remotely, monitoring and operating indoor plants over the IoT.

Federica Paganelli et al. [20]recommended the WoT Framework for RESTful Applications and Its Experimentation in a Smart City. The framework comprises a web resource information model, middleware, and tools for creating and sharing digital versions of smart objects over the web. This research reviews the framework's adherence to REST standards and its key implementation decisions. Lastly, the author presents results from the SmartSantander European Project, in whichthe framework is evaluated in a smart city environment to assess its effectiveness.

Dominique Guinard et al. [21] discussed a resource-oriented architecture for the WoT. The author begins by summarizing the RESTful principles that have contributed to the popularity, scalability, and flexibility of the traditional web, then delves into the WoT's design and best practices. The author then discusses several prototypes that follow these guidelines and connect web-based nodes for environmental sensors and an energy monitoring system. Finally, the author shows how to use Web-enabled smart objects through "physical mash-ups," small, ad hoc apps.

Zakaria Benomar et al. [2] presented the Stack4Things-based WoT Architecture. The WoT paradigm aims at improving application-layer interoperability by embedding smart objects in the web and establishing their interaction using the standard protocols and formats used on the Web. The essence of smart objects in applications and services is geared to be greatly simplified by web standards and protocol and other device standards, e.g. HTTP and WebSocket. To connect smart devices online, this article proposes a method that leverages the capabilities provided by the Stack4Things (S4T) OpenStack middleware. IoT devices can share their resources with user agents, such as web browsers and smart devices, using globally resolvable Uniform Resource Locators (URLs) that are independent of the protocol stacks used by these agents.

Ege Korkan et al.[22] introduced WoTify, a platform to bring the Web of Things to your devices. The IoT and other commercially available WoT gadgets, such as Philips Hue lighting, and platforms, like Azure IoT, have already seen meteoric rises in popularity. None of these devices or platforms is compatible with the World Wide Web Consortium's (W3C) newly released WoT standards; instead, each has its own method of specifying how devices communicate with one another. In contrast, there is currently no infrastructure independent of programming languages, allowing the integration of popular hardware components into the WoT, as withinstallingsoftware packages. In this article, the author presents a framework to address these issues and promote the adoption of the W3C WoT. Users may WoTify their current hardware by installing new software on it, or by describing their IoT and WoT devices using a Thing Description.

Junior Dongo et al. [23] deliberated the domOS: an "Operating System" for Smart Buildings. In this article, the author provides a definition of the domOS ecosystem, a set of rules for centralized management of energy services that allow authorized applications to access various connected home appliances and devices. Any current IoT platform can be easily built according to the standard, which utilizes legacy IoT technology. By separating the building's infrastructure plane from its application plane, a compliant IoT platform manages the building's operations. The architecture of WoT, as described by the W3C, is further elaborated upon in the domOS ecosystem standard. A Building Description (BD) electronic sign is present in compliant structures. For monitoring and controlling local energy processes, the BD is a machine-and human-readable document that includes pertinent information (such as building type, size, energy system, etc.). Energy service standardization and unification in buildings is the result of the domOS ecosystem definition.

Benedikt Ostermaier et al. [24]examineWebPlug, a framework for the Web of Things. A foundation for the soon-to-be-born IoT is introduced here with WebPlug. It is composed of many components that facilitate the incorporation of objects onto the web, including their sensors and actuators. For example, WebPlug facilitates the creation of physical mash-ups by supporting versioning and eventing for any web resource, such as sensor data. After describing the research that inspired WebPlug, the author goes over its key features. In the latter section of the article, the author demonstrates how our system works in practice.

Markel Iglesias-Urkia et al. [25]analysed the Automatic generation of WoT servient using Thing Descriptions. Achieving interoperability across the ever-increasing number of Internet-connected devices is the ultimate goal. In contrast, Model-Driven Engineering (MDE) methods aim to accelerate software development, facilitate design and code reuse, and enhance software quality by leveraging models to increase abstraction levels. The efficient creation of WoT clients is the goal of this study, which proposes the use of MDE approaches. It is possible to generate WoT servants in C++ with CoAP communication capabilities using either a model-based or a textual syntax; both are fully consistent with the WoT standard. Under an open-source license, the public may access and use a tool that implements this concept, covering the entire development process.

Borui Li et al. [6] investigated flexible, efficient WebAssembly-based WoT services for resource-constrained IoT Devices. This article proposes a Wasm-based architecture for efficient, adaptable WoT services, calledWaWoT. The flexible use of annotations and automated segmentation in WaWoT service development enables this. Additionally, it facilitates the use of WebAssembly modules for dynamic service migration, enabling the adaptation of placement between web clients and IoT devices. Also shown is an ahead-of-time compiler that utilizes streaming compilation and trimming, among other optimizations, to reduce memory utilization. To ensure energy efficiency, the writer adopts optimizations like accessing the relevant data via direct I/O and bulk writing of instructions. Through compile-time and run-time evaluation, the author offers a guarantee of sandboxed execution. As per the results, WaWoT is more flexible than existing WoT Development Methods. Also, WaWoT can reduce energy consumption by 1.9–4.9 times and RAM usage by 84.9 times compared to current WebAssembly runtimes. It allows secure and efficient provision of WoT services on resource-constrained IoT devices.

Luca Sciullo et al. [7] proposed the WoT on The Extreme Edge (WoTTEE): Enabling the W3C Web of Things for Micro-controllers. This study adds to this line of research by assessing mechanisms to upgrade the performance and interoperability on microcontrollers by making them W3C WoT compliant. The author proposes a re-engineered Web Thing (WT) architecture conforming to the WoT standard that caters according to devices' limited resources in response to the previous proposal's complexity. The WoTTEE software package supports edge-oriented WT deployment, installation, and monitoring. Lastly, the author demonstrates that WoTTEE can verify its operations in a small testbed and enable adaptive IoT systems, where microcontrollers can dynamically transition between multiple IoT network protocols using the W3C Web of Things' Thing Description (TD) mechanism.

García Mangas et al. [26]proposed a Python-based Web of Things framework that was experimentally tested and included bindings for HTTP, CoAP, WebSocket, and MQTT. Action Invocation latency, Event latency, Property read latency, and corresponding errors were measured during the evaluation, which was conducted in LAN, WLAN, proxy, and cloud scenarios. The author experimented in a variety of settings, including unconstrained, single-board computer (SBC), and constrained systems, across various scenarios.

Table 1: Summary of Existing Methods

Reference	Language / Platform	Target Environment	Resource-Constrained Suitability	Protocol Support	Key Contributions	Limitations
2010, WebPlug[24]	JavaScript	Cloud/Web	No	REST	Mash-up creation, sensor eventing	High overhead, no embedded deployment support
2019, WoTify[22]	Custom (TD-based)	Mixed	—	TD, Custom	Retrofits existing hardware	Requires additional components; integration complexity
2020, TD Generator [25]	C++ (MDE)	Embedded	Yes	CoAP	Auto-generates TD-compliant clients	Focused only on client-side; lacks runtime interaction logic
2022, SmartPortWoT[16]	Microservices	Industrial IoT	—	Event-based	Reusable real-time microservices	Gateway-focused; high-level deployment only
2022, domOS[23]	Mixed	Smart Buildings	—	HTTP, MQTT	Building-level energy orchestration	Complex architecture; not MCU-friendly
2022, WoTTEE[7]	C / MicroPython	Microcontrollers	Yes	HTTP, CoAP	Dynamic protocol switching, TD support	Early stage; limited scalability analysis
2023, Node-wot (0.8.8) [18]	Node.js/ TypeScript	Gateway, PC	No	HTTP, CoAP	W3C official stack; good tooling	High memory footprint; unsuitable for MCUs
2024, IR. WoT[27]	JS/C	Edge Devices	Yes	HTTP, CoAP	Efficient micro-WoT for embedded	Limited scalability; lacks full protocol flexibility
2024, WOTPY [26]	Python	SBCs	No	HTTP, CoAP, WebSocket, MQTT	All protocol binding implementations	Not suitable for embedded controllers, Heavy under high concurrent invocations
2025, WoT.NET [17]	.NET Core	Gateway, PC	No	HTTP, MQTT	Secure building automation	.NET Core is unsuitable for most embedded devices
2025, WaWoT[6]	WebAssembly (C/C++)	IoT Devices	Yes	HTTP	Secure, modular, Wasm-based WoT	Complex deployment pipeline; experimental tooling
2025, WoTC++ (Proposed)	Native C++	Embedded (ESP32, RPi)	Yes	HTTP, CoAP, Zigbee	Portable and customizable, Zigbee and Wi-Fi bridging	Currently limited to exposure thing, and not all binding interfaces are supported

5. Proposed Method

The WoTC++ framework offers a native C++ solution that bridges the gap between WoT architecture and smart home IoT devices with limited resources. Most contemporary WoT implementations use high-level interpreted languages such as Python or JavaScript, which are not well-suited for microcontrollers due to their limited memory and processing power. These frameworks are not viable for real-time applications in limited contexts because they consume too many resources, do not provide portable development environments, and do not directly enable interaction with legacy devices. With WoTC++, these restrictions are nullified, and the WoT Producer stack can be directly deployed on popular embedded controllers such as ESP32 and ESP8266, thanks to its native C++implementation.Also, WoTC++ is a lightweight, native C++ framework specifically designed for

seamless WoT integration on smart home embedded platforms ranging from battery-operated devices to always-on gateways and central controllers. In C++, WoTC++ implements the WoT stack natively without the overhead of interpreted languages.

This proposed framework is advantageous in two ways. At first, WoTC++ gets rid of having a device-specific or aggregated gateway for achieving WoT compatible IoT devices with constrained resources, which is illustrated by the drawings shown in Figure 2. These IoT devices are able to expose their services directly to the Web. The same framework can be scaled to higher platforms, such as gateways and central controllers, through configuration. The framework fixes compatibility issues of different implementations and maturity issues. Because it achieves minimal overhead, executes natively and works with many protocols and devices. WoTC++ intends to develop an efficient and deployable version of WoT which is scalable for IoT smart-home and real-time applications.

