Universiteit van Amsterdam

Vrije Universiteit Amsterdam

Master Thesis

Energy Consumption Analysis of Application Layer Protocols in Realistic IoT Uplink Scenarios: An Empirical Study on an ESP32 Microcontroller

Author: Daniele Quartinieri (UvA:14843986 VU:2767570)

1st and 2nd supervisor: Dr. Marios Avgeris, Dr. Adam Belloum
2nd reader: Dr. Shashikant Ilager

A thesis submitted in fulfillment of the requirements for the joint UvA-VU Master of Science degree in Computer Science

March 24, 2026

Energy Consumption Analysis of Application Layer Protocols in Realistic IoT Uplink Scenarios: An Empirical Study on an ESP32 Microcontroller

***Abstract*—IoT has affected plenty of fields; its applications range from domestic use to industrial environments. However, the success of these applications hinges on critical design decisions regarding hardware platforms and the software running on them due to the limited hardware resources and energy budgets that characterize IoT devices. This paper aims to assess the energy usage of different application layer protocols in possible IoT uplink scenarios. To do so, an ad hoc bench was built with an ESP32 acting as the data sender, an INA226 acting as the power data collector, and an Arduino Pro Micro acting as the data logger. Three increasingly demanding benchmarks were chosen from real-world scenarios: light (0.56 bps), intermediate (0.43 kbps), and heavy (14 Mbps). Four different protocols were tested: MQTT (QoS0, QoS1, QoS2), CoAP, WebSocket, and HTTP. Protocols tested were deemed significant over the others only if statistically significant and with a median difference exceeding the noise threshold. Most protocols tested under light benchmark showed no significant differences, only Websocket clearly resulted more energy hungry than the other protocols tested. In the intermediate benchmark, we observed a different pattern, most protocols showed significant differences in energy usage. CoAP performed best, followed by Websocket, MQTT QoS0, MQTT QoS1 and QoS2 (practically indistinguishable from QoS1) and finally HTTP, which was the most energy hungry. These results help developers make informed, constraint-driven choices: when uplink data rate is low, protocol selection is mainly driven by factors such as deployment constraints, since the choice of an application-layer protocol over another does not generally seem practically significant; as data rates increase, however, protocol choice can heavily affect energy usage.**

***Index Terms*—IoT, TCP/IP, network, protocols, OSI, RTOS, energy, power, ESP32, INA226, Arduino, benchmark, MQTT, HTTP, CoAP, WebSockets.**

I. Introduction

THE Internet of Things (IoT) is considered one of the most significant technological developments of the last two decades. IoT has found plenty of applications in recent years, ranging from home automation [1] to industrial environments such as agriculture [2] and, more broadly, in Industry 4.0 contexts [3]. The wide range of usage scenarios has led to the development of multiple types of hardware and protocols in order to adapt to a wide series of situations.

However, this variety of options brings the developer in front of a series of challenging choices, ranging from selecting an appropriate hardware platform, to determining whether to use a real-time OS (RTOS) and, if so, which one. Additionally, the developer might have to deal with the selection of a network stack and one or more libraries to make it work.

Each of these decisions can significantly affect the energy consumption and resource usage of an IoT device, two of the main constraints of such systems. Energy efficiency is repeatedly identified as a core factor for IoT devices because they often operate under limited resources and battery power constraints [4], [5]. Accordingly, the software stack interacting with these devices must account for such restrictions and avoid introducing significant overhead [6]. In the context of application-layer protocols such as MQTT, CoAP, and AMQP, [7] highlights that protocol choice is highly application-specific and that protocols must be compact to fit on constrained devices, which often operate over error-prone channels [5]. The focus on efficiency extends below the application layer; as noted in [8], network encapsulation and routing protocols must be efficient in terms of power consumption and conservative in transmitter energy usage. Given the limited resources and energy restrictions of IoT devices, solutions such as lightweight IP stacks and low-power modes are usually required [9]. For this reason, energy consumption is often treated as a key metric when comparing IoT protocol solutions, including at the application layer [10].

Building on the above constraints and the practical need to choose among competing protocol options, this paper investigates two research questions (RQs) across three real-world-inspired IoT scenarios (see IV-B2) on the ESP32, a widely adopted and resource-constrained microcontroller platform [11], extending existing practical studies with empirical evidence across diverse use cases (see Table IV).

- **RQ1** Does the choice of application-layer protocol over another significantly influence the energy consumption of the board?
- **RQ2** In terms of energy usage, are the protocols consistent across all benchmarks, or does a particular protocol perform best under specific conditions?

In Section II (Background), we present the rationale for selecting the protocols evaluated in this study and provide a brief introduction to real-time operating systems (RTOSs). Section III (Related Work) reviews prior research, summarizing what similar studies have measured or focused on and, where possible, the methodologies they employed. In Section IV (Methodology), we describe the hardware and software environment, outline the characteristics of the benchmarks and metrics used, and explain our data collection procedure along with its design and rationale. Section V (Implementation / Execution) details the experimental design and the mitigation techniques adopted to address carry-over effects. Provides also details on how to replicate the tests on our hardware. Section VI (Results and Data Analysis) presents and analyzes the data obtained from our three benchmarks. In Section VII (Discussion), we interpret these findings and

explore their implications. Section VIII (Threats to Validity) examines potential factors that might undermine the validity of our research. In Section IX (Conclusion), we summarize our key findings by answering the RQs and highlight our contributions. Finally, in Section X (Future Work), we discuss possible extensions and directions for future research building upon this work.

II. Background

A. *Common communication protocols in IoT*

To determine which protocols to include in our benchmarks, a brief literature study was conducted on the most cited protocols in papers published between 2019 and 2025. The study was carried out using the search terms 'IoT protocols literature review' and 'IoT protocols overview' on Google scholar and IEEE Xplore. The results are presented in Table I. By selecting only the protocols mentioned 50% or more of the time, avoiding closed-source technologies, and excluding protocols that cannot run on our ESP32 board without additional hardware, the selection narrows down to the following application-layer protocols: HTTP/HTTPS, MQTT, AMQP, CoAP, DDS, and XMPP. To ensure a fair comparison, external factors such as the RTOS (the operating layer used when not running bare metal) needed to be consistent across all protocols. This consideration, merged with the possible wide adoption of the libraries tested, given the board we were working with, led us to opt for protocols relying on libraries running through Arduino Core on ESP-IDF, as shown in Fig.1. This further narrowed down the selection to HTTP/HTTPS, MQTT, and CoAP, to which the WebSocket protocol was added due to the authors' interest in its performance.

Protocol	[12]	[13]	[14]	[4]	[15]	[16]	[17]	[18]	[19]
HTTP/HTTPS	Y				Y	Y		Y	Y
MQTT	Y		Y	Y	Y	Y	Y	Y	Y
WebSocket								Y	
AMQP	Y			Y	Y	Y	Y	Y	Y
CoAP	Y		Y	Y	Y	Y	Y	Y	Y
Lorawan		Y			Y		Y		
Zigbee		Y	Y	Y	Y		Y		
6LoWPAN	Y	Y	Y	Y	Y	Y	Y		Y
Thread									
Matter									
CORBA	Y								
OPC UA	Y								
DDS	Y			Y	Y		Y	Y	Y
ZeroMQ	Y								
XMPP	Y				Y	Y	Y	Y	Y
Zwave		Y	Y		Y		Y		Y
802.15.4		Y	Y	Y	Y	Y	Y		Y
Lora		Y			Y				
Sigfox		Y	Y		Y		Y		
NBIoT		Y			Y		Y		
LTE		Y							
5G		Y							
BT		Y							
BLE			Y		Y				Y
WirelessHArt			Y		Y				
LLAP				Y					
SMCP				Y					
TSMP				Y					
LWPAN					Y				
Modbus					Y	Y			
DNP3					Y				
OLE					Y				
mDNS									Y
DNS-SD									Y
RPL									Y
EPCglobal									Y
LTE-A									Y

TABLE I: Protocols mentioned by papers (protocols that were specifically mentioned as not relevant have not been considered)

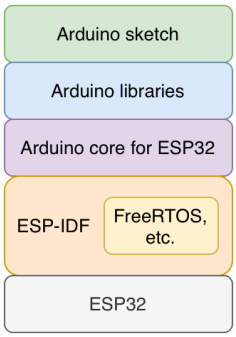

Fig. 1: Simplified view of the software stack for Arduino code execution on ESP32. The original FreeRTOS, also referred to as vanilla FreeRTOS, is a single-core RTOS. To support dual-core ESP targets (like our board), it has been modified and integrated into ESP-IDF (Espressif IoT Development Framework) [20], forming what is known as IDF FreeRTOS [21] or ESP-IDF FreeRTOS [22]. In order to use it through the Arduino IDE with Arduino libraries, we rely on a compatibility layer [23], the Arduino core for ESP32.

B. *RTOS*

An RTOS is an operating system whose correctness depends on both the logical accuracy of its output and the time at which that output is produced. RTOSs have found application in commercially available embedded systems such as IoT and consumer electronics, and represent the vast majority of the IoT OS market, a market that has been expanding at a compound annual growth rate of 41% from 2017 to 2023 [24].

RTOSs found wide usage in resource constrained devices (like the ESP32 board used in this paper) in those situations where the bare metal approach of using a "combination of superloop that handles non-time-critical work, with time-critical work being done in interrupt handlers" [25] would become difficult due to complexity of project. For example, in the case of a networked device interacting with multiple devices, traffic could be unpredictable and or bursty, a multitasking approach would be key.

Relying on an RTOS grants access to a framework and its abstractions to handle the challenges that multitasking can introduce. These abstractions support scheduling, task prioritization, timing constraints, and preemption, allowing us to ensure that time critical operations are executed in deterministic and predictable manner. RTOSs can offer also other features such as file systems services, communications services, and security services [25].

Kernel-wise, many RTOS designs adopt a microkernel architecture, which consists in running minimal OS services (e.g., memory management and inter process communication) in kernel space, while keeping other services such as networking in user space. This approach results to be easier to

maintain and more stable compared to traditional architectures like monolithic kernel where most services run in kernel space [24].

A further classification on RTOS kernel can be done between tickless and tick-based kernels. While a tick-based kernel periodically interrupts the CPU at a predetermined frequency (regardless of power state of CPU) the tickless kernel replaces periodic interrupts with on-demand interrupts [26]. Thanks to the on-demand interrupts tickless kernel allows idle CPU's in low power state to remain in such state until a new task is scheduled for processing.

Schedulers can be classified into two main groups, cooperative and preemptive. In preemptive scheduling, the running task can be interrupted by the scheduler in order to allow other tasks to run [27] [28]. This may be based on a round robin approach (e.g., if tasks have same priority) or on a priority based approach to let resources to higher priority tasks. In contrast, in cooperative scheduling, it is up to the tasks to yield themselves in a cooperative manner.

