Monitoring Performance Metrics in Low-Power Wireless Systems

Fabian Graf, Thomas Watteyne, Michael Villnow

Monitoring Performance Metrics in Low-Power Wireless Systems

Fabian Graf[a,*], Thomas Watteyne[b], Michael Villnow[a]

[a] *Siemens AG, Erlangen, Germany*
[b] *Inria, Paris, France*

Abstract

Application Performance Monitoring (APM) is key for ensuring computer systems perform well. While most APM tools target servers and networking infrastructure, here we focus on APM for devices with strict resource constraints: extremely limited in terms of power, memory and bandwidth. We tailor this article to be both a survey and a tutorial. In the survey part, we investigate APM approaches for low-power wireless networks, with a particular focus on Time Synchronized Channel Hopping solutions, as they are well-suited for critical industrial applications. We survey performance metrics characterizing the network health condition and show how, to capture the health of a network universally, it is important to constantly monitor hardware-related, network-related and network-wide metrics. We present a collection of metrics that serves as a checklist for the design of an APM system, describe related work on APM concepts suitable for low-power wireless system, and provide core concepts for collecting, exporting and processing performance metrics. The tutorial part consists of a hands-on example of running commercial APM and networking solutions. We use the active APM framework from *Memfault*, which periodically creates heartbeats including the performance metrics. We run this framework on top of the SmartMesh IP protocol stack, a commercial product by Analog Devices that offers wired-like high reliability and a decade of battery lifetime, and integrate it with the *Zephyr* operating systems. This tutorial allows the readership to experiment with a complete ready-to-deploy mote-to-cloud APM chain.

Keywords: Application Performance Monitoring (APM), Constrained Devices, Memfault, SmartMesh IP, Zephyr

1. Introduction

The increase in Internet of Things (IoT) devices continues unabated and is expected to reach an amount of 16.7 billion endpoints by the end of 2023 [1]. This massive number comprises a large variety of different device classes, ranging from the smallest sensors to complex industrial machinery and everything in between. These devices have the capability to collect, transmit, and process data in real-time, forming the backbone of the modern connected world. To maintain such a network of devices, Application Performance Monitoring (APM) is essential. Monitoring IoT applications has become indispensable due to several reasons. IoT devices often operate in dynamic and challenging environments, making them susceptible to various operational and connectivity issues. APM is the key in identifying and resolving these issues in real-time, ensuring uninterrupted data flow and functionality. Moreover, the insights gained from APM can optimize device performance, ensure reliability and reduce operational costs. Observability goes beyond APM and is a commonly addressed and well-understood topic for more advanced IoT device classes [2]. When speaking of advanced device classes, we mean full-fledged microcomputers like, e.g., a Raspberry Pi. However, in the diverse array of IoT networks, ultra low-power wireless systems have garnered particular attention, too. These systems commonly rely on battery-driven devices with resource-constrained Microcontroller Unit (MCU) architectures such as the 32 bit ARM Cortex family. These devices are designed for efficiency and longevity while causing only small costs. At the same time, high requirements in terms of reliability are demanded in Industrial Internet of Things (IIoT) environments. Wireless Sensor Networks (WSNs) are an example for such low-power wireless systems and are widely deployed in industrial environments to ensure a flawless operation of all the entities in a factory. It is therefore crucial to install a system that the user can count on. In fact, according to a survey published by the International Society of Automation (ISA), the overwhelming majority of people talks about reliability when asked for the most important features in a WSN [3]. Obviously, low-power wireless systems only offer limited amounts of battery lifetime. This might be an explanation for the tendency of vendors to neglect APM capabilities in such constrained devices. In our opinion, this seems inconsistent when recalling that reliability is the most important feature from a customer's point of view. APM allows retriev-

*Corresponding author
Email addresses: fabian.graf@siemens.com (Fabian Graf), thomas.watteyne@inria.fr (Thomas Watteyne), michael.villnow@siemens.com (Michael Villnow)

ing the health condition of not only a single device, but of the overall system. Having a platform where all important performance metrics are collected, processed and visualized makes it possible to detect critical trends or bottlenecks. A broad range of metrics is also essential for detecting the impact of external events or, e.g., a critical firmware upgrade with negative consequences on the overall system. But also, in case of sporadic failures or in case of a connection-loss, APM provides valuable insights to identify the bug and saves the user from laborious hardware debugging or error reconstruction. Although there has been limited effort in the academic community towards APM tools and metrics for these low-power wireless systems, many research topics remain open and unexplored.

This work aims to bridge this gap by offering the following contributions:

- Conducting a comprehensive survey on performance metrics that may serve as a checklist for developers of low-power wireless systems on what parameters they should consider for their APM implementation.
- Presenting a novel comparison of Real-Time Operating System (RTOS) for constrained IoT devices in terms of in-built performance metrics and APM features.
- Providing a survey and taxonomy of state-of-the-art APM frameworks covering the steps of collecting, exporting and processing performance metrics.
- Demonstrating the practical usability and performance of an APM platform in a low-power wireless system based on SmartMesh IP as part of an in-depth step-by-step tutorial.

This article is organized as both a survey (Sections 2–4) and a tutorial (Section 5). Specifically, Section 2 starts with an introduction about the technical background of low-power wireless systems. We compare different standards in the IIoT and present the IPv6 over the TSCH mode of IEEE 802.15.4e (6TiSCH) protocol stack promising high reliability and low-power consumption at the same time. Section 3 contains an exhaustive list of performance metrics delivering a detailed and complete picture of the system's health status. Section 4 deals with the results of our literature research on state-of-the-art methods of collecting metrics and exporting them out of the system. Additionally, some strategies for processing the data on the network edge are presented. Section 5 is the tutorial part of this work and shows an approach of monitoring a Zephyr [4] application that uses SmartMesh IP [5] for networking.

2. Technical Background

Before diving into the topic of APM in low-power wireless systems, we present a set of standards widely used in this type of networks. Besides WiFi (IEEE 802.11) or Bluetooth (IEEE 802.15.1), another standard dominates the IIoT world, namely Low-Rate Wireless Personal Area Networks (LR-WPAN), or simply IEEE 802.15.4 [6]. In general, it is not as power-hungry as WiFi [7] and the mesh topology of IEEE 802.15.4 allows to cover much larger areas compared to other low-power solutions such as Bluetooth Low Energy (BLE) [8, 9]. In 2012, Time-Slotted Channel Hopping (TSCH) was first proposed as an enhancement of IEEE 802.15.4 (known as IEEE 802.15.4e) [10, 11] and in 2015 it was included in the related standard specification [12]. TSCH deals with external interference and multi-path fading at the Medium Access Control (MAC) layer. When two neighbor nodes exchange frames, they send subsequent frames at different frequencies, resulting in channel hopping. The idea is that, if external interference or multi-path fading causes the transmission of a frame to fail, the retransmission happens at a different frequency, and therefore has a higher chance of succeeding than if retransmitted on the same frequency [13]. For that reason, TSCH makes the standard more robust and suited for industrial environments [14]. As IEEE 802.15.4 defines the characteristics of the PHY and MAC layer in the Open Systems Interconnection (OSI) model, there are several standards building up on IEEE 802.15.4 in the upper layers. Zigbee [15], Thread [16] and Matter [17] are popular examples that target mainly smart-home applications. Industrial solutions mostly rely on WirelessHART [18] and ISA-100.11a [19] which both employ TSCH. Besides the mentioned technologies, the Internet Engineering Task Force (IETF) has put effort in defining protocols for the integration of constrained devices, such as sensors, into the Internet. These protocols include IPv6 over Low-Power Wireless Personal Area Networks (WPAN) (6LoWPAN) [20], Routing Protocol for Low-power and Lossy Networks (RPL) [21] and Constrained Application Protocol (CoAP) [22]. The IETF 6TiSCH Working Group (WG) [23] was founded to create a standard that enables the use of them on top of the IEEE 802.15.4-2015 TSCH link layer [12]. The resulting 6TiSCH stack is fully implemented in at least 4 open source projects [24]: OpenWSN [25], Contiki(-NG) [26], RIOT OS [27] and TinyOS [28]. Analog Devices' SmartMesh IP product line [5] also implements a pre-6TiSCH protocol stack. The technical overview of SmartMesh IP [29] and the results of 6TiSCH performance evaluations [30, 31] indicate that this standard may fulfill our ambitious requirements in terms of reliability. Since we consider reliability as the major Key Performance Indicator (KPI) of our system, the 6TiSCH architecture is chosen as the underlying model in the remainder of this work.

3. Performance Metrics in Low-Power Wireless Systems

The goal of this section is to give a broad overview about what metrics are worth monitoring, in order to capture the health condition of a low-power wireless system. Furthermore, we want to clearly define certain terms which are tending to be mixed up or misinterpreted in the academic community. The following collection of metrics is based on a thorough literature survey.

"A survey on the metrics that matter" was already given by Yuan et al. [32]. All the metrics from this survey can be found in our presented overview, too. However, we do not only want to extend this collection of metrics, but also take a different approach for clustering the metrics in different groups. Yuan et

al. arrange the metrics in a matrix-like structure. One dimension rates the metrics as node-centric, hop-centric, path-centric, end-to-end or network-centric. The other dimension classifies the metrics in terms of the layers of the OSI model.

Recently, Ojeda et al. [33] published a review "On Wireless Sensor Network Models: A Cross-Layer Systematic Review". The authors cover different methods of modelling WSNs. In this context, they propose a set of metrics, which are relevant to estimate the performance of the system model. The metrics are once again clustered based on the OSI model.

Another important source in our survey are the metrics contained in the health reports of SmartMesh IP [34, Chapter 5.4]. The concept of health reports, also called heartbeats [35], is explained in detail in Section 4. We include all these health report metrics in our overview, as well as the aggregated metrics used for monitoring the health condition of a SmartMesh IP network [36, Chapter 6].

After an extensive literature study, we identified three main groups of metrics: device metrics (covering both Hardware (HW) and Software (SW)), node-centric networking metrics and system-wide networking metrics. In the following, we do not simply list these metrics, but also show the relevance of monitoring them, provide practical guidance on acquiring and discuss the associated costs.

3.1. Device HW Metrics

This first set of metrics is summarized in Table 2 and focuses on indicators that inform the user about potential HW defects of their device, bugs in software that either become visible just after long time of operation or which are caused by erroneous software updates and harmful physical impacts from the surrounding. This set can once again be divided in HW and SW metrics, respectively.

An important KPI in low-power systems is battery lifetime. Thus, keeping accurately track of power consumption is crucial [37]. *Battery Voltage* and *Charge Consumption* are the most important metrics to determine the State-of-Charge (SoC) of the battery and to detect aging problems [38]. To determine the SoC precisely, a fuel gauge Integrated Circuits (ICs) such as the MAX17048 [39] and high-resolution Analog-to-Digital Converters (ADCs) can be used. A Coulomb-counter is a widespread method to determine the lifetime charge consumption of the battery [40]. Measuring the *Energy Consumption* of specific atomic operations may be very useful in estimating the impact of certain events related to the energy performance of the device. Nevertheless, complex on-board instruments are the reason why capturing these metrics appears not be very popular and why most scenarios measurements are restricted to an ADC sampling the battery voltage [41]. Temperature is another physical metric that also has strong influence on the battery lifetime and the device's overall performance in general. We differentiate between the metrics *Core Temperature* and *Ambient Temperature*. The core temperature is measured at a location close to the Central Processing Unit (CPU) to detect overheating of the chip. The ESP32-S2 for instance, has a built-in temperature sensor designed for this monitoring use case [42]. To detect harmful external influences in the environment, capturing the temperature in the surrounding of the mote is essential. For this purpose, a Digital Humidity Temperature (DHT) sensor can be connected to the pins of the mote. However, a common DHT 11 sensor consumes about 2.5 mA during the sampling process [43], whereas capturing the other mentioned HW metrics just consumes current in the μA range. Therefore, ambient temperature shall just be reported in reasonable intervals or even on demand.

We also recommend monitoring metrics related to the crystal. It is a central component in a 6TiSCH device since time-synchronization among all motes is the basis of TSCH [44]. The observation of the *Clock Skew* cannot just be used to draw conclusions on potential HW issues, but also to calculate the *Experimental Clock Drift*. However, floating-point operations and the collection of drift samples require additional memory, which is typically limited in low-power wireless systems [45]. Nonetheless, the *Experimental Clock Drift* is a valuable parameter for the rejoin process of the mote helping to save time and energy by sticking to a low *Radio Duty Cycle (RDC)* [46].

In wireless systems, motes are naturally equipped with a radio which is another key element that wants to be monitored and delivers useful insights. Although the metric *Radio Duty Cycle (RDC)* is listed in the radio statistics list, anomalies usually indicate errors in software. Since the strict schedule in 6TiSCH networks allows the motes to turn off their radio most of the time, high RDC values warn the user in case of unwanted behavior of the mote resulting in higher power consumption. Besides that, there are several different metrics which allow to draw conclusions about the quality of a wireless link. We present the *Receive Signal Strength Indicator (RSSI)*, *Idle Receive Signal Strength Indicator (RSSI)*, *Link Quality Indicator (LQI)* and *Signal to Interference plus Noise Ratio (SINR)* in the following.

RSSI is one of the most commonly monitored metrics in a wireless system and is measured separately for each incoming link at the mote. In the SmartMesh IP health reports [34] it is also proposed to monitor a metric called *Idle RSSI*, which is not captured per link, but per channel. In TSCH networks, motes turn their radio on and off according to a fixed schedule. The Scheduling Function (SF) determines when and on which channel the radio should be turned on for listening or receiving. Thus, motes have receive links during which they wake up but their neighbor has no packet to transmit on the link. From the receiver's perspective, we call this event an idle listen. At the end of every idle listen, the mote takes a quick low-power measurement of the RSSI on the channel it was listening to. There should be no traffic from the network on the channel at this time [36]. Thus, the *Idle RSSI* metric is used to detect interference near the mote on a particular channel.

LQI is abstractly defined by the IEEE 802.15.4 standard [6] as a characterization of the strength and/or quality of a received packet and it is intended be reported as an integer ranging from 0 to 255 [32]. The LQI is based on the quality of the first 8 symbols after the Start-of-Frame Delimiter (SFD) and therefore determined by a different strategy compared to the RSSI. Consequently, we also propose to monitor the LQI in order to get

a more complete and meaningful picture of the quality of the wireless links. Sample values of *LQI* and *RSSI* are captured during the frame receive process by default and easily accessible via HW-registers on most MCU platforms with RF elements.

Signal to Noise Ratio (SNR) is a central parameter in communications engineering and used in formulas computing the channel capacity and Bit Error Rate (BER) of theoretical channel models like the Additive White Gaussian Noise (AWGN) channel. SNR is defined as the ratio between the signal power P_S and the background noise power P_N. In reality, the wireless link is not just disturbed by the background noise, but also by interference and thus the Signal to Interference plus Noise Ratio (SINR) is commonly used to determine the quality of the wireless link. SINR is computed similarly to the SNR, but the the power of the interfering signals is added up on the background noise power, resulting in P_{I+N}. In general, measuring SINR is a non-trivial task. Qin et al. [47] present a method estimating the SNR based on the RSSI and a Kalman filter. Additionally, they introduced the term *Effective SNR* which combines the metrics SNR and LQI again in a Kalman filter operation.

Lastly, the metrics *Angle of Arrival (AoA)* and *Energy Detection (ED)* are listed among the radio statistics. Angle of Arrival (AoA) is a technique of estimating the angle at which the signal arrives at the receiver. Obviously, not all types of motes allow the calculations of the AoA, since the receiver needs to be equipped with an antenna array, where the distance between adjacent antennas is less than half of the signal's wavelength [48, 49]. However, if possible, monitoring the AoA is recommended to detect potentially interfering objects on the transmission path. Many chips that additionally support Carrier-Sense Multiple Access with Collision Avoidance (CSMA/CA) based protocols have a Clear Channel Assessment (CCA) algorithm implemented. This algorithm is used in CSMA/CA to decide for transmission or waiting. The decision is based on an Energy Detection (ED) threshold used to detect any other type of interfering RF transmissions. Chips such as the CC2420 [50] and the nRF52840-DK [51] provide the ED value in a dedicated register. Besides its use for CSMA/CA, ED also turns out to be helpful for WiFi interference quantification [52] and jamming detection [53, 54].

3.2. Device Application Metrics

The list of SW metrics includes various statistics on the state of the running application and can easily be fetched via counters and timers. A central element of a wireless sensor node is the message queue containing the packets that are either generated by the mote itself or need to be forwarded. The SmartMesh IP health reports [34] propose to periodically send the *Current/Average Occupancy* of the queue, as well as the number of *Queue Overflows*, i.e., the mote has experienced congestion. Constantly reported congestion events indicate to the user that there is a need to either enlarge the message queue size or check the routing paths to detect critical bottlenecks. Another commonly proposed performance metric is the *Computation Time* [32, 55, 33]. Assume a homogeneous network, i.e., all the nodes are based on the same HW and experience the same computational load, then the computation time for a certain task reveals the mote's capacity of executing additional workload that may be shared among the motes in the network. Thus, this metric is not necessarily relevant for extremely constrained systems, but for networks employing task allocation in order to collaboratively execute IoT applications [56]. More application related performance metrics [35] were proposed in the context of *Memfault* [57], a commercial monitoring solution which is explained more in detail in Section 4. These metrics include important counters for the total number of bytes sent and received by the mote and a connectivity counter that keeps track of connect/disconnect events to certain neighbors. Furthermore, timers are proposed for measuring the duration of interactions over peripheral buses (e.g., SPI, I^2C, ...) and the time that a display or LED was switched on. Monitoring the reboot causes is another procedure, which can be listed in the group of application metrics. The idea is to save the reboot cause to flash memory if an unplanned reboot occurs and report it to the monitoring platform once the mote is up and running again [35]. Lastly it is also recommended to keep track of the uptime of the device, i.e., keep the user informed of the time since the last reboot [35, 58].

3.3. RTOS Runtime Metrics

The next group of metrics on RTOS performance metrics even goes beyond the previously discussed application metrics. This group of metrics is particularly helpful for debugging purposes in the RTOS design phase. While it may seem superfluous at first glance to constantly keep track of these RTOS metrics at runtime, we present reasonable arguments that motivate to monitor them continuously. In Section 3.3.1, we explain each RTOS metric, highlight the purpose of monitoring it and give practical implementation hints. In Section 3.3.2, we list Operating Systems (OSs) that provide in-built functionality of capturing these metrics. We have observed that this topic has just been scarcely addressed in literature, yet. Although, numerous surveys and comparisons between RTOSs exist [59, 60, 61, 62], which all focus on performance and timing, there is no work which compares state-of-the-art RTOSs in terms of APM features. This is where we aim to contribute in the following.

3.3.1. Collection of RTOS Metrics

As the complexity of IIoT applications rises, the concept of a super-loop that periodically calls function modules reaches its limits. There is the need for splitting the work into tasks, which are sometimes also called threads. On a single CPU, only one task executes at any given time [63]. It is the job of the scheduler in an RTOS to invoke, suspend and resume tasks based on their assigned priorities. Besides the priority, the user must reserve a certain amount of bytes in Random-Access Memory (RAM) for the task stack when initialising the task. Since this value is fixed, an uncontrolled growth of the stack may result in a stack overflow, one of the most common SW issues in RTOSs. To prevent this from happening, numerous authors in literature recommend monitoring the *Task Stack Size* continuously [60, 64, 65, 63, 35, 55]. This is usually done by initializing the allocated stack memory with a certain pattern [66] and

then identify the "watermark level" by searching for the lowest memory address which violates this pattern, assuming that the stack grows from high to low memory addresses [63, Section 5].

Another important metric when it comes to task monitoring is the *Task Utilization*. Reporting a detailed overview to the user on how much time the RTOS has spent in a certain task, is extremely beneficial for checking if the system behaves as intended. In many IIoT scenarios, battery-powered devices are forced to save energy wherever it is feasible. Therefore, in such environments it is an established design pattern for RTOSs to achieve a high percentage of time spent in "sleep" mode, i.e., the idle task, where the CPU activity is brought to a minimum [55]. The system is woken up again based on an event-driven premise, e.g., when a timer expires.

