Programmable Solutions for Low-power Lossy Wireless Networks: A Study of SDN and Femto Containers

Ahmad Mahmod, Julien Montavont and Thomas Noel

Abstract Low-power Lossy Wireless Networks (LLWNs) are characterized by constraints in memory, processing, and power consumption, coupled with an inherently dynamic wireless environment. In this context, a suite of programmable communication protocols is essential to efficiently adapt to varying network conditions, optimize resource utilization, and maintain performance within the stringent limitations of LLWN devices. In this work, we review and compare state-of-the-art network programmability techniques to assess their suitability for LLWNs. Based on the findings, we propose a new network architecture for LLWNs, utilizing software-defined networking for control plane programmability and Femto Containers lightweight virtualization for data plane programmability, ensuring it respects the constraints of LLWN devices. We have conducted a proof-of-concept validation to demonstrate the feasibility of Femto Containers to implement the data plane in LLWN on the FIT IoT-LAB testbed. The results show that our architecture successfully achieves a substantial reduction in update size while adhering to memory and power consumption constraints of LLWN devices, although this comes at the cost of a slightly acceptable increased packet processing delay.

1 Introduction

A Low-power Lossy Wireless Network (LLWN) is a branch of Internet of Things consisting of a large number of embedded devices connected using lossy wireless communication links. LLWNs include sensors, actuators and gateways which are

Ahmad Mahmod
ICube, University of Strasbourg, Pole API, 67412 Illkirch, France, e-mail: mahmod@unistra.fr

Julien Montavont
ICube, University of Strasbourg, Pole API, 67412 Illkirch, France, e-mail: montavont@unistra.fr

Thoms Noel
ICube, University of Strasbourg, Pole API, 67412 Illkirch, France, e-mail: noel@unistra.fr

advantageous where the installation of infrastructure-based networks (e.g., 5G) is not possible or is expensive. The application areas of LLWN include environmental monitoring, healthcare, industrial automation and many other areas [1].

LLWN devices are typically constrained by limited battery power, as well as limited processing and memory capacity which result in short-range, low data rate and possibly multi-hop communications [2]. These constraints highlight the need for network protocols that cope with the limited resources of LLWN by using available power efficiently and reducing memory and processing overhead.

The wireless nature of LLWN makes communication inherently dynamic. This dynamicity results from environmental changes, mutual interference between devices, power depletion, and mobility requirements in some applications, which cause topology variations and affect communication performance, making the communication links unstable and prone to high packet loss. The dynamic environmental conditions, combined with the diverse Quality of Service (QoS) requirements in applications, require reconfiguring network protocols. This underscores the need for *network programmability*: the ability to reconfigure the protocol suite according to varying conditions and requirements to achieve optimal performance.

This reconfiguration may involve adjusting the parameters of specific protocols or replacing the entire protocol. At the application level, when a new application prioritizes the integrity of the data, enabling checksum in the transport layer ensures data integrity by detecting corruption in the header and payload during transmission. For routing, hop-by-hop routing is particularly beneficial in scenarios where the nodes in the network are static and possess sufficient memory and energy resources to maintain routing tables. However, as node density increases, the size of routing tables on intermediate nodes escalates substantially. This scalability necessitates a transition to the source-routing, where centralized route management mitigates the burden on individual nodes.

At the medium access level, in dense IoT networks where devices experience high collision rates and energy inefficiency due to contention-based CSMA, the transition to time-slotted medium access (TSMA) ensures deterministic communication and improved energy efficiency [3]. Three programmability levels are defined in [4]:

- **Monolithic** defines n protocols and switches between them (e.g., switch from CoAP to MQTT).
- **Parametric** modifies some protocol parameters (e.g., backoff time of the radio).
- **Modular** defines functions in modules and interconnects them to construct the entire protocol logic representing the highest programmability level.

Implementing a monolithic solution by providing the operating system with multiple concurrent protocols is impractical due to constraints in memory and processing capacity. Over-The-Air (OTA) firmware updates resolve this issue by enabling the replacement of the running firmware with a new version that includes the necessary protocols. However, OTA requires transmitting the whole firmware image over a multi-hop constrained network and then rebooting the nodes to install the new version. This process is likely to increase the power consumption in nodes and requires

a large memory footprint, in addition to network congestion and service disruption as nodes exchange a large volume of messages to converge to a stable state [5]. Finally, some operating systems offer Application Programming Interfaces (APIs) to modify specific parameters of the network stack, such as RIOT [6]. However, this solution provides limited configuration options.