RTOSs can be classified in two macro groups depending on the strictness of timing constraints provided. The two groups are:

- Hard RTOS, where not being able to respond to an input within a specific time frame can be considered a system failure eg. causing catastrophic events [24].
- Soft RTOS response time is still bounded but less strictly constrained, hence missing a deadline is undesirable but tolerable since would just bring to a degradation of system [24].

III. Related work

There are several papers concerning energy consumption of application layer protocols [18], [29]–[35], and even more simply analyzing their performances [36]. Among the papers concerning energy usage, common conclusion is lower energy consumption of CoAP over MQTT, and of MQTT over HTTP, as we can see from [18], [29], [30], [33]–[35]. This result appears to be justified by the fact that protocol overhead heavily affects energy usage [31], and CoAP has the smallest overhead of the three, followed by MQTT and HTTP [34]. Among the reasons why CoAP has the smallest overhead of the three is its use of UDP at the transport layer, which enables a fire and forget approach [34] rather than TCP's connection setup and acknowledgments. Among the advantages of MQTT over HTTP, in addition to the smaller overhead mentioned earlier, is its interaction pattern. As stated in [33], the publish-subscribe (pub/sub) pattern used by MQTT, compared with the request-response (req/res) pattern typically used by HTTP, allows energy savings, primarily because it avoids repeated active work on the consumer side.

It is worth highlighting that, despite what was previously stated about CoAP, MQTT and HTTP, not all papers are aligned in their findings. An example concerns the AMQP protocol: both [29] and [34] covered AMQP and HTTP, but [29] in its tests on energy usage and network traffic, reported AMQP as the worst performing protocol, even suggesting excluding it from future measurements. By contrast, in [34]'s qualitative analysis, Naik classifies AMQP, in terms of energy consumption, as better performing than HTTP.

Most of the papers reviewed mentioned MQTT QoS levels but did not analyze the performance of all of them. QoS (Quality of Service), in the context of MQTT, defines the delivery guarantee for a message between a publisher and a subscriber. With QoS0, the message is delivered at most once, with no guarantees. With QoS1, the message is delivered at least once, with retransmissions until an acknowledgement is received. With QoS2, the message is delivered exactly once using a four-way handshake. The few studies that evaluated all QoS levels reported increasing energy consumption as QoS increased [29], [33]. With Eyers et al. [33] observing an increase of 27% from QoS0 to QoS1, 60% from QoS1 to QoS2, and 92% from QoS0 to QoS2.

Few works evaluate MQTT-SN, a lightweight variant of MQTT designed for sensor networks to save resources and reduce power consumption [18]. It typically operates over UDP and uses topic identifiers to reduce message overhead [18], [32], [35]. Studies report better energy performance compared to CoAP [30], [32] as well as to MQTT (no QoS specified) and HTTP [30].

A few studies consider HTTP/2, the successor to HTTP/1.1, which introduces header compression and the possibility of multiple concurrent exchanges on the same connection (multiplexing). These features, among others, allow HTTP/2 to reduce latency and overhead, making it particularly attractive for constrained devices [35]. Overall HTTP/2 improvements make it consume less energy than its predecessor HTTP/1.1 [29]. In the remainder of this paper, unless otherwise specified, HTTP denotes HTTP/1.1; the same generic term is used when cited works do not specify the HTTP version.

From the reviewed literature, the following gaps emerge:

- **Lack of broad protocol coverage under consistent conditions.** Dizdarević et al. [35] note a lack of studies that evaluate a broad range of protocols under consistent conditions. This is also reflected in our recap Table II: among papers that run their own tests, most evaluate only two to three protocols [31]–[33].
- **Benchmark characteristics being often arbitrary rather than scenario driven.** Across the analyzed studies, when provided, payload sizing proves very heterogeneous, both in terms of size and in terms of rationale behind it. Some works do not report payload sizes at all, as in the case of [30], while others adopt a small set of pseudo-representative values without a particular justification, as in the case of [29] which uses packets with either 10 bytes or 1 kB payloads, or [36] which spans from 10 B up to 1 MB. A smaller subset provides an explicit rationale: [31] frames message sizes as a progression from a "typical sensor value" (6 B) to increasingly large, human readable content (sentence, paragraph, and ultimately large documents up to 1 MB), whereas [33] starts from a presumed common IoT payload (24 B) and scales it (48, 240, 1320, 1560 B), motivating the two largest sizes via the MTU boundary (Maximum Transmission Unit, i.e., the largest packet size that can be carried without fragmentation). Krug et al. [32] instead,

use payloads of 21 bytes for data and 17 bytes for status of packets, specifically to avoid fragmentation. Even among studies that justify payload sizes, the baseline used to define a usual message varies: as noted, [31] anchors it to a minimal sensor reading (6 B), whereas [33] assigns it to a "typical IoT payload" (24 B), and the rationale alternates.

A similar lack of consistency appears in transmission frequency. Many studies do not specify a fixed sending rate [31], [36], [29] (except for the MQTT inter-message delay); some rely on an ACK-driven frequency, as in the case of [36], which sends the next payload only after the previous packet has been acknowledged. Where rates are explicitly controlled, the choices vary: [33] evaluates 1–16 messages per second, while [32] adopts a much lower cadence (every 15 minutes).

Overall, both payload size and message frequency are heterogeneous across the literature. Regarding payload sizes, some overlap is observable on powers of 10 values, but this convergence appears incidental. Such heterogeneity would not be a problem if protocol behavior scaled similarly across payload sizes and message frequencies but as stated by [29], [36] payload size and frequency cause different behaviors in different protocols.

- **Hardware and software stacks being often not representative of constrained IoT deployments.** Stefanec and Kusek [29] utilize a Raspberry Pi for testing; however, as they note in their future work section, the tests should be run on more constrained devices, such as our ESP-32[1]. They are not the only ones, plenty of the papers that executed their own tests relied on Raspberry Pis, like in [31], [36], or even desktop computers, as in [33]. Some relied on simulations, as in [30], in which case the representativeness of the software stack depends on the accuracy of the simulation used. Overall, the lack of usage of constrained boards (practical or simulated) implied also the lack of usage of constrained software stack, which may affect the results not only in terms of energy consumption but also in terms of feasibility.

Overall, we believe that a more realistic representation of the tasks the hardware would execute within its limitations, might provide a more complete and reliable picture of its energy performance.

Source	Protocols covered	Platform	Main Focus
D. Glaroudis et al. [18]	MQTT, COAP, XMPP, AMQP, DDS, REST HTTP and Websocket	No platform, is a review	Latency, bandwidth and throughput, Energy/Power consumption
T. Stefanec and M. Kusek [29]	HTTP, HTTP/2, CoAP, MQTT (QoS0–2), AMQP	Real HW (Raspberry Pi 3B)	Energy use & network traffic vs payload/QoS
A. Shahrokhi and M. Ahmad [30]	MQTT, MQTT-SN, CoAP, HTTP	Simulation (COOJA/-Contiki, Z1 mote)	Power consumption in controlled simulation
J. Hofer and S. Pawaskar [31]	MQTT vs REST (HTTP/S)	Real HW (Raspberry Pi)	Energy consumption, throughput and overhead
S. Krug et al. [32]	CoAP, MQTT-SN, Custom	Analytical + experimental (not specified)	Energy usage of complete stacks and of specific tasks they execute
D. Eyers et al. [33]	MQTT (QoS0–2), HTTP	Real HW (Laptop and desktop)	Payload size energy, interaction pattern energy & protocol energy
N. Naik [34]	MQTT, CoAP, AMQP, HTTP	No platform, is a qualitative analysis	Pros and cons of protocols aspects like overhead, QoS, power usage, security etc.
J. Dizdarevi´c et al. [35]	REST/HTTP, MQTT, CoAP, AMQP, DDS, XMPP, HTTP/2	No platform, is a survey	Characteristics of protocols and performances in latency, bandwidth, energy consumption, security etc.
U. Tandale et al. [36]	CoAP, MQTT (QoS0–2), REST/HTTP	Real HW (Raspberry Pi 3)	Bandwidth & time performance over 4G/broadband

TABLE II: Summary of related work: protocols covered, platform used (if any), and the paper's main focus (not necessarily for every listed protocol).

IV. Methodology

A. *Hardware and Software Setup*

In order to address the research questions, we needed to run the benchmarks and collect power consumption metrics from the benchmark runs. To do so, we relied on a series of components. These components can be classified into four macro groups based on the tasks they perform: 1) Measurements & Data Acquisition, 2) Monitored Uplink, 3) Network Infrastructure, and 4) Data Reception (see Fig. 3 for hardware overview). From this point forward in this paper, we will refer to the core components of our experiments as our test bench (see Fig. 4). Our test bench was composed of: a board to run the benchmarks on (ESP32), a board to measure power consumption (INA226) of benchmarks and a board to log the measures on our laptop (Arduino Pro Micro) (see Fig. 2).

A deeper view into the board's specifications and capabilities is provided below, highlighting how the boards features support our requirements:

- **ESP32** ESP32 is a System on Chip (SoC) microcontroller developed by Espressif Systems, and can be considered a successor of ESP8266 [38]. Since its introduction, several variants of the ESP32 SoC have been released, each featuring different specifications. The module used in our research was the ESP32-WROOM-32 with SoC ESP32-D0WDQ6 hence equipped with dual-core Xtensa 32-bit LX6 microprocessor. The system incorporates 448 kB of ROM for booting and core functions, 520 kB of on-chip SRAM and 4 MB of integrated SPI flash memory. The module has an average operating current of 80 mA. The ESP32 can operate in 5 power modes [39]:

[1]We refer to our ESP32 as a constrained device conscious of the fact that the ESP32 specifications are well over RFC7228 [37] classification for Class 2 devices (since is over the ∼250 KiB of flash memory and ∼50 KiB of ram). We do so because, as per RFC7228, the boundaries proposed would shift over time, and RFC7228 was proposed in 2014, more than 10 years before the writing of this paper. Moreover, as stated by RFC7228, devices that exceed Class 2 specifications can still be constrained in terms of energy usage. Plus the ESP32 fits well in the description provided for Class 2 devices, which states that such devices are fundamentally capable of supporting most of the same protocols stacks as notebooks and servers while still benefiting from lightweight and energy efficient protocols.

  - Active mode: chip radio powered on, can receive, transmit and listen
  - Modem-sleep mode: CPU operational but WiFi/BT baseband and radio are disabled
  - Light-sleep mode: CPU is paused (state retained). RTC memory, peripherals and ULP (ultra low power co-processor) stay running, wake up events will wake up the chip
  - Deep-sleep mode: CPU off, ULP functional, RTC memory and peripherals on, WiFi and BT data are stored in RTC memory.
  - Hibernation mode: Only RTC timer and RTC GPIO's remain active as wake up sources.