How to practically implement measuring the *CPU Utilization* is studied in [67, 55]. Grabbing *CPU Utilization* at run-time and reporting it constantly is an important step to better understand the power consumption of your mote [60, 65, 35, 68]. On top of that, the measured *CPU Utilization* can be compared with the estimated CPU workload calculated during the system design phase. When designing an RTOS, an analysis technique to determine if all tasks can be scheduled to run and meet their deadlines is essential. Such an analysis is obviously dependent on the priority assignment of the tasks. Although there has been considerable research on this topic [69], the most commonly used technique is still the Rate Monotonic Analysis (RMA) which was introduced by Liu and Layland in their seminal paper [70]. They estimate the CPU utilization U by

$$U = \sum_{k=1}^{n} \frac{E_k}{T_k} \leq n\left(2^{\frac{1}{n}} - 1\right), \tag{1}$$

Where E_k is the worst-case execution time for the task and T_k the period that the task will run. n is defined as the total number of tasks and used to derive an upper bound for CPU utilization. A key feature of the RMA is the ability to prove a priori that a given set of tasks will always meet its deadlines, even during periods of transient overload [71]. It is essential to recognize RMA as a sanity check and in many cases as a model that is not calculated once, but tested using our initial assumptions and then periodically updated based on real-world measurements and refined assumptions as the system is built [55, Section 4].

The next metric in Table 2 is called *Memory Usage*. This term needs to be defined more precisely since there are different types of memory on the nodes of such a low-power wireless system. In general, we differentiate between the program Read-Only Memory (ROM), RAM and the external storage [32]. The memory usage of the program ROM is determined during compile time. Thus, it is constant and does not need to be monitored. The monitoring of the RAM usage is already captured by the metric *Task Stack Size*. The remaining memory unit whose size needs to be tracked during runtime is the external storage, which can either be another flash memory or even an SD card [32]. These forms of external storage are often used to store logs and therefore monitoring the remaining free memory space is essential to adapt the logging verbosity or being aware of the fact that logs might get lost.

At this point, we have covered the RTOS related metrics which occur the most in APM-related literature. However, there are many other metrics mainly proposed by Labrosse [63] and Hoffman [35]. Both propose to keep track of the heap size by monitoring the metric *Heap bytes used/free*. Tasks can allocate stack space from the heap by calling `malloc()`. However, if the allocated space is not set free again, this can cause the heap to fragment, which is not desirable in embedded systems [63]. Whereas heap and stack are located on the RAM, Hoffman [35] also proposes to monitor interactions with the flash memory. In particular, we include the metrics *Flash Operation Time*, *Flash Bytes Written* and *Flash Sector Erase* in Table 2. The goal is to get a detect errors related to flash aging effects caused by a large number of Program/Erase (P/E) cycles [72, 73]. Besides an increasing BER at P/E cycles, HW fingerprinting techniques relying on Physical Unclonable Functions (PUFs) may suffer under decreasing accuracy due to flash aging problems, too [74].

The remaining metrics in the RTOS runtime statistics subset in Table 2 deal with kernel objects. A Mutual Exclusion Semaphores (Mutex) is such an object and prevents multiple tasks from accessing the same shared resource simultaneously. Since Mutexs are used to protect critical global variables or message queues, monitoring *Mutex Lock Failures*, i.e., how often a certain Mutex lock function call failed, and the *Mutex Waiting Time*, i.e., the time spent waiting for a certain blocked Mutex, is proposed [63, 35]. Further RTOS metrics introduced by [63] are *Task Context Switches*, *Nested Interrupt Counter*, *Max. Interrupt Disable Time*, *Interrupt Queue Length* and *Interrupt Queue Overflows*. *Task Context Switches* is a counter which tracks the total number of performed task switches. As a stand-alone number this metric has little significance, but when comparing it across the whole system it helps to detect anomalies. The *Nested Interrupt Counter* contains the interrupt nesting level. Level 1 means servicing the first level of interrupt nesting, level 2 means the interrupt was interrupted by another interrupt. This metric is useful to spot SW issues and saves the user from time-consuming debug sessions. The *Maximum Interrupt Disable Time* can be tracked globally or on a per-task basis. This metric shows how each task affects interrupt latency. By disabling interrupts, e.g., check and set operations happen without any other Interrupt Service Routine (ISR) having the chance to execute in between. Monitoring the *Interrupt Queue Length* shows the user how many calls the interrupt handler has put in the queue. Again, this is a metric which has more significance when regarded in comparison to the reference values reported by the other motes. Lastly, the metric *Interrupt Queue Overflows* is listed. It indicates how many times an interrupt was not being able to be serviced by its corresponding task because the queue was not large enough. If the value is non-zero it indicates the user to redesign the queue size or that the processor was not fast enough.

3.3.2. APM Features in State-of-the-Art OSs

The previously presented concepts make it possible to implement the proposed metrics regardless of the used OS. Nevertheless, we provide a survey on different popular OSs and their in-built monitoring functionalities in the following. The

intention is to simplify the task of designing the APM module of such a constrained device by leaning back to these features offered out of the box. The OSs covered in this section include *Free-RTOS* [75], *Zephyr* [4], *Contiki-NG* [76], *RIOT OS* [77], *Mbed OS* [78] and *µC/OS-III* [63]. According to a recent survey by the Eclipse Foundation, FreeRTOS is still the most widespread RTOS used on MCUs [79]. The reason therefore is its small and simple kernel and the portability to a large range of different MCU HW architectures. Zephyr is an RTOS developed by the Linux Foundation and offers a small kernel, the `west` configuration and build system and a broad set of protocol stacks for IPv4 and IPv6, CoAP, Message Queue Telemetry Transport (MQTT), IEEE 802.15.4 [6], Thread [16] or BLE. Especially due to its advanced security features, Zephyr has recently become also one of the most popular choices for an RTOS. Although Contiki-NG is designed for resource-constrained IoT devices, it is not an RTOS but a common OS. Furthermore, Contiki-NG provides a 6TiSCH stack implementation and is popular in the research community thanks to the supplied *"Cooja"* simulator [80], which is able to imitate the behavior of a WSN consisting of multiple nodes running Contiki-NG. RIOT OS is an RTOS designed to fit on IoT devices equipped with just minimal memory in the order of ≈10kByte [27]. Additionally, RIOT OS comes with a native implementation of the 6TiSCH stack and offers the possibility to run OpenWSN in an own *"thread"*, the term used for a task in RIOT OS [81]. According to [79], Mbed OS is another RTOS which is especially popular among developers using ARM Cortex-M platforms. Mbed OS supports 6LoWPAN mesh architectures and the *ARM TrustZone*, a technology reducing the potential for attack by isolating the critical security firmware, assets and private information from the rest of the application [82]. In the end, we also want to mention µC/OS-III, a kernel which shines through its clean and slim architecture. µC/OS-III manages a nearly unlimited number of application tasks, priority levels and features an interrupt disable time close to zero [63]. On top of that, µC/OS-III is famous for its various monitoring features.

Once again, as in Section 3.3.1, we now go through the corresponding metrics of Table 2 and name different OS implementations that support monitoring these metrics out of the box. To the end of this section the presented OSs and their APM functionalities are summarized in Table 1.

The first metric of the RTOS runtime statistics group in Table 2 is *Task Stack Size*. Free-RTOS comes with the `uxTaskGetStackHighWaterMark` function returning the amount of stack that remained unused when the task stack was at its greatest (deepest) value [83]. In Zephyr, the compiler flag `-DCONFIG_THREAD_ANALYZER=y` needs to be set. Then calling the function `thread_analyzer_run()` prints out the stack usage for each *"thread"*, which is the term used for a task in Zephyr. Contiki-NG offers the *Stack checker library* including a `stack_check_init` function which initializes the stack area with a known pattern and the `stack_check_get_usage` function to calculate the maximal stack usage so far. Also RIOT OS implements a flag `THREAD_CREATE_STACKTEST` that can be used after a thread creation for measuring the stack's memory usage. In Mbed OS a macro has to be set, which is called `MBED_STACK_STATS_ENABLED=1`. µC/OS-III already provides variables `StkUsed` and `StkFree` for each task.

Following the order of the metrics in Table 2, the next ones are *Task Utilization* and *CPU Utilization*. Since these metrics are closely linked, whereas *CPU Utilization* just includes measuring the time spent in the "Idle" task, we cover both metrics together in the following. In Free-RTOS there is an Application Programming Interface (API) function call `vTaskGetRunTimeStats()` which returns the absolute time as well as percentage time values for each task (including the Idle Task) in a tabular format. In the case of using Zephyr, the same information (also including the time spent in the Idle Task) is again contained in the output of calling the `thread_analyzer_run()` function. Since Contiki-NG is not an RTOS relying on a task based architecture, there is no list showing the metric *Task Utilization*. However, Contiki-NG has an own module called `energest` for monitoring *CPU Utilization*. There are five predefined `energest` states, indicating whether the CPU is active, in low-power mode or deep low-power mode and whether the radio is in transmitting or listening mode. Periodically reporting information about the time spent in these states is extremely useful, since the states can directly be translated in current consumption and ultimately energy consumption based on the datasheets of some HW platforms, such as the Zolertia Z1 [84]. Threads in RIOT OS contain a "struct" called `schedstat_t` if the module `schedstatistics.h` is included. `schedstat_t` contains a timestamp of the thread's last start time, how often the thread was scheduled to run and the total runtime of the thread. For Mbed OS there is no in-built method to monitor the CPU usage. µC/OS-III offers the variable `CPUUsage` on a per-task basis and the variable `OSStatTaskCPUUsage` for the whole RTOS. `OSStatTaskCPUUsage` shows to the user the percentage the CPU was active and not idle. `CPUUsage` expresses the CPU usage of a certain task as a percentage of the total CPU usage, i.e., `OSStatTaskCPUUsage`. However, there is not a detailed overview on how much time was spent in a certain task and its fraction of the total runtime.

In Section 3.3.1, we define *Memory Usage* as a metric monitoring the external storage like another flash memory. Since this metric is rather highly dependent on the used HW platform than on the OS there are no specific implementations available and we do not include it Table 1.

The next metric in the list is *Heap bytes used/free* and can be monitored in FreeRTOS using the function call `xPortGetFreeHeapSize()`, depending on the used memory allocation implementation. FreeRTOS actually offers five different options (`heap_1.c`, ..., `heap_5.c`) for heap memory allocation. `heap_1.c` and `heap_2.c` allow monitoring the amount of heap space that remains unallocated via the API function call `xPortGetFreeHeapSize()`. However, this function is not available when using `heap_3.c`. The remaining `heap_4.c` and `heap_5.c` both support the use of `xPortGetFreeHeapSize()` again and additionally provide the `xPortGetMinimumEverFreeHeapSize()` function which returns the lowest amount of free heap space that

has existed since the FreeRTOS application has booted. At this point it might be worth mentioning that the Espressif 32 IoT Development Framework (ESP 32-IDF) [85] offers an even more verbose set of methods for monitoring the heap size on an ESP 32 HW using FreeRTOS. Zephyr and Contiki-NG currently do not offer an in-built functionality for heap memory monitoring. In RIOT OS, the function `memarray_available()` returns the number of blocks available in the memarray pool, the concept used in RIOT OS for allocating heap memory. Similarly to the stack monitoring functionality, Mbed OS comes with a macro that needs to be set to 1, i.e., `MBED_HEAP_STATS_ENABLED=1`, to enable heap size monitoring. By calling `mbed_stats_heap_get()` detailed information is reported on currently and maximum allocated bytes, on the sum of bytes ever allocated and on the number of failed allocations. To avoid memory fragmentation μC/OS-III comes with an alternative to the common heap memory concept, which is based on `malloc()` and `free()` commands. This alternative is the use of memory blocks, all having the same size. Memory blocks are again part of a partition, which are created by the function `OSMemCreate()`. There are multiple partitions possible and the memory blocks of different partitions are also allowed to have different sizes. The RTOS runtime statistics related to the memory blocks in μC/OS-III are the variable `OSMemQty` containing the number of partitions created and for each partition the variables `BlkSize`, `NbrMax` and `NbrFree` holding information about the partition's block size, the maximum number of blocks and the number of free blocks, respectively.

Following the order of metrics in Table 2, the next ones would be related to flash memory monitoring. Similarly to *Memory Usage* these metrics monitoring the flash memory are not included into Table 1 due to their strong correlation with the used HW platform.

All the remaining metrics are be summarized under the term *Kernel Objects*. μC/OS-III is basically the only RTOS which provides such a detailed monitoring functionality for Mutexs, *Task Context Switches*, *Nested Interrupt Counter*, *Max. Interrupt Disable Time*, *Interrupt Queue Length*, *Interrupt Queue Overflows* and many more.

Table 1
Comparison between APM functionalities in (RT)OS implementations.

OS	Task Stack Size	Task Utilization	CPU Utilization	Heap bytes used/free	Kernel Objects
FreeRTOS [75]	✓	✓	✓	✓	
Zephyr [4]	✓	✓	✓		
Contiki-NG [76]	✓		✓		
RIOT OS [77]	✓	✓	✓	✓	
Mbed OS [78]	✓	✓	✓	✓	
μC/OS-III [63]	✓		✓	✓	✓

3.4. Networking Metrics (node-centric)

The second set of metrics in Table 3 lists all the statistics that we consider as important to monitor the mote's reliability in terms of networking and to identify the problem when packet losses occur. Motes in 6TiSCH based systems are arranged in a mesh topology. Thus, all motes have at least one neighbor node. Neighbors may be other motes or the network manager, which is also called Border Router (BR). The first subset of metrics comprises packet statistics characterize the path to the mote's neighbors. Therefore, a constantly updated list of neighbors should be reported to the monitoring platform [34]. The counters *Neighbor Transmitted Packets*, *Neighbor Transmit Failures*, *Neighbor Retransmissions* and *Neighbor Received Packets* are straight forward and build the basis for calculating the metrics *Expected Transmission Count (ETX)* [95], *Mote Packet Delivery Ratio (PDR)* and *Mote Packet Retransmission Ratio (PRetR)*. The formulas are given in Table 3. We have observed that in literature different terms and definitions for the metric Packet Delivery Ratio (PDR) are used. We also agree on the following definition of these terms [32, Section 4.2.2].

$$\mathrm{PDR_A} = \mathrm{PRR_B} = 1 - \mathrm{PLR_A} \tag{2}$$

The equation expresses the relationship between PDR, Packet Reception Rate (PRR) and Packet Loss Ratio (PLR). The PDR of mote A for its link to mote B is equal to the Packet Reception Rate (PRR) of mote B for its link to mote A. The complement of PDR is called Packet Loss Ratio (PLR).

Further neighbor metrics are the hop depth and the *β-factor* [32]. The hop depth is defined as the number of motes that lie on the path of a message from the mote to the manager. A changing hop depth indicates a dynamic in terms of lost neighbors or new routing paths. The β-factor measures the link burstiness and allows to reason about how long a protocol should pause after encountering a packet failure to reduce its transmission cost [96].

Besides the mentioned mesh topology, TSCH is a main characteristic of 6TiSCH networks. The concept of TSCH consists in iteratively changing between the available frequency channels according to a fixed schedule. The SmartMesh IP health reports [34] propose to monitor the number of total *Unicast Attempts* and *Unicast Failures* for each of the channels. The resulting complement ratio is called *Channel Stability* and usually shows the presence of interference and multi-path fading when comparing the values across the different channels [13, 91, 92].

The following subset of metrics in Table 3 is subsumed under the term *Mote Scheduling Statistics* and deals with node-centric networking statistics related to the employed scheduling algorithm. In TSCH networks time is cut in timeslots which typically have a duration of 10ms [11]. Since the frequency band is also cut in 16 different channels for communication, a matrix structure arises. Each cell of this matrix is characterized by a time- and channel-offset, respectively. The task of the SF is to assign a cell to each communication link between two neighbors. Thus, the motes know exactly at which time to transmit, receive or sleep. The *Average Time between Transmits* is listed as one of the KPIs in 6TiSCH networks [88]. This metric is used to check the fairness of the SF. Another metric closely linked to the SF is bandwidth [5]. We differentiate between *Assigned* and *Needed Bandwidth*, respectively. Based on these values, it can be evaluated if the used SF is still suited for the ongoing traffic at each mote in the system [94]. Another KPI of

Table 2
Overview of Device HW and SW Metrics

Metric	Unit	Explanation	References
Device HW Metrics			
Physical Metrics			
Battery Voltage	V	Voltage measured with a fuel gauge IC or high-res. ADCs to estimate battery SoC	[34, 38]
Charge Consumption	mC	Lifetime charge consumption measured with Coulomb counters to detect aging problems	[34, 40, 14, 38, 68]
Energy Consumption	J	Instantaneous amount of energy required for certain operations captured by on-board instrument-based measurement	[14, 86, 26, 32, 33]
Core Temperature	°C	Temperature measured near CPU core to detect overheating issues	[42]
Ambient Temperature	°C	Temperature measured in environment to detect harmful external influences	[34]
Crystal Metrics			
Resynchronization Time	s	Time elapsed since last synchronization (ΔT) in TSCH network	[45, 87, 44, 46]
Clock skew/Time Offset	s	Desynchronization/offset ϵ to time master measured at each resynchronization	[45, 87, 44, 46, 32]
Experimental Clock Drift	ppm	The experimental clock drift rate r_{exp} is defined as $r_{\text{exp}} = \frac{\epsilon}{\Delta T}$	[45, 87, 44, 46, 33]
Radio Statistics			
Radio Duty Cycle (RDC)	%	Ratio between the cumulative time that the radio chip is powered and the measurement period	[88, 60, 8, 26, 32, 35]
Receive Signal Strength Indicator (RSSI)	dBm	RSSI is a metric which is captured for each neighbor-link individually	[34, 89, 31, 90, 91, 32, 92, 93, 53, 33]
Idle RSSI	dBm	Idle RSSI is measured during Idle listens for each channel individually in TSCH networks	[34]
Link Quality Indicator (LQI)	-	LQI is abstractly defined by the IEEE 802.15.4 standard as a characterization of the strength and/or quality of a received packet	[32, 33, 6]
Signal to Interference plus Noise Ratio (SINR)	dB	SINR is defined as SINR = $\frac{P_S}{P_{I+N}}$,where P_S is the power of the signal and P_{I+N} is the power of the interference plus background noise	[47, 32]
Angle of Arrival (AoA)	°	AoA is used for localization and detection of external influences on the signal paths	[48]
Energy Detection (ED)	J	ED is used to measure the level of energy in the frequency band	[32, 52, 53]
Device Application Metrics			
Message Queue Occupancy	%	Latest occupancy of the message queue	[34]
Avg. Message Queue Occ.	%	Average occupancy of the message queue over a certain time interval	[34, 60]
Message Queue Congestion	-	Counter of message queue overflows	[34, 94]
Computation Time	s	Duration for the CPU to complete a certain piece of computation	[32, 55, 33]
Bytes sent/received	bytes	Counter for bits sent/received via the communication module	[35]
Connectivity Counter	-	Counter of connect/disconnect events	[35]
Peripheral Interaction Timer	s	Timer for how long the interaction to certain peripheral devices such as sensor, flash storage, etc. via communication buses (SPI, I^2C) has taken	[35]
Display/LED Timer	s	Timer for how long a display or LED of the device was on	[35]
Reboot Cause + Counter	-	Counter and root cause of reboot events	[35]
Device Uptime	d	Time for how long the device is up and running since last reboot	[35, 58]
RTOS Runtime Metrics			
Task Stack Size	%	Percentage indicating how much size of the initially allocated task stack size is already occupied in order to detect the danger of stack overflows	[83, 60, 64, 65, 63, 35, 55]
Task Utilization	s	Timer for how long the OS has spent time in certain task	[35, 55]
CPU Utilization	%	Percentage of time that RTOS not spends in idle task	[67, 60, 65, 63, 35, 55]
Memory Usage	%	Monitoring of HW that provide external memory rescources concerning free disk space	[60, 90, 64, 65, 32, 35]
Heap bytes used/free	%	Amount of heap memory used/free	[63, 35]
Flash Operation Time	s	Time spent in flash operations (P/E-Cycles)	[35]
Flash bytes written	bytes	Counter for bytes written to flash memory	[35]
Flash sector erase	-	Counter for how often sectors in flash have been erased	[35]
Mutex Lock Failures	-	Counter for how often a certain Mutex lock function call failed	[63, 35]
Mutex Waiting Time	s	Time spent waiting for a certain Mutex	[63, 35]
Task Context Switches	-	Variable that accumulates the number of context switches performed	[63]
Nested Interrupt Counter	-	Variable containing the interrupt nesting level.	[63]
Max. Interrupt Disable Time	s	Timer for capturing the maximum interrupt disable time	[63]
Interrupt Queue Length	-	Variable indicating the current number of entries in the interrupt handler queue	[63]
Interrupt Queue Overflows	-	Counter for how often an interrupt was not being able to be serviced	[63]

every communication system is obviously the data rate which is also closely linked to the time between transmit cells allocated by the SF. Yuan et al. [32] propose to measure the data rate at the link-layer and define it as the amount of link-layer payload (excluding link-layer header) per certain unit of time. Furthermore,the authors [32] present the *Next Hop Switch Rate* metric in order to have an indicator for the routing stability. The routing topology, i.e., the set of links a message crosses from a mote through the mesh to the manager, is determined by the RPL. The motes lying on this path are called *parent* nodes. High values of the *Next Hop Switch Rate* might be a sign for dynamics in the network caused by moving objects or decreasing link quality, which influences the Objective Function (OF) of the RPL. This means the routing topology keeps changing with the radio environment even when the number of nodes in the network do not change [97]. Of course, the RPL topology also varies with nodes leaving or joining the network. Thus, we recommend tracking the number of *Network Joining Events* of the mote [5, 98]. This value becomes essential when comparing it to the *Reboot Counter* introduced in Section 3.1. If the numbers do not match, this may be an indicator for a connection loss due to SW issues or external disturbances of the wireless link.