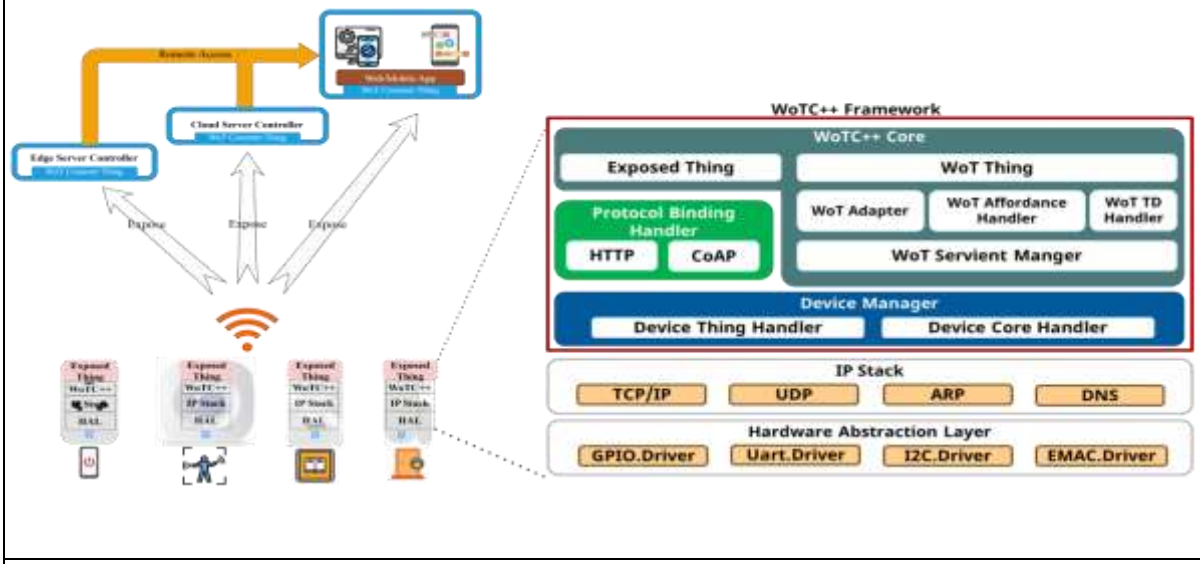


Figure 2:WoT System with WoTC++ framework

Framework Design

The need for a lightweight, portable, and real-time-compliant WoT stack optimized for both constrained and unconstrained smart home devices drove the creation of the WoTC++ framework. The proposed WoTC++ framework is a modular solution that follows the W3C WoT reference architecture[28]. By using flexible configuration options and optimized data handling to reduce RAM requirements, the WoT stack achieves a significant architectural improvement, making it portable across devices with and without limited resources. The WoTC++ framework, which maintains full compliance with the W3C model, introduces another architectural enhancement: native integration of heterogeneous network technologies, specifically Zigbee and Wi-Fi, within the device behaviour layer.In addition to an overview of the WoT system, Figure 2 provides an architectural overview of the WoTC++ modules and components. The overall framework consists of three logicalmodules, each with clearly defined responsibilities:WoTC++ Core, Protocol binding Handlers, and Device Manager. Figure 3 explains the architecture of the WoTC++ framework compared with the architectural aspects specified by W3C.This WoT's architecture aspects define four fundamental pillars: device behaviour, interaction affordances, protocol bindings, and security configuration. While security configuration is not the primary emphasis of this implementation, the suggested framework is designed to be compatible with proven WoT-compliant security methods, such as DTLS, OAuth 2.0, or HTTPS-based transport, making it security-ready. This WoTC++ framework offers a scalable, efficient, and standards-aligned solution for real-time IoT system integration, providing end-to-end WoT support from microcontroller-level hardware to web and cloud interfaces. The framework has been designed to execute on a single thread or main loop, which makes it runnable on non-RTOS or bare-metal systems.

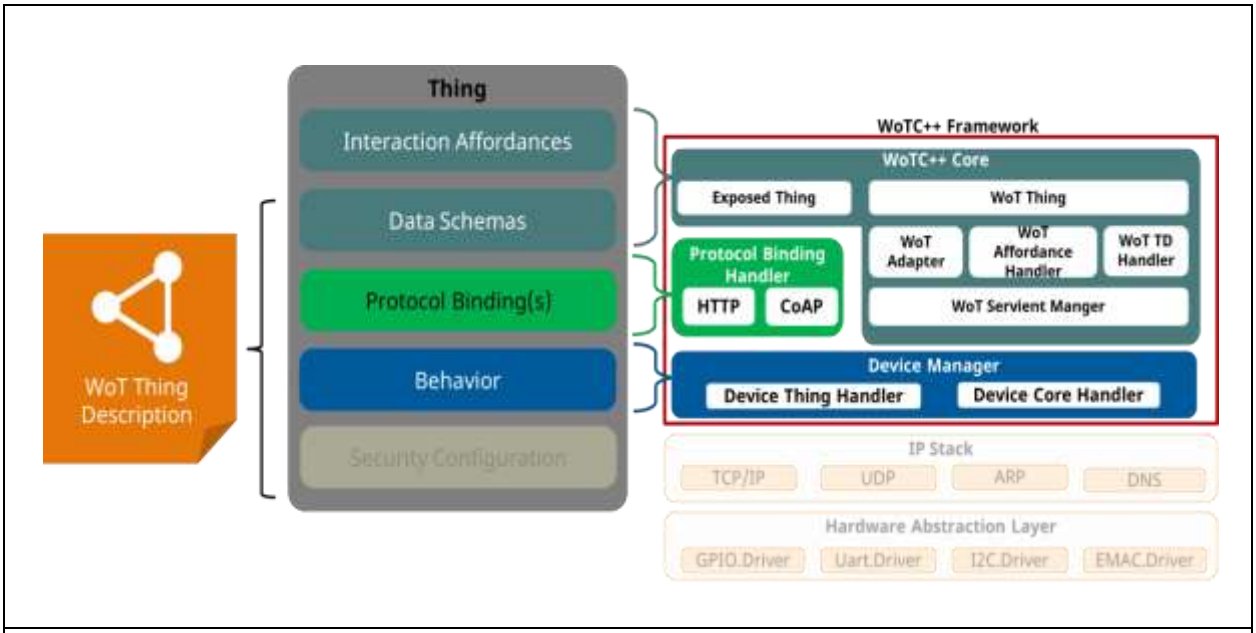


Figure 3: The Proposed WoTC++ Architecture mapping with WoT Architectural Aspects of a Thing (re-arranged)

WoTC++ Core

The WoT ecosystem's semantic and logic core is embodied at the top of the architecture by the WoTC++ core. In the proposed framework, the WoTC++ core satisfies the W3C specification. It can be logically separated into WoT Thing, WoT Adaptor, WoT TD Handler, WoT Affordance Handler, WoT Servient Manager, and Exposed Thing. To improve adaptability, the WoT++ Core implementation has adopted the concepts and semantic conventions of the WoT Scripting API, even though it does not fully implement the W3C-suggested scripting API[29]. WoT Servient manager is the WoTC++ Runtime Engine, a compact C++ engine that runs the WoT's basic functions and acts as the primary interface to the application task. It is a unified entity that conforms to W3C WoT standards and integrates producer and consumer features. By use of WoT Thing Descriptions (TDs)[30], the Exposed Thing and WoT Thing modules reveal the operational semantics and metadata of physical devices, both abstract and concrete, respectively.

The WoT Adapter is a component of the WOTC++ Core that contains the custom standard API for reading and writing objects and their affordances. This custom standard API provides a way to scale additional binding interfaces without modifying WoT Core components. The WoT Affordance Handler manages the Properties, Actions, and Events of a WoT Thing, and provides APIs to read and write at runtime. The built-in TD Handler processes and organizes these thing descriptions (TDs), and the Things Description is in the JSON-LD format specified by the W3C. Affordance handlers have an inbuilt serialize method that converts the affordance into machine-readable JSON-LD. While the Affordance Handler enables the real-time invocation and monitoring of specified affordances, the WoT Adapter serves as a bridge between hardware signals and semantic operations.

Protocol binding Handler

Protocol binding is the Process of translating from interaction affordances to machine-readable messages of an underlying communication protocol, such as HTTP [RFC7231], WebSocket [RFC6455], MQTT [RFC9431], and CoAP [RFC7252]. The protocol binding handler serves as a protocol server for bindingWoT objects. Protocol binding handlers are heavily reliant on the platform's Internet Protocol (IP) stack, such as Linux, ESP32, or ARM. The Internet Protocol stack employs data communication methods over Ethernet, as per the OSI (Open Systems Interconnection) or TCP/IP (Transmission Control Protocol/Internet Protocol) model. Most platform vendors deliver the Internet Protocol stack or library as part of their Software Development Kit (SDK), including a Hardware Abstraction Layer (HAL), to enable rapid application development. The protocol-binding

servers in this proposal are created using the APIs provided by the platform's IP stack. The IP stack handles the lower-level interfaces to the Hardware Ethernet controller.

The protocol binding interfaces generated by the WoTC++ framework's protocol binding handlers are serialized as hypermedia controls (in the form of Thing Description) with a descriptive way to activate interaction affordances, such as Properties, Actions, and Events. The hypermedia controls are formed using URIs [RFC3986] accessible via the REST API from clients. The media type [RFC2046] used by WoTC++ is application/JSON[RFC8259], as specified by W3C WOT.

Device Manager

The Device Manager in the WoTC++ framework is responsible for implementing a Thing's functionality in accordance with the W3C WoT architecture. It controls the actuators, reads the sensors, and handles the affordances of interactions like property access, action invocation, and event notification. To facilitate portability and reuse across different hardware versions, a Hardware Abstraction Layer (HAL) from the platform vendor decouples hardware dependencies by providing consistent driver interfaces for EMAC, I^2C, UART, and GPIO to the device manager.

The Device Thing Handler and the Device Core Handler are the two primary parts of WoTC++'s Device Manager that provide modularity and maintainability. While the Device Core Handler communicates directly with hardware components via the underlying platform's Hardware Abstraction Layer (HAL), the WoTC++ Core processes interaction requests and builds compatible responses via a bespoke API that the Device Thing Handler later uses. The Device Thing handler harmonizes the interfaces between the WoTC++ core and the custom API to handle interaction requests and responses. Maintaining atomicity, streamlining platform interaction, and improving code clarity are all benefits of this tiered separation. The architectural enhancement of WoTC++ lies in its comprehensive device handler support, which extends beyond virtual devices to include native bindings for both Zigbee- and Wi-Fi-based smart devices.

Framework Configuration

The implementation avoids dynamic memory allocation, which is the primary limitation when porting the stack to memory-constrained IoT platforms. Despite static memory allocation being chosen to fulfill portability, the WoTC++ stack has been armed with multiple configuration parameters to adjust the sizes of affordances and their parameters, as well as the number of clients and servers that can be supported. The configuration strategy aids in porting the WoTC++ core across different platforms and in deploying according to the deployment patterns described in the Common Deployment Patterns section of the WoT Architecture. These configurable parameters help port the WoTC++ core to any platform based on the available memory for its Thing requirements. Table 2 lists the configurable parameters used in the WoTC++ stack: Nt, Np, Na, Ne, Npp, Nei, Nai, Nao, Nservers, and Nclients.

Table 2: Configurable parameters in WoT++ Core

Configuration Name	Legend	Definition
WOT_THINGS_MAX_SUPPORT	N_t	No of things to be exposed or consumed by the device
WOT_THINGS_MAX_PROPERTY	N_p	No of properties for each exposed or consumed thing
WOT_THINGS_MAX_ACTION	N_a	No of actions for each exposed or consumed thing
WOT_THINGS_MAX_EVENT	N_e	No of events for each exposed or consumed thing
WOT_PROPERTY_MAX_PARAMS	N_{pp}	No of parameters supported by the property affordance
WOT_EVENT_MAX_INPUTS	N_{ei}	No of input parameters for event affordance
WOT_ACTION_MAX_INPUTS	N_{ai}	No of input parameters for action affordance
WOT_ACTION_MAX_OUTPUTS	N_{ao}	No of output parameters for action affordances
WOT_SERVIENT_MAX_SERVERS	$N_{servers}$	No of servers supported by the servient for the exposed thing
WOT_SERVIENT_MAX_CLIENTS	$N_{clients}$	No of clients supported by the servient for the consumed thing

The 64-bit platform is used to derive the sizes of the data structures (*PropParameters_t*, *EventParameters_t*, and *ActionParameters_t*) under the following assumptions, which are statically defined in the framework.

- String size limited to 24 Bytes
- double size 8 Bytes as per the platform
- uint64 size 8 Bytes as per the platform
- int size 8 Bytes as per the platform
- bool size 1 Bytes as per the platform

Each Property affordance size M_{pp}depends on its parameter size Npp and the*PropParameters_t*size. Denoting $M_{properties}$as the total RAM required for the property affordances in the framework, it depends on Np and Mpp and is calculated by formula (1).

$M_{properties} = (N_p * M_{pp}) + N_p * 8$ Bytes .... (1)

Where M_{pp} is derived from,

M_{pp} = sizeof (*PropParameters_t*) * N_{pp} ... (2)

In the WoTC++ implementation, *PropParameters_t*requires 92 Bytes to fulfill its requirements, and the formula (2) can be rewritten as

$M_{pp} = 92 \times N_{pp}$Bytes ... (3)

Each Event affordance size M_{ei} depends on its input parameter size N_{ei} and the*EventParameters_t* size. Denoting M_{events}as the total RAM required for the event affordances in the framework, it depends on Ne and Mei and can be calculated using formula (4).