  With Wi-Fi enabled, the chip will switch between Active and Modem-sleep modes. In Modem-sleep, the CPU frequency will automatically adjust to the load [39]. For wireless connectivity, the module supports Wi-Fi 802.11 b/g/n with data rates up to 150 Mbps in the 2.4 GHz band as well as BLE [39]. ESP32 provides also GPIO's, PWM channels and supports various interfaces like I2C, SPI and UART [38]. For our research, two ESP32-WROOM-32 modules were used: one acting as a sender, monitored through INA226 and logged via Arduino Pro Micro, and another unmonitored module used only to enable the monitored ESP32 REF.2 to communicate in our network-related benchmarks.
- **INA226** The INA226 is a current shunt and power monitor with an I2C - or SMBUS-compatible interface. [40]. It is capable of measuring the voltage drop across a shunt resistor through IN+ and IN- pins and to measure bus voltage via VBUS pin. Then INA226 is capable of calculating through its internal ADC the current and the power. Bus voltage, shunt voltage, current and power values are respectively accessible at registers 02h, 01h, 04h and 03h of the INA226.
- **Arduino Pro Micro** The Arduino Pro Micro is a micro-controller based on ATmega32U4, has a clock speed of 16 MHz, a flash memory of 32 KB and 2.5 KB of SRAM [41]. It is used to collect measurements through I2C from INA226.

In order to power the ESP32, a USB-C power delivery sink trigger board is used. The trigger board relies on a CH224K USB-C PD controller to negotiate the output voltage from the power adapter with the power adapter. In our setup has been set to negotiate voltage at 5V. See Fig. 2 for the test bench hardware setup.

Given the advantages that an RTOS provides over the bare metal approach, see Section II, we opted to rely on an RTOS. In order to choose the RTOS to use, a series of RTOSs has been taken into consideration from sources released in the last 10 years (2015-2025): FreeRTOS, RIOT, Contiki, Contiki-ng, TinyOS, mbedOS, Zephyr, µC/OS-II, µC/OS-III and Azure RTOS ThreadX (also known as Eclipse ThreadX), [42] OpenWSN, NuttX, ecos, uCLinux, ChibiOS/RT, CoOS, nanoRK and Nut/OS [43].

By selecting only RTOSs with recent contributions (within the year), with more than 500 stars on GitHub, and no

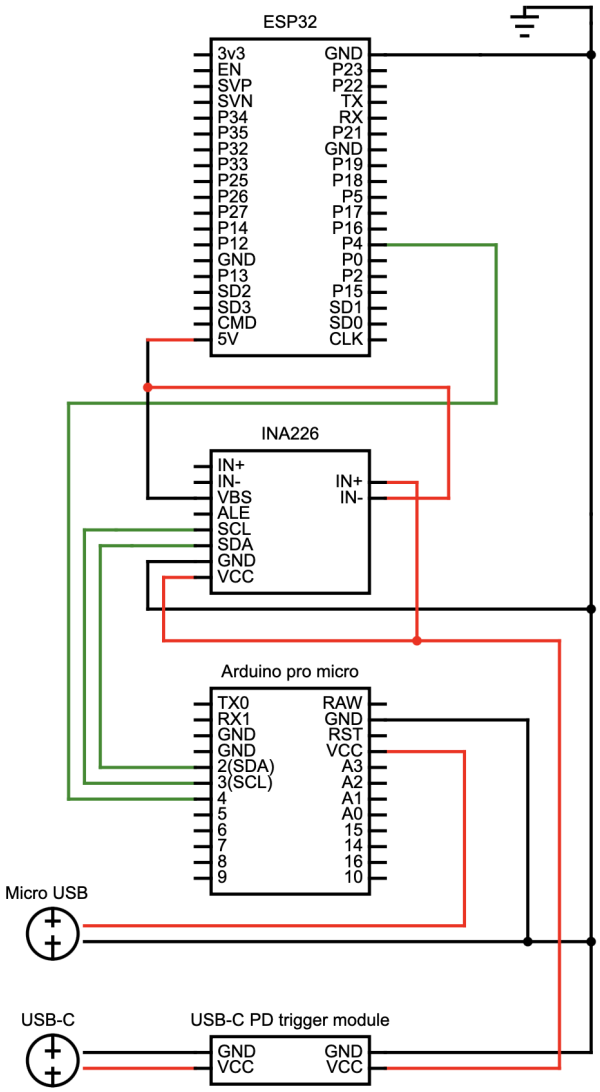

Fig. 2: Test bench hardware wiring layout

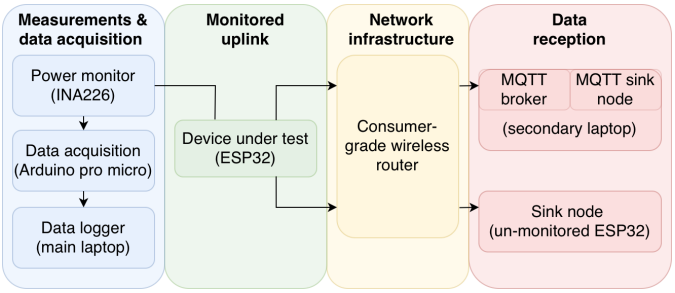

Fig. 3: Overview of the hardware employed to conduct our tests and their roles.

announced end of life (as is the case for Mbed OS), the selection narrows down to FreeRTOS, Zephyr, NuttX, RIOT, TinyOS, Eclipse ThreadX, ChibiOS/RT and Contiki-ng.

Given the board used, ESP32 based on Xtensa, the selection of RTOSs has been narrowed down to RTOSs that appear to have out-of-the-box (OOTB) support for our board. This leads us to select ESP-IDF FreeRTOS [44] [22] (also due its

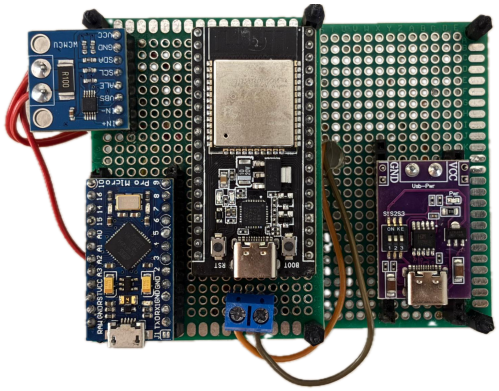

Fig. 4: From left to right: INA226, Arduino Pro Micro, ESP-32 board, USB-C power-delivery trigger board.

popularity in its non ESP-IDF version [45]), Zephyr [46] Nuttx [47] and RIOT [48]. See Table III for a brief overview of RTOS characteristics.

OS	Kernel arch.	Scheduler	Network
ESP-IDF FreeRTOS	microkernel [43]	Tick-based preemptive	LWIP
Zephyr	nanokernel + microkernel [27]	Tick-based preemptive	Native
Nuttx	microkernel or monolithic [43]	Tick-based preemptive	Native
RIOT	microkernel [49]	Tickless Preemptive	GNRC [27]

TABLE III: characteristics of RTOSs evaluated

As mentioned in Section II-A, the potential wide adoption of the libraries tested, the need for support of the application-layer protocols we aimed to evaluate, and the requirement for RTOS to support OOTB the board we were using led us to choose ESP-IDF FreeRTOS used through the Arduino core see Fig. 1.

B. *Benchmarking Tools, Metrics and criteria*

1) Tools and Metrics: In order to answer the research questions we need to collect power consumption metrics from our monitored ESP32 executing benchmarks. In order to obtain this data we rely on INA226. First we collect power through INA226 internal ADC [40]. The power (W) represents the current (A) multiplied by bus voltage (V). Then we calculate the energy. The energy (mWh or J) is obtained by integrating power over time; since power is sampled at discrete time instants, energy is approximated using the rectangular rule. It is worth noting that the minimum amount of current detectable by the INA226 depends on the sizing of the shunt resistor. More details on INA226 measurement equations, data it can collect and shunt resistor sizing are available at Appendix A.

In order to do the comparisons necessary for our study, inspired by studies that measured energy usage in IoT contexts [29] [32] the metric of Energy (J) usage is collected.

2) Criteria: In order to ensure the relevance of our measurements, we designed a set of benchmarks inspired by real world scenarios. These scenarios were selected for their significantly different data rate requirements, while all of them share concern for energy consumption. The real world scenarios collected and the technical limits of the test board (ESP32) shaped the data rate (defined as payload size divided by transmission interval), the payload size and the transmission interval for our benchmarks.

In the *light benchmark*, payload size represents data returned from a humidity and temperature sensor like a DHT11; [50] presents 4 payload sizes from meteorological sensors: 2 bytes for temperature only, 4 bytes for temperature + humidity, 8 bytes for temperature + humidity + pressure and 12 bytes for timestamp + complete data. The choice has fallen to temperature + humidity since we thought that 4 bytes were a reasonable lower bound for our benchmarks payloads, once formatted in JSON for transfer e.g., `{"t": 24.5, "h": 60.2}` would become 21 bytes. For the transmission rate a consumer weather station has been taken as example, bringing us to opt for a transmission rate of 5 min [51]. This would bring us to a data rate of 0.56 bps with *light benchmark*.

In the *intermediate benchmark*, payload size represents the data necessary to track bird movement. While in [52] only transmission frequency is specified, with GPS data collected every 3 seconds in the high-frequency case, the payload is not specified but can be inferred from the following fields: id, timestamp, latitude, longitude, altitude, speed, battery, satellite number. The payload once JSON formatted could be for example: `{"id": 53100, "timestamp":"2025-08-07T12:30:45Z", "latit": 43.47126774300904, "longit": 11.28529038302559, "altit": 578, "speed": 3.2, "batt": 5, "satnumb": 3}`. Given the payload of 160 bytes and the transmission every 3 seconds we could expect a datarate of 0.4267 kbps.

The *heavy benchmark* represents a data stream derived from a UAV (unmanned aerial vehicle) vSLAM (Visual Simultaneous Localization and Mapping) use case. Requirements in [53] and [54] specify RGB-D (Red, Green, Blue + Depth) at 480p or 720p resolution and at least 30 fps, requiring a minimum bandwidth of 350 Mbps. Since this exceeds the ESP32's hardware capabilities (Section IV-A), the transmission specifications were reduced. Following [55], the RGB and depth channels can be compressed separately down to 14 Mbps for RGB at 480p/30 fps using JPEG lossy compression, and 40 Mbps for depth using RVL compression. This work focuses exclusively on the RGB stream, which at 480p and 30 fps yields 14 Mbps and a per-frame payload of approximately 58 kB.

A summary of benchmark characteristics is provided in Table IV.

Benchmark	payload	frequency	datarate
Light	21 bytes	5 mins	0.56 bps
Intermediate	160 bytes	3 seconds	0.43 kbps
Heavy	58 kB or 0.462 Mb	0.033 seconds	14 Mbps

TABLE IV: Benchmark characteristics: payload used, frequency at which payload is sent and resulting datarate.

C. *Data Collection Procedure*

*1) **Data Acquisition and Logging**:* Data was collected on a laptop through our Arduino Pro Micro thanks to serial

communication over USB. The data transferred through serial was formatted in CSV by the Arduino and saved upon arrival on the laptop. Data obtained and scripts used are available at [56]. The Arduino logged the metrics collected via I2C from the INA226, along with information about the run number and benchmark being run obtained by the ESP32 through GPIO[2]. More details are provided in Section V.