The next subset of metrics contains general *Mote Packet Statistics* and consists exclusively of counters and ratios built from these counters. At first glance it may seem a bit petty to implement so many counters, but it turns out to be worth the effort when it comes to finding the reason for packet losses. Since we have already introduced the counters for successful and failed packet transmissions to certain neighbors of the mote, it is obvious to sum over all neighbors and obtain the total number of packets transmitted and transmission failures, respectively. The *Packets Dropped at the MAC Layer* due to exceeded retry count, aging or no route are tracked as a subset of transmit failures [5] . Furthermore, *Retransmissions* are essential to be monitored [90, 92]. They form a sub-quantity of the transmitted packets. Analogous to the neighbor packet statistics, we can also calculate the *PDR* and the *PRetR* of the mote according to the formulas in Table 3. Yuan et al. [32] listed the metric *Information fan-out* as the ratio of received packets over transmitted packets over a short period of time at an intermediate node. This metric is useful for congestion detection and requires a counter for received packets, which is simply the sum of all received packets over all neighbor links of the mote. The next counters listed in Table 3 are called *Transmit Ready Packets* and *Transmit Errors*. Both are implemented in the health reports of SmartMesh IP [5] and describe the border from the NET layer to the MAC layer in the OSI model. *Transmit Ready Packets* counts the packets handed successfully from the network to the MAC layer and thus are ready to be sent. Contradicting are the *Transmit Errors*, i.e., packets that were not sent and dropped due to congestion and timeouts. These two counters are used for the calculation of the *Mote Availability* [94]. To determine the reliability of a mote we need to track the number of packets correctly received by the manager from this mote as well as the number of packets lost on the way to the manager from this mote. We denote by these metrics *Packets received Mote → Manager* and *Packets lost Mote → Manager*, respectively. Consequently, the mote's reliability can be computed according to the formula in Table 3.

The last subset of metrics in Table 3 is summarized under the term *Mote Packet Validation*. The SmartMesh IP health reports [5] have a counter for the number of packets that were discarded due to validation errors. This quantity can even be subdivided in counters, which specify the reason why validation has failed. Validation may fail because of *Decryption Errors* which are detected depending on the employed security concept. For encryption, the 6TiSCH architecture, e.g., supports Object Security for Constrained RESTful Environments (OSCORE) [99], an IETF standard compatible with CoAP. SmartMesh IP relies on a concept of Pre-Shared Keyss (PSKs) [36]. These security mechanisms are independent of IEEE 802.15.4, which also comes with an own in-built encryption algorithm based on Advanced Encryption Standard (AES) in order to achieve data integrity through a Message Integrity Code (MIC). If the check of this Message Integrity Code (MIC) fails, the *Authentication Error* counter is incremented. Additionally, there is a counter for *Cyclic Redundancy Check (CRC) Errors*. For each message, a Cyclic Redundancy Check (CRC) code is computed and appended to the message as redundancy, which gets validated at the receiver's side. In our literature survey, we found that different authors [5, 32, 33] propose to count the number of CRC errors in order to detect the presence of unusual traffic or jamming that is interfering with the network [100, 101]. Based on the CRC error counter and the previously introduced *Received Packets* counter, i.e., packets that have passed the preamble and SFD check, one can compute the *Packet Corruption Rate (PCR)*. The Packet Corruption Rate (PCR) is not necessarily a measure for the detection of harmful attacks but also for co-existing, interfering networks, respectively.

3.5. Networking Metrics (global)

Finally, Table 4 shows a collection of metrics that cannot be monitored on a single device, since system-wide knowledge is required. Typically these metrics can be obtained at the network manager, i.e., the Low-Power and Lossy Network (LLN)-BR in 6TiSCH networks [23]. The first two metrics *Network Efficiency* and *Network Fairness* were defined by Hull et al. [104] and are also included in the survey of Yuan et al. [32]. The definition of *Network Efficiency* η [104] relies on a variable U which quantifies the set of "useful packets". If we define every packet as useful in our 6TiSCH scenario, η is simply the inverse metric of PRetR averaged over the whole network and thus forms an important metric for energy consumption estimations. *Network Fairness* is an indicator whether the packets arriving at the manager are fairly shared among the motes in the network. In 6TiSCH networks that are based on the RPL and relying on a fixed schedule, this metric's significance is minor, since it is clearly dependent on the SF. *Network Throughput* [32, 33] is another global metric and is defined as the total amount of data received at the collection points, i.e., the manager, over a certain period of time (e.g., the health-report/heartbeat interval). Anomalies in the *Network Throughput* over time indicate crucial events such as joining motes, connection-losses or unusual

Table 3
Overview of node-centric Networking Metrics

Metric	Unit	Explanation	References
Networking (node-centric)			
Neighbors (Mesh)			
Neighbor (Nbr.) ID List	-	A list of all registered neighbor device names/IDs/MAC addresses	[34]
Nbr. Transmitted Packets	-	Counter for packets transmitted to a certain neighbor	[34]
Nbr. Transmit Failures	-	Counter for failed transmissions to a certain neighbor	[34]
Nbr. Retransmissions	-	Counter for packets retransmitted to a certain neighbor	[90]
Nbr. Received Packets	-	Counter for packets received from a certain neighbor	[34]
Nbr. Packet Delivery Ratio (PDR)/Nbr. Path Stability	%	PDR of a link to a certain neighbor: $\text{PDR}_{\text{Nbr.}} = 100 \cdot \left(1 - \frac{\text{Nbr. Transmit Failures}}{\text{Nbr. Transmitted Packets}}\right)$	[34, 94, 36, 91, 26, 32]
Nbr. Packet Retransmission Ratio (PRetR)	%	PRetR of a link to a certain neighbor: $\text{PRetR}_{\text{Nbr.}} = 100 \cdot \left(\frac{\text{Nbr. Retransmissions}}{\text{Nbr. Transmitted Packets}}\right)$	[90, 26]
Hop Depth	-	Number of hops to the manager	[34, 32]
β-factor	%	Characterization of the *burstiness* of a wireless link	[32]
Channel Statistics (TSCH)			
Channel (Ch.) Number	Hz	A list of all available channels and their corresponding frequency	[34]
Ch. Unicast Attempts	-	Counter for Unicast attempts over a certain channel	[34]
Ch. Unicast Failures	-	Counter for missed Acknowledgements (ACKs) over a certain channel	[34]
Ch. Stability	%	Stability over a certain channel: $\text{Stability}_{\text{Ch.}} = 100 \cdot \left(1 - \frac{\text{Ch. Unicast Failures}}{\text{Ch. Unicast Attempts}}\right)$	[31, 91, 92]
Scheduling Statistics (Mesh)			
Avg. Time between Tx	s	Time interval between two transmit slots of a mote according to schedule	[88]
Assigned Bandwidth	ms	Bandwidth assigned to the mote according to the schedule	[94, 36]
Needed Bandwidth	ms	Total bandwidth needed by the mote to cope with the traffic	[94, 36]
Data Rate	$\frac{\text{bytes}}{\text{s}}$	Amount of link-layer payload (excluding link-layer header) per unit of time	[8, 32]
Next Hop Switch Rate	$\frac{\text{switch}}{\text{h}}$	Rate of routing change events affecting the destination paths of the mote	[32]
Number of Joining Events	-	Counter of network join events to detect mobility and undesired reboots	[94, 98]
Packet Statistics			
Transmitted Packets	-	Counter for total number of packets (to all N neighbors including the manager) transmitted: $\sum_{i=1}^{N}(\text{Nbr. Transmitted Packets})_i$	[94]
Transmit Failures	-	Counter for total number of packets packets (to all N neighbors including the manager) that did not reach their destination correctly: $\sum_{i=1}^{N}(\text{Nbr. Transmit Failures})_i$	[94]
Pkts Dropped MAC Layer	-	Number of packets dropped by MAC Layer (subset of Transmit Failures)	[34]
Retransmissions	-	Counter for total number of packets packets (to all N neighbors including the manager) retransmitted: $\sum_{i=1}^{N}(\text{Nbr. Retransmissions})_i$	[90, 92]
Mote PDR	%	PDR over all links of the mote: $\text{PDR}_{\text{Mote}} = 100 \cdot \left(1 - \frac{\text{Transmit Failures}}{\text{Transmitted Packets}}\right)$	[94]
Mote PRetR	%	PRetR over all links of the mote: $\text{RetR}_{\text{Mote}} = 100 \cdot \left(\frac{\text{Retransmissions}}{\text{Transmitted Packets}}\right)$	[90]
Mote ETX	-	The Expected Transmission Count (ETX) of a link is the predicted number of data transmissions required to send a packet over that link, including retransmissions.	[95, 102, 103]
Received Packets	-	Counter for total number of packets packets (from all N neighbors including the manager) received $\sum_{i=1}^{N}(\text{Nbr. Received Packets})_i$	[94]
Information fan-out	%	It is defined as the ratio $100 \cdot \left(\frac{\text{Received Packets}}{\text{Transmitted Packets}}\right)$ and used for congestion detection	[32]
Transmit Ready Packets	-	Counter for packets handed from NET to MAC layer and ready to be sent	[34]
Transmit Errors	-	Counter for packets that were not sent and dropped due to congestion and timeouts	[34]
Mote Availability	%	Avail. of a mote is defined as $\text{Avail}_{\text{Mote}} = 100 \cdot \left(1 - \frac{\text{Transmit Errors}}{\text{Transmit Ready Packets+Transmit Errors}}\right)$	[94, 36]
Pkts. rec. Mote → Manager	-	Counter for total number of packets received by the manager from the mote	[5]
Pkts. lost Mote → Manager	-	Counter for total number of packets sent by the mote and lost on the way to the manager	[5]
Mote Reliability	%	The rel. of the mote is defined as $\text{Rel}_{\text{Mote}} = 100 \cdot \left(1 - \frac{\text{Packets lost Mote}\rightarrow\text{Manager}}{\text{Packets received Mote}\rightarrow\text{Manager}}\right)$	[32, 89]
Packet Validation			
Received Invalid Packets	-	Number of packets discarded by NET layer due to due to validation errors	[34]
Decryption Errors	-	Packets that fail in the decryption process	[34]
Authentication Error	-	Message Integrity Code (MIC) failed due to authentication error	[34]
CRC Errors	-	Number of incoming packets with MAC-layer Cyclic Redundancy Check (CRC) errors to indicate the presence of unusual traffic or jamming that is interfering with the network	[34, 32, 100, 101]
Mote PCR	%	Packet Corruption Rate (PCR) defines the ratio of of the received corrupted packets over the received packets (passing the preamble and SFD check)	[32, 33]

traffic patterns. The κ-factor is a metric originally defined by Srinivasan et al. [105] and also mentioned in the collection of Yuan et al. [32]. κ impartially captures to what degree packet reception on different links is correlated. This means that κ is not a global metric for the whole network, but one that compares two links which share the same transmitter. Since the required information to compute κ goes beyond the knowledge of a single mote, this metric is listed under system-wide metrics in Table 4. The κ-factor is not trivial to compute, however, it can deliver information about the protocol performance. The following counters in Table 4 are called *Total Transmit Packets Ready*, *Total Transmit Errors*, *Total Packets Generated* and *Total Packets Lost*. They are simply obtained by summing over the corresponding node-centric metric in Table 3 for all motes [94]. Similarly as in Section 3.4, one can compute the *Network Availability* and the *Network Reliability*, which is considered as the most important KPI by most papers included in this survey.

The last subset of global metrics is summarized under the term *Timing*. The *Network Lifetime* is a timer which falls into this group. It is captured at the network manager and starts at the moment at which motes are allowed to join the network. The metric is also listed by Yuan et al. [32] and Ojeda et al. [33] as a useful metric for the detection of aging problems. Another timer metric is called *Network Formation Time* [88]. It is measured once at the initial phase when the network is forming and refers to the end of the secure joining phase of the network. The *Network Formation Time* becomes interesting when running multiple different, but comparable, deployments. All the remaining timing metrics are different forms of the KPI *Latency*. Besides reliability, latency is the metric that appeared the most when it comes to most important metrics in a wireless system. However, when talking about latency, most authors actually refer to the metric *Network Upstream Latency*. It is usually defined as the *Upstream Mote Latency* averaged over all motes in the network, where *Upstream Mote Latency* is the average time taken for packets generated at a certain mote to reach the manager. Obviously, the same timers exist for *downstream* direction, i.e., from the manager to the mote, and for *point-to-point* directions. *Point-to-Point Mote Latency* can be stored in form of a dictionary with neighbors (keys) and corresponding average time taken for packets to reach this neighbor (values). It is worth mentioning that the manager is not part of this neighbor list, since the value would correspond to the *Upstream Mote Latency* in this case.

4. Application Performance Monitoring Frameworks for Low-Power Wireless Systems

In the last section, we covered what metrics we consider as important to monitor in low-power wireless systems. In this section, we want to shift our focus on APM frameworks, which may be used for this purpose. We begin with a literature survey on APM frameworks in constrained, wireless systems in Section 4.1. In the following Section 4.2 we discuss to what extent the metrics from Section 3 are taken into account in each of the frameworks. Eventually, we describe how metrics find their way from the device to the user's APM platform in an active monitoring system. Thus, we cover the steps of metric collection on the device in Section 4.3, exporting in Section 4.4 and processing on receiver side in Section 4.5.

4.1. Related Work

There are numerous APM techniques for IoT addressing different network architectures and technologies. We cluster these techniques in 6 different groups. After explaining the core concept of each group, we spend a subsection on each of them to list corresponding frameworks in literature. The first group is called active monitoring and often referred to as "traditional monitoring", i.e., the metrics make their way to the user by being sent in packets dedicated for health monitoring. In active monitoring, nodes periodically send out health information in form of additional packets, called health reports, heartbeats or snapshots, depending on the project. These terms are interchangeable and describe the same contextual concept. Clearly, active monitoring means an additional overhead on top of the actual application resulting in performance degradation of the system in terms of energy consumption, bandwidth and device resources, i.e., additional storage and CPU load, respectively. Passive monitoring systems do not penalize neither the operation nor the performance of the system. The approach relies on spying the communications among the motes and inferring on the health condition. However, such passive tools may deliver insufficient information for a complete analysis. The third group covers hybrid approaches trying to combine the advantages of the afore-mentioned two groups. Group 4 contains *"Piggyback"* Methods for APM. In order to mitigate the drawbacks arising with active monitoring, the piggyback method aims to enrich the packets generated by the actual application with metrics in the frame's payload and Information Element (IE) field. The fifth group of network monitoring techniques is called Alternate Marking Performance Measurement (AM-PM). The idea is that every packet of the monitored flow in the downstream, i.e., from the manager to the mote, carries one or two marking bits used for signaling and coordinating measurement events across the measurement points. In upstream communication, these marking bits are used for monitoring. In contrast to active monitoring or piggybacking, AM-PM brings no extra overhead on the monitored packets because AM-PM makes use of already existing bits in the IEEE 802.15.4 header. In the last group we present In-Band Network Telemetry (INT), which forms an alternative to the traditional monitoring techniques. INT is proposed as a framework allowing the collection and reporting of network state, by collecting metrics per-hop and per-frame as packets traverse the network. Recently, the INT approach was also adapted to be compatible with the 6TiSCH stack.