$M_{events} = N_e * M_{ei}) + N_e * 8$ Bytes ... (4)

Where M_{ei} is derived from,

M_{ei} = sizeof (*EventParameters_t*) * N_{ei} ... (5)

In the WoTC++ implementation, *EventParameters_t*requires 51 bytes to fulfill its requirements, and the formula (5) can be further considered as

$M_{ei} = 51 \times N_{ei}$ Bytes ... (6)

Each Action affordance's input and output memory sizes, Mai and Mao, depend on the input and output parameter sizes Nai and Nao, respectively, as well as the*ActionParameters_t*size. Denoting M_{action}as the minimum RAM required for the action affordances, it can be calculated using formula (7).

$M_{action} = (N_a * (M_{ai}+M_{ao})) + N_a * 8$ Bytes ... (7)

Where M_{ao} and M_{ai} are derived from,

M_{ai} = sizeof (*ActionParameters_t*) * N_{ai} ... (8)

M_{ao} = sizeof (*ActionParameters_t*) * N_{ao} ... (9)

In the WoTC++ implementation, *ActionParameters_t statically* requires 50 bytes to fulfill its requirements, and the M_{ai} and M_{ao} formulas are rewritten as

$M_{ai} = 50 \times N_{ai}$ bytes ... (10)

$M_{ao} = 50x N_{ao}$ bytes ... (11)

Each server and client enrolled to WoTC++ will be stored in a pointer, and $M_{servers}$ and $M_{clients}$ can be derived from it.

$M_{servers}$= 8 Bytes * $N_{servers}$ ... (12)

$M_{clients}$ = 8 Bytes * $N_{clients}$ ... (13)

At the end, M_{total}, the RAM required based on the configuration parameters, is calculated from formulas (1), (4), (7), (12), and (13) and formulated as (14).

$M_{total} = M_{servers}+M_{clients}+ (N_t * (M_{properties}+M_{action}+M_{events}))$ .... (14)

6. Implementation Details

This section discusses the implementation of the suggested framework on the platforms (P1 and P2) as discussed in the next section. Written entirely in native C++, this framework is designed to improve execution performance and reduce memory overhead, in conformance with the W3C WoT architecture.

Implementation Setup

Prior to delving into the specifics of the implementation, it is necessary to specify the platform on which it will be experimented. Two platforms were selected to support the objective of creating a portable WoT core system. Because of its limited resources, the first platform, P1, is an ESP32 and is considered a constrained platform. Furthermore, ESP32 has a flexible development environment that facilitates quicker prototyping. The Raspberry Pi (RPi), the second platform designated P2, is typically regarded as an unconstrained platform due to its abundant resources. The Raspberry Pi was selected due to its widespread use as a prototyping platform by researchers [31] and its suitability for benchmarking against current WoT implementations. The specifics of the platforms utilized for the implementation are provided in Table 3.

Table 3: Implementation setup details

Setup Information	ESP32 (P1)	Raspberry Pi (P2)
Hardware Platform	ESP32-WROOM-32 (dual-core Xtensa LX6 @ 240 MHz, 520 KB SRAM)	Raspberry Pi 5 (4GB) (2.4GHz quad-core 64-bit Arm Cortex-A76 CPU)
Operating System	FreeRTOS (Real-Time Operating System, built-in support on ESP-IDF)	Debian GNU/Linux 12 (bookworm), Linux 6.12.20+rpt-rpi-2712
IoT Devices	Sensor Module	Virtual Clock, Philips Hue Bulb

ESP32 Platform (P1)

The ESP32-WROOM-32, which features a dual-core Xtensa LX6 @ 240 MHz, 520 KB of SRAM, and built-in Wi-Fi connectivity, is used on the ESP32 platform. Because the ESP32 platform lacks an Ethernet interface, the device is connected to the router via a Wi-Fi access point. ESP32 is connected to a sensor module labelled D1, as shown in the Device Manager section of Figure 4 under the platform P1 setup.

Raspberry Pi Platform (P2)

The Raspberry Pi platform uses a Generation 5 Raspberry Pi device, also known as the RPi 5, which has a 64-bit ARM Cortex-A17 quad-core CPU running at 2.5GHz, 4GB of RAM, gigabit Ethernet, and support for 802.11ac Wi-Fi and Linux 6.12 running on the Debian 12 (Bookworm) operating system. The Raspberry Pi 5 has been configured with the 20+rpt-rpi-2712 Kernel. This platform simulates a lightweight virtual device, D1, for performance benchmarking by hosting a virtual clock on the Raspberry Pi. The Raspberry Pi 5 is also connected to a Philips Hue Zigbee bulb (referred to as D2).

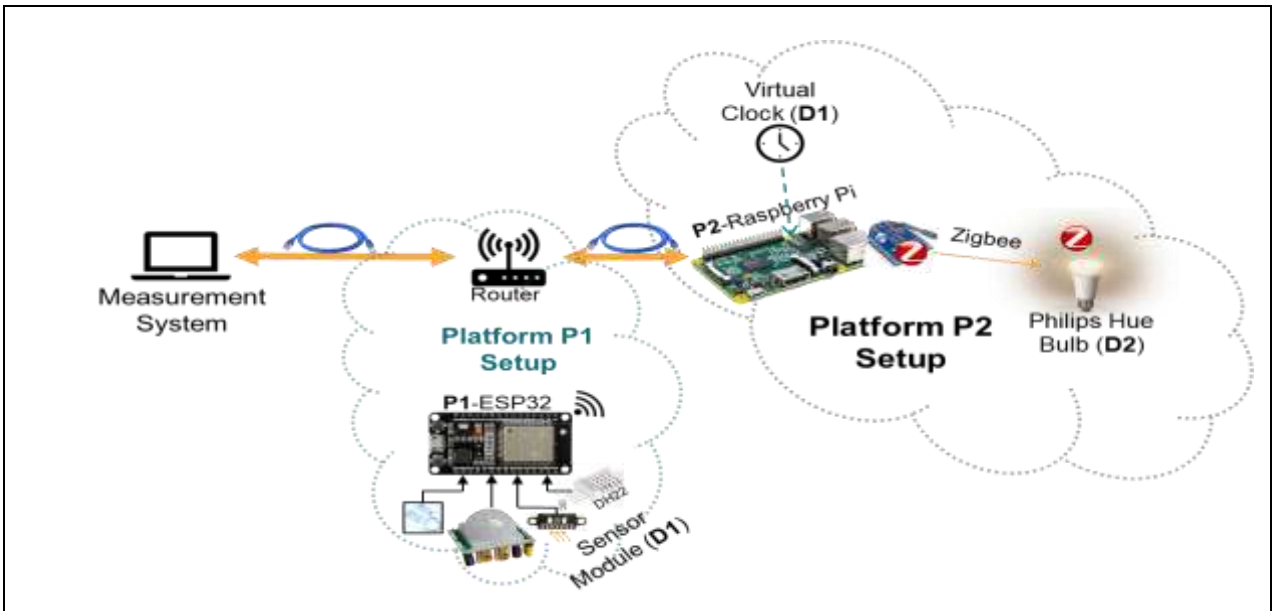


Figure 4: Implementation bench setup with Platform P1 and P2

To illustrate the framework's support for legacy protocols, the Zigbee Hue Bulb (D2) connects to the Raspberry Pi via an XBee Zigbee adapter. With standardized Thing Descriptions made available by the suggested framework, external WoT clients can access and manage these devices. This setup exemplifies WoTC++'s ability to unify device interactions across heterogeneous communication stacks, providing a scalable WoT runtime for gateways. To eliminate Wi-Fi transmission overhead in the measurements, the Raspberry Pi 5 and the measurement laptop are connected to a router via an Ethernet cable.

WoTC++ Core

Being central to the proposal, the WoTC++ core implementation is targeted for reuse across P1 and P2. It can be integrated into any device, regardless of device class, as stated in the Device Categories section of the WoT Architecture [28], provided necessary resources are available.

As previously mentioned, the implementation of WoT requirements is owned by the WoTC++ core and is primarily dependent on the realization of the Exposed thing and its interaction affordances. A sample ExposedThing collaboration diagram describing the general WoTC++ core architecture is shown in Figure 5. *ExposedThing*, a derived component in the WoTC++ core, inherits all of the W3C WoT functions from its base affordance classes, such as*ThingProperty*, *ThingAction*, and *ThingEvent*. In the proposed work, the WoTC++ core implementation is common across both P1 and P2 platforms, and the configuration parameters are selected accordingly based on the number of exposed devices.The implementation source codeis publicly available at https://github.com/boobalkrishnan-git/WoT-Cpp-Core.

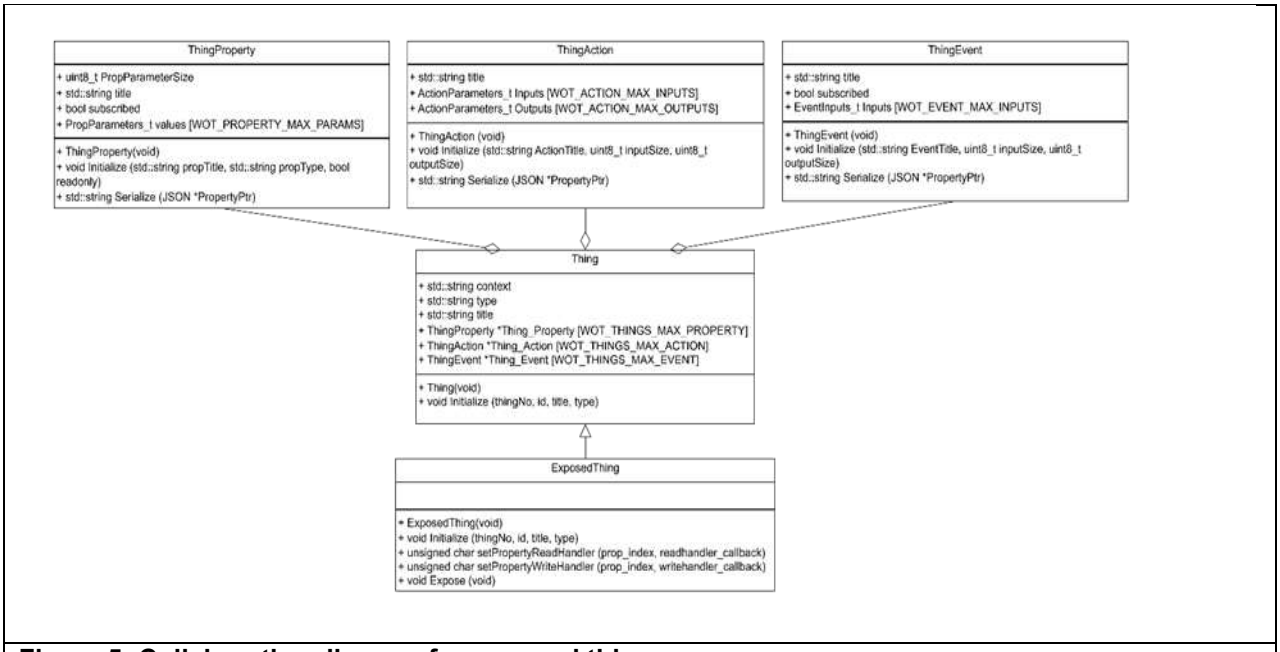


Figure 5: Collaboration diagram for exposed thing

Configuration details

As previously discussed, configuration parameters help to scale the WoT core to multiple platforms and it offers unified stack across platforms. The details of the configurations used among P1 and P2 is listed in Table 4.