*2) **Sampling Rate rationale**:* The INA226 supports 2 operating modes, continuous and triggered, as well as a power-down state for reduced power consumption. For convenience, we opted for continuous mode with no power-down. Trigger mode was not used because the range of sampling frequencies we were aiming for was quite high. Additionally, given that we wanted to cover as much data as possible, if we dictated sampling frequency externally (unless mitigated by using overlapping sampling windows), we would have risked missing some data due to delays in the dictation of sampling times. The power-down state was not used since we are not constrained in powering the INA226. From now on, when we refer to INA226 usage, it will be taken for granted that we are talking about it in continuous mode.

The INA226 offers a wide variety of settings for sampling. Unlike "classic" sampling where we obtain a snapshot of our data at specific time intervals, the INA226 can use a block-based averaging approach with non-overlapping sampling windows. This means that when we query the INA226 register for measurements, what we obtain is the average of the last N samples. These samples occur at an interval dictated by the sum of the conversion times we set up. To give an example: with both ShuntVoltage and BusVoltage (the two measurements from which INA226 computes current and power [40]) conversion times set at 1.1 ms and a sample averaging of 16, the window of time covered by one query to the INA226 register would be (1.1 + 1.1) × 16 = 35.2 ms, and the data obtained would be the result of the average of 16 samples collected at an interval of 2.2 ms. The advantage of this approach over classic sampling is that it reduces the possibility of undersampling, making us less likely to miss short events like power spikes. The tradeoff of this approach is that short events like power spikes would be averaged across the window, resulting in them appearing flatter than they really are. However, this is not a concern for us since our goal is not to profile instantaneous power but to calculate energy usage, which depends on the mean over time.

In order to gain a better idea of which settings to use, conscious of the sampling rate of 10 ms used by [29], a series of tests around the 10 ms mark were executed. The tests consisted of running intermediate benchmarks with HTTP using various sampling averages and conversion times. The results obtained were as follows Fig.5:

As we can notice from the boxplot, the medians between the 8 ms setting with conversion times of 1.1 ms and averaging at 4 samples are very similar to the 18 ms setting at 0.588 ms conversion times and averaging of 16 samples. The choice fell to the 8 ms window option since it is within the 10 ms of T. Stefanec and M. Kusek [29], since the boxplot resulted more compact, and since, as per the Texas Instruments datasheet [40] longer conversion times (in our case, 1.1 ms against 0.588 ms) result in less noise.

The choice of settings to use was made from the set of available options described in [40] and implemented in the library we used [57] (ver. 0.6.4)

[2]In retrospect, communication via GPIO increased implementation complexity. A UART based approach would likely have resulted in a cleaner and more maintainable solution, while providing comparable functionality for our use-case

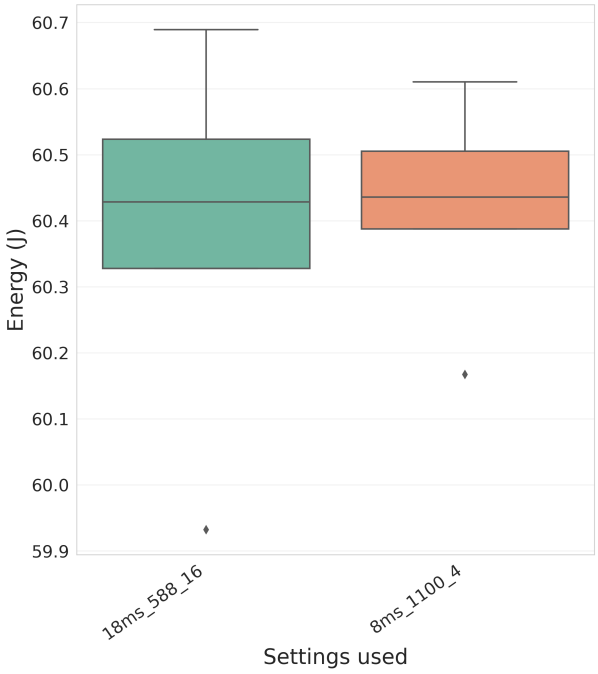

Fig. 5: Energy (J) used by HTTP intermediate benchmark with runs of 3 mins with different settings of INA226. 18ms window at 0.588 ms conversion time and averaging of 16 samples (on the left) and 8ms window at 1.1 ms with averaging of 4 samples (on the right). Boxes show the interquartile range (Q1–Q3) with the median line; whiskers extend to the most extreme non-outlier values (±1.5×IQR), and points beyond are outliers.

D. Statistical Analysis

In order to analyze data collected we relied on descriptive statistics visualizations, statistical tests and practical significance assessments.

- The descriptive statistics visualizations consisted of box plots to display the median, interquartile range (IQR), and data distribution for each protocol for each benchmark level.
- The statistical analysis relied on non-parametric tests, motivated by the boxplots of energy consumption, which indicated that most distributions deviated from normality. We first applied a Kruskal–Wallis test to assess if there were statistical significance differences across the protocols of each benchmark. When statistically significant differences were found we proceeded with pairwise Mann–Whitney U tests with Bonferroni correction as

post-hoc test to assess where the statistical differences occurred.
- The practical significance assessment consisted in evaluating whether observed differences between protocols were meaningful relative to the system's measurement variability. Practical significance was assessed by comparing the absolute difference between protocol medians against a noise threshold derived from the 12 repeated runs we executed for Fig. 8. The threshold was calculated by scaling the MAD obtained from the 12 runs to its SD equivalent value. If the absolute delta between the protocols medians did not exceed the noise threshold, the difference between protocols was deemed not practically significant.

More details on statistical analysis and its rationale are provided at section VII. Results are presented in section VI.

V. Implementation / Execution

A. *Experimental Design*

*1) **Experimental Design Structure**:* Given the selection of protocols done in Section II, we tested: MQTT, HTTP, CoAP, and WebSockets. MQTT was tested with all three possible QoS levels; considering each QoS setting as a separate protocol variant, this yielded a total of six protocol configurations. Each protocol was tested under two network traffic intensity levels: light and intermediate. Protocols supporting it were also tested under a heavy benchmark; see Table IV for benchmark details. This means, we used a 2-factor (protocol and benchmark level) design with an unbalanced structure,

$$\text{Total Runs} = (\text{Protocols} \times \text{Benchmark level}) \times \text{Replicas}$$

which given the 5 replicas and given that MQTT and HTTP were the only protocols capable of running heavy benchmark, resulted in a total of 80 experimental runs. This approach allows us to investigate main effects across all benchmarks and two-way interactions for the light and intermediate benchmarks.

*2) **Carryover Effect Mitigation**:* To ensure that no potential ordering effects influenced the results across runs of the same protocol benchmark, a series of tests was conducted. The intermediate HTTP benchmark was executed over 12 runs, and the energy consumption for each run was recorded. Measurements were obtained using a conversion time of 1100 microseconds with an averaging of four samples. The recorded energy values were plotted against the execution order, and a Spearman correlation test was performed to assess the relationship between run order and energy usage. The analysis revealed a significant ordering effect, as illustrated in Fig. 6 and supported by the Spearman test results. For the 0-minute cooldown interval, the test yielded a Spearman's rho of 0.6713 and a p-value of 0.0168.

Cooldown times between runs were increased to 1 minute, see Fig. 7, resulting in a Spearman's rho of 0.2937 and a p-value of 0.3541.

Since the runs in Fig. 6 and Fig. 7 were conducted with a reduced duration (3 minutes instead of 6 minutes), an additional test was performed with 12 consecutive runs, this time using the maximum run time employed in our final tests (6 minutes)[3].

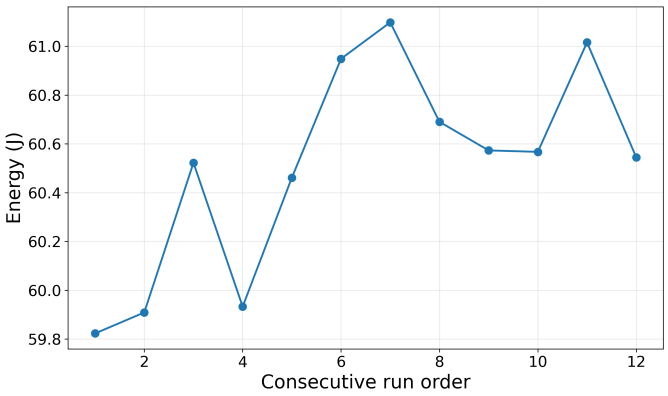

Fig. 6: Energy used (J) over consecutive runs of intermediate HTTP benchmark, with 0 seconds cooldown between runs and runs of 3 minutes

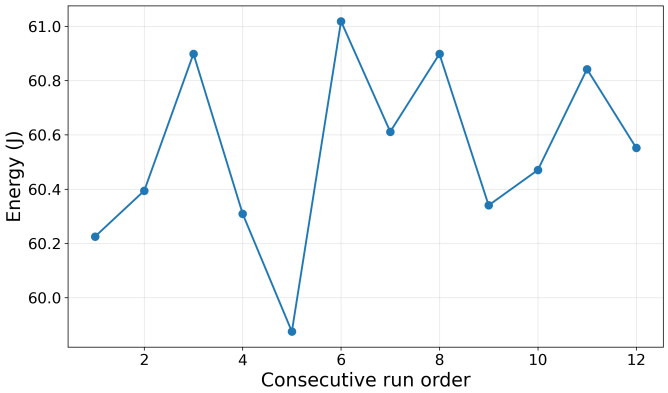

Fig. 7: Energy used (J) over consecutive runs of intermediate HTTP benchmark, with 60 seconds cooldown between runs and runs of 3 minutes

The 1 minute cooldown proved to be sufficient across 12 repetitions of the intermediate benchmark with our standard run time of 6 minutes. As we can observe from Fig. 8, we do not notice visible increments in energy usage across consecutive runs. This is confirmed by our Spearman test, which returned a Spearman's rho of -0.2098 and a p-value of 0.5128, well above the 0.05 threshold. This means that the weak negative correlation between energy used per run and run order is not statistically significant, confirming that our cooldown times are likely sufficient.

Assessing ordering effects was crucial because no balancing strategy across multiple protocols was possible, the code for multiple protocols could not fit on the board. Ensuring that individual protocol runs were free from significant ordering effects was therefore necessary to validate our energy measurements.

[3]The test shown in Fig. 8 was conducted with the latest code version, the same version that generated the data shown in the boxplots in our results, unlike the data shown in Fig. 6 and Fig. 7, which relied on an older version of the code.