4.1.1. Active Monitoring

Two early active APM frameworks are *"Sympathy"* [106] and *"Memento"* [107], which both provide failure detection and symptom alert service. In 2010, a poller-pollee concept for monitoring a distributed wireless system, not yet relying on the RPL, was introduced [108]. Later, Liu et al. [109] present a

Table 4
Overview of System-wide/ Global Networking Metrics

Metric	Unit	Explanation	References
Networking (global)			
Packet Statistics (global)			
Network Efficiency	%	Network efficiency measures the average fraction of transmissions in a data collection that contributes to a packet's eventual delivery at the sink	[32, 104]
Network Fairness	%	Network fairness measures whether the packets received by the sink are fairly shared among all source nodes	[32, 104]
Network Throughput	$\frac{\text{bits}}{\text{s}}$	Throughput is defined as the total amount of data received at the collection points (manager) over a certain period of time (e.g., the network lifetime)	[32, 33]
κ-factor	%	Inter-link reception correlation is defined as the normalized correlation coefficient between the packet receptions of two links that share the same transmitter	[32, 105]
Total Pkts. Transmit Ready	-	System-wide number of packets sent from NET to MAC layer and ready to be sent of all M motes: $\sum_{i=1}^{M}(\text{Transmit Ready Packets})_i$	[94, 36]
Total Transmit Errors	-	System-wide number of packets packets that were not sent and dropped due to congestion and timeouts of all M motes: $\sum_{i=1}^{M}(\text{Transmit Errors})_i$	[94, 36]
Network Availability	%	$\text{Availability}_{\text{Network}} = 100 \cdot \left(1 - \frac{\text{Total Transmit Errors}}{\text{Total Packets Transmit Ready+Total Transmit Errors}}\right)$	[94, 36]
Total Packets Generated	-	System-wide number of packets generated by all M motes: $\sum_{i=1}^{M}(\text{Received Packets})_i$	[94, 36]
Total Packets Lost	-	System-wide number of packets lost by all M motes: $\sum_{i=1}^{M}(\text{Packets lost Mote} \rightarrow \text{Manager})_i$	[94, 36]
Network Reliability	%	The network reliability is the portion of the packets injected into the network that were received by their final destination. $\text{Rel}_{\text{Network}} = 100 \cdot \left(\frac{\text{Total Packets Generated}}{\text{Total Packets Lost}}\right)$	[94, 36, 88, 98, 89, 14, 8, 32]
Timing			
Network Lifetime	d	Time span elapsed since the deployment of a network	[32, 33]
Network Formation Time	s	Duration of the initial phase until the secure join process of all motes is finished	[88]
Upstream Mote Latency	ms	Average time taken for packets generated at a certain mote to reach the manager	[34, 94, 88, 98, 89, 14, 8, 26, 32, 33]
Downstream Mote Latency	ms	Average time taken for packets generated at the manager to reach a certain mote	[88, 89, 32]
Network Upstream Latency	ms	Upstream Mote Latency averaged over all N motes $\frac{1}{N}\sum_{i=1}^{N}\text{Upstream Mote Latency}_i$	[34, 88, 89, 8, 32]
Network Downstream Latency	ms	Downstream Mote Latency averaged over all N motes $\frac{1}{N}\sum_{i=1}^{N}\text{Downst. Mote Latency}_i$	[88, 98, 89, 8, 32]
Point-to-Point Mote Latency	ms	Dictionary with neighbors (keys) and corresponding average time taken for packets to reach this neighbor (values)	[88, 89]
Point-to-Point Network Latency	ms	P2P Mote Latency averaged over all N motes $\frac{1}{N}\sum_{i=1}^{N}\text{P2P Mote Latency}_i$ - Latency of traffic among motes excluding the manager	[88, 89, 8]

self-diagnosis concept for large-scale WSNs. The authors even refined this concept by designing fault detectors based on Finite State Machiness (FSMs) [110] . For industrial environments, an active monitoring framework is presented by Raposo et al. [111]. The authors provide a survey on IIoT standards such as WirelessHART and ISA 100.11a and propose a monitoring structure evaluated on a WirelessHART testbed. In 2017, the IETF has standardized the Representational State Transfer Configuration Protocol (RESTCONF) protocol [112] which uses structured data (Extensible Markup Language (XML) [113] or JavaScript Object Notation (JSON) [114]) and Yet Another Next Generation (YANG) [115] to provide a REST-like API enabling the programmatic access to different network devices. However, the RESTCONF APIs use Hypertext Transfer Protocol (HTTP) methods. Therefore, a similar approach for a management interface for CoAP instead of HTTP, was designed. The first initiative was called CoAP Management Interface (CoMI) [116] and later referred to as CORECONF [117]. Recently a study on RESTCONF, CORECONF and their performance in constrained IIoT environments has been published [118]. Although CORECONF is frequently proposed as an option for active monitoring [119], it has not yet been considered in recent work because at the time of writing, CORECONF does not have a production-ready implementation [120]. In Section 3, we have already presented a large number of metrics contained in the health reports of SmartMesh IP [5]. These health reports are sent in fixed intervals and thus can also be categorized as an active monitoring framework. A *NetworkHealthAnalyzer.py* [94] Python script is also part of this active monitoring framework for SmartMesh IP. The script is part of the SmartMesh Software Development Kit (SDK), a set of Python applications interacting with the serial API of the SmartMesh IP devices. The *NetworkHealthAnalyzer.py* script is motivated by practical concepts on monitoring the health condition of a SmartMesh IP network [36, Chapter 6]. The goal is to aggregate and combine different metrics of the health reports and interpret them in order to track the status of the network. In the book *"Embedded Software Design"* [55], Beningo names two other tools for active monitoring of WSNs: the *Percepio Tracealyzer* [121] and the reliability platform *Memfault* [57].

The Percepio Tracealyzer is a tool for runtime monitoring optimized for FreeRTOS-based applications. Besides the streaming mode which requires a wired connection just like with a HW debugger, there is the snapshot mode, where the desired metrics are stored in a buffer, that needs to be transferred to the computer running the Tracealyzer SW [122]. Memfault is a company offering an IoT reliability platform supporting the design of more robust devices via performance monitoring, debugging and Over-the-Air (OTA) updates. The framework offered by Memfault comprises the whole chain starting from data collection inside the OS running on the device, up to a cloud including a front-end for visualization and analysis of the collected metrics. The concept of Memfault is to pack metrics in so-called heartbeats and send them periodically via the communication interface of your device to an edge device where the heartbeat *"chunks"* are pushed to the Memfault cloud.

4.1.2. Passive Monitoring

The concept of passive WSN monitoring was introduced by Ringwald et al. [123] in form of a tool called *Sensor Network Inspection Framework (SNIF)*. In contrast to active approaches, SNIF does not require additional bandwidth for the monitoring traffic. The idea of SNIF is to install a Deployment Support Network (DSN) alongside the actual WSN. The algorithms used by the SNIF tool for packet sniffing, packet decoding and the "Path Analyzer" are later formalized by the same authors [124]. Passive Distributed Assertions (PDA) [125] and *"Pimoto"* [126] are examples for other passive monitoring mechanism based on packet sniffing. Both rely on Bluetooth nodes for sending the sniffed data in the DSN. For more passive and also active monitoring techniques, the interested reader is referred to the survey paper on *"Diagnostic Tools for Wireless Sensor Networks"* [127].

4.1.3. Hybrid Approaches

Another survey on active and passive APM methods is given by Mendoza et al. [128]. However, the authors also present hybrid monitoring solutions which try to combine both, active and passive approaches, to realize greater observability of the monitored system. Additionally, they come up with the Hybrid Monitoring Platform (HMP), a new hybrid approach harvesting the information both actively, i.e., directly from the sensor nodes, and passively, i.e., by means of messages captured from the wireless system, causing a very low intrusion in the network. In 2013, Keller et al. [129] introduced the first hybrid health monitoring system. It utilizes passively reconstructed packet information while only adding one bit of extra information to improve the failure detection accuracy. Although the solution offered by the company Sternum IoT [58] has parallels to the active monitoring platform Memfault, we put it in the group of hybrid approaches. Sternum IoT comes with an observability SDK promising continuous monitoring and an end-to-end platform for device manufacturers, offering built-in security, granular remote visibility, and valuable business insights. The company has a strong focus on security and enhances Zephyr's in-built security features. Due to their offered on-board device exploitation prevention to detect dangers, threats and attacks, the classification as hybrid approach seems valid.

4.1.4. Piggyback Method

Based on the results of Dunkels et al. [130], the motivation for piggybacking has arised. Actually, the authors showed that piggybacking multiple beacons in a single transmission significantly reduces energy costs compared to the generation of new packets. Soon, Dressler et al. [102] have adopted the piggybacking concept for energy-efficient monitoring of WSNs. At that time, the concept of piggybacking has also been introduced for 6LoWPAN and the RPL [131]. Hereby, it is counted on the underlying APM concept based on a poller-pollee approach similar to the one in the group of active monitoring approaches [108]. Fanucchi et al. [103] set up on this work and present piggybacking for IEEE 802.15.4e networks by defining a method exploiting the IE. The IE is part of a IEEE 802.15.4 frame and defined by the standard [10]. Finally, Gaillard et al. [132] extend the idea of piggybacking the IE also to 6TiSCH networks.

4.1.5. Alternate-Marking Performance Measurement (AM-PM)

Alternate Marking is a monitoring method which goes back to the idea of *"traffic coloring"* which first has come up in 2011 as an approach by Telecom Italia for measuring packet loss [133]. In the following years, efforts in standardizing have resulted in AM-PM, a method for packet loss, delay, and jitter measurements on live traffic [134]. Thus, AM-PM has become an efficient measurement method for monitoring network flows, i.e., loss and delay measurements, with low overhead, namely at the cost of one or two bits per data packet. Experimental results for the evaluation of the impact of AM-PM on large deployments with wired cellular networks, have been published, too [135, 136]. Karaagac et al. [137] are the first and only ones, who integrate the AM-PM method in low-power wireless systems. They provide an AM-PM extension to the 6TiSCH stack, which comprises an adapted IEEE 802.15.4 frame format with certain bits reserved for AM-PM and a network monitoring application used to collect and analyze AM-PM telemetry data at each hop. Although AM-PM is the monitoring method with the least impact on the system performance, it is not compatible with transferring verbose information, such as a large set of metrics. Liu et al. [138] earlier proposed an APM technique called Passive Diagnosis (PAD). They declared PAD as a passive monitoring approach used for inferring the root causes of abnormal phenomena in a WSN. However, PAD employs a packet marking and packet parsing algorithm, respectively. The inferred model is capable of reasoning root causes based on passively observed symptoms, which has also been evaluated in a sea monitoring testbed. Just like AM-PM, PAD does not incur additional traffic overhead for collecting desired information in contrast to the active approaches. Due to the similarities of PAD to AM-PM, we decided to put the APM frameworks into the same group.

4.1.6. In-Band Network Telemetry (INT)

The idea of INT has been born in 2015 during the efforts for practical applications of the P4 language [139]. As a result, an approach to gather so-called telemetry metadata for packets traversing network segments has been presented [140]. Soon, the P4 Language Consortium has adopted the concept [141] and paved the way for recent research related to INT. Gulenko et al. [142] for instance demonstrate the use of INT in a virtual switch. The core idea is to attach monitoring information directly to application traffic flowing through the different network devices. The host or device that generates the traffic of interest initiates the data collection by attaching INT commands to the outgoing traffic. In general, INT differs from piggybacking in the fact that monitoring data can be appended at each intermediate hop that the message traverses from source to the sink. Gaillard et al. [132] present in their paper on piggybacking the concept of "shared" and "exclusive" containers. Shared containers in this context actually defines the same as INT just under a different name, because the term INT has been restricted to wired applications in the beginning. But recently there has also been considerable efforts in the wireless field, too [143]. In 2020, Karaagac et al. [119] have adopted the INT concept to 6TiSCH networks. They prove that the INT technique enables ultra-efficient network monitoring operations without any effect on the network behavior and performance. Very recently, an analytical traffic model of 6TiSCH using INT has been derived [144]. The presented model can calculate the network traffic overhead in a 6TiSCH network, using INT to predict the number of transmitted and received bytes.

4.2. Comparison of Metrics used in APM Frameworks

The literature reviewed on APM frameworks in Section 4.1 emphasizes methodology over the specific metrics utilized. However, the types of metrics discussed in these literature sources illustrate the distinct characteristics of each framework class. Consequently, we have checked the literature on these frameworks for the surveyed metrics presented in Section 3. The result is shown in Table 5. We can draw multiple conclusions from this comparison. Firstly, no framework captures the entire set of metrics proposed in Section 3. We also observe, that the group of node-centric networking metrics is mentioned the most in the context of the reviewed frameworks. The reason for this is that most of the frameworks focus on the networking aspects and do not define the "health status" of the low-power wireless network by monitoring all groups of metrics in their entirety. Additionally, we observe that passive monitoring approaches exclusively deliver information on networking statistics, since the sniffed messages do not reveal any insights on the device HW and SW metrics, respectively. Among the passive monitoring frameworks, only the PDA strategy [125] allows to draw conclusions on the mote resynchronization time intervals, as well as on the device uptime. Furthermore, Table 5 shows that the group of active monitoring approaches aims to deliver a more verbose set of metrics, whereas AM-PM is driven by keeping the impact on the network as low as possible. Finally, we also want to remark that the group of application-related SW metrics and RTOS runtime statistics is quite rarely taken into consideration by the surveyed literature on APM frameworks. The explanation for this fact is that developers of APM solutions may see the constant monitoring of these metrics as superfluous and rather put their focus on a small set of performance metrics, which results in a smaller overhead in terms of traffic, resources and energy consumption. However, in Section 3.2 and Section 3.3 we have already provided compelling reasons why monitoring these groups of metrics makes sense, thereby contradicting the aforementioned assertion of being "superfluous".

4.3. Collecting Metrics

In this section, we focus on concepts dealing with the collection of metrics on the device. Traces and logs give an in-depth timeline view of what happened on a single device and allow to spot issues as long as you know what you are looking for. The amount of data collected by tracing and logging tools is immense and most of the data does not get used or processed. Additionally, the data in its raw form, which is usually a string, is hard to compress or aggregate [35]. Therefore, we focus on the collection of metrics instead of traces or logs in this work. The values of the metrics are usually stored in the APM engine, a SW block which runs in the OS on the device. Conceptually, there are two approaches for collecting the values for the APM engine: on-demand and continuously. On-demand means that the value of each metric is gathered at the moment when it is needed, e.g., when the timer indicating the end of a heartbeat interval expires, if we stick to the concept of Memfault. Collecting data continuously describes a concept where we call a function to update the metric in the APM engine each time a code block which modifies the metric has been passed. The answer to the question on which approach for collecting to choose is as too often: It depends. If a certain metric tends to change at an extremely high frequency, it is advantageous to query the value at the end of each heartbeat interval, i.e., gather the metric value on demand.

4.4. Exporting Metrics

By "exporting", we mean the process of serializing and sending the metrics at the end of the heartbeat interval. Serialization defines the process of converting an object into a byte-stream for saving it in a file or database or for sending it via a communication module. Common text formats for serialization are XML [113], JSON [114] and YAML [145]. XML is probably the most wide-spread format and offers much more than the capability to serialize objects, since it is a mark-up language and thus also comes with a significant overhead. JSON and YAML are very easy to read compared to XML, whereas YAML is even more minimalistic than JSON, since YAML does not require the use of brackets, rather line indentation. In general, JSON is preferred for data exchange, whereas YAML is more popular in configuration files. To transmit the payload as part of an User Datagram Protocol (UDP) packet in an IEEE 802.15.4 frame, the data must not be in text format, but binary. By default, the aforementioned serialization formats can be converted into a binary representation by using e.g., UTF-8 encoding. However,

Table 5
Comparison of literature on Application Performance Monitoring frameworks and the metrics used in these frameworks.

Metric Group	APM approach	Active Monitoring						Passive Monitoring				Hybrid Appr.			Piggyback Method				AM-PM	INT		
	Framework / Metric	Sympathy [106]	Memento [107]	Raposo et al. [111]	SmartMesh IP [34, 36, 94]	Tracealyzer [121, 122]	Memfault [57, 35]	SNIF [123]	Ringwald et al. [124]	PDA [125]	Pimoto [126]	HMP [128]	Keller et al. [129]	Sternum IoT [58]	Dressler et al. [102]	Lahmadi et al. [131]	Fanucchi et al. [103]	Gaillard et al. [132]	Karaagac et al. [137]	Tan et al. [143]	Karaagac et al. [119]	Van Leemput et al. [144]
Device HW Metrics	Battery Voltage		✓	✓	✓		✓							✓			✓					
	Charge Consumption				✓																	
	Ambient Temperature				✓																	
	Resynchronization Time		✓							✓		✓										
	RDC						✓					✓		✓								
	RSSI			✓	✓		✓							✓						✓	✓	
	Idle RSSI				✓																	
	LQI			✓	✓		✓							✓								
Device Application Metrics	Msg. Queue Occupancy				✓									✓						✓	✓	
	Avg. Msg. Queue Occupancy				✓																	
	Msg. Queue Congestion		✓		✓		✓						✓	✓								
	Computation Time						✓															
	Bytes sent/received						✓							✓								
	Connectivity Counter						✓							✓								
	Peripheral Interaction Timer						✓							✓								
	Display/LED Timer						✓							✓								
	Reboot Cause + Counter						✓							✓								
	Device Uptime	✓					✓			✓				✓								
RTOS Metrics	Task Stack Size					✓	✓							✓								
	Task Utilization					✓	✓							✓								
	CPU Utilization					✓	✓							✓								
	Memory Usage					✓	✓							✓								
	Kernel Objects					✓	✓															
Networking Metrics (node-centric)	Neighbor (Nbr.) ID List	✓	✓	✓	✓		✓								✓		✓	✓		✓	✓	✓
	Nbr. Transmitted Packets	✓		✓	✓		✓	✓	✓		✓				✓		✓	✓	✓			✓
	Nbr. Transmit Failures			✓	✓		✓								✓		✓	✓	✓			✓
	Nbr. Retransmissions							✓	✓		✓						✓					✓
	Nbr. Received Packets	✓		✓	✓		✓	✓	✓		✓				✓			✓	✓			✓
	Nbr. PDR	✓			✓		✓								✓		✓	✓				✓
	Hop Depth	✓			✓			✓	✓		✓				✓							✓
	Channel Stability			✓	✓			✓	✓		✓							✓		✓	✓	✓
	Avg. Time between Tx						✓															
	Assigned Bandwidth				✓																	
	Needed Bandwidth				✓																	
	Data Rate						✓	✓	✓		✓			✓								
	Next Hop Switch Rate		✓																			
	Number of Joining Events				✓		✓	✓	✓	✓	✓			✓								
	Transmitted Packets	✓		✓	✓		✓	✓	✓		✓			✓		✓	✓	✓				✓
	Expected Transmission Count															✓	✓					✓
	Transmit Failures			✓	✓		✓							✓				✓				✓
	Pkt. Dropped MAC Layer			✓	✓																	✓
	Retransmissions							✓	✓		✓											✓
	Mote PDR			✓	✓		✓							✓				✓				✓
	Received Packets	✓			✓			✓	✓		✓			✓								
	Transmit Ready Packets				✓																	
	Transmit Errors				✓																	
	Mote Availability				✓																	
	Pkts. rec. Mote → Manager	✓			✓		✓	✓	✓		✓			✓				✓	✓			✓
	Pkts. lost Mote → Manager	✓			✓		✓	✓	✓		✓			✓				✓	✓			✓
	Mote Reliability				✓		✓	✓	✓		✓			✓				✓				✓
	Decryption Errors			✓	✓									✓								
	Authentication Errors			✓	✓									✓								
	CRC Errors	✓		✓	✓		✓	✓	✓	✓	✓			✓								
Networking Metrics (global)	Network Availability				✓																	
	Network Reliability				✓			✓						✓				✓	✓			✓
	Network Lifetime						✓															
	Network Upstream Latency				✓				✓				✓									
	Network Downstream Latency								✓				✓									
	Network P.t.P Latency								✓				✓						✓	✓	✓	

we focus on ultra low-power wireless systems where every bit counts in terms of energy consumption. Thus, in the following we highlight some binary serialization formats which are more efficient than a binary representation of a text serialization format such as JSON. MessagePack [146] is an efficient binary serialization format which offers encoders for JSON in many different programming languages. Since small integers are encoded into a single byte and typical short strings require only one extra byte in addition to the strings themselves, MessagePack promises to be faster and smaller than JSON. Another binary serialization format is Binary JSON (BSON), which has been developed for the storage of JSON objects in the MongoDB database [147]. BSON can be compared to binary interchange formats, like Protocol Buffers (Protobufs) [148]. BSON is more "schema-less" than Protobuf. This may be beneficial in terms of flexibility but when it comes to space efficiency, Protobuf outperforms BSON, because BSON has an overhead for key names within the serialized data. With Protobuf, key names are encoded with an integer "field number" and like in MessagePack integers are encoded using a variable length scheme so that small integer numbers can be represented in very few bytes [149]. Another fallback solution with a very low resulting payload size is the use of packed C structures. However, there are strong arguments why packed C structures should not be considered for serialization [149]. The main reason is that value encoding is not standardized and is architecture dependent which quickly results in endianness issues and manual encoding/decoding algorithms. To this end we present Concise Binary Object Representation (CBOR) [150], which is a binary data serialization format based on JSON. CBOR was built to be able to represent all JSON data types and the most common data formats in Internet standards without any ambiguity, while maintaining a compact encoder/decoder structure. Furthermore, data must be able to be decoded without a schema description and the serialization must be reasonably compact. CBOR is designed to be implemented on constrained nodes and that its format is extensible.