Table 4: Configuration details of P1 and P2 platforms

Configuration Name	Legend	P1 Platform	P2 Platform
WOT_THINGS_MAX_SUPPORT	N_t	1 thing	2 things

WOT_THINGS_MAX_PROPERTY	N_p	4 properties	10 properties
WOT_THINGS_MAX_ACTION	N_a	0 actions	10 actions
WOT_THINGS_MAX_EVENT	N_e	0 events	10 events
WOT_PROPERTY_MAX_PARAMS	N_{pp}	1 value	5 values
WOT_EVENT_MAX_INPUTS	N_{ei}	0 inputs	10 inputs
WOT_ACTION_MAX_INPUTS	N_{ai}	0 inputs	5 inputs
WOT_ACTION_MAX_OUTPUTS	N_{ao}	0 outputs	5 outputs
WOT_SERVIENT_MAX_SERVERS	$N_{servers}$	1 server	5 servers
WOT_SERVIENT_MAX_CLIENTS	$N_{clients}$	0 client	0 client

Protocol binding Handler

WoTC++'s protocol binding interfaces, as explained in the framework design section, leverage native support to build upon the target platforms' existing IP communication stacks rather than starting from scratch with new networking layers. Each platform (P1 and P2) requires distinct protocol binding handlers due to IP stack dependencies. Custom standard APIs are defined to interact between the WoT Core and the Protocol binding handler, delivering two benefits. To begin with, the custom API enables scaling to additional protocol servers without interfering with those already in place. Second, it allows the WoTC++ core to remain unchanged across platforms. This design choice maintains runtime performance efficiency while guaranteeing hardware and operating system compatibility.

The layered implementation of the WoTC++ protocol binding handler on P1 and P2 is depicted in Figure 6. Through its modular layers, the stack illustrates the interplay between the Exposed Thing interface and the platform-specific hardware abstraction layer. The WoT Adaptor, located at the very top, uses HTTP and CoAP Protocol Handlers to connect application logic to network protocol affordances, and these handlers communicate with their corresponding protocol drivers from the IP stack.

On the Raspberry Pi platform, the Poco C++ network library [32], Release 1.13.4, has been used as the IP stack to build our protocol binding interfaces, due to widespread support for data parsing libraries such as XML and JSON. Poco::Net, Poco::JSON, Poco::Util and Poco::DNSSD are used.

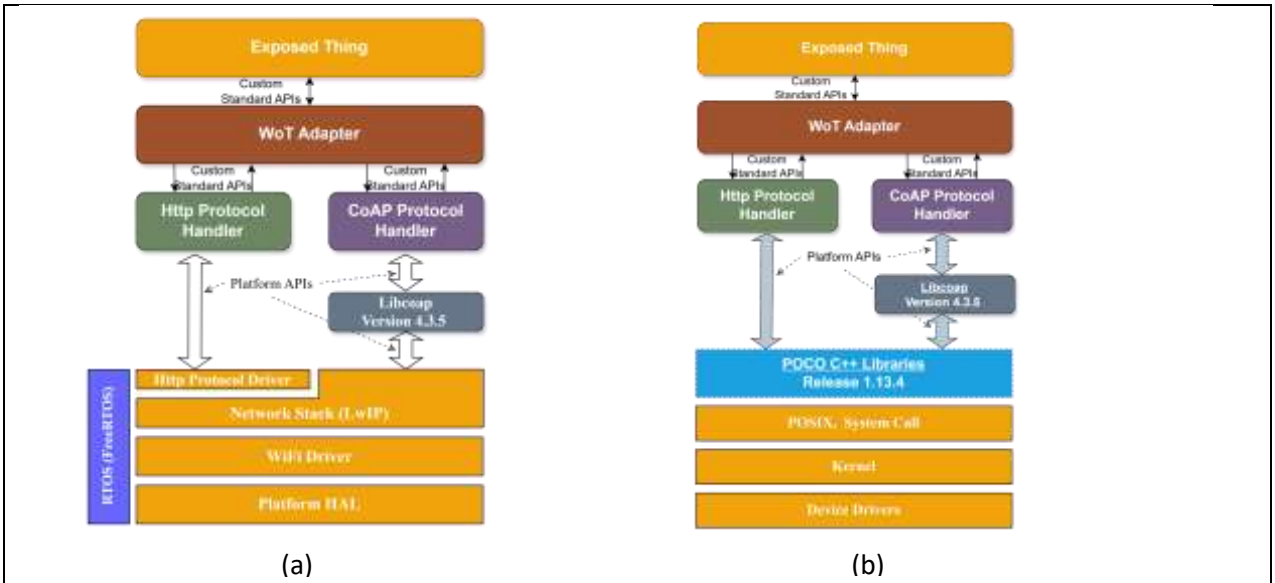


Figure 6(a) Interface betweencommunication stack and WoT core stack in P1, (b) Interface between communication stack and WoT core stack in P2

Using a UML collaboration class diagram, Figure 7 illustrates the implementation of an HTTP protocol binding handler. Upon receiving an incoming HTTP request, the *BindHttpServer* class parses the URI and calls the appropriate affordance classes, including *BindHttpPropertyHandler*, *BindHttpActionHandler*, and

BindEventObservHander. The base class *Poco::Net::HTTPRequestHandler* is the source of these affordance handlers.

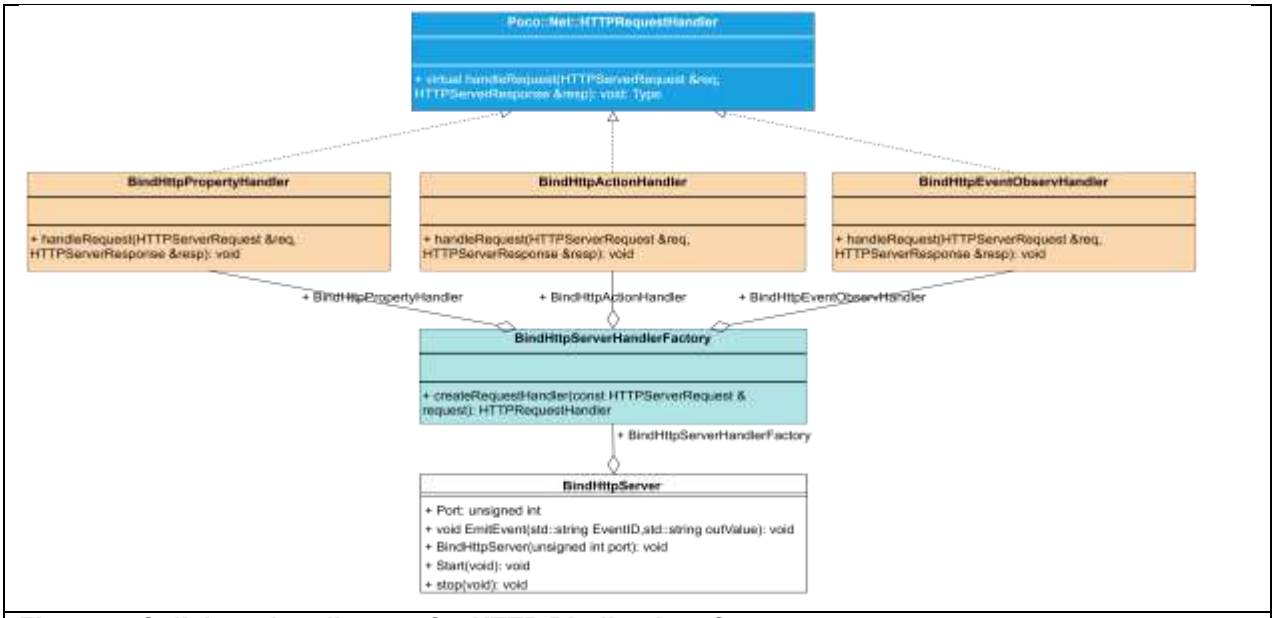


Figure 7: Collaboration diagram for HTTP Binding interface

The Constrained Application Protocol (CoAP) binding interface has been implemented on top of libcoap[33] and Poco sockets.Libcoap is an ideal candidate for a WoT framework, as it is designed with multi-platform support, interoperability, andextended support for Security frameworks.Thoughlibcoap supports multiple security extensions, this proposal does not use them to evaluateWoTC++ performance under additional load.CoAP binding interfaces were designed to support the Request/Response model with the confirmable (CON) messagetype. In addition, the binding interface is architected to support piggybacked responses to avoid overhead.

On the ESP32 platform, the HTTP protocol binding handler is built on top of the HTTP protocol driver from the ESP network stack, and the CoAP protocol binding handler is built on top of the LibCoAP library version 4.3.5, which is again built on top ofLwIP. For real-time job scheduling and predictable behaviour, the entire stack runs in an RTOS (such as FreeRTOS), and the Wi-Fi Driver communicates with the hardware-level Platform HAL (Hardware Abstraction Layer). Modular concern separation, protocol extension, and integration of WoT-compliant communication across resource-constrained IoT devices are the key features of this system.

Device Manager

To evaluate the functionality and interoperability of the WoTC++ framework, three distinct device managers have been implementedfor the devices as described in the Implementation setup section. Because these device managers represent a variety of device classes and communication protocols, the framework can be thoroughly tested in both virtual and real-world settings.

Sensor Module

The temperature sensor (DH22), light sensor (TCS34725), PIR motion sensor, door magnetic switch, and other common smart home sensors are all controlled centrally by a smart sensor module. The ESP32 and temperature sensor are linked via a 1-wire bus. After the ESP32 issues the start signal, the DH22 responds with 40-bit data containing temperature, humidity, and a checksum. The temperature and humidity will be extracted from the 40-bit data in human-readable formats. The I2C interface, a two-wire bus, has been used to connect the Light Sensor to the ESP32. The main function of the TCS34725 is to measure the ambient light's red, blue, green, and clear hues. A software algorithm computes illumination and colour temperature. The

digital IO pins on the ESP32 are used to connect the PIR motion sensor and the door magnetic switch. Open-source drivers for the DH22 and TCS34725 sensors are used in conjunction with C++ on the ESP-IDF platform to develop the application. The sensor logic serves as the foundation for the WoT server framework, which exposes the sensors to the client system.

Zigbee Bulb

The Zigbee protocol is a legacy home automation protocol used in many battery-operated smart home devices due to its low power consumption. The Zigbee device manager is implemented only on the Raspberry Pi P2 platform to evaluate gateway functionality using the proposed framework. The Philips Hue lightbulb is utilized as the Zigbee target in our proposal. The focus of the proposal is mapping the Zigbee Cluster Library (ZCL), which defines the application messaging of the Zigbee protocol, to WoTaffordances. In this proposal, a pre-configured method has been employed to map between the Zigbee cluster library andWoTThingaffordanceclasses.The pre-configured mapping method consists oftwo steps. The first step is to create a static structure that converts data from the Zigbee cluster library into an iterable. During the Zigbee discovery process, the second step is the runtime mapping of Zigbee cluster library clusters, attributes, and commands to WOT affordances, as shown in Table 5 below.

Table 5: Static Mapping between Zigbee Cluster Library and WoT Thing Model

Zigbee	WoT
Endpoint	**Thing**
Cluster Attributes	**Properties**
Server Generating Commands	**Events**
Server Receiving commands	**Actions**

At step 1, a Boolean flag called Exposable is added to the Zigbee cluster library's static structures, such as Endpoint Information, Cluster List, and Attribute List, to control the exposure of the corresponding cluster's Attributes and Commands when creating new objects. With this Boolean flag, protocol-level endpoints, clusters, and attributes corresponding to ZDO and OTA upgrades—which are not functional clusters—are not unnecessarily exposed to users or clients. When constructing WoT objects with properties, actions, and event affordances, algorithm 1 usesa Boolean flag to verify the presence of each.