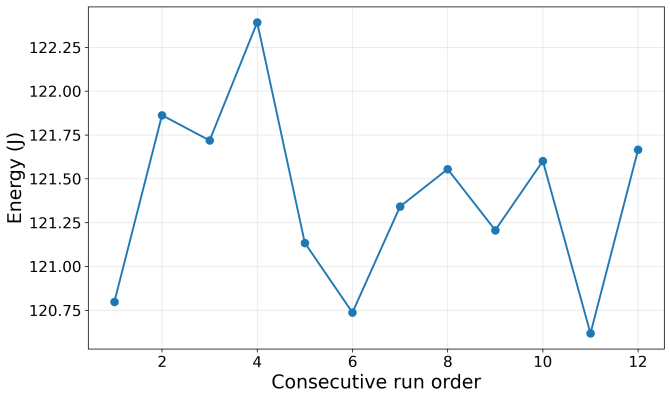

Fig. 8: Energy used (J) over consecutive runs of intermediate HTTP benchmark, with 60 seconds cooldown between runs and runs of 6 minutes

B. *Test bench Realization*

The boards were mounted on a perforated prototyping board according to the scheme shown in Fig. 2. All boards were installed in sockets to facilitate replacement in the event of damage. Signal and power routing was kept as short as possible, and cables of approximately equal cross-section were used where required.

C. *Boards provisioning*

The Arduino board was flashed with the datalogger script. The ESP32 sender was flashed with the script implementing the protocol under test to enable transmission, while the receiver ESP32 was flashed with the corresponding script for reception. Flashing was performed through the Arduino IDE (version 2.3.6), with ESP32 board support by Espressif System installed (version 3.3.0). The benchmark executed by the sender script is determined by the loop value. To run the *heavy* benchmark, this value was modified accordingly. For MQTT, the receiver was kept on the laptop, which also acted as the broker running Mosquitto (version 2.0.18).

D. *Network setup*

On the dedicated consumer-grade wireless router the DHCP server was configured with DHCP reservations outside the DHCP pool, ensuring that the boards and the broker always received the same IP addresses.

E. *Data collection start and stop procedure*

Measurements are collected using a fixed startup/shutdown procedure: the laptop serial logger script is started in the desired output directory, the Arduino is powered via USB, and the broker (if applicable) is started, followed by the receiver/subscriber. The ESP32 sender is then powered via the USB-C PD trigger. Data acquisition ends by terminating the logger via a keyboard interrupt, which finalizes and stores the recorded data as CSV files. The Arduino and ESP32 sender are then powered down, and the broker, if used, is stopped.

Detailed steps for data logging start and stop procedure as well as board provisioning are available at Appendix C.

F. *Data Handling Pipeline*

After all data of interest has been collected through the serial logger script, a sequential data processing pipeline is executed to calculate, aggregate, and statistically analyze the energy values across all experimental protocols. The pipeline is orchestrated by a single python script, which follows the instructions specified in a JSON configuration file that defines the order and execution context of each processing script.

Detailed steps of the data pipeline and the tasks performed by each script are provided in Appendix D.

G. *Environment Configuration*

The choice of libraries used to execute the benchmarks was dictated by their popularity, maintenance status and features availability. Further heuristics have been applied if library was not deemed sufficient to execute benchmarks selected and or if problems were encountered. This brought us to select what we think were the most common libraries with the most user friendly experience possible. The following table, Table V, presents the protocols tested, the libraries used to test them, and the settings applied that may affect our measurements. To reduce author familiarity bias in benchmark implementation, the benchmark code was developed with assistance from large language models (ChatGPT by OpenAI and Claude by Anthropic). These models were provided with documentation and usage examples from the respective libraries' GitHub repositories. The resulting code was manually consolidated, reviewed, and critically evaluated. Examples of prompts used are provided at Appendix B.

Protocol	Libraries	Extra settings applied
MQTT	ArduinoMqttClient.h (ver. 0.1.8), WiFi.h	QoS (0,1,2), port, topic, random client id
CoAP	WiFi.h, WiFiUdp.h, coap-simple.h (ver. 1.3.28)	512 bytes buffer
Websockets	WiFi.h, WebSocketsClient.h	port, automatic reconnection after 5 seconds
HTTP	WiFi.h, HTTPClient.h	Timeout time 4 seconds

TABLE V: Protocols tested and their sender-side settings, along with the libraries used and version. The libraries with no version reported are part of ESP32 core by Espressif System (ver. 3.3.0)

H. *Benchmark Execution*

In order to start and stop data collection from INA226 the status change in ESP32 GPIO has been used as a lock/unlock mechanism. Algorithm 1 shows the pseudocode of how the ESP32 running the benchmarks signals Arduino to start and stop collecting data from INA226. Algorithm 2 shows pseudocode of how the arduino handles the GPIO status change.

VI. Results and Data Analysis

A. *Light benchmark*

1) Visual Analysis: As we can notice from Fig. 9 by looking at boxplots, all protocols consumed a very similar amount of energy in light benchmark. With values (excluding outliers)

Algorithm 1: Data logger section (ESP32) pseudocode

```
Function setup_leds_and_gpio():
    set GPIO as output
    set led as output
    set GPIO low
    set led low
Function run_or_benchmark_change(int extra_cooldown_time_between_runs):
    // HIGH signal
    set GPIO high
    set led high
    delay(high_duration_ms)
    delay(extra_cooldown_time_between_runs)
    // LOW signal
    set GPIO low
    set led low
Function finished_with_all_benchmarks():
    while true do
        set led high
        set GPIO high
```

Algorithm 2: Data logger (Arduino Pro Micro) pseudocode

```
Function setup()
    enable I2C to INA226
    allow arduino to read GPIO
    set INA226 upperbound to 0.6A
    set INA226 shunt resistor at 0.100 Ohms
    set INA226 at 4 sample averaging
Function collect_INA_data_and_calc()
    if INA226 is conversion ready then
        fetch power data from INA226
        if (is not first reading of the run) then
            // log the run data in CSV
               through printl()
            Serial println (current_runs, current_benchmark,
              previoustime, current_time, currentPower_mW)
Function void loop()
    if (gpio state is HIGH AND we just came from a LOW state) then
        mark that we are no longer in a LOW state
        mark that a HIGH state has happened
    else if (gpio state is LOW) then
        mark that we are in a LOW state
        if (this is a valid LOW, which means that comes after a HIGH state) then
            if (this is the start of a new LOW state) then
                increment run and benchmark counters
            collect_INA_data_and_calc()
    else if (gpio state is HIGH AND we are still in same HIGH) then
        // no operation
    else
        we are in another gpio state, print debug data
```

ranging from 111 joules used per run to 112 joules used per run. We can notice a major grouping at the bottom of the graph made of CoAP and MQTT (QoS0, QoS1 and QoS2)

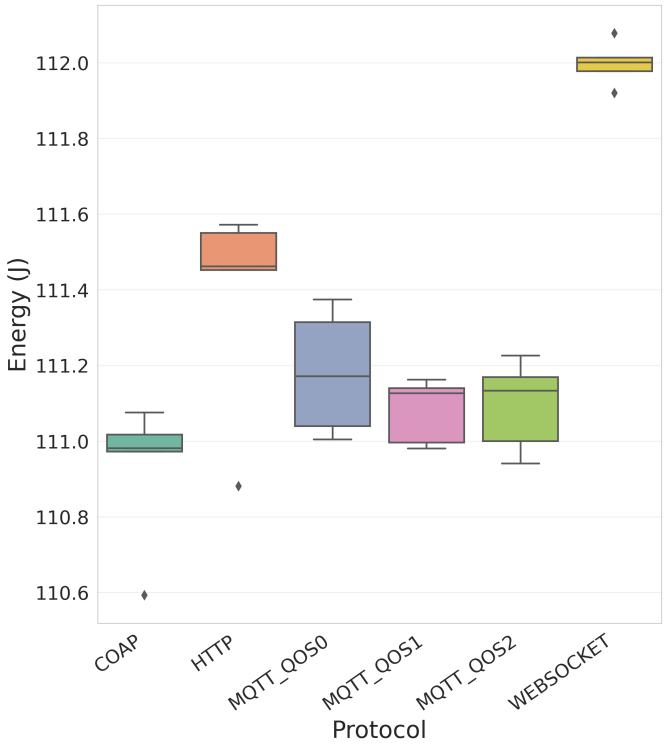

Fig. 9: Energy (J) used by a single run while executing *light* benchmark, see Table IV.

with HTTP slightly above, visually only Websocket protocol seem to have a significant difference in energy used from other application layer protocols in light benchmark, with a gap from HTTP median of circa 0.5 joules. All protocols except MQTT QoS0 show a heavily skewed distribution.

2) Statistical Hypothesis Testing: Kruskal-Wallis test confirmed the presence of statistically significant differences ($p<0.05$) among the six protocols tested with $H(5)=17.975$ and a $p=0.003$.

According to Mann–Whitney U test with Bonferroni correction, the only statistically significant differences ($p<0.05$) are between COAP and Websocket ($p = 0.0079$), HTTP and Websocket ($p = 0.0079$), MQTT(QoS0 ($p = 0.0079$), QoS1 ($p = 0.0079$), QoS2 ($p = 0.0079$)) and Websocket, with comparison between CoAP and MQTT QoS0 barely passing as not significant ($p = 0.0556$).

3) Practical Significance Assessment: Practical significance was assessed by comparing the absolute difference between protocol medians to a dispersion threshold derived from the Median Absolute Deviation (MAD) from Fig. 8. MAD was computed with respect to the median and scaled by a factor of 1.4826 to obtain a standard deviation equivalent estimate of 0.4333. Differences exceeding this threshold were classified as practically significant. According to our assessment the protocols that pairwise result practically significant are COAP and HTTP with a difference of 0.481, CoAP and Websocket with 1.020, HTTP and Websocket with 0.539, MQTT QoS0 and Websocket with 0.829, MQTT QoS1 and Websocket with 0.875, MQTT QoS2 and Websocket with 0.867.

B. *Intermediate benchmark*

1) Visual Analysis: As we can notice from Fig. 10 by looking at the boxplots, CoAP appears to be the protocol that consumed the less per energy per run, followed by WebSocket with a median of almost one Joule more, followed by MQTT QoS0. Looking upward after MQTT QoS0, we can notice a gap of circa 2 Joules where no other protocols fall in, to then restart with MQTT QoS1, followed by MQTT QoS2 and HTTP. Unlike in the light benchmark Fig. 9, here in the intermediate benchmark the protocols appear more scattered but with what looks like a marked division between the CoAP, WebSocket, MQTT QoS0 group and the MQTT QoS1, QoS2 and HTTP group. The boxplot shows no outliers in the data collected and skewed distributions for CoAP, MQTT QoS0, QoS1, and WebSocket.