In Fig. 1 we show the results of a simulation of an active monitoring system, which is based on heartbeats with an example payload containing a small set of metrics [149]. The source code for reproducing the results is publicly available on GitHub [151]. In the simulation, we compute the overall transmitted payload size of the heartbeat packets over time. We compare the resulting accumulated payload bytes for different heartbeat interval lengths and different serialization formats, respectively. Fig. 1 shows that choosing a more compact serialization format allows much more verbose heartbeat information. In general, the advantage in terms of a smaller payload when using Protobuf, packed C structure or CBOR is huge compared to JSON and MessagePack. When comparing the performance of JSON and MessagePack for a heartbeat interval length of 60 min to the compact representations like CBOR with an interval length of 30 min, i.e., double heartbeat frequency, the resulting payload size for CBOR is still smaller. It is true that sending a larger data payload increases power consumption, but it is more efficient to send fewer large payloads than it is to send more small payloads. Since there is a fixed overhead for all

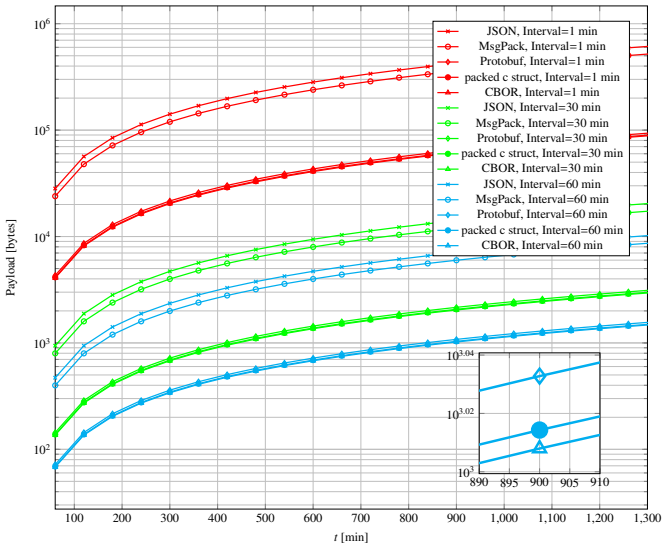


Figure 1. Comparison of serialization formats.

packets in IEEE 802.15.4, sending a big payload does not cost much more than sending a small payload [36]. Nevertheless, the simulation shows that implementing CBOR as serialization format in such constrained systems is crucial, since it allows much verbose monitoring due to the possibility of including a larger set of metric into the heartbeat.

4.5. *Processing Metrics*

In this section, we highlight some state-of-the-art concepts in terms of further processing of the metrics on the network edge. We locate the network manager at the network edge, i.e., the LLN-BR in 6TiSCH networks [23]. The manager needs to implement the application protocol of the network, e.g., MQTT or HTTP when the system relies on the Transmission Control Protocol (TCP) or CoAP when relying on UDP as used in 6TiSCH. Node-Red [152] is a flow-based tool which is commonly used to extract the payload of the incoming notifications at the manager, apply functions to it and forward it to higher-level protocols [153]. The next steps are usually to grab the received data, store it in a database and visualize it on a dashboard. A widespread solution covering all the mentioned steps is the so-called Telegraf, InfluxDB, Grafana (TIG) stack. Telegraf [154] is a server-based agent for collecting and sending all metrics from a large variety of systems and protocols into InfluxDB [155]. Thus, Telegraf offers the possibility to specify parsers in its configuration file in order to convert the metrics to InfluxDB Line Protocol. InfluxDB is an open-source time series database used for storage and retrieval of time series data. The tool is optimized for storing application metrics, IoT sensor data, and real-time analytics. For visualization of time series, Grafana [156] is a charming solution as it allows fast and flexible creation of different kinds of graphs and promises easy interoperability with InfluxDB. It is worth mentioning that all parts of the TIG stack can be run as Docker containers and thus, the TIG solution can simply be deployed in different OSs.

To conclude this section, we present some different approaches for metrics processing presented in literature. Capodiferro et al. [157] address the problem of data visualization in IoT and also come up with the TIG stack as the most flexible solution for metrics processing. A different monitoring solution based on MQTT and *"Zabbix"* [158] for 6TiSCH networks running Contiki-NG as OS has been presented by Gajica et al. [159]. In contrast to the traditional 6TiSCH architecture relying on UDP at the transport layer and CoAP at the application layer, the authors chose TCP and MQTT, respectively. An MQTT broker is installed at the network edge. The 6TiSCH nodes publish their messages to this broker and the Zabbix network monitoring software subscribes to it, stores the readings in a database and offers the possibility for visualization. Lastly, we also want to present a metrics processing solution for SmartMesh IP. For user-friendly operation of SmartMesh IP networks, the developers provide a set of tools in the SmartMesh SDK, such as the *JsonServer.py* application [160]. The JsonServer connects to one or more embedded SmartMesh IP Managers and acts as a command-line tool that turns the SmartMesh IP Manager serial API into an JSON-based HTTP API. Subsequently, Node-Red, MQTT and the TIG stack can be used for further processing.

5. Tutorial: Monitoring a Zephyr Application that uses SmartMesh IP

In this last section, we present a practical hands-on tutorial demonstrating a simple monitoring solution of an TSCH based IEEE 802.15.4 network relying on SmartMesh IP. The source code used in this section is publicly available on GitHub [161].

As part of this tutorial, we build up a low-power wireless network. It consists of a SmartMesh IP network manager and an arbitrary amount of motes. We use the Zephyr RTOS for implementing the application running on the mote. The actual application is just a Real-time Clock (RTC) incrementing a counter variable. Besides that, we install an active monitoring approach, where the motes send a set of performance metrics in certain heartbeat intervals. For this, we use the framework provided by Memfault [57]. Thus, we do not use the fixed set of metrics implemented in SmartMesh IP, the so-called *Health Reports* [5, Section 5.4], but create an own customized group of metrics. The SmartMesh IP protocol stack runs on the LTC5800 chip [162]. We show in this tutorial how SmartMesh IP can be used in combination with another monitoring framework, such as Memfault, by using the SmartMesh IP C-Library. This library is publicly available on GitHub [163] and provides an interface, which establishes a Universal Asynchronous Receiver Transmitter (UART) connection to the LTC5800 chip. In the remainder, we refer to the SmartMesh IP LTC5800 chip as the *networking chip*. The other HW platform's chip is called *application chip* in the following.

The application chip runs the Zephyr RTOS [4] in this tutorial. In Zephyr, we do not need to care about the networking stack but implement exclusively the actual application and the monitoring framework Memfault. The Memfault SDK collects the metrics and periodically puts them into packets. The application chip sends these packets via UART to the networking chip by handing them to the transmit function of the SmartMesh IP C-Library. The networking chip sends the monitoring data to the SmartMesh IP manager, which is connected via USB to an Edge Device. On the Edge Device, we run a Python script which turns the SmartMesh IP Manager serial API into an JSON-based HTTP API. Another Python script on the Edge Device posts these HTTP messages to the Memfault cloud, where the metrics are finally stored in a database and ready for visualization and analysis. A block diagram for all the previously described steps of this tutorial is shown in Fig. 2.

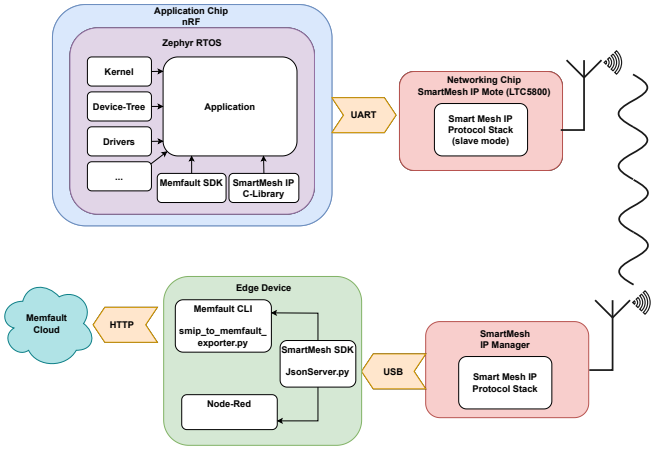


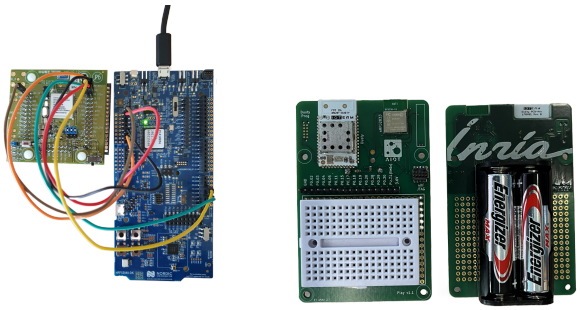
Figure 2. Way of the performance metrics from a Zephyr application to the Memfault cloud using SmartMesh IP.

Section 5.1 explains the HW setup in detail. Section 5.2 focuses on the port of the SmartMesh IP C-Library in order to be used in combination with Zephyr. Section 5.3 deals with the questions on how to integrate Memfault into the Zephyr application and how to transfer the heartbeat packets, the so-called *"chunks"* up to the Memfault cloud. Section 5.4 concludes this tutorial with a performance analysis of the presented solution in this tutorial.

5.1. Hardware Setup

As described in the beginning of this section, we use a HW setup which comprises a SmartMesh IP manager connected to a computer with Internet connection, and a number of motes forming a wireless network based on a mesh topology. The motes consist of a *networking chip* (LTC5800) [162] and an *application chip*. Although the choice of the application chip is up to the user, we show two different setups based on Nordic's nRF chips. While the used nRFs have a basic radio functionality, we do not use it in this tutorial and count on the networking capabilities of the LTC5800. Using the Nordic Connect SDK (NCS) [164] offers easy programmability of the application chip via the `nrfjprog` Command Line Interface (CLI) and natively supports the use of Zephyr. The two setups shown in Fig. 3 are almost technically identical, but they vary in terms of programming/debugging process, battery voltage supply and size.

On the left hand side in Fig. 3a, a picture of an nRF52840-DK [51] and a SmartMesh IP mote (DC9003A-B) is shown.



(a) nRF52840DK + SmartMesh IP mote (b) AIOT Play

Figure 3. The used hardware setups.

They are connected via jumper wires to establish the UART connection and provide a power supply for the LTC5800, if desired. In general, both boards run on coin cell batteries. Table 6 shows a list of pins on both boards to which the jumper wires need to be connected.

Table 6
Wiring between the Application Chip and Networking Chip for Setup A.

Application Chip (nRF52840DK)	Networking Chip (LTC5800)
GND	GND
VCC	VSUPPLY
UART1 Tx Pin	RX
UART1 Rx Pin	TX
GND	TX CTSn
VCC	RX RTSn

The advantage of this solution is that the nRF52840-DK comes with an on-board SEGGER J-Link debug chip, which allows to program it simply via USB. The SmartMesh IP mote needs to be set into *"tail"* mode. Switching between head and tail mode is possible by connecting the SmartMesh IP mote to the Eterna Interface Card (DC9006A). This operation only needs to be done once.

Obviously, this hardware setup is large, fragile and cumbersome. The Inria-AIO team has therefore designed the *"AIOT Play"* board [165], shown in Fig. 3b. The two core elements are still the LTC5800 as networking chip and an nRF as application chip. The major difference to the first setup is that both chips and their connections are already soldered on a common Printed Circuit Board (PCB). In contrast to the nRF52840DK from the first setup, the AIOT comes with a BC833M module containing an nRF52833, which is also based on a 64 MHz ARM Cortex-M4 micro-controller. Another difference between the setups is the fact that the AIOT relies on 2×AA batteries as its voltage supply. Additionally, an nRF JTAG connector is part of the board and allows to program the BC833M module via an external J-Link debugger or even via another nRF Development Kit (DK) with an on-board debug chip. The AIOT Play is designed to easily set up and deploy custom applications by making use of the prototyping area, which contains a breadboard, allowing you to build circuits without needing to solder. On top of that, the AIOT Play is also well suited for teaching purposes [166], motivated by concepts for academic courses on 6TiSCH [167]. The source code used on the AIOT Play is published on GitHub [168].

5.2. *SmartMesh IP C-Library*

As mentioned in the beginning of this section, the SmartMesh IP C-Library is an essential part of the SW running on the application chip. It handles the communication via between the application chip and the networking module via UART. The C-Library is open-source and available on GitHub [163]. It can simply be included in the project by dropping the directory `sm_clib` into your application and add it to the build configuration, i.e., to `CMakeLists.txt` in case of Zephyr, which relies on CMake.

Depending on the HW platform of the application chip, a port of the library needs to be done. At most, three files, namely, `dn_uart.h`, `dn_lock.h` and `dn_endianness.h`, need to be modified. The functions in `dn_uart.h` allow the SmartMesh IP C Library to send bytes over the serial port and receive bytes from the serial port. A *"flush"* function is provided in case the UART driver of the platform is buffer-oriented rather than byte-oriented, e.g., if the serial port is driven through a Direct Memory Access (DMA) module. The SmartMesh C Library doesn't handle flow control. However, the networking chip does not need incoming flow control (`TX CTSn`, `RX RTSn`) when the application chip is sending. If the application chip cannot wake up on data, monitoring the UART flow control pins is required. In case of the used nRF in the presented HW setup this is not the case and we can connect the flow control lines to `GND` and `VCC`, respectively. The `dn_lock.h` file contains functions allowing the library to operate in a multi-threaded environment by defining Mutexs. Lastly, `dn_endianness.h` provides functions for byte swapping operations. Although the LTC5800 on the networking chip is a little-endian processor, all communication in the network is carried out in big-endian order according to the TCP/UDP convention. If the used application chip is based on a little-endian processor, the byte-swap functions found in `dn_endianness.c` need to be used [163].

In this tutorial, we use the Zephyr RTOS which builds up on the device tree concept which makes the port to different HW platforms very easy, because it manages the configuration of the Hardware Abstraction Layer (HAL) and driver functions. In the device tree overlay file, we just need to activate the UART module which we aim to use. When using the AIOT, `P0.09` for Tx and `P0.10` for Rx of the application chip are connected to the network chip. Therefore, these pins need to be assigned to a UART device object in Zephyr's device tree and afterwards just set to active. Thus, in `dn_uart.c` we only need to fill the `dn_uart_init()` and `dn_uart_txByte()` with the corresponding UART functions provided by Zephyr. Furthermore an ISR for the reception of bytes via the UART interface needs to be defined.

To make the mote join the network spanned up by the SmartMesh IP manager, we just need to kick off the FSM of the library. Therefore, we may take over the concept of AIOT sample applications [168] where a file called `ntw.c` was created to handle the interaction with the FSM. In Zephyr's `main()` function we just need to call `ntw_init()` from `ntw.c` to initialize a timer and schedule the first event in the FSM. At first, the UART

interface is initialized and a check if the SmartMesh IP mote has booted is performed. The dn_ipmt_init() function then issues a getParameter<moteStatus> command. If the network chip is idle, i.e., the mote has just booted, the FSM keeps on asking for the status until the mote is in operational mode. Then the command getServiceInfo is sent. The library sets a 500 ms serial response timeout for the mote to respond, which is rather conservative, as commands are expected to be answered within 125 ms. Either a reply arrives, canceling the timer and scheduling the next event, or it times out and the FSM returns to the starting state. At the lower levels, each API call results in a command buffer being constructed and passed to the function dn_serial_mt_sendRequest(), which in turn calls a series of High-Level Data Link Control (HDLC) functions, which ultimately call the UART send function in dn_uart.c. After that, the dn_ipmt_join() function transmits a join command to the network chip. As soon as a reply arrives, the FSM moves to the next state, otherwise it times out and the FSM restarts. Note that it can take 10-20 seconds between the join command and the mote becoming operational depending on the conditions. So polling the state every second instead of instant reset may be an option, too. Although this seems less efficient, it does not have a large impact on the energy consumption, since joining is an infrequent activity in general. When the SmartMesh IP mote has reached the operational state, packets can be sent via the dn_ipmt functions, which offer the possibility to open and bind a communication socket and ultimately also send packets via this socket. To obtain networking metrics such as the ones in Table 3, dn_ipmt.c also offers a series of API calls.

The entire joining process is handled under the hood once the FSM got started, so that simply calling the functions from ntw.c inside the main application is sufficient to initiate the joining process.

5.3. Integration of Memfault

In this section, we highlight the necessary steps to integrate Memfault as monitoring framework into our described setup.

5.3.1. Including Memfault into Zephyr Project

Memfault can be included in Zephyr by editing the west.yml file. *"west"* acts as a configuration and build system for Zephyr and thus also manages the integration of different Git repositories into the project. Therefore, the GitHub-URL of the Memfault Firmware (FW) SDK [169] just has to be provided in west.yml. When creating a new project in the Memfault cloud, the generated project key needs to be pasted in the prj.conf file behind the corresponding identifier CONFIG_MEMFAULT_NCS_PROJECT_KEY when using the NCS.

5.3.2. Custom Settings for Metrics Collection

In the next step, we need to configure the process of metrics collection. The NCS offers a small set of metrics out of the box which are collected in a default heartbeat interval length of 1 hour. Obviously, we want to add custom metrics and adjust the interval length. Therefore, a config directory in the root folder can be created. It may contain a memfault_platform_config.h file for setting a heartbeat interval length by using the define MEMFAULT_METRICS_HEARTBEAT_INTERVAL_SECS. Additionally, in a memfault_metrics_heartbeat_ config.def file, we can define custom metrics via the command MEMFAULT_METRICS_KEY_DEFINE(), which takes the metric name and its corresponding type as arguments. After that, we can place the corresponding Memfault heartbeat functions for metric collection, i.e., for counters, timers or gauges, at the appropriate places of the application source code. The metrics intended to be collected at end of the heartbeat interval must be sampled in the memfault_metrics_heartbeat_collect_data function, which is invoked when the heartbeat interval timer expires.

5.3.3. Data Packetizer

The data packetizer is the Memfault module that handles the transformation of the collected metrics into a Memfault chunk, which is then handed to the send function. A function template showing the usage of the data packetizer is available in the Memfault documentation [170]. The function is called send_memfault_data_multi_part. In the beginning the function checks if there is data available. This is the case when metrics are ready to be sent due to the elapsed timer of the heartbeat interval. Thus, the function can be theoretically called at any time since new data is just available when a heartbeat interval is over. Therefore, it is recommended to call the function immediately after the end of an interval. If metrics are available, a data buffer is created. The buffer should have a size which fits into the payload element of the network frame. IEEE 802.15.4 frames have a length of 127 B. However, due to the header and multiple control fields, the size is significantly smaller and depends on the used protocol architecture [24]. In our setup based on SmartMesh IP, a payload of up to 90 B is supported [171]. The buffer is handed to the packetizer, which grabs the available metrics and enriches them with some additional meta data and heartbeat information. Additionally, a header is appended before the payload and a CRC of the payload is computed and appended. The rest of the provided buffer is filled with a certain pattern. The complete structure of a Memfault chunk is shown in Fig. 4. In this tutorial we try to keep the Memfault chunk as minimalistic as possible to find out the smallest size a single chunk can have. It turns out that there are several metrics that Memfault and the NCS report by default. For testing purposes, we disable them by setting CONFIG_MEMFAULT_METRICS_DEFAULT_SET _ENABLE = n and CONFIG_MEMFAULT_NCS_STACK _METRICS = n in Zephyr's proj.conf file. Furthermore, we define a test metric, which is a simple gauge metric incremented by a periodic timer. We end up with a metrics section consisting of just 7 B. The *"Metric Values"* fields in our example just consist of 1 timer metric (4 B), 2 counter metrics (1 B each) and 1 gauge metric (1 B), i.e, in total 4 metrics (7 B). In total, the Memfault chunk in Fig. 4 has a payload size of 43 B, from which 39 B are the actual payload. Besides the metrics, Metadata and Heartbeat Information fills the payload block. These values have a fixed size in general, but we can specify the Device Version Info fields, where we chose "aiot"

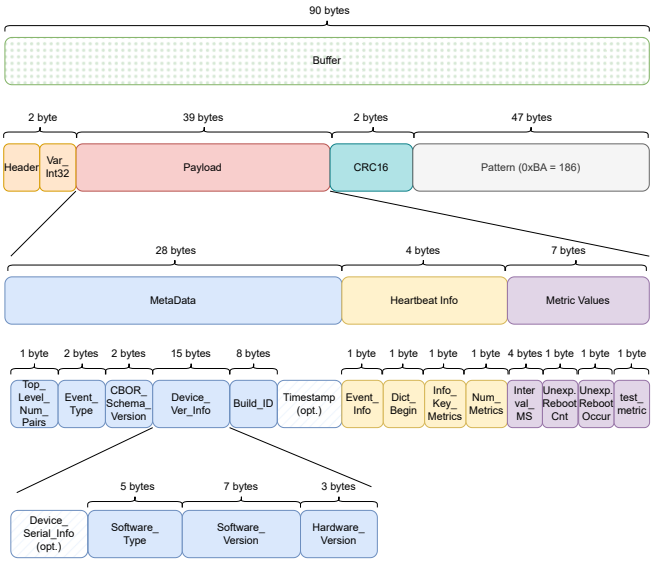


Figure 4. Structure of a minimalistic Memfault chunk.

as `CONFIG_MEMFAULT_NCS_FW_TYPE`. Since this information is transmitted with every chunk, choosing short device version information is desirable in scenarios, where every byte counts.