The available solutions for enabling programmability in LLWN are insufficient, as they either do not respect the constraints of LLWN, or offer only limited programmability. In this article, we propose a new architecture that ensures high programmability of the protocol suite, including low-level functions essential for wireless communications, while also adhering to the constraints of LLWN devices. To the best of our knowledge, we are the first to use virtualization techniques to implement network protocols in constrained LLWN environments. The contributions of this article are:

1. Reviewing various network programmability techniques and studying their feasibility for LLWN.
2. Proposing a novel architecture for LLWN using Software Defined Network (SDN) and Femto Container (FC) lightweight virtualization.
3. Validating our approach with a proof-of-concept implementation.

2 Background and Existing Works

Network processes are divided into two main planes: the *control plane* and the *data plane*. The control plane serves the intelligence of the network, responsible for decision-making and rule-setting for data forwarding. In contrast, the data plane applies these rules and handles the actual forwarding of data packets. Achieving a high level of programmability necessitates reconfigurability in both the decision-making (control plane) and decision-applying (data plane) components. We detail here background notions on the state-of-the-art of control plane and data plane programmability.

2.1 Control Plane Programmability

The Software Defined Networking (SDN) paradigm redefines network architecture by separating the control plane from the data plane [7]. In SDN, the control plane is centralized within an entity known as the SDN controller. This controller maintains a comprehensive, global view of the network and oversees the data plane functions that remain distributed across network devices. Centralization allows the control plane to be programmable, enabling the SDN controller to dynamically adjust network behavior and optimize performance based on real-time conditions.

In the LLWN context, the SDN paradigm enables the offloading of complex control tasks to the central controller. This approach allows devices to prioritize efficient data transmission and energy conservation. Given that LLWN networks typically operate in a multi-hop fashion, many proposals focus on decentralized routing, where path computation is handled by the central controller. SDN-WISE [8]

replaces the packet processing pipeline of devices with Match-Action flow tables managed by the controller. Ouhab *et al.* have proposed a hybrid approach where a distributed routing protocol is utilized at a small scale, while the large-scale management of routing paths is delegated to an SDN controller [9]. Other solutions have been developed to manage the scheduling of time-slotted networks. SDN-TSCH [10] introduced a novel SDN-based scheduling approach that isolates flows, which helps to meet and guarantee their QoS requirements, and ensures a reliable control plane through the use of dedicated slots. We observe that the majority of SDN-based works in LLWN focus on specific tasks in the network stack such as scheduling and forwarding. In this article, our objective is to expand on this contribution by advocating for the comprehensive management of the entire communication protocol suite.

2.2 Data Plane Programmability

In this section, we review some state-of-the-art technologies that can be used to program the data plane and compare their feasibility for LLWN.

2.2.1 P4 Programming Language

Programming Protocol-independent Packet Processors (P4) is a high-level programming language dedicated to programming the data plane of network devices such as routers or switches [11]. This architecture is hardware-agnostic and consists of three main stages: the Parser, responsible for understanding the packet header; the Processing stage, which manipulates packets in a key-action manner; and the Deparser, which reconstructs the processed packet. For example, P4 has been used to define the data plane of IEEE802.11 in the Linux network stack, facilitating access to previously inaccessible management frames [12].

2.2.2 eBPF

The extended Berkeley Packet Filter (eBPF) is a virtual machine for programming the kernel of Linux-based operating systems, enabling versatile applications in security, monitoring, and networking [13]. The eBPF virtual machine is event-based, triggered by specific events using hooks—checkpoints installed in the operating system to monitor particular events. Networking hooks include eXpress Data Path (XDP) at the lowest layer of the Linux network stack, offering fast packet processing with basic and limited actions, and Traffic Control (TC) in the upper layers, which offers broader processing capabilities, striking a balance between performance and flexibility. The virtual machine is lightweight, featuring 11 registers and a 512-byte stack, and can be updated and connected without the need to modify the kernel. eBPF has many applications in networking, such as extending the TCP stack with new arbitrary options [14].

2.2.3 Femto Container

Femto Container (FC) is a new middleware that enables the deployment of lightweight virtual machines on resource-constrained devices [15]. This technology extends the eBPF virtual machine to Real-Time Operating Systems (RTOS) used in LLWN devices, offering a minimal memory footprint and affordable processing overhead. Moreover, FCs are hardware-agnostic and therefore compatible with various hardware specifications or boards. FC is lightweight, featuring 11 registers and a 512-bytes stack, and operates on an event-based model similar to eBPF. However, FCs extend its functionality with user-defined hooks that can be installed at any point in the operating system, from the driver to the application layer. The launching and updating of FCs are transparent to the operating system, and do not require firmware updates. In [15], FC was used to read sensor data at the driver level and transmit it using the Constrained Application Protocol (CoAP) at the application level.

We can conclude that, compared to eBPF, FC maintains the same virtual machine architecture but introduces a new engine for eBPF virtual machines within RTOS. Moreover, unlike eBPF, which is restricted to predefined hooks, FC allows users to define hooks at any point within the operating system.