Zigbee discovery begins upon receiving a Device Announce message or a Node Information message. The Zigbee reception process runs in a separate thread to ensure that all messages from other devices are recorded. Upon arrival of a new data frame, the reception process verifies each time whether the frame is a ZDO Device Announce Cluster or a Node Information Frame. Consequently, as illustrated in Figure 8, the received data is passed to the Discovery process to obtain additional ZDO cluster data, including the IEEE (64-bit) Address, Node Descriptor, Active Endpoints, and the Simple Descriptor of each Endpoint. After the ZDO cluster data has been gathered, the newly discovered device can be added to the Zigbee Thing queue and made public. Algorithm 1 describes the Process for creating a new Web of Things object and its property affordance using ZDO information, such as NodeDescriptor, Endpoint details, and SimpleDescriptor, from the discovered Zigbee device.

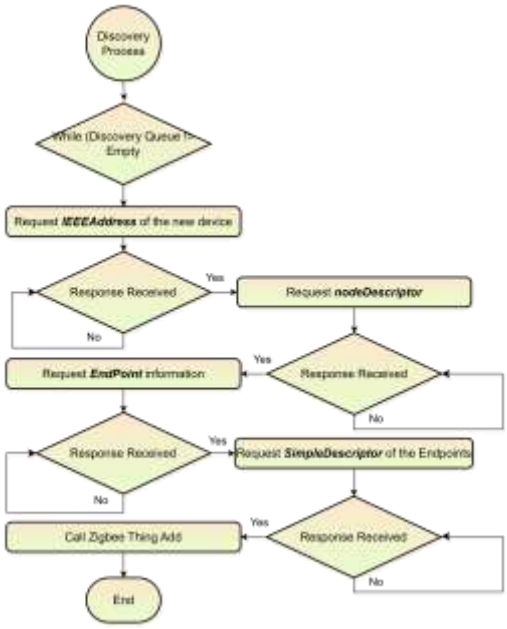


Figure 8: Flowchart of Zigbee Discovery process

Algorithm 1 Runtime WoT Things construction algorithm for new Zigbee Device

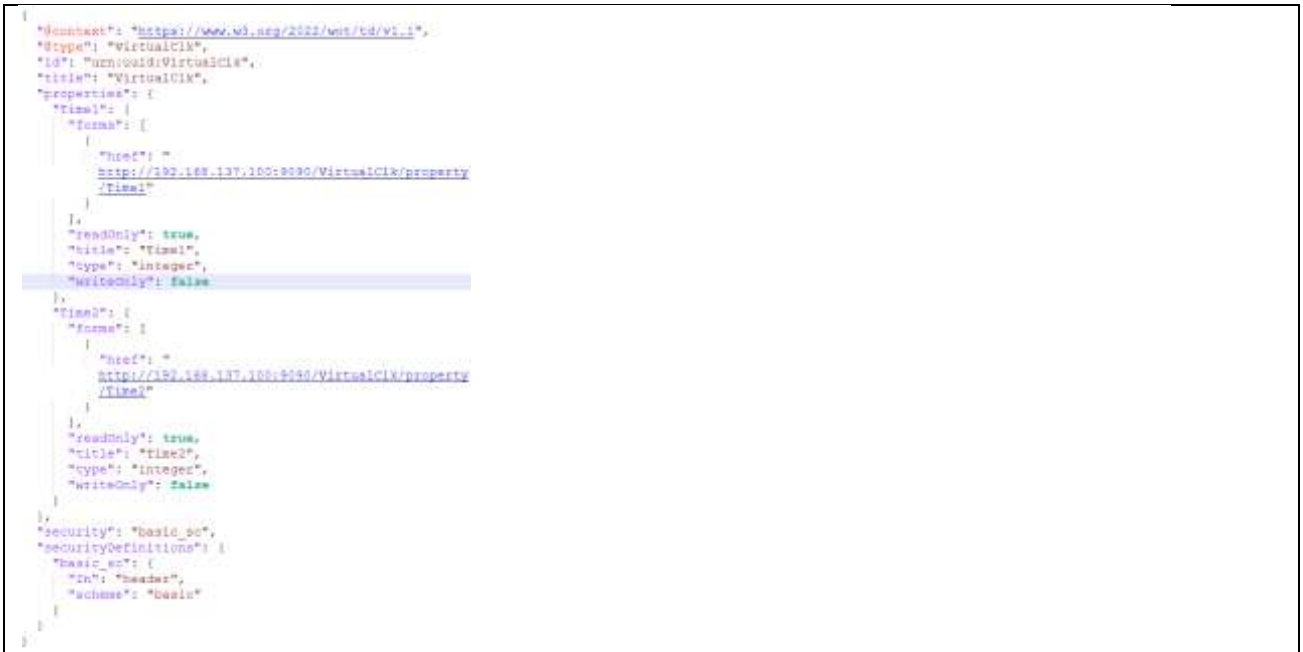
```
input:ZigbeeDeviceInput, clusterInput, WoT
NoEndPoints← 0, NoClusters ← 0
ZigbeeNode ← ZigbeeDeviceIn
ZigbeeDeviceId ← convert_int(ZigbeeDeviceIn.Bit64Address)
NoEndPoints ← ZigbeeDeviceIn.EpCount
/* Ep means EndPoint in Zigbee Protocol*/
for EpIndex ← 0 to NoEndPoints do
  CurrentEpInfo ← ZigbeeDeviceIn.EndPointInfo[EpIndex]
  if CurrentEpInfo.Exposable then
    ZigbeeThing ← new ExposedThing()
    title ← CurrentEpInfo.SimpleDescriptor.DeviceString
    ZigbeeThing.Initialize(ZigbeeDeviceId,title)
    NoClusters ← CurrentEpInfo.SimpleDescriptor.InputClusterLen
    for ClusterIndex ← 0 to NoClusters do
      CurrentCluster ← CurrentEpInfo.SimpleDescriptor.InputClusters[ClusterIndex]
      if CurrentCluster.Exposable then
        /*Create Property class FOR all the Attributes supported & Exposed */
        for attributeIndex ← 0 to CurrentCluster.AttributeSize do
          if CurrentCluster.AttributesList[attributeIndex].Exposable
          then
            ZigbeeProperty ← new ThingProperty()
            attributeid ← CurrentCluster.AttributesList[attributeIndex].AttributeID
            endpointid ← CurrentEpInfo.EpNumber
            /*PropertyType holds the string of property value's type*/
            PropertyType ← ZigbeeThing_GetPropertyValueType(clusterIn,attributeid)
            AttributeString ←CurrentCluster.AttributesList[attributeIndex].AttributeName
            Intialize ZigbeeProperty(AttributeString,PropertyType,false)
            propertyId_return ← ZigbeeThing.addProperty(ZigbeeProperty)
            /*Set Handler function FOR PUT Request */
            ZigbeeThing.setPropertyWriteHandler(prop_return,ZigbeeWriteHandler)
            /*Set Handler function FOR GET Request */
            ZigbeeThing.setPropertyReadHandler(prop_return,ZigbeeReadHandler)
            propertyinfo ={ZigbeeThing.title,ZigbeeProperty.title,ZigbeeDeviceInput,
                                                              endpointid,clusterInput,attributeid}
            Push PropertyList← propertyinfo
          end if
        end for
      end if
    end for
    WoT.AddThing(ZigbeeThing)
  end if
end for
```

The sample Things Description (TD) received from the Zigbee device thing, 'Colour Temperature Light', is shown in Figure 9. The Colour Temperature Light device is pre-configured using a Boolean flag to expose only the OnOff Property and the Off, On, and Toggle actions.

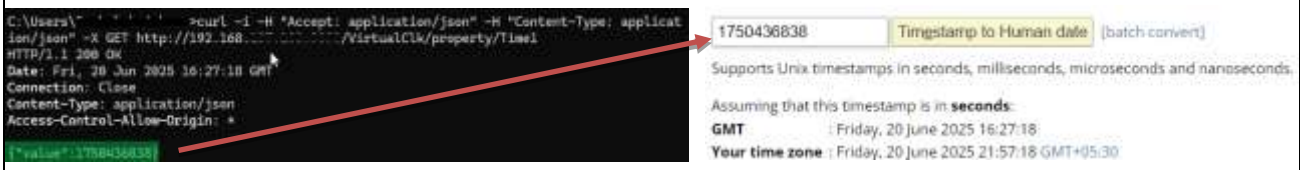
Figure 9:Sample TD received from the Zigbee Thing.

Virtual Clock

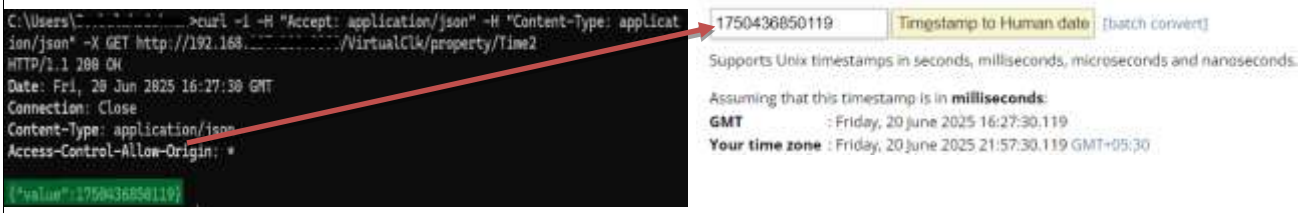
Virtual Clock serves as simulated device on P2 that retrieves the system's internal time in Unix epoch format. It is primarily used for performance benchmarking across different WoT implementations. As illustrated in Figure 10 (a), the Virtual Clock Thing Description (TD) defines two readable properties: Time1, which returns the system time as a 10-digit epoch (seconds only), and Time2, which provides a 13-digit epoch format including milliseconds. This dual-format design supports testing of time granularity and serialization performance. A sample property read response from the Virtual Clock running on a Raspberry PI is shown in Figure 4, demonstrating the consistent structure and response time that the WoTC++ protocol stack can achieve in a virtualized scenario. Figure 10(b) explains the time received from time1 Property is as "*GMT: Friday, 20 June 2025 16:27:18*" and figure 10(c) explains the time received from time2 Property is as "*GMT: Friday, 20 June 2025 16:27:30.119*" with milliseconds.



(a) Sample TD retrieved from WoTC++ implementation



(b) Sample response from the Time1 property of the virtual clock thing



(c): Sample response from the Time2 property of the virtual clock thing

Figure 10: Virtual clock

7. Evaluation Setup

To thoroughly evaluate the WoTC++ framework's performance, efficiency, and compatibility, two distinct experimental evaluations were conducted using platforms P1 and P2, as listed in Table 6. The first evaluation assesses the framework's performance on the ESP32 platform, a device with limited resources via asensor module. The second evaluation involves incorporating the WoTC++ framework into a gateway design using the Raspberry Pi platform, which aims to connect various IoT ecosystems, such as Zigbee- and Wi-Fi-enabled TP-Link smart devices, to the WoT interface, thereby complying with W3C standards. In addition, the Gateway evaluation is essential for assessing the framework's interoperability. The legacy devices were made available as WoT Things via the gateway configuration, enabling standardized interaction with WoT consumers via HTTP and CoAP bindings.

Table 6: WOTC++ experimentation evaluations on Platform P1 and P2				
Evaluation Name	Platform used	Frameworks used	Binding Protocols	Devices used
ESP32 Evaluation	P1	WOTC++	HTTP, COAP	Sensor Module
Gateway Evaluation	P2	WOTC++	HTTP, COAP	Zigbee

To evaluate multi-protocol interoperability and performance, two protocol bindings were employed, as stated earlier in the implementation details section: HTTP, the conventional web application layer protocol, and CoAP (Constrained Application Protocol), optimized for constrained environments. For each test scenario, 100 read property invocations were executed to measure response time, with the number of concurrent threads varied across seven predefined levels—1, 2, 5, 10, 20, 50, and 100—to simulate varying workloads and concurrency levels. This controlled configuration enabled a detailed performance analysis across three experimental implementations.