In the following step, the packetizer serializes the the Memfault chunk using CBOR and finally writes the chunk into the buffer. After that, the buffer and the length of the chunk are passed into send function, i.e, `ntw_transmit` from `ntw.c` calling the send function of the the SmartMesh IP library. In the library's `dn_serial_mt_sendRequest` function, the buffer is copied into the UART buffer up to the chunk length, so that the random pattern is not transmitted. Then the packet arrives via UART at the networking chip, where it is ultimately sent out via the chip's radio into the network.

5.3.4. Network Edge

At the network edge, i.e, the BR in 6TiSCH networks, the network manager is connected to an edge device with an internet connection. In our setup the network edge consists of the SmartMesh IP manager which is connected via USB to a computer. From this point on, several options are possible to ensure a reliable transfer of the Memfault chunks into the Memfault cloud. In this tutorial we decide to run the *JsonServer.py* application from the SmartMesh SDK [160] as explained in Section 4.5. *JsonServer.py* connects to the SmartMesh IP Manager serial API and converts the incoming notifications into JSON-based HTTP messages. As part of this tutorial, we also provide a second Python script, *smip_to_memfault_exporter.py* [161]. This script takes the HTTP messages and pushes the payload via the Memfault CLI [172] to the cloud. The Memfault CLI is a tool, which acts as a client to Memfault's HTTP API and can be installed as Python package via `pip3 install memfault-cli`. The CLI contains a `post-chunk` command, which takes the Memfault chunk in hexadecimal format and the project key as arguments.

5.3.5. Memfault Cloud

The Memfault chunks finally arrive in the Memfault cloud, where they need to be decoded. Therefore, one always needs to upload the compiled FW file, e.g., `zephyr.elf`, which currently runs on the application chip. Based on the FW file, Memfault parses the names of the metrics and combines them with the meta data and values of the received Memfault chunk to write the metrics into a database and visualize them properly. In general, the Memfault cloud also stores older FW images and thus is able to automatically assign older chunks to their corresponding FW file based on a matching `build_id`, which is part of the chunk's metadata.

5.4. Performance Analysis

We conclude this tutorial section with a performance analysis of the proposed monitoring solution in terms of power consumption, security and efficiency. Furthermore, we propose improvement concepts, which may serve as ground for research in future work.

5.4.1. Power Consumption

For estimating the current consumption of the motes in the network, we rely on the SmartMesh IP Power and Performance Estimator [173]. The tool is publicly available and estimates the power consumption of a SmartMesh IP network based on different parameters, such as payload size, reporting interval, number of motes, hop-depths, temperature, etc. On top of that, it allows to draw conclusions on the battery lifetime based on the calculated average current draw [36, Section 24]. In order to use the estimator for our HW setup we need to make some assumptions on the network. First of all we assume that the power consumption of the application chip is comparatively small in contrast to the radio activity at the networking chip. Furthermore, a constant neighbor link PDR/path stability of 80% is presumed. The resulting simulation to estimate the average current draw of each mote is done based on a network consisting of 20 motes in total and a maximum hop-depth of 4. We assume that the motes are split up equally along the hops, i.e., 5 motes on each hop-depth. Further simulation parameters are a temperature of 25 C and a constant payload size of 80 B. In the simulation, the average current draw is calculated based on varying heartbeat intervals. It is not the aim to show the overhead of the monitoring solution on the actual system performance, but to demonstrate the impact on power consumption of increasing reporting interval lengths. Thus, we simply assume that the transmission rate of the actual application increases proportionally with the heartbeat rate and that the application payload is already part of the simulated payload.

The results of the Power and Performance Estimator based on the described setup are shown in Fig. 5.

We observe that the motes consume less when having a larger hop-depth. Obviously, these hops have less children than for instance the 1-hop motes, which need to forward the traffic from the deeper motes in the mesh. Furthermore, the simulation shows a high slope in current draw for the 1-hop, 2-hop and 3-hop motes when shortening the heartbeat interval lengths. In

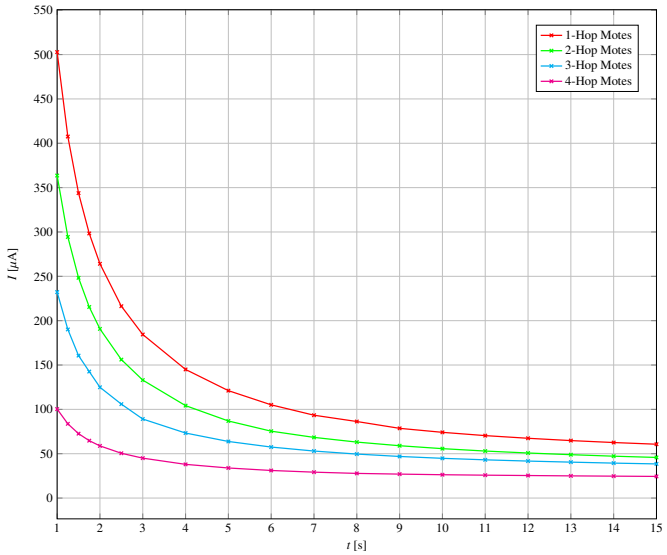


Figure 5. Estimated average current draw of the networking chip for increasing heartbeat interval lengths.

contrast to that, a saturation is visible for almost all hop-depths when the interval is greater than 10s. There is a minimum report rate below which the impact on power consumption is negligible. The reason is that in a SmartMesh IP network, there is some quantity of radio traffic needed to maintain time synchronization across the network [36]. Although, many assumptions have been made, the simulation clearly shows the costs in power consumption related to an increasing heartbeat interval duration.

5.4.2. *Security Considerations*

According to the survey [3] in Section 1, security in IIoT WSNs is the most important feature behind reliability. Thus, we want to evaluate the monitoring solution proposed in this tutorial in terms of security. Memfault provides a framework for collecting and packetizing performance metrics on the device, as well as a cloud platform with a HTTP API. It is the responsibility of the user that the packets make their way from the device to the Memfault cloud reliably and securely.

Security already starts at the mote, i.e., the application chip and the networking chip. Therefore, concepts like the *Arm TrustZone* [82] and secure boot mechanisms are in the focus of current research [174]. However, the broad range of topics related to security on IoT devices is out of scope of this work and we look towards security in the mesh network.

Anyone can theoretically eavesdrop wireless packets transmitted through the air. Indeed, there are so-called packet sniffers that can listen to all 16 channels in the IEEE 802.15.4 spectrum of the 2.4 GHz ISM band at the same time. Thus, TSCH alone is not sufficient to protect data from outside listeners [36]. The goal of security protocols is that attackers, who manage to grab the raw bits of every single packet still cannot decrypt the information. Consequently, in this tutorial the security features implemented in SmartMesh IP [36] are used and thus, the network is secured in terms of:

- Message Integrity: two 32-bit MICs are used at the link layer and network layer, respectively. The goal is to guarantee that the packet is not altered at any hop on its path and to decrypt and authenticate the packet at the destination.
- Access Control: A mote can only join the network by presenting a correct 128-bit join key.
- Confidentiality: The message payload is encrypted with a Counter with Cipher Block Chaining Message Authentication Code (CCM) stream cipher based on 128-bit AES.
- Replay Protection and Denial of Service (DoS) resistance: A monotonically increasing 32-bit nonce counter is used in the encryption process to avoid the sending of duplicates, for instance by a third party.

In general, 6TiSCH networks rely on the OSCORE protocol [99] for providing end-to-end security between two CoAP endpoints. The IETF Lightweight Authenticated Key Exchange (LAKE) WG recently proposes the Ephemeral Diffie-Hellman Over COSE (EDHOC) protocol as a standard which provides a compact handshake implementation and supplies the session keys to OSCORE [175]. Due to EDHOC's efficiency regarding message footprints, it becomes a charming alternative to the established Datagram Transport Layer Security (DTLS) solution for protecting UDP messages [176].

Coming back to our tutorial and looking at the edge device, we observe that the heartbeat messages are vulnerable for man-in-the-middle attacks, as they need to find their way from the SmartMesh IP manager API to the Memfault cloud. Since, the messages may be manipulated or bugged at this point, a security concept is part of future work. CBOR Object Signing and Encryption (COSE) [177] is an IETF standard for CBOR encryption, which is also used in EDHOC. Therefore, COSE may be an obvious solution to secure the Memfault chunks, which are already serialized with CBOR. In this case, the chunks would be decrypted immediately before calling the Memfault CLI, which sends the chunks to the cloud. On the way to the Memfault cloud, the data is again secured by Hypertext Transfer Protocol Secure (HTTPS).

5.4.3. *Efficiency Considerations*

Lastly, we analyze the performance of the monitoring solution presented in this tutorial regarding efficiency. When analyzing the structure of the Memfault chunks from Fig. 4, we have observed that the majority of the fields did not change. In particular, only the metric values and consequently the CRC change with each heartbeat interval. Thus, we obviously carry a significant overhead in each heartbeat message. To post the complete chunk to the Memfault cloud is certainly necessary, since the metadata and heartbeat info fields serve for storing the metric values correctly in the cloud database. However, the question arises if there are methods to reduce this overhead in each heartbeat message and what is the resulting gain in terms of lowering the power consumption.

This problem has been extensively addressed in literature under the topic of data aggregation and data compression. Data aggregation describes the process of reducing the packet size by combining the payload of packets when traversing through the hops in the mesh network. The goal is to remove redundant parts by manipulating some extracted features and statistics of the data sets collected from sensor nodes like the minimum, maximum and/or mean [178]. Data compression is a methodology which has its roots in information theory and finds application in all kinds of communications engineering. A recent literature survey on data compression in constrained networks is given by Nassra et al. [179]. They sorted the presented compression algorithms in two groups, namely *lossy* and *lossless* data compression. However, we just consider lossless data compression algorithms for our use case of reducing the message size of the monitoring information. Massey et al. [180] have presented a lossless packet compression algorithm for WSNs based on a dictionary approach. Their approach relies on the distribution of the same dictionary to all the motes, so that every mote can compress data by searching for data patterns in the payload, which is already stored in the dictionary. If the pattern is included in the dictionary, the mote sends a much smaller key instead of the original data. Since we observe that the vast majority of the fields in the Memfault chunks remains identical, the proposed algorithm [180] delivers a simple and charming solution to this problem.

In Fig. 5 we have seen the increase in current draw of the networking chip, when choosing a smaller heartbeat interval length, i.e., sending more packets. The current draw of motes in wireless systems based on IEEE 802.15.4 is indeed dominated by the number of packets and not so much by the packet size [36, 130]. This means it is more efficient to put more payload in one packet instead of splitting it up on two packets. Nevertheless, there is still an increase in power consumption with growing packet size. To analyze the impact of a larger packet size on the power consumption, we set up another simulation, which still relies on the same network parameters as for the simulation results in Fig. 5. However, we now fix the heartbeat reporting interval length to 20 s and vary the transmitted payload size. The average current draw of the SmartMesh IP LTC5800 networking chip is shown in Fig. 6.

When comparing the difference in terms of average current draw between a payload size of 5 B and a payload size of 90 B, we observe values of 10μA for the 1-hop motes and only 2μA for the 4-hop motes. Obviously, the amount of energy saved by choosing smaller heartbeat interval lengths is an order of magnitude higher than the saving by smaller payload sizes achieved through data aggregation and data compression. Nevertheless, especially in ultra low-power wireless systems, every possibility of saving energy must be exploited in order to extend the battery lifetime of the motes. Consequently, a task for future work is to develop and analyze methods for data aggregation and data compression in the context of APM.

We conclude this tutorial section by discussing the benefits brought by the presented APM framework in contrast to the introduced overhead in terms of power consumption. We have highlighted that choosing a reasonable heartbeat interval length,

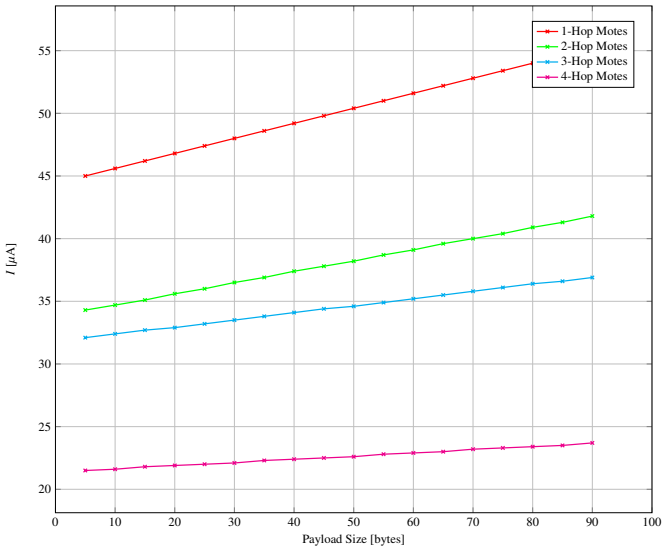


Figure 6. Estimated average current draw of the networking chip for increasing packet sizes.

i.e., not smaller than 10s, as well as an efficient way of compressing the payload containing the metrics and the corresponding meta-information, is the key to tipping the balance in favor of the benefits. However, the remaining space in the message frame can be filled with a larger number of metrics, since Memfault is highly scalable thanks to an efficient strategy avoiding sending the metric identifiers in each packet. The insights gathered by such a broad set of performance metrics clearly help to identify application dependent bottlenecks, aging related problems, networking issues and the performance of the entire network. Moreover, many SW bugs remain undiscovered during the testing process and just become visible once deployed in the field under harsh conditions and after a significant runtime. Implementing an active monitoring framework as in this tutorial, which stores these metric values in a time series database is essential to give the developer the opportunity to detect these bugs and fix them as part of a SW update.

6. Conclusion

This article is both a survey and a tutorial.

In the survey part, we address the challenge of designing APM solutions for low-power wireless systems. We survey performance metrics from a broad set of literature sources. The resulting overview of metrics forms the basis for designing a strong APM framework. For each metric, we have defined a motivation for continuously monitoring it during the operational phase of the system. The goal is to obtain a complete and meaningful picture of the system's health condition. Additionally, we have highlighted performance metrics of an RTOS, which is at the core of each mote in the network. To the best of our knowledge, this work is the first comparing different RTOS implementations in terms of their in-built APM features. Furthermore, we give practical implementation hints for these RTOS metrics. The survey part of this article further

contains a section on state-of-the-art APM approaches in wireless systems. We cluster these approaches in different groups and evaluate them based on verbosity and impact on the system performance. In the remainder of this article, we analyze *active APM* solutions and introduce common concepts for collecting, exporting and processing the metrics. For the problem of exporting the metrics, comparing different serialization strategies confirms that CBOR is the most desirable solution for low-power systems, where a small reduction in terms of payload size directly translates into a crucial power consumption enhancement.

In the tutorial part, we provide a hands-on tutorial showing a Proof of Concept (PoC) that the active monitoring framework Memfault can be used inside a low-power wireless system. SmartMesh IP is used to build up a network based on IEEE 802.15.4. The source code for this tutorial is available on GitHub [161] and comprises the FW running on the application chip of the mote and a Python script running on the Edge Device. Two possible HW setups for the motes are presented. Both HW setups consist of an application chip and a networking chip. In the first setup, the UART connection between the two chips is realized by jumper wires, whereas the second setup, the so-called *AIOT*, has both chips and their connection already soldered on a common PCB. The FW of the application chip also includes a version of the SmartMesh IP C-Library. This library has been ported to Zephyr as part of this work and handles the UART communication between the two chips. We guide the reader through the integration of the APM solution Memfault step-by-step. In the performance analysis of the tutorial, we show that it is advantageous in terms of power consumption to pack more metrics into one frame instead of splitting them up on multiple frames. The Memfault monitoring framework delivers helpful tools for metric collection and exporting on the device and for visualization in the cloud. We emphasize that it is the responsibility of the system architect to ensure that the Memfault heartbeats reliably make their way from the device to the cloud. We propose methods for future work to enhance the presented APM solution in terms of security and compression.

Acknowledgments

This document is issued within the frame and for the purpose of the OpenSwarm project. This project has received funding from the European Union's Horizon Europe Framework Programme under Grant Agreement No. 101093046. Views and opinions expressed are however those of the author(s) only and the European Commission is not responsible for any use that may be made of the information it contains.

References

[1] S. Sinha, State of IoT 2023: Number of connected IoT devices growing 16% to 16.7 billion globally. [Online]. Available: https://iot-analytics.com/number-connected-iot-devices/

[2] M. Usman, S. Ferlin, A. Brunstrom, J. Taheri, A Survey on Observability of Distributed Edge and Container-Based Microservices, IEEE Access 10 (2022) 86904–86919. doi:10.1109/ACCESS.2022.3193102.

[3] M. Hatler, Wireless sensor networks: Expanding opportunities for Industrial IoT, InTech - ISA's Flagship Publications. [Online]. Available: https://www.isa.org/intech-home/2017/september-october/features/expanding-opportunities-for-industrial-iot

[4] The Linux Foundation, Zephyr RTOS - A proven RTOS ecosystem, by developers, for developers. [Online]. Available: https://www.zephyrproject.org/

[5] Analog Devices, Analog Devices SmartMesh IP. [Online]. Available: https://www.analog.com/en/products/rf-microwave/wireless-sensornetworks/smartmesh-ip.html

[6] IEEE Computer Society, IEEE Standard for Information technology–Local and metropolitan area networks– Specific requirements– Part 15.4: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low Rate Wireless Personal Area Networks (WPANs), IEEE Std 802.15.4-2006 (Revision of IEEE Std 802.15.4-2003) (2006) 1–320doi:10.1109/IEEESTD.2006.232110.

[7] N. Ahmed, H. Rahman, M. I. Hussain, A comparison of 802.11ah and 802.15.4 for IoT, ICT Express 2 (3) (2016) 100–102, special Issue on ICT Convergence in the Internet of Things (IoT). doi:https://doi.org/10.1016/j.icte.2016.07.003.

[8] P. R. Narendra, S. Duquennoy, T. Voigt, BLE and IEEE 802.15.4 in the IoT: Evaluation and Interoperability Considerations, in: Internet of Things. IoT Infrastructures: Second International Summit, IoT 360° 2015, Rome, Italy, Vol. 170, 2016, pp. 427–438. doi:10.1007/978-3-319-47075-7_47.

[9] K. Mikhaylov, N. Plevritakis, J. Tervonen, Performance Analysis and Comparison of Bluetooth Low Energy with IEEE 802.15.4 and SimpliciTI, Journal of Sensor and Actuator Networks 2 (3) (2013) 589–613. doi:10.3390/jsan2030589.

[10] IEEE Computer Society, IEEE Standard for Information technology–Local and metropolitan area networks– Specific requirements- Part 15.4: Wireless Medium Access Control (MAC) and Physical Layer (PHY)-Amendment 1: MAC Sublayer, IEEE Std 802.15.4e-2012 (Amendment to IEEE Standard 802.15.4-2011) (2012) 1–225.

[11] D. De Guglielmo, S. Brienza, G. Anastasi, IEEE 802.15.4e: A survey, Computer Communications 88 (2016) 1–24. doi:https://doi.org/10.1016/j.comcom.2016.05.004.