2.2.4 Comparison

Table 1 compares the reviewed technologies. While P4 and eBPF are robust solutions for programming the data plane in devices with high performance, they present challenges for deployment in LLWN devices due to hardware limitations and no radio management capabilities. P4 requires more powerful hardware than typically available in LLWN devices, and lacks P4 targets for such resource-constrained devices. eBPF, despite its small memory footprint, is originally designed for Linux OS, which imposes hardware requirements that exceed those of LLWN devices. Both P4 and eBPF primarily focus on post-packet reception processing and do not directly manage radio-related operations. Although eBPF can perform some driver-level tasks, its capabilities are limited to basic operations such as packet dropping, redirection, and forwarding.

Table 1 Comparison between Technologies

	P4	**eBPF**	**Femto-Container**
Scope	Domain-specific for data plane of network devices	Programming Linux Kernel including network stack	Event-driven applications in constrained devices
Footprint	Large memory and processing requirements	Small memory footprint	Small memory footprint
Limitations	Need high performance hardware, no radio management	Limited to Linux Kernel, no radio management	Limited to some RTOSs until now

In contrast, FC is a promising solution for implementing isolated network protocols and managing radio-related operations through specific hooks at different operating system levels. With its minimal memory footprint, light processing overhead, and event-triggered architecture, FC is well-suited for the resource-constrained nature of LLWN. A modular approach can be adopted, where elementary functions are implemented in independent FCs. By interconnecting these FCs, we can create complex application logic. These applications include communication protocols attached to different hooks across the protocol stack, allowing runtime updates. While FCs are compatible with various hardware platforms, their current limitation to certain RTOS exists. However, as a novel technology, there is potential for FCs to expand support to additional operating systems in the future.

3 Proposed Architecture

For programming constrained LLWN, we propose an architecture that integrates the SDN paradigm and lightweight virtualization, featuring an SDN controller that serves as the central manager of the network and runs the control plane that pushes the protocols in the data plane distributed on LLWN devices. The data plane in the devices adopts a micro-service approach, where fundamental functions are implemented within lightweight virtual machines. These virtual machines, each representing a micro-service, offer secure and isolated functionalities that can be easily updated. By interconnecting these micro-services, a complete protocol can be constructed within the data plane. Based on our previous review, we propose Femto Containers (FCs) to define these micro-services, *but any other lightweight virtualization technique could play this role*. Fig. 1 illustrates the architecture, which will be detailed in the following sections.

3.1 Control Plane

The SDN controller continuously receives updates on environmental conditions from LLWN devices, including metrics such as the packet delivery rate and interference level. Based on the evaluation of these conditions and performance targets, the

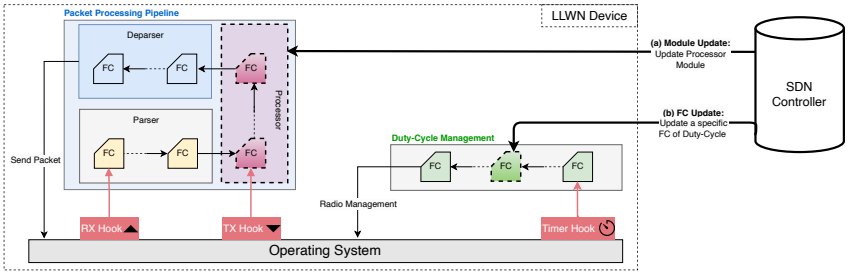

Fig. 1 Proposed Architecture

controller defines the appropriate protocols in the form of FC chains and distributes them to the data plane in the devices. The proposed modifications can range from adjusting specific protocol parameters to updating entire protocol or individual functions as needed. For example, if the packet delivery rate drops significantly due to increased interference, the controller might switch from a standard MAC protocol to a more robust, interference-tolerant protocol to maintain network performance and reliability.

3.2 Data Plane

The data plane is distributed in all LLWN devices and consists of sequences of FCs, each responsible for fundamental functions such as medium access control and packet processing. Each FC may be triggered by a hook (event) such as packet reception and transmission events or timing events, or it may be triggered by another FC in the chain. The FC-based data plane can handle high-level network tasks, for instance, consider the implementation of a Packet Processing Pipeline (Fig. 1). Upon receiving a message from the radio (RX Hook), the Parser is activated to decompose the message header. Subsequently, the processing stage determines the appropriate output before initiating the Deparser to reconstruct the message for transmission. Additionally, FCs can manage low-level networking aspects such as pre-reception functions related to the radio using specific timing hooks, such as Duty-Cycle Management (Fig. 1). These functionalities are crucial and cannot be achieved using P4 or eBPF.

3.3 Architecture Features

Our proposed architecture introduces a **lightweight**, **programmable**, and **modular** network stack for LLWNs, leveraging SDN principles and lightweight FCs for virtualization.