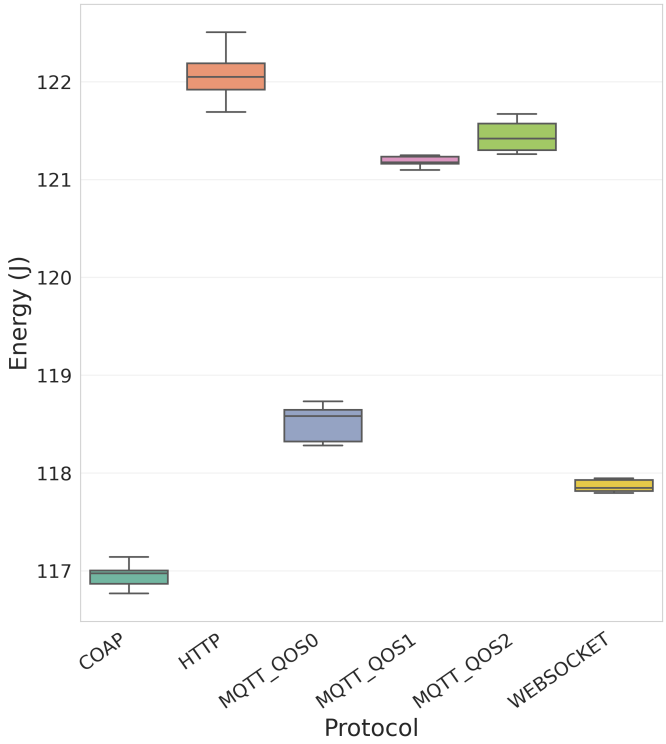

Fig. 10: Energy (J) used by a single run while executing *intermediate* benchmark, see Table IV.

2) Statistical Hypothesis Testing: Kruskal–Wallis test confirmed the presence of statistically significant differences ($p<0.05$) among the six protocols tested with $H(5)=28.2258$ and a $p=0.00$.

According to Mann–Whitney U test with Bonferroni correction, all protocols result to be statistically significantly different ($p<0.05$).

3) Practical Significance Assessment: Practical significance was assessed by comparing the absolute difference between protocol medians to a dispersion threshold derived from the Median Absolute Deviation (MAD). MAD was computed with respect to the median and scaled by a factor of 1.4826 to obtain a standard deviation equivalent estimate of 0.4333. Differences exceeding this threshold were classified as practically significant. According to our assessment all protocols result pairwise practically significant except for MQTT QoS1 and MQTT QoS2 where absolute difference of medians in Joules is 0.243.

C. *Heavy benchmark*

Under heavy benchmark conditions, only the HTTP and MQTT protocols managed to run. Attempts with CoAP and WebSocket were conducted at a frequency of 0.05 seconds before attempting with our frequency of 0.033 seconds. These attempts resulted in either a connect/disconnect loop in the case of WebSocket or in the payload being corrupted in the case of CoAP. After several attempts at tweaking buffers for CoAP and monitoring memory heap for WebSocket, we concluded that achieving the data rate required by the heavy benchmark at our current frequency and payload size would likely not be possible with our specific hardware/software setup without significant code optimization.

After executing the runs to collect energy usage, while checking the payload sent via HTTP and MQTT at its various QoS levels, we noticed that the payload was either truncated in the case of MQTT or not visible in the case of HTTP. To verify the correctness of the observed data behavior, Wireshark (version 4.2.2) was used to capture the traffic passing through our MQTT broker laptop, which for the occasion has been used with FastAPI to receive HTTP data. A rapid examination of the captured Wireshark data confirmed that in the case of HTTP, the payload was missing, and in the case of MQTT, the payload was truncated at 256 bytes.

This finding is in line with our earlier statements regarding CoAP and WebSocket, namely that achieving the data rate required by the heavy benchmark at the current frequency and payload size would likely not be possible with our specific hardware and software setup without significant code optimization. Consequently, due to the inability to achieve reliable data transmission that adheres to the chosen payload size and frequency, the heavy benchmark results are excluded from further analysis and discussion.

VII. Discussion

A. *Rationale behind data analysis*

We started by observing the boxplots Fig. 9 and Fig. 10, given the lack of normality in most distributions we opted for non parametric tests. We first executed Kruskal–Wallis test on each benchmark level of dataset to assess if there were any statistically significant differences between the protocols tested, to then proceed, if significant differences were found, with a post hoc pairwise comparison through Mann–Whitney U test to assess if there were statistically significant differences between protocols. Given the number of tests we needed to do (6 per benchmark) and being concerned about familywise error rate (FWER) we opted for Bonferroni correction. Bonferroni correction is quite conservative, would allow us to heavily limit the risk of Type I errors (false positives) with the con of increasing the likelihood of Type II errors (false negatives).

In addition to statistical significance, a practical significance assessment was performed to evaluate whether observed differences between protocols were meaningful relative to the

system's measurement variability. Practical significance was assessed by comparing the absolute difference between protocol medians against a noise threshold derived from repeated measurements. Medians were used to avoid being influenced by outliers like the ones we see in Fig. 9. The noise threshold was estimated using median absolute deviation (MAD) scaled to its standard deviation equivalent (MAD × 1.4826) [58]. It was computed from the 12 consecutive runs of intermediate benchmark with HTTP that we used to assess cooldown times, see Fig. 8. The choice of using the 12 runs to estimate our noise treshold was driven by the number of datapoints which allowed a more reliable estimate of system variability than the 5 data points per protocol that we got in our benchmarks. The resulting threshold was applied to both benchmark levels as a conservative upper bound on system noise, given that the same hardware and acquisition modalities were used across experiments. This choice is further supported by the lack of a consistent scaling between signal magnitude and dispersion, as protocols with similar median energy values exhibited substantially different MAD-based standard deviations.

B. *Combined Statistical and Practical Outcomes*

1) Light benchmark: The practical significance assessment matches the statistical difference assessment made with Mann–Whitney U test with Bonferroni correction except for the pairing CoAP HTTP which does not result statistically different according to Mann–Whitney but results practically different according to our practical assessment, see Table VI.

Protocol A	**Protocol B**	**Statistically different ($p<0.05$)**	**Practically different ($\Delta>0.4333$)**
CoAP	HTTP	N	Y
CoAP	MQTT QoS0	N	N
CoAP	MQTT QoS1	N	N
CoAP	MQTT QoS2	N	N
CoAP	Websocket	Y	Y
HTTP	MQTT QoS0	N	N
HTTP	MQTT QoS1	N	N
HTTP	MQTT QoS2	N	N
HTTP	Websocket	Y	Y
MQTT QoS0	MQTT QoS1	N	N
MQTT QoS0	MQTT QoS2	N	N
MQTT QoS0	Websocket	Y	Y
MQTT QoS1	MQTT QoS2	N	N
MQTT QoS1	Websocket	Y	Y
MQTT QoS2	Websocket	Y	Y

TABLE VI: summary of pairwise significancy of protocols in light benchmark

2) Intermediate benchmark: The outcome of the Mann–Whitney U test with Bonferroni correction is confirmed also by our practical significance test, all pairwise comparisons result to be significant except for MQTT QoS1 and MQTT QoS2, which result to be not practically significant, see Table VII.

Protocol A	**Protocol B**	**Statistically different ($p<0.05$)**	**Practically different ($\Delta>0.4333$)**
CoAP	HTTP	Y	Y
CoAP	MQTT QoS0	Y	Y
CoAP	MQTT QoS1	Y	Y
CoAP	MQTT QoS2	Y	Y
CoAP	Websocket	Y	Y
HTTP	MQTT QoS0	Y	Y
HTTP	MQTT QoS1	Y	Y
HTTP	MQTT QoS2	Y	Y
HTTP	Websocket	Y	Y
MQTT QoS0	MQTT QoS1	Y	Y
MQTT QoS0	MQTT QoS2	Y	Y
MQTT QoS0	Websocket	Y	Y
MQTT QoS1	MQTT QoS2	Y	N
MQTT QoS1	Websocket	Y	Y
MQTT QoS2	Websocket	Y	Y

TABLE VII: summary of pairwise significancy of protocols in intermediate benchmark

C. *Implications of Findings*

1) Light benchmark: Since WebSocket showed differences that were consistently both statistically and practically significant in pairwise comparisons against the other protocols, we can conclude that WebSocket consumes more energy in the light benchmark than the other protocols tested.

For MQTT, none of the QoS variants differed significantly from either HTTP or CoAP. Therefore, we cannot conclude whether MQTT consumes more or less energy than CoAP or HTTP, and vice versa, in the light benchmark.

Interestingly, the comparison between CoAP and HTTP was not statistically significant but was practically significant. This indicates that the observed median difference exceeds the estimated noise threshold; however, it cannot be confirmed as statistically significant with the available sample size.

2) Intermediate benchmark: Given the results obtained by our statistical and practical pairwise assessment, we can conclude that HTTP is the most energy consuming protocol. CoAP is the least energy consuming protocol, followed by Websocket, and then by MQTT QoS0. Statistically, MQTT QoS1 and MQTT QoS2 can be distinguished, however, the difference between the two falls within the noise threshold we established, making them not passing our practical significance assessment.

D. *Comparison with Existing Work*

When looking at the protocols tested in common, all papers are aligned with the classification we made with our intermediate benchmark: [34] results for power consumption are consistent with our findings, since it lists among the protocols it shows CoAP as the less power hungry protocol followed by MQTT and HTTP. [29] tests reflects our results in intermediate benchmark as well, with CoAP being the best choice, followed by MQTT and by HTTP. [30] is aligned with our findings, ranking CoAP as best performing, followed by MQTT followed by HTTP. [31] shows MQTT as more efficient than HTTP. [33] states that under same interaction pattern HTTP consumed more than MQTT and then states that QoS's consumption of MQTT are in increasing order QoS0, QoS1 and QoS2. Also here we are aligned, with the little difference that in our test QoS1 and QoS2 although statistically different, exhibited a performance gap smaller than noise threshold, preventing a confident assessment of which performed better.

[35] mentions CoAP as the most efficient followed by MQTT, followed by HTTP. [18] classifies the protocols from most energy efficient to less energy efficient as CoAP followed by MQTT followed by HTTP. No paper analyzed reported the energy usage of Websocket protocol. However, since protocol overhead influences energy usage [31], and WebSocket overhead is larger than CoAP and MQTT QoS0 but smaller than MQTT QoS2 [29], the expected situation is reasonably consistent with the intermediate benchmark results we obtained, where WebSocket falls between CoAP and MQTT QoS2, with the difference that in our benchmark it performs slightly better than MQTT QoS0. The concept of WebSocket's limited overhead is also reiterated by [18] which notes that after the initial handshake the HTTP headers are removed, resulting in minimal overhead that is significantly lower than HTTP. Given the substantial transmission interval of the light benchmark, where payloads are sent every 5 minutes over a 6 minute run, idle energy usage can have a significant impact on our light-benchmark measurements, providing a view of the higher idle energy usage of Websocket that is not amortized/hidden by the advantages WebSocket provides at higher transmission frequencies, like in intermediate benchmark.

Some notable omissions in our paper, compared to previously mentioned literature, are AMQP and MQTT-SN protocols, for which we had difficulty finding suitable working libraries; additionally, HTTP/2 was not tested because the selected HTTP library does not provide HTTP/2 support.