[12] IEEE Computer Society, IEEE Standard for Information technology–Local and metropolitan area networks– Specific requirements– Part 15.4: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low Rate Wireless Personal Area Networks (WPANs), IEEE Std 802.15.4-2015 (Revision of IEEE Std 802.15.4-2011) (2016) 1–709.

[13] J. Muñoz, P. Muhlethaler, X. Vilajosana, T. Watteyne, Why Channel Hopping Makes Sense, even with IEEE802.15.4 OFDM at 2.4 GHz, in: 2018 Global Internet of Things Summit (GIoTS), 2018, pp. 1–7. doi:10.1109/GIOTS.2018.8534544.

[14] T. Watteyne, J. Weiss, L. Doherty, J. Simon, Industrial IEEE802.15.4e Networks: Performance and Trade-offs, in: 2015 IEEE International Conference on Communications (ICC), 2015, pp. 604–609. doi:10.1109/ICC.2015.7248388.

[15] ZigBee Alliance, The ZigBee Specification (2015). [Online]. Available: https://zigbeealliance.org/wp-content/uploads/2019/11/docs-05-3474-21-0csg-zigbee-specification.pdf

[16] Thread Group, Thread Network Fundamentals (2020). [Online]. Available: https://www.threadgroup.org/Portals/0/documents/support/Thread%20Network%20Fundamentals_v3.pdf

[17] Connectivity Standards Alliance, Matter Specification Version 1.0 (2022). [Online]. Available: https://csa-iot.org/wp-content/uploads/2022/11/22-27349-001_Matter-1.0-Core-Specification.pdf

[18] HART Communication Foundation, HART Field Communication Protocol Specification, Revision 7.0, HART Communication Foundation.

[19] International Society of Automation (ISA), 100.11 a-2009: Wireless systems for industrial automation: Process control and related applications, International Society of Automation: Research Triangle Park, NC, USA.

[20] G. Montenegro, J. Hui, D. Culler, N. Kushalnagar, Transmission of IPv6 Packets Over IEEE 802.15.4 Networks, RFC 4944, RFC Editor (09 2007).

[21] R. Alexander, A. Brandt, J. Vasseur, J. Hui, K. Pister, P. Thubert, P. Levis, R. Struik, R. Kelsey, T. Winter, RPL: IPv6 Routing Protocol for Low-Power and Lossy Networks, RFC 6550, RFC Editor (03 2012).

[22] Z. Shelby, K. Hartke, C. Bormann, The Constrained Application Protocol (CoAP), RFC 7252, RFC Editor (06 2014).

[23] P. Thubert, An Architecture for IPv6 over the Time-Slotted Channel Hopping Mode of IEEE 802.15.4 (6TiSCH), RFC 9030, RFC Editor (05 2021).

[24] X. Vilajosana, T. Watteyne, T. Chang, M. Vučinić, S. Duquennoy, P. Thubert, IETF 6TiSCH: A Tutorial, IEEE Communications Surveys and Tutorials 22 (1) (2020) 595–615. doi:10.1109/COMST.2019.2939407.

[25] T. Watteyne, X. Vilajosana, B. Kerkez, F. Chraim, K. Weekly, Q. Wang, S. Glaser, K. Pister, OpenWSN: A Standards-Based Low-Power Wireless Development Environment, Wiley Transactions on Emerging Telecommunications Technologies 23 (2012) 480–493. doi:10.1002/ett.2558.

[26] S. Duquennoy, A. Elsts, B. A. Nahas, G. Oikonomo, TSCH and 6TiSCH for Contiki: Challenges, Design and Evaluation, in: 2017 13th International Conference on Distributed Computing in Sensor Systems (DCOSS), 2017, pp. 11–18. doi:10.1109/DCOSS.2017.29.

[27] E. Baccelli, O. Hahm, M. Günes, M. Wählisch, T. C. Schmidt, RIOT OS: Towards an OS for the Internet of Things, in: 2013 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS), 2013, pp. 79–80. doi:10.1109/INFCOMW.2013.6970748.

[28] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, D. Culler, TinyOS: An Operating System for Sensor Networks, Vol. 00, Springer, 2005, pp. 115–148. doi:10.1007/3-540-27139-2_7.

[29] T. Watteyne, L. Doherty, J. Simon, K. Pister, Technical Overview of SmartMesh IP, in: 2013 Seventh International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing, 2013, pp. 547–551. doi:10.1109/IMIS.2013.97.

[30] X. Vilajosana, T. Watteyne, M. Vučinić, T. Chang, K. S. J. Pister, 6TiSCH: Industrial Performance for IPv6 Internet-of-Things Networks, Proceedings of the IEEE 107 (6) (2019) 1153–1165. doi:10.1109/JPROC.2019.2906404.

[31] L. Doherty, W. Lindsay, J. Simon, Channel-Specific Wireless Sensor Network Path Data, in: 2007 16th International Conference on Computer Communications and Networks, 2007, pp. 89–94. doi:10.1109/ICCCN.2007.4317802.

[32] D. Yuan, S. S. Kanhere, M. Hollick, Instrumenting Wireless Sensor Networks – A Survey on the Metrics that Matter, Pervasive and Mobile Computing 37 (2017) 45–62. doi:https://doi.org/10.1016/j.pmcj.2016.10.001.

[33] F. Ojeda, D. Mendez, A. Fajardo, F. Ellinger, On Wireless Sensor Network Models: A Cross-Layer Systematic Review, Journal of Sensor and Actuator Networks 12 (4) (2023) 50.

[34] Linear Technology, SmartMesh IP Embedded Manager API Guide. [Online]. Available: https://www.analog.com/media/en/reference-design-documentation/design-notes/smartmesh_ip_embedded_manager_api_guide.pdf

[35] T. Hoffman, Monitoring Fleet Health with Heartbeat Metrics (September 2020). [Online]. Available: https://interrupt.memfault.com/blog/device-heartbeat-metrics

[36] Linear Technology, SmartMesh IP Application Notes. [Online]. Available: https://www.analog.com/media/en/technical-documentation/application-notes/smartmesh_ip_application_notes.pdf

[37] X. Vilajosana, Q. Wang, F. Chraim, T. Watteyne, T. Chang, K. S. J. Pister, A Realistic Energy Consumption Model for TSCH Networks, IEEE Sensors Journal 14 (2) (2014) 482–489. doi:10.1109/JSEN.2013.2285411.

[38] Analog Devices, How to achieve greater accuracy in battery capacity readings for portable designs. [Online]. Available: https://www.analog.com/en/technical-articles/how-to-achieve-greater-accuracy-in-battery-capacity-readings-for-portable-designs.html

[39] Texas Instruments, MAX17048 - 3μA 1-Cell/2-Cell Fuel Gauge with ModelGauge. [Online]. Available: https://www.analog.com/en/products/max17048.html#product-overview

[40] S. Naderiparizi, A. N. Parks, F. S. Parizi, J. R. Smith, μMonitor: In-situ energy monitoring with microwatt power consumption, in: 2016 IEEE International Conference on RFID (RFID), 2016, pp. 1–8. doi:10.1109/RFID.2016.7488017.

[41] C. Guo, S. Ci, Y. Zhou, Y. Yang, A Survey of Energy Consumption Measurement in Embedded Systems, IEEE Access PP. doi:10.1109/ACCESS.2021.3074070.

[42] Espressif, ESP32-S2 - ESP-IDF Programming Guide. [Online]. Available: https://docs.espressif.com/projects/esp-idf/en/latest/esp32s2/api-reference/peripherals/temp_sensor.html#:~:text=The%20ESP32%2DS2%20has%20a,ranges%20with%20specific%20measurement%20errors.

[43] Gay, Warren, DHT11 sensor, Advanced Raspberry Pi: Raspbian Linux and GPIO Integration (2018) 399–418.

[44] T. Chang, T. Watteyne, K. Pister, Q. Wang, Adaptive synchronization in multi-hop TSCH networks, Computer Networks 76 (2015) 165–176. doi:https://doi.org/10.1016/j.comnet.2014.11.003.

[45] Brzozowski, Marcin and Salomon, Hendrik and Langendoerfer, Peter, On efficient clock drift prediction means and their applicability to IEEE 802.15. 4, in: 2010 IEEE/IFIP International Conference on Embedded and Ubiquitous Computing, IEEE, 2010, pp. 216–223.

[46] T. Claeys, F. Rousseau, B. Tourancheau, A. Duda, Clock Drift Prediction for Fast Rejoin in 802.15.4e TSCH Networks, in: 2017 26th International Conference on Computer Communication and Networks (ICCCN), 2017, pp. 1–9. doi:10.1109/ICCCN.2017.8038401.

[47] F. Qin, X. Dai, J. E. Mitchell, Effective-SNR estimation for wireless sensor network using Kalman filter, Ad Hoc Networks 11 (3) (2013) 944–958. doi:https://doi.org/10.1016/j.adhoc.2012.11.002.

[48] T. Savić, X. Vilajosana, T. Watteyne, Constrained Localization: A Survey, IEEE Access 10 (2022) 49297–49321. doi:10.1109/ACCESS.2022.3171859.

[49] I. Dotlic, A. Connell, H. Ma, J. Clancy, M. McLaughlin, Angle of arrival estimation using decawave DW1000 integrated circuits, 2017 14th Workshop on Positioning, Navigation and Communications (WPNC) (2017) 1–6doi:10.1109/WPNC.2017.8250079.

[50] Texas Instruments, CC2420 - 2.4 GHz IEEE 802.15.4 / ZigBee-ready RF Transceiver - Datasheet revision C. [Online]. Available: https://www.ti.com/product/de-de/CC2420

[51] Nordic Semiconductor, nRF52840DK - Product Specification v1.1. [Online]. Available: https://infocenter.nordicsemi.com/pdf/nRF52840_PS_v1.1.pdf

[52] J. Ansari, T. Ang, P. Mähönen, Wispot: Fast and reliable detection of Wi-Fi networks using IEEE 802.15.4 radios, in: Proceedings of the 9th ACM International Symposium on Mobility Management and Wireless Access, MobiWac '11, 2011, pp. 35—-44. doi:10.1145/2069131.2069138.

[53] A. Cortés-Leal, C. Del-Valle-Soto, C. Cardenas, L. J. Valdivia, J. A. Del Puerto-Flores, Performance Metric Analysis for a Jamming Detection Mechanism under Collaborative and Cooperative Schemes in Industrial Wireless Sensor Networks, Sensors 22 (1). doi:10.3390/s22010178. [Online]. Available: https://www.mdpi.com/1424-8220/22/1/178

[54] D. Ljepojević, G. Gardašević, An Approach to Link Quality Measurement in 6TiSCH Networks, in: 2023 30th International Conference on Systems, Signals and Image Processing (IWSSIP), 2023, pp. 1–5. doi:10.1109/IWSSIP58668.2023.10180236.

[55] J. Beningo, Embedded Software Design: A Practical Approach to Architecture, Processes, and Coding Techniques, Apress, 2022. [Online]. Available: https://books.google.de/books?id=cIoPzwEACAAJ

[56] A. Niccolai, F. Grimaccia, M. Mussetta, R. Zich, Optimal task allocation in wireless sensor networks by means of social network optimization, Mathematics 7 (4) (2019) 1–15.

[57] Memfault Inc., Memfault - Device Reliability Platform for IoT Monitoring, Debugging and OTA Updates. [Online]. Available: https://memfault.com/

[58] Sternum, Sternum IoT - Embedded Security and Observability Platform. [Online]. Available: https://sternumiot.com/

[59] A. Serino, L. Cheng, Real-Time Operating Systems for Cyber-Physical Systems: Current Status and Future Research, in: 2020 International Conferences on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData) and

IEEE Congress on Cybermatics (Cybermatics), IEEE, 2020, pp. 419–425.
[60] A. Musaddiq, Y. B. Zikria, O. Hahm, H. Yu, A. K. Bashir, S. W. Kim, A Survey on Resource Management in IoT Operating Systems, IEEE Access 6 (2018) 8459–8482. doi:10.1109/ACCESS.2018.2808324.
[61] M. D. Marieska, P. G. Hariyanto, M. F. Fauzan, A. I. Kistijantoro, A. Manaf, On performance of kernel based and embedded Real-Time Operating System: Benchmarking and Analysis, in: 2011 International Conference on Advanced Computer Science and Information Systems, 2011, pp. 401–406.
[62] K. Baynes, C. Collins, E. Fiterman, B. Ganesh, P. Kohout, C. Smit, T. Zhang, B. Jacob, The performance and energy consumption of embedded real-time operating systems, IEEE Transactions on Computers 52 (11) (2003) 1454–1469. doi:10.1109/TC.2003.1244943.
[63] J. J. Labrosse, MC/OS-III: The Real-time Kernel, Micrium Press, 2009. [Online]. Available: https://books.google.de/books?id=c_kBRQAACAAJ
[64] A. Sehgal, V. Perelman, S. Kuryla, J. Schonwalder, Management of resource constrained devices in the internet of things, IEEE Communications Magazine 50 (12) (2012) 144–149. doi:10.1109/MCOM.2012.6384464.
[65] X. Tan, I. Hakala, StateOS: A Memory-Efficient Hybrid Operating System for IoT Devices, IEEE Internet of Things Journal 10 (11) (2023) 9523–9533. doi:10.1109/JIOT.2023.3234106.
[66] N. Pendleton, Measuring Stack Usage the Hard Way (June 2023). [Online]. Available: https://interrupt.memfault.com/blog/measuring-stack-usage
[67] Embedded Magazine, How To Calculate CPU Utilization (July 2004). [Online]. Available: https://www.embedded.com/how-to-calculate-cpu-utilization/
[68] T. Hoffman, Understanding Battery Performance of IoT Devices (July 2023). [Online]. Available: https://interrupt.memfault.com/blog/monitoring-battery-life
[69] R. I. Davis, L. Cucu-Grosjean, M. Bertogna, A. Burns, A review of priority assignment in real-time systems, Journal of Systems Architecture 65 (2016) 64–82. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1383762116300200
[70] C. L. Liu, J. W. Layland, Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment, J. ACM 20 (1) (1973) 46–61. doi:10.1145/321738.321743.
[71] M. Barr, Three Things Every Programmer Should Know About RMA (August 2010). [Online]. Available: https://www.embedded.com/three-things-every-programmer-should-know-about-rma/
[72] L. M. Grupp, A. M. Caulfield, J. Coburn, S. Swanson, E. Yaakobi, P. H. Siegel, J. K. Wolf, Characterizing flash memory: Anomalies, observations, and applications, in: 2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 2009, pp. 24–33. doi:10.1145/1669112.1669118.
[73] A. Prodromakis, S. Korkotsides, T. Antonakopoulos, MLC NAND Flash memory: Aging effect and chip/channel emulation, Microprocessors and Microsystems 39 (8) (2015) 1052–1062. doi:https://doi.org/10.1016/j.micpro.2015.06.007.
[74] S. Sakib, A. Milenković, M. T. Rahman, B. Ray, An Aging-Resistant NAND Flash Memory Physical Unclonable Function, IEEE Transactions on Electron Devices 67 (3) (2020) 937–943. doi:10.1109/TED.2020.2968272.
[75] Amazon Web Services Inc., FreeRTOS - Real-time operating system for microcontrollers. [Online]. Available: https://www.freertos.org/index.html
[76] G. Oikonomou, S. Duquennoy, A. Elsts, J. Eriksson, Y. Tanaka, N. Tsiftes, The Contiki-NG open source operating system for next generation IoT devices, SoftwareX 18 (2022) 101089. doi:https://doi.org/10.1016/j.softx.2022.101089. [Online]. Available: https://github.com/contiki-ng/contiki-ng/wiki
[77] The RIOT Open Community, RIOT OS - The friendly operating system for the IoT. [Online]. Available: https://www.riot-os.org/
[78] ARM Limited, Mbed OS - The RTOS for Arm Cortex M Devices. [Online]. Available: https://os.mbed.com/
[79] The Eclipse Foundation, IoT and Edge Developer Survey. [Online]. Available: https://outreach.eclipse.foundation/iot-edge-developer-survey-2022
[80] The Contiki Community, COOJA - the network simulator for Contiki. [Online]. Available: https://github.com/contiki-os/contiki/wiki/An-Introduction-to-Cooja
[81] T. Claeys, F.-X. Molina, M. Vučinić, T. Watteyne, E. Baccelli, RIOT and OpenWSN 6TiSCH: Happy Together, in: 2020 9th IFIP International Conference on Performance Evaluation and Modeling in Wireless Networks (PEMWN), 2020, pp. 1–6. doi:10.23919/PEMWN50727.2020.9293070.
[82] ARM Limited, TrustZone for Cortex-M. [Online]. Available: https://www.arm.com/technologies/trustzone-for-cortex-m
[83] OpenRTOS.net, FreeRTOS - uxTaskGetStackHighWaterMark. [Online]. Available: http://www.openrtos.net/uxTaskGetStackHighWaterMark.html
[84] I. N. R. Hendrawan, I. G. N. W. Arsa, Zolertia Z1 energy usage simulation with Cooja simulator, in: 2017 1st International Conference on Informatics and Computational Sciences (ICICoS), 2017, pp. 147–152. doi:10.1109/ICICOS.2017.8276353.
[85] Espressif Systems (Shanghai) Co., ESP32-IDF Programming Guide. [Online]. Available: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/get-started/
[86] W. Du, D. Navarro, F. Mieyeville, Performance Evaluation of IEEE 802.15.4 Sensor Networks in Industrial Applications, International Journal of Communication Systems 28 (10) (2015) 1657–1674.
[87] D. Stanislowski, X. Vilajosana, Q. Wang, T. Watteyne, K. S. J. Pister, Adaptive Synchronization in IEEE802.15.4e Networks, IEEE Transactions on Industrial Informatics 10 (2014) 795–802. [Online]. Available: https://api.semanticscholar.org/CorpusID:10601926
[88] M. Vucinic, T. Chang, B. Škrbic, E. Kocan, M. Pejanovic-Djurišic, T. Watteyne, Key Performance Indicators of the Reference 6TiSCH Implementation in Internet-of-Things Scenarios, IEEE Access 8 (2020) 79147–79157.
[89] V. Pereira, J. S. Silva, E. Monteiro, A framework for Wireless Sensor Networks performance monitoring, in: 2012 IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks (WoWMoM), 2012, pp. 1–7. doi:10.1109/WoWMoM.2012.6263784.
[90] T. O'Donovan, N. Tsiftes, Z. He, T. Voigt, C. J. Sreenan, Detailed Diagnosis of Performance Anomalies in Sensornets, in: Proceedings of the 6th Workshop on Hot Topics in Embedded Networked Sensors, HotEmNets '10, Association for Computing Machinery, New York, NY, USA, 2010, pp. 1–5. doi:10.1145/1978642.1978654.
[91] K. Brun-Laguna, P. Gomes, P. Minet, T. Watteyne, Moving Beyond Testbeds? Lessons (We) Learned About Connectivity, IEEE Pervasive Computing 17 (4) (2018) 15–27. doi:10.1109/MPRV.2018.2873847.
[92] T. Watteyne, C. Adjih, X. Vilajosana, Lessons learned from large-scale dense IEEE802.15.4 connectivity traces, in: 2015 IEEE International Conference on Automation Science and Engineering (CASE), 2015, pp. 145–150. doi:10.1109/CoASE.2015.7294053.
[93] S. Fu, M. Ceriotti, Y. Jiang, C.-Y. Shih, X. Huan, P. J. Marron, An Approach to Detect Anomalous Degradation in Signal Strength of IEEE 802.15.4 Links, in: 2018 15th Annual IEEE International Conference on Sensing, Communication, and Networking (SECON), 2018, pp. 1–9. doi:10.1109/SAHCN.2018.8397126.
[94] Linear Technology, SmartMesh SDK 1.0.5.138 (2015). [Online]. Available: https://github.com/twatteyne/smartmeshsdk3/blob/develop/libs/SmartMeshSDK/protocols/NetworkHealthAnalyzer/NetworkHealthAnalyzer.py
[95] De Couto, Douglas S. J. and Aguayo, Daniel and Bicket, John and Morris, Robert, A high-throughput path metric for multi-hop wireless routing, in: Proceedings of the 9th Annual International Conference on Mobile Computing and Networking, MobiCom '03, Association for Computing Machinery, New York, NY, USA, 2003, pp. 134—-146. doi:10.1145/938985.939000.
[96] K. Srinivasan, M. A. Kazandjieva, S. Agarwal, P. Levis, The β-Factor: Measuring Wireless Link Burstiness, in: Proceedings of the 6th ACM Conference on Embedded Network Sensor Systems, SenSys '08, Association for Computing Machinery, New York, NY, USA, 2008, pp. 29—-42. doi:10.1145/1460412.1460416.
[97] Y. Tanaka, P. Minet, M. Vučinić, X. Vilajosana, T. Watteyne, YSF: A 6TiSCH Scheduling Function Minimizing Latency of Data Gathering in IIoT, IEEE Internet of Things Journal 9 (11) (2022) 8607–8615. doi:

10.1109/JIOT.2021.3118017.