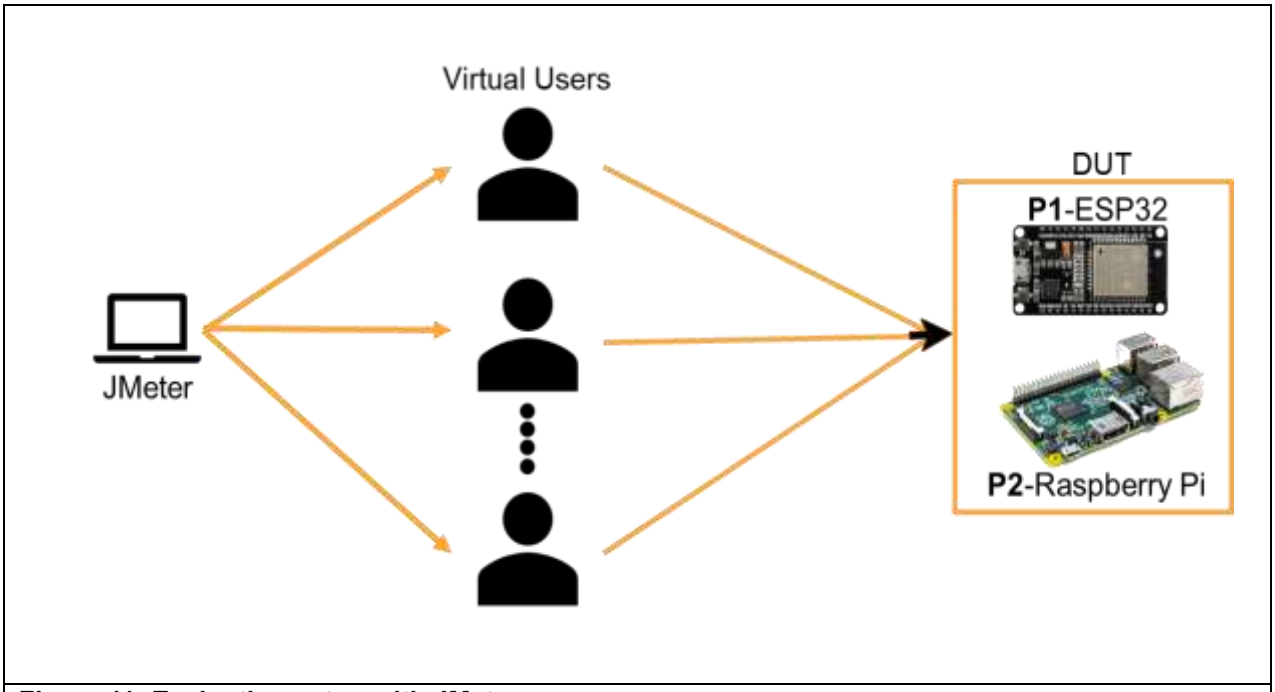


Figure 11: Evaluation setup with JMeter

The evaluation parameters in the experiments are Response Time, CPU utilization, and RAM consumption. These three metrics are sufficient to demonstrate performance, as CPU utilization and RAM consumption are critical indicators of constrained device performance, and Response Time is a universal metric for analyzing web system performance. The JMeter [34] client, an open-source tool for evaluating web application performance, is used to measure response time as shown in Figure 11. Despite JMeter's lack of native support for COAP, the proposal has been tested using jmeter-coap, an open-source third-party plug-in[35]for JMeter. The OriginPro learning edition[36] tool is used to create the box charts. Time stamps obtained from the JMeter response message were used to correlate CPU and memory usage measurements recorded with the "top" Linux command.

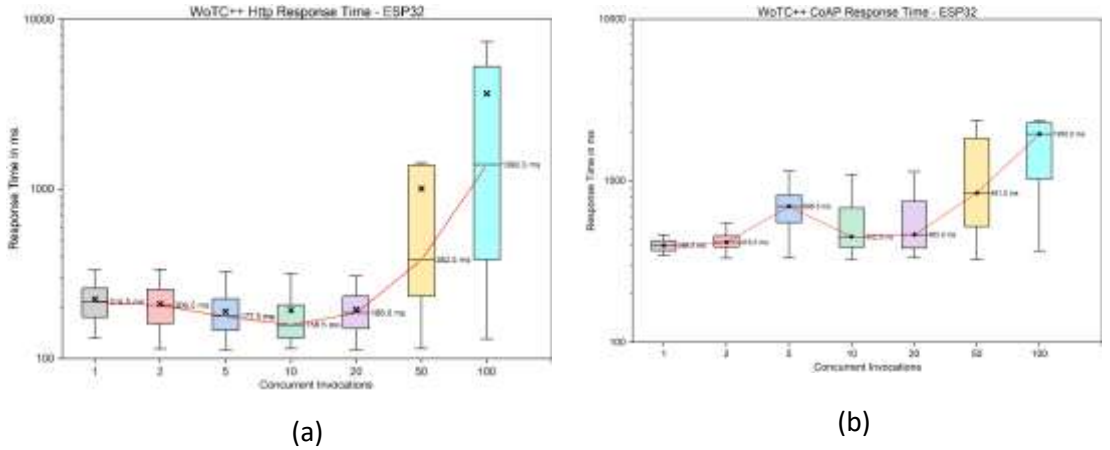
Boxplots have been used to illustrate response time because they provide a visual representation of the data and help determine the data's average. This proposal focuses on the analysis of the median and interquartile range (IQR) of response times. The quartiles for Q1 and Q3 are in the 25th and 75th percentiles, respectively. The upper and lower whiskers were chosen as Q3+1.5(IQR) and Q1-1.5(IQR). Whiskers are considered outliers, and occasional outliers are ignored in all results due to the nature of the study, which benchmarks against different systems and analyses average performance. The Interval chart has been used to show the CPU load

variability. First two evaluations, Benchmarking and Constrained platform evaluations, are analysed using a linear scale as the upper and lower whiskers, as the response times are not too far from the median.

8. Results and Discussion

ESP32 Sensor Module Results

Figure 12 presents a boxplot analysis of response times of ESP32 in logarithmic scale across different invocation concurrent levels (1, 2, 5, 10, 20, 50, and 100). As the number of threads increases, particularly beyond 20, response time becomes more variable and the median increases. At 100 threads, the system exhibits a significant spike in latency, as shown by the wider whisker, indicating potential bottlenecks or saturation of processing capacity. The median response time is around 200ms with HTTP binding and around 400ms with CoAP, and it increases with thread concurrency levels. As previously discussed, higher concurrency is not expected at end-sensor devices, as the sensor servers would have very few clients, and the additional concurrent requests are handled at Edge or cloud servers or digital twins.



(a) (b)

Figure 12: Response Time performance of the WoTC++ framework for a constrained platform under varying levels of concurrency (logarithmic scale)

OnESP32 CPU utilization, CPU usage remains minimal, under 10% at low thread counts (1, 2, 5). However, as threads increase, especially from 20 to 100, CPU utilization rises sharply, reaching nearly 50% at 100 threads. This trend indicates that WoTC++ maintains energy efficiency at low to moderate loads but may start to stress the processor significantly as concurrency increases.

Zigbee Gateway Results

This section describes the response time results of the gateway evaluation for the Zigbee smart home protocol.In the gateway evaluation, a linear scale has been used because the upper whiskers are not too far from the median response time comparing to ESP32 results.

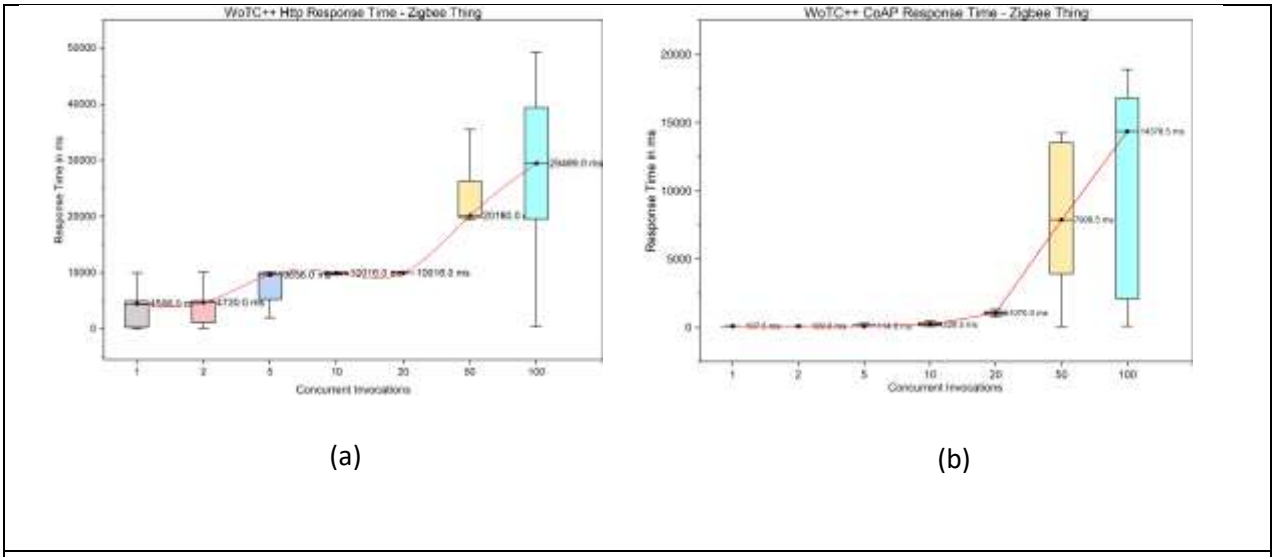


Figure 13: Response Time performance of the WoTC++ framework for Zigbeesmart device under varying levels of concurrency (Linear Scale)

In Figure 13(a), the HTTP-based response times exhibit relatively steady but higher values, with a mean around 1000 ms and a varying median. Although the performance is consistent across thread counts, it shows elevated latency and moderate variability. In contrast, Figure 13(b) highlights the CoAP-based results, where the framework initially demonstrates low latency and stable performance at lower thread counts (1–20), maintaining response times around 100 ms. However, as concurrency increases to 50 and 100 threads, CoAP response times degrade sharply, with spikes into the tens of thousands of milliseconds. This sharp rise suggests possible scalability constraints in the Zigbee protocol stack. Overall, the figure highlights WoTC++'s initial efficiency with CoAP in Zigbee environments while also revealing its vulnerability to performance breakdowns under high concurrency, unlike the more stable, though slower, HTTP performance. It is import to note that the error rate is at 20% at highest concurrent invocation and it is removed from the chart for better comparison.

9. ComparisonStudy

This section describes a thorough comparison of WoTC++ and the existing Python-based WoT implementation (WoTPy) for the two binding interfaces—HTTP and CoAP—implemented in the proposal. On WoTC++, virtual clocks are used for benchmarkingcomparison. Similarly, WoTPy have been used for measurements using a sample benchmark application that is a component of the framework.

Comparison on HTTP Binding Interface

Comparison results for HTTP-based Property read response time are presented in Figure 14(a), which is conducted on the P1 platform across twoWoT implementations: WoTC++and WoTPy. The results are measured over varying thread counts (1, 2, 5, 10, 20, 50, and 100), highlighting both performance stability and scalability. In the HTTP protocol, the overall lower and upper whisker response times range from 8 ms to 16 ms, and the WOTC++ response time is closely coupled with its median. When compared to Python-based frameworks, WoTC++ consistently performs better.WoTPy exhibits the least efficient performance, with median response times exceeding 19 ms and a discernible fluctuation up to 36 ms as the number of threads increases. It is important to note that WoTPy's response time occasionally exceeded 2 seconds, even though the outlier is not relevant to the discussion. This result suggests inadequate scalability, likely due to the constraints of interpreted execution and Python's Global Interpreter Lock (GIL).