VIII. Threats to validity

A. *Internal validity*

- **Fixed execution order** Protocol benchmarks were run in a single fixed order. Some protocol scripts, once flashed, required up to approximately 80 percent of the ESP32's program storage space, preventing us from packaging multiple protocol benchmarks in a single script and thus preventing us from implementing counterbalancing techniques such as a Latin square design. Even if this problem were overcome by optimizing the script or by using boards with higher memory capacity, counterbalancing would still not be practical due to the need for having a receiver ready for each protocol. In this situation, any time-dependent external factor (temperature, RF congestion, etc.) could affect some protocols more than others; a counterbalancing design would have distributed possible effects evenly.
- **Uncontrolled RF environment** Experiments were executed in a household environment where radio-frequency (RF) conditions could not be fully controlled. Even though they were run on a dedicated consumer-grade wireless router, nearby Wi-Fi networks and devices may have introduced variability in the ESP32's wireless activity, possibly influencing our energy usage measurements.
- **INA226 settings and setup** As mentioned by [40] and by us in IV-C2 the INA226 settings of conversion time and the averaging settings introduce a tradeoff between noise reduction and update rate. Shunt resistor sizing also introduces a tradeoff that affects measurements, since a higher shunt resistance increases sensitivity but maximum measurable current decreases and vice versa with a smaller shunt resistance.
- **Limited statistical power** Due to the small sample size (5 replicas per protocol), the statistical analysis has limited power.
- **Practical Significance Assessment** While a shared noise floor is reasonable given the shared hardware setup, small protocol or benchmark specific induced noise as well as noise introduced by external factors, may not have been fully expressed by our shared noise floor.

B. *Construct validity*

- **Window averaged power** As mentioned in IV-C2, because the INA226 reports block-averaged power, our measurements are suitable for comparing total energy use but not for analyzing instantaneous peak power, which is flattened within each averaging block.

C. *External validity*

- **Platform diversity** Our benchmarks were all run on an ESP32 relying on Arduino core and ESP-IDF FreeRTOS. Our setup might not generalize to the wide variety of possible combinations of hardware platforms and software.

IX. Conclusion

Given the results obtained from our benchmark levels we can answer:

RQ1, Does the choice of application layer protocol over another significantly influence the energy consumption of the board? The answer is yes but not always. We noticed statistical and practically significant differences in the energy used by some protocols over the others within the same benchmark in both our benchmarks levels. However, it is worth highlighting that, within the same benchmarks, we also encountered protocols for which the differences didn't result as statistical and or practically significant, and hence of which no preference of one over the other (energy wise) can be expressed. Looking at the data collected in light benchmark, no protocol preference can be expressed within the group of CoAP, HTTP and MQTT (across all QoS's), but the preference of those protocols over Websocket can be expressed. Looking at the data collected in intermediate benchmark we cannot express (energy wise) the preference of MQTT QoS0 over MQTT QoS1, and vice versa. However, we can say that CoAP resulted as the most energy frugal protocol, followed by Websocket, MQTT QoS0, MQTT QoS1 and QoS2 (practically indistinguishable from QoS1), with HTTP as worst performing of the bunch.

RQ2, In terms of energy usage, are the protocols consistent across all benchmarks, or does a particular protocol perform best under specific conditions? The answer to first section of RQ2 is no, not all protocols energy usage is consistent across benchmarks, since energy usage of protocols tested doesn't directly scale with benchmark requirements, we can notice this with Websocket which resulted as the

worst performing (energy wise) in light benchmark but as the second best performing in intermediate benchmark. However we can notice some patterns across the benchmarks such as CoAP being in the group of the best performing in light benchmark and being also the best performing in intermediate benchmark, can be worth noticing also MQTT QoS0 being in the group of the best performing in light benchmark and being the third best performing protocol in intermediate benchmark. To answer the second section of RQ2 and not repeating what stated about CoAP and MQTT QoS0 constancy, yes, an example of protocol performing best under specific conditions is Websocket which seem to perform drastically better with the higher data rate of intermediate benchmark.

Beyond answering RQ1 and RQ2, this study proposed also two motivated benchmarks that could be reused as reference workloads in future research on resource constrained devices. It further proposes a third, more computationally demanding benchmark which exceeded the capabilities of the board, in the modality tested here, but may still be useful as a baseline for developing heavier workloads, particularly as video streaming and other high throughput applications become increasingly common and as hardware capabilities continue to improve. In addition, the results suggest that the INA226 can be a practical low cost solution for non simulated energy measurement experiments, without requiring expensive laboratory equipment.

X. Future studies

We believe that the topic of energy consumption of network protocols on hardware-constrained and power-constrained devices still has plenty of possible studies to conduct in order to obtain a more complete view. This could start from extending the work of this paper with downlink performances, moving to introducing more benchmarks in order to obtain a more granular view, testing without Arduino core on top of FreeRTOS, or even applying these benchmarks across different RTOSs such as FreeRTOS and Zephyr to determine whether protocol performance rankings are RTOS-dependent, and then extending to additional RTOSs.

Appendix

A. INA226 calculations and shunt resistor sizing

By using the INA226, we can collect a series of data points that can be further processed into additional information. Some of the information we can obtain from our INA226 are:

- **Shunt voltage** (V) across IN+ and IN-. This value consists in the voltage drop obtained when current flows through the resistor of INA226.
- **Bus voltage** (V) at VBUS pin, it represents the voltage going to our ESP32 board.
- **Current** (A) Obtained through INA226 internal ADC [40], the current represents shunt voltage divided by shunt resistance. Can be obtained by reversing Ohm's law.

$$\text{Voltage (V)} = \text{Current (A)} \times \text{Resistance } (\Omega)$$

$$\text{Current (A)} = \frac{\text{shunt voltage (V)}}{\text{Resistance } (\Omega)}$$

- **Power** (W) Obtained through INA226 internal ADC [40], the power represents current multiplied by bus voltage

$$\text{Power } (W) = \text{Current } (A) \times \text{Bus voltage (V)}$$

- **Energy** (mWh or J) is obtained by integrating power over time. Where $P(t)$ is instantaneous power in watts, dt is an infinitesimal small time step (we are going to approximate since our INA226 collects data at discrete time intervals and not continuously), t0 and t1 are start and end times.

$$E = \int_{t_0}^{t_1} P(t)\, dt$$

Must be taken into consideration that the minimum amount of current detectable by the INA226 depends on the sizing of the shunt resistor. According to Ohm's Law, using a shunt resistor with a value of $0.1\ \Omega$, given INA226 theoretical voltage resolution of $2.5\ \mu$V [40], would result in a theoretical minimum current detectable of $\frac{2.5\ \mu\text{V}}{0.1\ \Omega} = 25\ \mu$A. By using a shunt resistor with a value of $0.5\ \Omega$ we would get a theoretical minimum detectable current of $\frac{2.5\ \mu\text{V}}{0.5\ \Omega} = 5\ \mu$A and so on.

B. Prompts used to assist benchmarks code writing

Initial prompts were formatted in the following way: *Given the README and the example code provided from library <application-layer protocol library>, generate two simple sender and receiver scripts to send a JSON-formatted string from one ESP32 to another using <application-layer protocol>. Keep it simple, use an hardcoded JSON string.* If the README was unavailable, only example code was provided.

C. Board provisioning and data collection start and stop procedures

Boards provisioning The Arduino board was flashed with `master_datalogger_mk6.ino`, while the ESP32s were flashed with the sender and receiver scripts of the protocol under test; for MQTT, the receiver was kept on the laptop running the broker.

Data collection start and stop procedure

In order to collect the data a specific system initialization procedure needs to be followed:

1) Execute the Python script `master_serial_to_csv_mk3.py` to collect serial data from the Arduino in the folder where we want to save all our measurements.
2) Power on the Arduino by plugging it into laptop.
3) Power on the broker (in the case of MQTT).
4) Power on the ESP32 receiver or start the subscriber on the laptop (in the case of MQTT).
5) Power on the ESP32 sender by plugging in our USB-C PD trigger module.

In order to stop collecting the data and save all data collected so far:

1) Press Ctrl+c on the terminal running `master_serial_to_csv_mk3.py`, by doing this all data collected so far will be saved in a folder in CSV.

2) Power off the ESP32 sender by un-plugging our USB-C PD trigger module.
3) Power off the Arduino by un-plugging it from laptop.
4) Power off/ stop receiver and broker if available.

D. Data handling pipeline steps and details

After all data of interest has been collected through master_serial_to_csv_mk3.py, a sequential data processing pipeline is executed to calculate, aggregate, and statistically analyze the energy values across all experimental protocols. The pipeline is orchestrated by pipeline_runner_mk2.py, which follows the instructions dictated by the configuration file (pipeline_config.json) that defines the order and execution context of each processing script.

The pipeline is organized into three distinct phases, each targeting a specific directory level within the project structure. The first phase acts on each experiment folder, performing:

- 01_energy_calculation.py Calculates energy consumption in Joules for each measurement section using the formula $E = P \times \Delta t$ (we use rectangular rule for our discrete approximation of energy formula mentioned in section IV-B). Power measurements in milliwatts are converted to watts and time intervals $\Delta t = current\_time - previous\_time$ are converted from milliseconds to seconds. The calculated energy values are appended as a new column (energy_used_by_section) to the existing dataset.
- 02_energy_aggregator.py Aggregates the energy values by grouping data according to current_runs and current_benchmark, summing the energy_used_by_section values to produce total_energy_J for each unique combination of run and benchmark. The aggregated values are rounded to four decimal places.
- 03_add_prot_col_and_order.py Adds protocol column, sorts the data by current_benchmark and by current_runs, Renames the resulting file from this operations by appending _to_merge

The second phase operates at the base directory level running:

- 04_merge_all_protocol_csvs.py Recursively searches all subdirectories for files with name ending in master_energy_to_merge.csv, concatenates them into a single DataFrame, and sorts the unified dataset by protocol, benchmark, and run number. The merged output is saved as all_protocols_master_energy.csv in a newly created directory called results.

The third phase operates in the results directory running:

- 05_do_boxplot.py A boxplot is created from all_protocols_master_energy.csv for each benchmark level collected, each containing all protocols tested under that benchmark level.
- 06_kruskall_wallis.py Executes Kruskal-Wallis tests to compare energy consumption across protocols for each benchmark. If significant differences are detected the script performs post-hoc pairwise Mann-Whitney U tests with Bonferroni correction to identify which protocol pairs differed significantly. Descriptive statistics are computed too.
- 07_check_practical_difference.py Calculates pairwise absolute differences between the median energy usage of protocols. It then compares the resulting absolute differences against a MAD based standard deviation threshold provided by the user to determine whether the difference is practically significant.

REFERENCES

[1] M. Al-Kuwari, A. Ramadan, Y. Ismael, L. Al-Sughair, A. Gastli, and M. Benammar, "Smart-home automation using iot-based sensing and monitoring platform," in *2018 IEEE 12th International Conference on Compatibility, Power Electronics and Power Engineering (CPE-POWERENG 2018)*, pp. 1–6, IEEE, 2018.

[2] P. Jayashankar, S. Nilakanta, W. J. Johnston, P. Gill, and R. Burres, "Iot adoption in agriculture: the role of trust, perceived value and risk," *Journal of Business & Industrial Marketing*, vol. 33, no. 6, pp. 804–821, 2018.