[98] G. Martinovic, J. Balen, D. Zagar, A Cross-Layer Approach and Performance Benchmarking in Wireless Sensor Networks, in: Proceedings of the 2nd WSEAS International Conference on Sensors, and Signals and Visualization, Imaging and Simulation and Materials Science, SENSIG'09/VIS'09/MATERIALS'09, World Scientific and Engineering Academy and Society (WSEAS), Stevens Point, Wisconsin, USA, 2009, pp. 76–81.

[99] G. Selander, J. P. Mattsson, F. Palombini, L. Seitz, Object Security for Constrained RESTful Environments (OSCORE), RFC 8613, RFC Editor (07 2019).

[100] F. Hermans, O. Rensfelt, T. Voigt, E. Ngai, L.-A. Norden, P. Gunningberg, SoNIC: Classifying interference in 802.15.4 sensor networks, in: 2013 ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN), 2013, pp. 55–66. doi:10.1145/2461381.2461392.

[101] D. Liu, J. Raymer, A. Fox, Efficient and timely jamming detection in wireless sensor networks, in: 2012 IEEE 9th International Conference on Mobile Ad-Hoc and Sensor Systems (MASS 2012), 2012, pp. 335–343. doi:10.1109/MASS.2012.6502533.

[102] F. Dressler, D. Neuner, Energy-efficient monitoring of distributed system resources for self-organizing sensor networks, in: 2013 IEEE Topical Conference on Wireless Sensors and Sensor Networks (WiSNet), 2013, pp. 145–147. doi:10.1109/WiSNet.2013.6488662.

[103] D. Fanucchi, R. Knorr, B. Staehle, Impact of network monitoring in IEEE 802.15.4e-based wireless sensor networks, in: 2015 IEEE 16th International Symposium on A World of Wireless, Mobile and Multimedia Networks (WoWMoM), 2015, pp. 1–3. doi:10.1109/WoWMoM.2015.7158174.

[104] B. Hull, K. Jamieson, H. Balakrishnan, Mitigating Congestion in Wireless Sensor Networks, in: Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems, SenSys '04, Association for Computing Machinery, New York, NY, USA, 2004, pp. 134–147. doi:10.1145/1031495.1031512.

[105] K. Srinivasan, M. Jain, J. I. Choi, T. Azim, E. S. Kim, P. Levis, B. Krishnamachari, The κ Factor: Inferring Protocol Performance Using Inter-Link Reception Correlation, in: Proceedings of the Sixteenth Annual International Conference on Mobile Computing and Networking, MobiCom '10, Association for Computing Machinery, New York, NY, USA, 2010, pp. 317–328. doi:10.1145/1859995.1860032.

[106] R. Nithya, K. Chang, R. Kapur, L. Girod, E. Kohler, D. Estrin, Sympathy for the sensor network debugger, in: Proceedings of the 3rd international conference on Embedded networked sensor systems, 2005, pp. 255–267. doi:10.1145/1098918.1098946.

[107] S. Rost, H. Balakrishnan, Memento: A Health Monitoring System for Wireless Sensor Networks, in: 2006 3rd Annual IEEE Communications Society on Sensor and Ad Hoc Communications and Networks, Vol. 2, 2006, pp. 575–584. doi:10.1109/SAHCN.2006.288514.

[108] C. Liu, G. Cao, Distributed Monitoring and Aggregation in Wireless Sensor Networks, in: 2010 Proceedings IEEE INFOCOM, 2010, pp. 1–9. doi:10.1109/INFCOM.2010.5462033.

[109] K. Liu, Q. Ma, X. Zhao, Y. Liu, Self-diagnosis for large scale wireless sensor networks, in: 2011 Proceedings IEEE INFOCOM, 2011, pp. 1539–1547. doi:10.1109/INFCOM.2011.5934944.

[110] K. Liu, Q. Ma, W. Gong, X. Miao, Y. Liu, Self-Diagnosis for Detecting System Failures in Large-Scale Wireless Sensor Networks, IEEE Transactions on Wireless Communications 13 (10) (2014) 5535–5545. doi:10.1109/TWC.2014.2336653.

[111] D. Raposo, A. Rodrigues, S. Sinche, J. Sá Silva, F. Boavida, Industrial IoT Monitoring: Technologies and Architecture Proposal, Sensors 18 (10). doi:10.3390/s18103568. [Online]. Available: https://www.mdpi.com/1424-8220/18/10/3568

[112] A. Bierman, M. Björklund, K. Watsen, RESTCONF Protocol, RFC 8040, RFC Editor (01 2017).

[113] M. A. García, G. Camarillo, Extensible Markup Language (XML) Format Extension for Representing Copy Control Attributes in Resource Lists, RFC 5364, RFC Editor (10 2008).

[114] T. Bray, The JavaScript Object Notation (JSON) Data Interchange Format, RFC 8259, RFC Editor (12 2017).

[115] M. Björklund, YANG - A Data Modeling Language for the Network Configuration Protocol (NETCONF), RFC 6020, RFC Editor (10 2010).

[116] P. Van der Stok, A. Bierman, M. Veillette, C. Bormann, CoAP Management Interface (CoMI), draft-vanderstok-core-comi-11, RFC Editor (01 2017).

[117] M. Veillette, P. Van der Stok, A. Pelov, A. Bierman, C. Bormann, CoAP Management Interface (CORECONF), draft-ietf-core-comi-16, RFC Editor (09 2023).

[118] M. Ganesh Bhat, S. Bhattacharjee, C. Gündoğan, K. Alexandris, A. Gogolev, CORECONF, NETCONF, and RESTCONF: Benchmarking Network Orchestration in Constrained IIoT devices, TechRxiv Preprintdoi:10.36227/techrxiv.23987106.v1.

[119] A. Karaagac, E. De Poorter, J. Hoebeke, In-Band Network Telemetry in Industrial Wireless Sensor Networks, IEEE Transactions on Network and Service Management 17 (1) (2020) 517–531. doi:10.1109/TNSM.2019.2949509.

[120] D. Saif, A. Matrawy, A Proposal and Experimental Evaluation Towards Mass Configuration of Heterogeneous IIoT Nodes, TechRxiv Preprintdoi:10.36227/techrxiv.23816748.

[121] Percepio, Tracealyzer - Visual Runtime Insights. [Online]. Available: https://percepio.com/tracealyzer/

[122] M. Khomenko, O. Veligorskyi, The Use of Percepio Tracealyzer for the Development of FreeRTOS-based Applications, in: MC&FPGA-2020, 2020, pp. 26–29. doi:10.35598/mcfpga.2020.008.

[123] M. Ringwald, K. Römer, A. Vialetti, SNIF: Sensor Network Inspection Framework. Department of Computer Science, Tech. rep., ETH Zurich, Technical Report, 535 (2006).

[124] M. Ringwald, K. Römer, A. Vitaletti, Passive inspection of sensor networks, in: Distributed Computing in Sensor Systems: Third IEEE International Conference, DCOSS 2007, Santa Fe, NM, USA, June 18-20, 2007. Proceedings 3, Springer, 2007, pp. 205–222.

[125] K. Romer, J. Ma, PDA: Passive distributed assertions for sensor networks, in: 2009 International Conference on Information Processing in Sensor Networks, 2009, pp. 337–348.

[126] A. Awad, R. Nebel, R. German, F. Dressler, On the need for passive monitoring in sensor networks, in: 2008 11th EUROMICRO Conference on Digital System Design Architectures, Methods and Tools, IEEE, 2008, pp. 693–699.

[127] A. Rodrigues, T. Camilo, J. Sá Silva, F. Boavida, Diagnostic Tools for Wireless Sensor Networks: A Comparative Survey, Journal of Network and Systems Management 21. doi:10.1007/s10922-012-9240-6.

[128] M. N. Mendoza, J. C. Campelo Rivadulla, A. M. Bonastre Pina, J. V. Capella Hernández, R. Ors Carot, HMP: A Hybrid Monitoring Platform for Wireless Sensor Networks Evaluation, IEEE Access 7 (2019) 87027–87041. doi:10.1109/ACCESS.2019.2925299.

[129] M. Keller, J. Beutel, L. Thiele, The Problem Bit, in: 2013 IEEE International Conference on Distributed Computing in Sensor Systems, 2013, pp. 105–114. doi:10.1109/DCOSS.2013.72.

[130] A. Dunkels, L. Mottola, N. Tsiftes, F. Österlind, J. Eriksson, N. Finne, The announcement layer: Beacon coordination for the sensornet stack, in: Wireless Sensor Networks: 8th European Conference, EWSN 2011, Bonn, Germany, February 23-25, 2011. Proceedings 8, Springer, 2011, pp. 211–226. doi:10.1007/978-3-642-19186-2_14.

[131] A. Lahmadi, A. Boeglin, O. F. Inria, Efficient distributed monitoring in 6LoWPAN networks, in: Proceedings of the 9th International Conference on Network and Service Management (CNSM 2013), 2013, pp. 268–276. doi:10.1109/CNSM.2013.6727846.

[132] G. Gaillard, D. Barthel, F. Theoleyre, F. Valois, Monitoring KPIs in synchronized FTDMA multi-hop wireless networks, in: 2016 Wireless Days (WD), 2016, pp. 1–6. doi:10.1109/WD.2016.7461516.

[133] M. Cociglio, A. Capello, A. T. Bonda, L. Castaldelli, A packet-based method for passive performance monitoring, draft-tempia-opsawg-p3m-00, expired. [Online]. Available: http://www.watersprings.org/pub/id/draft-tempia-opsawg-p3m-00.txt

[134] G. Fioccola, M. Cociglio, G. Mirsky, T. Mizrahi, Z. Tianran, Alternate-Marking Method, RFC 9341, RFC Editor (12 2022).

[135] A. Riesenberg, Y. Kirzon, M. Bunin, E. Galili, G. Navon, T. Mizrahi, Time-Multiplexed Parsing in Marking-Based Network Telemetry, in: Proceedings of the 12th ACM International Conference on Systems and Storage, SYSTOR '19, Association for Computing Machinery, New York, NY, USA, 2019, p. 80–85. doi:10.1145/3319647.3325837.

[136] T. Mizrahi, G. Navon, G. Fioccola, M. Cociglio, M. Chen, G. Mirsky, AM-PM: Efficient Network Telemetry using Alternate Marking, IEEE

Network 33 (4) (2019) 155–161. doi:10.1109/MNET.2019.1800152.
[137] A. Karaagac, E. De Poorter, J. Hoebeke, Alternate Marking-based Network Telemetry for Industrial WSNs, in: 2020 16th IEEE International Conference on Factory Communication Systems (WFCS), 2020, pp. 1–8. doi:10.1109/WFCS47810.2020.9114490.
[138] Y. Liu, K. Liu, M. Li, Passive Diagnosis for Wireless Sensor Networks, IEEE/ACM Transactions on Networking 18 (4) (2010) 1132–1144. doi:10.1109/TNET.2009.2037497.
[139] Open Networking Foundation, Programming Protocol-independent Packet Processors (P4). [Online]. Available: https://opennetworking.org/p4/
[140] C. Kim, P. Bhide, E. Doe, H. Holbrook, A. Ghanwani, D. Daly, M. Hira, B. Davie, In-band network telemetry (INT) dataplane specification (2016).
[141] Telemetry, In-band Network, In-band Network Telemetry (INT) Dataplane Specification, P4. org Applications Working Group.
[142] A. Gulenko, M. Wallschläger, O. Kao, A Practical Implementation of In-Band Network Telemetry in Open vSwitch, in: 2018 IEEE 7th International Conference on Cloud Networking (CloudNet), 2018, pp. 1–4. doi:10.1109/CloudNet.2018.8549431.
[143] L. Tan, W. Su, W. Zhang, J. Lv, Z. Zhang, J. Miao, X. Liu, N. Li, In-band Network Telemetry: A Survey, Computer Networks 186 (2021) 107763. doi:https://doi.org/10.1016/j.comnet.2020.107763.
[144] D. Van Leemput, J. Hoebeke, E. De Poorter, Analytical traffic model of 6TiSCH using real-time in-band telemetry, Internet of Things 23 (2023) 100847. doi:https://doi.org/10.1016/j.iot.2023.100847.
[145] O. Ben-Kiki, C. Evans, B. Ingerson, YAML Ain't Markup Language (YAML) (2001). [Online]. Available: https://yaml.org/
[146] S. Furuhashi, Message Pack - It's like JSON. but fast and small. [Online]. Available: https://msgpack.org/
[147] S. M. Araque, I. Martinez, G. Z. Papadopoulos, N. Montavont, L. Toutain, Toward a Standard Time Series Representation for IoT based on CBOR Templates, in: 2022 Global Information Infrastructure and Networking Symposium (GIIS), 2022, pp. 13–19. doi:10.1109/GIIS56506.2022.9936910.
[148] Google Inc., Protocol Buffers. [Online]. Available: https://protobuf.dev/
[149] Memfault Inc., Event Serialization. [Online]. Available: https://docs.memfault.com/docs/mcu/event-serialization-overview/
[150] C. Bormann, Concise Binary Object Representation (CBOR), RFC 8949, RFC Editor (12 2020).
[151] F. Graf, Comparison of Serialization Formats. [Online]. Available: https://gist.github.com/fabiangraf96/4c213cd340612aa5382783e05b97bd92
[152] OpenJS Foundation, Node-RED. [Online]. Available: https://nodered.org/
[153] M. Lekić, G. Gardašević, IoT sensor integration to Node-RED platform, in: 2018 17th International Symposium INFOTEH-JAHORINA (INFOTEH), 2018, pp. 1–5. doi:10.1109/INFOTEH.2018.8345544.
[154] InfluxData Inc., Telegraf - the open source server agent for metric collection. [Online]. Available: https://www.influxdata.com/time-series-platform/telegraf/
[155] InfluxData Inc., InfluxDB - It's About Time. [Online]. Available: https://www.influxdata.com
[156] Grafana Labs, Grafana: The open observability platform. [Online]. Available: https://grafana.com/
[157] C. Capodiferro, M. Mazzei, An approach adopted for smart data generation and visualization problems, ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences 6 (2020) 41–45.
[158] Zabbix, Zabbix: The Enterprise-Class Open Source Network Monitoring Solution. [Online]. Available: https://www.zabbix.com/
[159] S. Gajica, Monitoring of 6TiSCH infrastructure with MQTT and Zabbix NMS software, in: 2020 International Symposium on Industrial Electronics and Applications (INDEL), 2020, pp. 1–6. doi:10.1109/INDEL50386.2020.9266161.
[160] Linear Technology, SmartMesh SDK 1.0.5.138 - JSonServer (2018). [Online]. Available: https://github.com/dustcloud/smartmeshsdk/tree/master/app/JsonServer
[161] F. Graf, T. Watteyne, M. Villnow, AIOT Play FW Zephyr. [Online]. Available: https://github.com/aiotsystems/aiot_play_fw_zephyr.git
[162] Analog Devices (Dust Networks), SmartMesh IP Node 2.4GHz 802.15.4e Wireless Mote-on-Chip. [Online]. Available: https://www.analog.com/media/en/technical-documentation/data-sheets/5800ipmfa.pdf
[163] Linear Technology, SmartMesh IP C-Library (2015). [Online]. Available: https://github.com/dustcloud/sm_clib
[164] Nordic Semiconductor, nRF Connect SDK. [Online]. Available: https://www.nordicsemi.com/Products/Development-software/nrf-connect-sdk
[165] Inria AIO Team, AIOT Play. [Online]. Available: http://aiotsystems.org/
[166] C. Thonier, T. Watteyne, F. Garde, T. Razafindralambo, B. Marcon, AIOT: the All-in-1 IoT Educational Tool You Need, FutureIoT PhD school, poster (Aug. 2022). [Online]. Available: https://inria.hal.science/hal-03820401
[167] T. Watteyne, P. Tuset-Peiro, X. Vilajosana, S. Pollin, B. Krishnamachari, Teaching Communication Technologies and Standards for the Industrial IoT? Use 6TiSCH!, IEEE Communications Magazine 55 (5) (2017) 132–137. doi:10.1109/MCOM.2017.1700013.
[168] Inria AIO Team, AIOT Play FW (2022). [Online]. Available: https://github.com/aiotsystems/aiot_play_fw
[169] Memfault Inc., Memfault firmware sdk (2023). [Online]. Available: https://github.com/memfault/memfault-firmware-sdk
[170] Memfault Inc., Memfault - Data from Firmware to the Cloud. [Online]. Available: https://docs.memfault.com/docs/mcu/data-from-firmware-to-the-cloud/
[171] Linear Technology, SmartMesh IP User's Guide. [Online]. Available: https://www.analog.com/media/en/technical-documentation/user-guides/SmartMesh_IP_User_s_Guide.pdf
[172] Memfault Inc., Memfault CLI tool. [Online]. Available: https://docs.memfault.com/docs/ci/install-memfault-cli/
[173] Linear Technology, SmartMesh IP Power and Performance Estimator - V2.05b. [Online]. Available: https://www.analog.com/media/en/simulation-models/software-and-simulation/SmartMesh_Power_and_Performance_Estimator.xls
[174] L. Luo, Y. Zhang, C. White, B. Keating, B. Pearson, X. Shao, Z. Ling, H. Yu, C. Zou, X. Fu, On Security of TrustZone-M-Based IoT Systems, IEEE Internet of Things Journal 9 (12) (2022) 9683–9699. doi:10.1109/JIOT.2022.3144405.
[175] G. Selander, J. P. Mattsson, F. Palombini, Ephemeral Diffie-Hellman Over COSE (EDHOC), Rfc, RFC Editor (09 2023).
[176] M. Vučinić, G. Selander, J. P. Mattsson, T. Watteyne, Lightweight Authenticated Key Exchange With EDHOC, Computer 55 (4) (2022) 94–100. doi:10.1109/MC.2022.3144764.
[177] J. Schaad, CBOR Object Signing and Encryption (COSE), RFC 8152, RFC Editor (07 2017).
[178] K. L. Ketshabetswe, A. M. Zungeru, B. Mtengi, C. K. Lebekwe, S. R. S. Prabaharan, Data Compression Algorithms for Wireless Sensor Networks: A Review and Comparison, IEEE Access 9 (2021) 136872–136891. doi:10.1109/ACCESS.2021.3116311.
[179] I. Nassra, J. V. Capella, Data compression techniques in IoT-enabled wireless body sensor networks: A systematic literature review and research trends for QoS improvement, Internet of Things 23 (2023) 100806. doi:https://doi.org/10.1016/j.iot.2023.100806.
[180] Massey, Travis L. and Mehta, Ankur and Watteyne, Thomas and Pister, Kristofer S. J., Packet Compression for Time-Synchronized Wireless Networks, in: 2010 7th Annual IEEE Communications Society Conference on Sensor, Mesh and Ad Hoc Communications and Networks (SECON), 2010, pp. 1–3. doi:10.1109/SECON.2010.5508212.