The CPU load during HTTP-based WoT property reads is also compared across two implementations (WoTC++, and WoTPy) for different numbers of concurrent invocation threads in Figure 16(b). WoTC++ CPU utilization is better around 6–7% at lower concurrent thread counts and increases with increasing concurrent thread requests. However,concurrent requests are not common on low-powered IoT devices like web servers, so

WoTC++ is well-suited for low-power IoT development because it is lightweight and requires few resources. WoTPy, on the other hand, shows much higher CPU consumption with increasing thread counts (over 90% at 100 threads), suggesting potential performance bottlenecks under load.These findings demonstrate that WoTC++ is just as efficient as web application frameworks while using less CPU power, making it much more appropriate for CPU-constrained IoT environments than its interpreted counterparts.

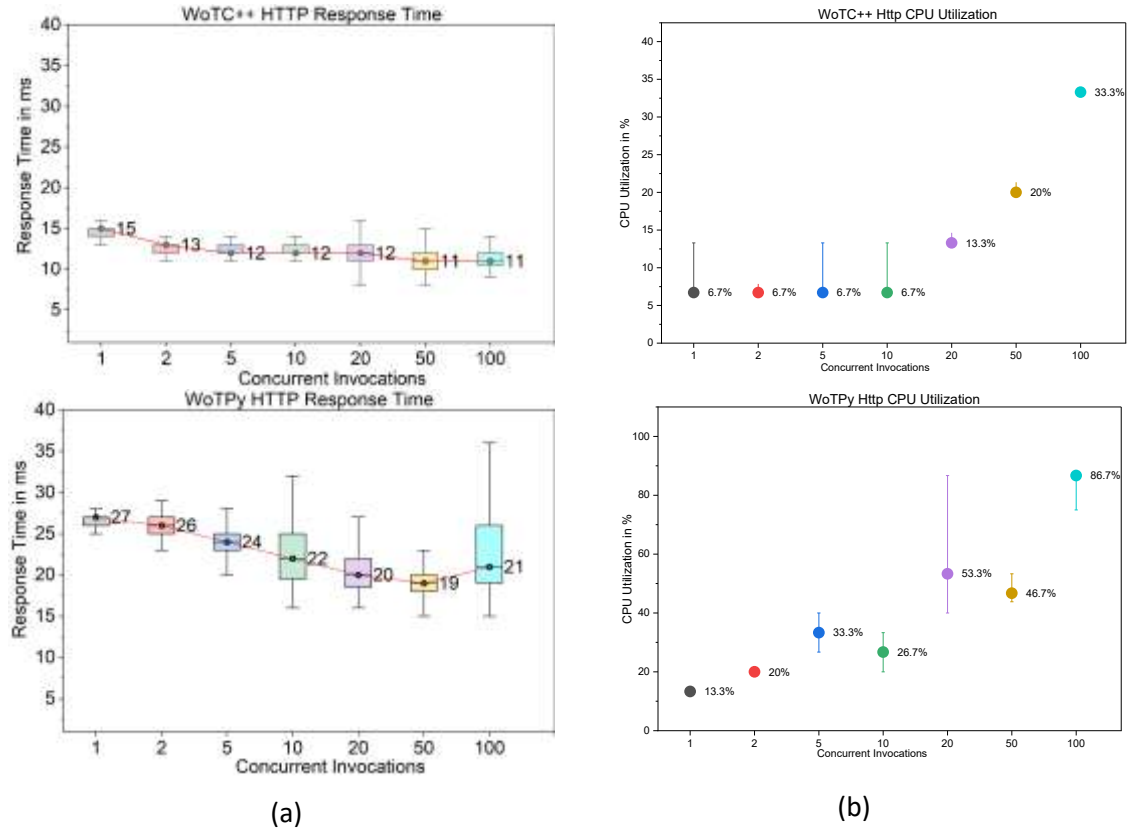


Figure 14: Comparison of HTTP Response Time & CPU Utilization performance across WoTC++, and WoTPy implementations over varying levels of concurrency

Comparison on CoAP Binding Interface

Comparison results for CoAP-based Property read response time are presented in Figure 15(a), which was conducted on the P1 platform across twoWoT implementations: WoTC++, andWoTPy. WoTC++ routinely achieves low median latencies less than 10ms, and narrow interquartile ranges, even under high concurrency. This excellent performance is a result of its design. WOTC++ is still preferable for bare-metal environments. As Python's WoTPy shows the greatest delay and volatility, particularly with more threads, this may be due to interpreter restrictions or inefficient concurrency handling. This identical set of thread levels' associated CPU usage percentages is shown in the right column.

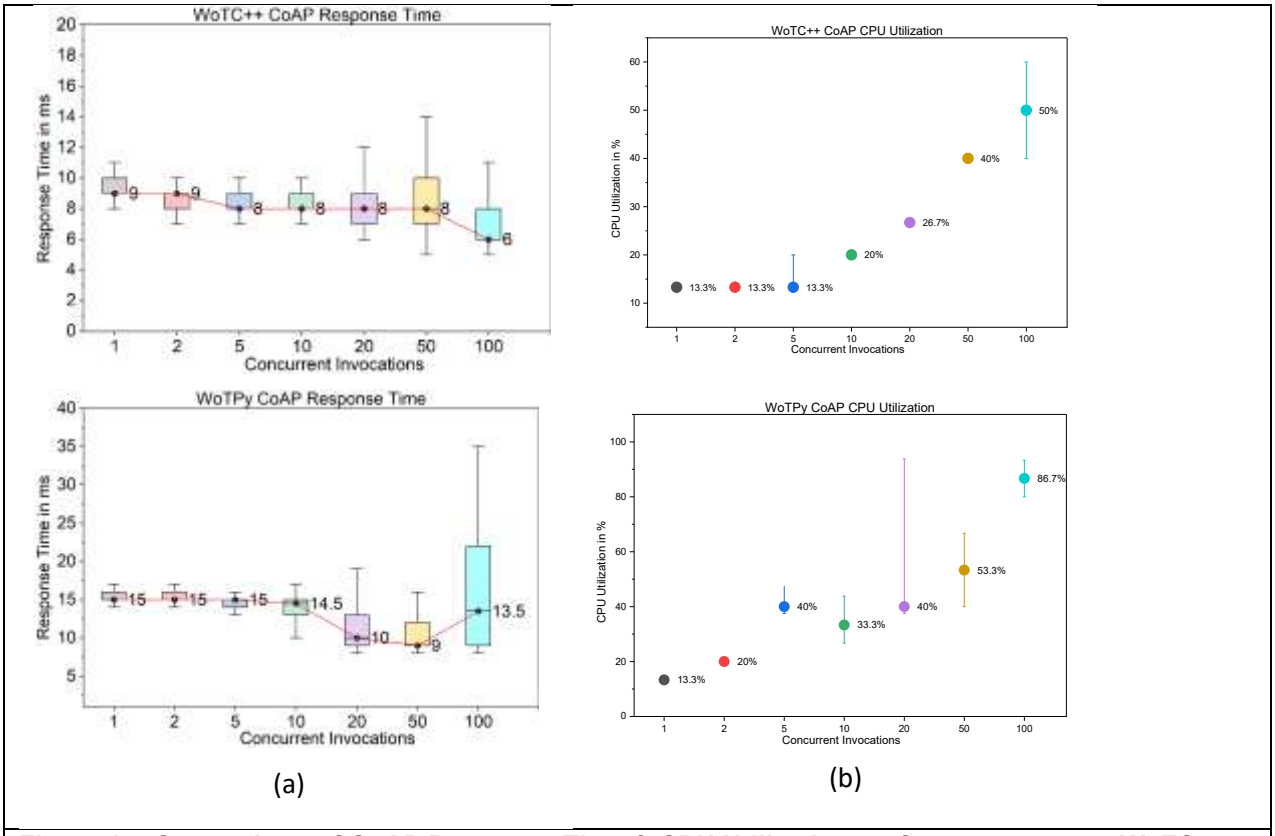


Figure 15: Comparison of CoAP Response Time & CPU Utilization performance across WoTC++, Node-wot, and WoTPy implementations over varying levels of concurrency

Figure 15(b) illustrates the CPU load associated with CoAP-based WoT property read operations across three implementations—WoTC++, and WoTPy—under varying concurrent invocations. Similar to HTTP finding, WoTC++ consistently exhibits low CPU consumption at low concurrency levels, gradually increasing as the number of concurrent threads increases. This observation may be due to limitations in the open-source Internet protocol stack. WOTC++ would be a reasonable choice while making a trade-off between lower power and high concurrent threads. Additionally, integrating optimized commercial solutions can reduce the extensive CPU load and match the performance of a competitive solution.

RAM Utilization

Another advantage of the proposal is RAM utilization and control. Figure 16compares RAM utilization acrosstwo benchmarking applications (WoTC++, and WoTPy). The grey bar representing WoTC++ shows the lowest RAM usage at 0.3%, highlighting its high efficiency and lightweight nature. WoTPy, as indicated by the blue bar, consumes 1.2% of RAM, resulting in a moderate increase in memory usage. This comparison clearly demonstrates that WoTC++ is significantly more memory-efficient, making it better suited for resource-constrained applications. The total size of the platform is 4Gbytes.