[3] McKinsey&Company, "What are industry 4.0, the fourth industrial revolution, and 4ir?," Aug 2022. [Accessed 25-10-2024].

[4] C. Sobin, "A survey on architecture, protocols and challenges in iot," *Wireless Personal Communications*, vol. 112, no. 3, pp. 1383–1429, 2020.

[5] M. Aboubakar, M. Kellil, and P. Roux, "A review of iot network management: Current status and perspectives," *Journal of King Saud University-Computer and Information Sciences*, vol. 34, no. 7, pp. 4163–4176, 2022.

[6] S. Sinche, D. Raposo, N. Armando, A. Rodrigues, F. Boavida, V. Pereira, and J. S. Silva, "A survey of iot management protocols and frameworks," *IEEE Communications Surveys & Tutorials*, vol. 22, no. 2, pp. 1168–1190, 2019.

[7] T. Salman and R. Jain, "Networking protocols and standards for internet of things," *Internet of things and data analytics handbook*, pp. 215–238, 2017.

[8] A. Triantafyllou, P. Sarigiannidis, and T. D. Lagkas, "Network protocols, schemes, and mechanisms for internet of things (iot): Features, open challenges, and trends," *Wireless communications and mobile computing*, vol. 2018, no. 1, p. 5349894, 2018.

[9] O. Ali, M. K. Ishak, M. K. L. Bhatti, I. Khan, and K.-I. Kim, "A comprehensive review of internet of things: Technology stack, middlewares, and fog/edge computing interface," *Sensors*, vol. 22, no. 3, 2022.

[10] B. H. Çorak, F. Y. Okay, M. Güzel, Ş. Murt, and S. Ozdemir, "Comparative analysis of iot communication protocols," in *2018 International symposium on networks, computers and communications (ISNCC)*, pp. 1–6, IEEE, 2018.

[11] M. Babiuch, P. Foltýnek, and P. Smutný, "Using the esp32 microcontroller for data processing," in *2019 20th International Carpathian Control Conference (ICCC)*, pp. 1–6, 2019.

[12] L. Tightiz and H. Yang, "A comprehensive review on iot protocols' features in smart grid communication," *Energies*, vol. 13, no. 11, p. 2762, 2020.

[13] K. Hofer-Schmitz and B. Stojanović, "Towards formal verification of iot protocols: A review," *Computer Networks*, vol. 174, p. 107233, 2020.

[14] J. Tournier, F. Lesueur, F. Le Mouël, L. Guyon, and H. Ben-Hassine, "A survey of iot protocols and their security issues through the lens of a generic iot stack," *Internet of Things*, vol. 16, p. 100264, 2021.

[15] E. Al-Masri, K. R. Kalyanam, J. Batts, J. Kim, S. Singh, T. Vo, and C. Yan, "Investigating messaging protocols for the internet of things (iot)," *IEEE Access*, vol. 8, pp. 94880–94911, 2020.

[16] S. Jaloudi, "Communication protocols of an industrial internet of things environment: A comparative study," *Future Internet*, vol. 11, no. 3, p. 66, 2019.

[17] M. Lombardi, F. Pascale, and D. Santaniello, "Internet of things: A general overview between architectures, protocols and applications," *Information*, vol. 12, no. 2, p. 87, 2021.

[18] D. Glaroudis, A. Iossifides, and P. Chatzimisios, "Survey, comparison and research challenges of iot application protocols for smart farming," *Computer networks (Amsterdam, Netherlands : 1999)*, vol. 168, pp. 107037–, 2020.

[19] M. N. Bhuiyan, M. M. Rahman, M. M. Billah, and D. Saha, "Internet of things (iot): A review of its enabling technologies in healthcare applications, standards protocols, security, and market opportunities," *IEEE Internet of Things Journal*, vol. 8, no. 13, pp. 10474–10498, 2021.

[20] Espressif, "Get started." [Accessed 10-07-2025].

[21] Espressif, "Esp-idf programming guide freertos (idf)." [Accessed 10-07-2025].
[22] Espressif, "Esp-idf programming guide freertos overview." [Accessed 10-07-2025].
[23] Arduino, "Installing additional cores." [Accessed 10-07-2025].
[24] R. Luna and S. A. Islam, "Security and reliability of safety-critical rtos," *SN Computer Science*, vol. 2, p. 356, 2021.
[25] A. Eliasz, *Zephyr RTOS Embedded C Programming: Using Embedded RTOS POSIX API*. Springer, 2024.
[26] R. Hat, "red hat enterprise linux power management guide 3.6. tickless kernel," 2010 (estimated). [Accessed 5-08-2025].
[27] M. Silva, D. Cerdeira, S. Pinto, and T. Gomes, "Operating systems for internet of things low-end devices: Analysis and benchmarking," *IEEE Internet of Things Journal*, vol. 6, no. 6, pp. 10375–10383, 2019.
[28] R. Oshana, M. Kraeling, and S. O. service), *Software engineering for embedded systems : methods, practical techniques, and applications*. Expert guide, Amsterdam: Newnes, first edition. ed., 2013.
[29] T. Stefanec and M. Kusek, "Comparing energy consumption of application layer protocols on iot devices," in *2021 16th International Conference on Telecommunications (ConTEL)*, pp. 23–28, 2021.
[30] A. Shahrokhi and M. Ahmadi, "Power evaluation of iot application layer protocols," in *2023 7th International Conference on Internet of Things and Applications (IoT)*, pp. 1–7, 2023.
[31] J. Hofer and S. Pawaskar, "Impact of the application layer protocol on energy consumption, 4g utilization and performance," in *2018 3rd Cloudification of the Internet of Things (CIoT)*, pp. 1–7, 2018.
[32] S. Krug, I. Shallari, and M. O'Nils, "A case study on energy overhead of different iot network stacks," in *2019 IEEE 5th World Forum on Internet of Things (WF-IoT)*, pp. 528–529, 2019.
[33] R. Canek, P. Borges, C. Taconet, S. Voulgaris, and D. Eyers, "Analysis of the impact of interaction patterns and iot protocols on energy consumption of iot consumer applications," in *Distributed Applications and Interoperable Systems*, vol. 13272 of *Lecture Notes in Computer Science*, pp. 131–147, Switzerland: Springer International Publishing AG, 2022.
[34] N. Naik, "Choice of effective messaging protocols for iot systems: Mqtt, coap, amqp and http," in *2017 IEEE International Systems Engineering Symposium (ISSE)*, pp. 1–7, 2017.
[35] J. Dizdarević, F. Carpio, A. Jukan, and X. Masip-Bruin, "A survey of communication protocols for internet of things and related challenges of fog and cloud computing integration," *ACM computing surveys*, vol. 51, no. 6, pp. 1–29, 2019.
[36] U. Tandale, B. Momin, and D. P. Seetharam, "An empirical study of application layer protocols for iot," in *2017 International Conference on Energy, Communication, Data Analytics and Soft Computing (ICECDS)*, pp. 2447–2451, 2017.
[37] C. Bormann, M. Ersue, and A. Keränen, "Terminology for Constrained-Node Networks." RFC 7228, May 2014.
[38] M. Babiuch, P. Foltỳnek, and P. Smutnỳ, "Using the esp32 microcontroller for data processing," in *2019 20th International Carpathian Control Conference (ICCC)*, pp. 1–6, IEEE, 2019.
[39] E. Systems, "Esp32-wroom-32." [Accessed 10-09-2025].
[40] T. Instruments, "Ina226 36v, 16-bit, ultra-precise i2c output current, voltage, and power monitor with alert," Sep 2024. [Accessed 16-06-2025].
[41] sparkfun, "Pro micro specsheet." [Accessed 10-09-2025].
[42] M. Borges, S. Paiva, A. Santos, B. Gaspar, and J. Cabral, "Azure rtos threadx design for low-end nb-iot device," in *2020 2nd International Conference on Societal Automation (SA)*, pp. 1–8, 2021.
[43] O. Hahm, E. Baccelli, H. Petersen, and N. Tsiftes, "Operating systems for low-end devices in the internet of things: A survey," *IEEE Internet of Things Journal*, vol. 3, no. 5, pp. 720–734, 2016.
[44] FreeRTOS, "Freertostm real-time operating system for microcontrollers and small microprocessors." [Accessed 10-07-2025].
[45] D. Dublensky, E. Kanunnikov, and T. Makhmudov, "Comparison of time characteristics of real-time operating systems (rtos) on microcontrollers with core architectures arm and risc-v," in *2025 International Russian Smart Industry Conference (SmartIndustryCon)*, pp. 630–635, 2025.
[46] Zephyr, "Zephyr rtos a proven rtos ecosystem, by developers, for developers." [Accessed 10-07-2025].
[47] nuttx, "Apache nuttx is a mature, real-time embedded operating system (rtos)." [Accessed 21-07-2025].
[48] riot os, "The friendly operating system for the internet of things." [Accessed 18-07-2025].
[49] C. Sabri, L. Kriaa, and S. L. Azzouz, "Comparison of iot constrained devices operating systems: A survey," in *2017 IEEE/ACS 14th International Conference on Computer Systems and Applications (AICCSA)*, pp. 369–375, 2017.
[50] S. K. Chakravarty, A. Batra, N. Singh, and G. Kumar, "Low power consumption analysis of communication protocol for the meteorological internet of things," in *2024 12th International Conference on Intelligent Systems and Embedded Design (ISED)*, pp. 01–06, 2024.
[51] netatmo, "Smart home weather station," unknown. [Accessed 6-08-2025].
[52] T. Schaub, A. Millon, C. De Zutter, R. Buij, J. Chadœuf, S. Lee, A. Mionnet, and R. H. G. Klaassen, "How to improve the accuracy of height data from bird tracking devices? an assessment of high-frequency gps tracking and barometric altimetry in field conditions," *Animal Biotelemetry*, vol. 11, no. 1, p. 31, 2023.
[53] E. Jharko, M. Mamchenko, and S. P. Khripunov, "Robot/uav indoor visual slam in smart cities based on remote data processing," in *2023 International Russian Smart Industry Conference (SmartIndustryCon)*, pp. 504–508, 2023.
[54] M. Mamchenko, "Modeling the performance of ieee 802.11ac wlan network for remote visual slam," in *2023 16th International Conference Management of large-scale system development (MLSD)*, pp. 1–5, 2023.
[55] J. Alves and A. Bernardino, "A remote rgb-d vslam solution for low computational powered robots," in *2020 IEEE International Conference on Autonomous Robot Systems and Competitions (ICARSC)*, pp. 214–220, 2020.
[56] Quartinocci, "Repository for "energy consumption analysis of application layer protocols in realistic iot uplink scenarios"." https://github.com/Quartinocci/ECAALPRIUS, 2026. GitHub repository, accessed 2026-03-10.
[57] R. Tillaart, "Ina226: Arduino library for the ina226 current/power monitor." https://github.com/RobTillaart/INA226, 2026. GitHub repository, accessed 2026-01-13.
[58] P. Rousseeuw and M. Hubert, "Anomaly detection by robust statistics," *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, vol. 8, 03 2018.