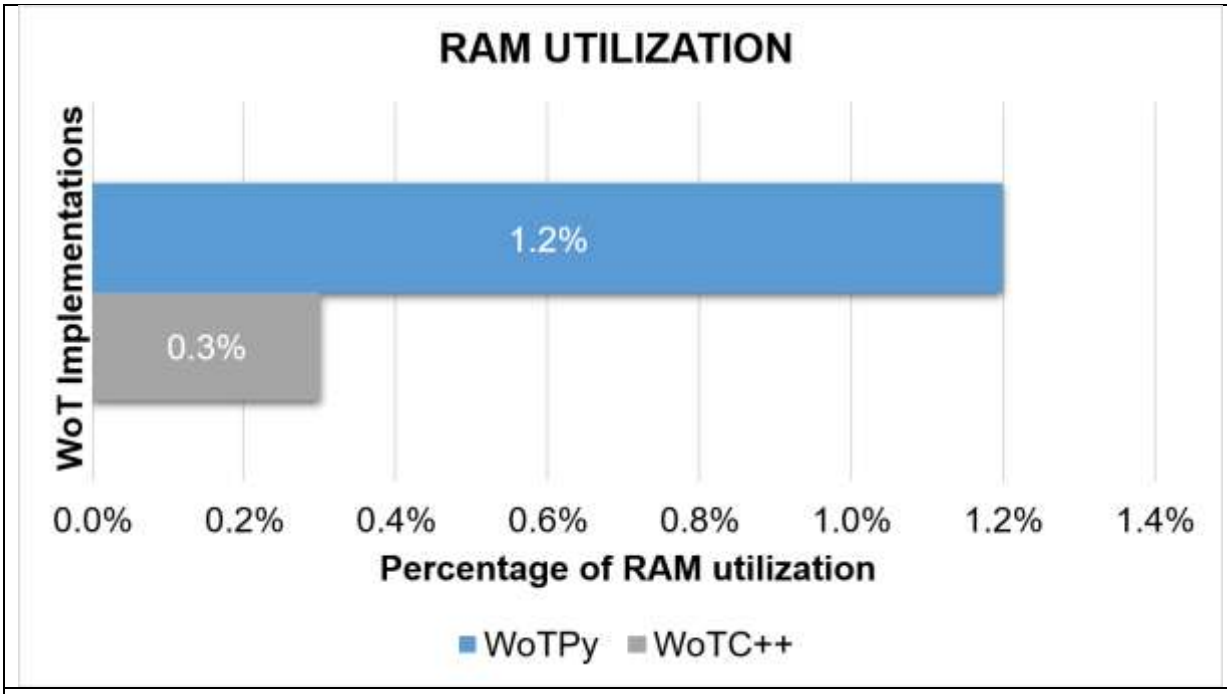


Figure 16: RAM Utilization Comparison

10. Conclusion

This research introduced WoTC++, a lightweight native C++ framework designed to make it easy to integrate WoT features into smart home devices with limited resources. The fundamental approach was creating a runtime environment and communication stack optimized for devices with constrained memory, CPU, and energy resources. WoTC++ uses a configurable, lightweight WoT stack that is portable across platforms.TheWoTC++ framework was evaluated across multiple real-world embedded platforms, including ESP32- and Raspberry Pi–based nodes, representative of constrained devices and unconstrained devices used in smart home environments. The framework demonstrated high operational reliability, maintaining memory usageacross varying thread loads and network types (Zigbee, Wi-Fi, and virtual). Under controlled stress conditions with up to 100 concurrent threads and 100 invocations, WoTC++ achieved amedian property read median response time of less than 20 ms for virtual and Wi-Fi Things, and under 2.2s for Zigbee-based devices, enabling near real-time responsiveness essential for automation tasks. When considering benchmark and constrained ESP32 platform evaluations, WOTC++ is a good fit for resource-constrained devices that prioritize stability and low power consumption over supporting more concurrent sessions. Future research will examine the integration of consumer functions and the security framework. Thus, WoTC++ could be used to build intermediary devices that serve as digital twins, Edge gateways, or proxies.

Author Contributions

Conceptualization, methodology and validation was done by B.S; writing—original draft was done by B.S; writing—review and editing, K.R; supervision, K.R, all authors have read and agreed to the published version of the manuscript.

References

1. L. Turchet and F. Antoniazzi, "Semantic Web of Musical Things: Achieving interoperability in the Internet of Musical Things," Journal of Web Semantics, vol. 75, p. 100758, 2023, doi: 10.1016/j.websem.2022.100758.
2. Zakaria Benomar, Marco Garofalo, Nikolaos Georgantas, Francesco Longo, Giovanni Merlino, and Antonio Puliafito, "Bridging IoT Protocols with the Web of Things: A Path to Enhanced Interoperability," 2024.
3. H.-T. Nguyen, V.-H. Pham, V.-T. Vu, and T.-T. Nguyen, "Web access as a service for CSA matter protocol: Bridging matter and the W3C Web of Things framework for smart home interoperability," Internet of Things, vol. 32, p. 101618, 2025, doi: 10.1016/j.iot.2025.101618.

4. S. Thapliyal, M. Singh, M. Wazid, D. P. Singh, and A. K. Das, "Design of SMOTE Analysis-Based Phishing Attack Detection Scheme for Web of Things Environment," SECURITY AND PRIVACY, vol. 8, no. 2, e70023, 2025, doi: 10.1002/spy2.70023.
1. Belhadi, Y. Djenouri, G. Srivastava, and J. C.-W. Lin, "Fast and Accurate Framework for Ontology Matching in Web of Things," ACM Trans. Asian Low-Resour. Lang. Inf. Process., vol. 22, no. 5, 2023, doi: 10.1145/3578708.
2. Li, H. Fan, Y. Gao, and W. Dong, "WaWoT: Towards Flexible and Efficient Web of Things Services via WebAssembly on Resource-Constrained IoT Devices," IEEE Transactions on Computers, vol. 74, no. 3, pp. 1094–1108, 2025, doi: 10.1109/TC.2024.3500385.
3. L. Sciullo, C. Castiglione, A. Trotta, and M. Di Felice, "WoT on The Extreme Edge (WoTTEE): Enabling W3C Web of Things for Micro-controllers," in 2022 IEEE 8th World Forum on Internet of Things (WF-IoT), 2022, pp. 1–6.
4. J. Domingo, J. I. Panach, and E. Dura, "EmintWeb: Creation of embedded web applications in C++ for specific systems," SoftwareX, vol. 27, p. 101809, 2024, doi: 10.1016/j.softx.2024.101809.
5. P. Branch and P. Weinstock, "Functional Programming for the Internet of Things: A Comparative Study of Implementation of a LoRa-MQTT Gateway Written in Elixir and C++," Electronics, vol. 13, no. 17, p. 3427, 2024, doi: 10.3390/electronics13173427.
6. Aldweesh, "Blockchain-Based Secure Firmware Updates for Electric Vehicle Charging Stations in Web of Things Environments," WEVJ, vol. 16, no. 4, p. 226, 2025, doi: 10.3390/wevj16040226.
7. Chakraborty, M. Khosravi, M. K. Khan, and H. H. Song, "Editorial: Multimodality, Multidimensional Representation, and Multimedia Quality Assessment Toward Information Quality in Social Web of Things," J. Data and Information Quality, vol. 15, no. 3, pp. 1–3, 2023, doi: 10.1145/3625102.
8. J. Llopis, M. Mena, J. Criado, L. Iribarne, and A. Corral, "A faceted discovery model architecture for cyber-physical systems in the web of things," ComSIS, vol. 20, no. 4, pp. 1639–1659, 2023, doi: 10.2298/CSIS230328049L.
9. F. Khan, F. Bukhari, R. Mehras, L. Rehman, and K. Kanwal, "Machines and Algorithms Integrating the Web of Things in Agriculture: Trends, Challenges and Opportunities," 2023.
10. N. Deb, A. Kollu, A. Alferaidi, L. M. Alkwai, and P. Kumar, "Suppressing the Spread of Fake News Over the Social Web of Things: An Influence Maximization-Based Supervised Approach," IEEE Systems, Man, and Cybernetics Magazine, vol. 9, no. 4, pp. 20–25, 2023, doi: 10.1109/MSMC.2023.3276575.
11. M. R. Faheem, T. Anees, M. Hussain, A. Ditta, H. Alquhayz, and M. A. Khan, "Indexing in WoT to Locate Indoor Things," IEEE Access, vol. 11, pp. 53497–53517, 2023, doi: 10.1109/ACCESS.2023.3272691.
12. G. Ortiz et al., "A microservice architecture for real-time IoT data processing: A reusable Web of things approach for smart ports," Computer Standards & Interfaces, vol. 81, p. 103604, 2022, doi: 10.1016/j.csi.2021.103604.
13. N. Ekren, M. Sensoy, and T. C. Akinci, "Smart Buildings Using Web of Things with .NET Core: A Framework for Inter-Device Connectivity and Secure Data Transfer," Information, vol. 16, no. 2, p. 123, 2025, doi: 10.3390/info16020123.
14. Eclipse, node-wot: Eclipse. Accessed: Jun. 29 2025. [Online]. Available: https://github.com/eclipse-thingweb/node-wot
15. Luca Sciullo, Ivan Dimitry Ribeiro Zyrianoff, Angelo Trotta, and Marco Di Felice, "WoT Micro Servient: Bringing the W3C Web of Things to Resource Constrained Edge Devices," 2021, doi: 10.1109/SMARTCOMP52413.2021.00042.
16. Federica Paganelli, Stefano Turchi, and Dino Giuli, "A Web of Things Framework for RESTful Applications and Its Experimentation in a Smart City," IEEE Systems Journal, 2016, doi: 10.1109/JSYST.2014.2354835.
17. D. Guinard, V. Trifa, and E. Wilde, "A resource oriented architecture for the Web of Things," in 2010 Internet of Things (IOT), 2010, pp. 1–8.
18. Ege Korkan, Hassib Belhaj Hassine, Verena Eileen Schlott, Sebastian Käbisch, and Sebastian Steinhorst, "WoTify: A platform to bring Web of Things to your devices," 2019.
19. J. Dongo et al., "domOS: An “Operating System” for Smart Buildings," CLIMA 2022 conference, 2022, doi: 10.34641/clima.2022.437.
20. B. Ostermaier, F. Schlup, and K. Römer, "WebPlug: A framework for the Web of Things," in 2010 8th IEEE International Conference on Pervasive Computing and Communications Workshops (PERCOM Workshops), 2010, pp. 690–695.
21. M. Iglesias-Urkia, A. Gómez, D. Casado-Mansilla, and A. Urbieta, "Automatic generation of Web of Things servients using Thing Descriptions," Personal Ubiquitous Comput, vol. 28, no. 1, pp. 325–341, 2020, doi: 10.1007/s00779-020-01413-3.
22. García Mangas and F. J. Suárez Alonso, "WOTPY: A framework for web of things applications," Computer Communications, vol. 147, no. 4, pp. 235–251, 2019, doi: 10.1016/j.comcom.2019.09.004.

23. Manta-Caro and J. M. Fernández-Luna, "IR.WoT: An Architecture and Vision for a Unified Web of Things Search Engine," Sensors (Basel, Switzerland), vol. 24, no. 11, 2024, doi: 10.3390/s24113302.
24. Kunihiko Toumura, Michael Lagally, Michael McCool, and Ryuichi Matsukura, "Web of Things (WoT) Architecture 1.1," W3C Recommendation, W3C, 2023.
25. Johannes Hund, Zoltan Kis, Kazuaki Nimura, Daniel Peintner, and Cristiano Aguzzi, "Web of Things (WoT) Scripting API," W3C Note, W3C, 2023.
26. Michael McCool, Ege Korkan, and Sebastian Käbisch, "Web of Things (WoT) Thing Description 1.1," W3C Recommendation, W3C, 2023.
27. S. E. Mathe, H. K. Kondaveeti, S. Vappangi, S. D. Vanambathina, and N. K. Kumaravelu, "A comprehensive review on applications of Raspberry Pi," Computer Science Review, vol. 52, no. 10, p. 100636, 2024, doi: 10.1016/j.cosrev.2024.100636.
28. POCO C++ Libraries - Simplify C++ Development. Accessed: Jun. 29 2025. [Online]. Available: https://pocoproject.org/
29. Olaf Bergmann, libcoap: A CoAP (RFC 7252) implementation in C: libcoap. Accessed: Jun. 29 2025. [Online]. Available: https://libcoap.net/
30. Apache Software Foundation, Apache JMeter. Accessed: Jun. 29 2025. [Online]. Available: https://jmeter.apache.org/
31. Tanganelli, jmeter-coap: JMeter Plugin to Evaluate CoAP Servers. Accessed: Jun. 29 2025. [Online]. Available: https://github.com/Tanganelli/jmeter-coap
32. OriginLab Corporation, OriginPro Learning Edition. Accessed: Jun. 29 2025. [Online]. Available: https://www.originlab.com/OriginProLearning.aspx
33. Suresh kumar. (2025). Integration of Renewable Energy Sources with Electric Drive Systems: Control Challenges and Performance Analysis. National Journal of Electric Drives and Control Systems, 33-42.
34. Mrunal Salwadkar, &P.Dineshkumar. (2025). Smart City Waste Management Using Sensor-Driven IoT Architecture and Predictive Analytics. Journal of Wireless Sensor Networks and IoT, 3(1), 48-55.
35. K P Uvarajan. (2025). Physics-Constrained Uncertainty-Aware Learning Architectures for Resilient Context Inference in Pervasive Intelligent Systems. National Journal of Ubiquitous Computing and Intelligent Environments, 2(3), 36–44.
36. Krishnapriya K S. (2026). Consumer Trust and Privacy Concerns in Digital Markets: Implications for Online Purchase Behavior. Journal of Social Science and Management Studies, 1-8.