The **lightweight** nature of the architecture is achieved through the integration of event-driven, lightweight FCs and the centralization of the control plane. This design significantly reduces the computational overhead on LLWN devices.

In terms of **programmability**, the adoption of the SDN paradigm in the control plane facilitates the dynamic specification of network protocols to meet the changing requirements and conditions of the network. Additionally, FC virtualization in the data plane provides a flexible framework that supports dynamic updates deployed by the SDN controller in the control plane.

The architecture is also inherently **modular**, incorporating a two-tier modularity in the data plane. The first level of modularity exists between protocols or services, allowing individual protocols to be updated or replaced independently without impacting other components. For instance, the replacement of the Processor module does not affect the other modules illustrated in Fig. 1-(a). The second level of modularity exists within the protocol itself, allowing individual FCs to be updated independently of the others. For instance, a specific FC in Duty-Cycle Management can be updated while the others remain unchanged as depicted in Fig. 1-(b).

Compared to traditional OTA updates, this architecture aligns well with the constraints of LLWNs. Rather than transmitting an entire firmware image for updates, it focuses solely on updating the relevant FCs. This approach minimizes network congestion, reduces power consumption and required memory footprint, and accelerates the update process, making it more efficient and sustainable for LLWN environments.

4 Evaluation

To validate the feasibility of using lightweight virtualization technique to implement network protocols, we implemented the UDP protocol using Femto-Containers in RIOT operating system as a proof-of-concept. We selected UDP because it is one of the simplest protocols in the network stack, making it an ideal candidate for initial implementation. Future work will focus on implementing protocol updates and extending the implementation to include other layers of the protocol stack proposed in our architecture. This open-source implementation[1] was compared to the default GNRC (Generic network stack) IP in RIOT. Fig. 2 shows the network stack of both implementations. In the GNRC stack, each layer has its own thread running permanently in the background along with the associated thread stack. In contrast, our implementation is event-based, with two Femto-Containers being triggered only when a packet is received by (UDP Recv) or sent from (UDP Send) the UDP layer. The checksum in UDP, is implemented as an update and can be optionally installed by the controller on LLWN devices when data integrity is required. This approach offers a significant advantage over GNRC, which requires a complete OTA firmware update when it becomes necessary.

The experiments were conducted on the FIT IoT-LAB testbed [16] using the IoT-LAB M3 board, which features an ARM Cortex M3 CPU, 2.4 GHz radio transceiver, 256KB of ROM, and 64KB of RAM.

We compared the FC and GNRC implementations on four metrics: update size, memory footprint, power consumption and execution time across various scenarios. To support reproducibility, we provide the raw results and processing scripts in the Git repository[1].

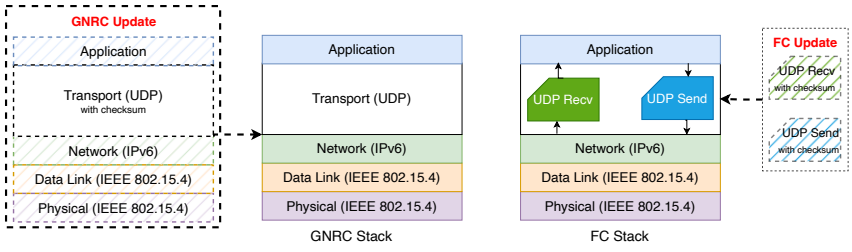

Fig. 2 GNRC Stack and FC Stack and Updating Methodology

[1] https://github.com/ahmahmod/UDP-Protocol-using-Femto-Containers

Table 2 Update Size (KB)

GNRC	UDP Send	UDP Recv
113.65 KB	0.59 KB	0.60 KB

4.1 Update Size

To evaluate the advantages of our proposed architecture, we compared the size of updates required to enable checksum functionality in both the FC-based and GNRC implementations as shwon in Table. 2. In the GNRC implementation, an OTA firmware update of 113.65 KB is necessary, even for this minor modification. In contrast, our FC-based architecture requires updating only two FCs (UDP Send and UDP Recv), with a total size of 1.19 KB. This demonstrates that the update size using our architecture constitutes merely **1.05%** of the OTA firmware update required by GNRC.

Large update sizes pose significant challenges in constrained LLWNs, particularly when fragmentation is employed, such as in 6LoWPAN over IEEE 802.15.4 networks. The loss of a single fragment leads to the complete failure of the packet, resulting in increased power consumption and elevated processing load due to retransmissions and recovery efforts.

This significant reduction in update size leads to substantial benefits, including decreased update time, reduced network congestion, lower power consumption, and minimized memory footprint and processing load on devices. These results highlight the efficiency and suitability of our architecture for resource-constrained LLWN environments.

In future work, we plan to implement more complex scenarios to further demonstrate the significant advantages of adopting this architecture, particularly in dynamic and high-density network conditions.

4.2 Memory Footprint

We compared the ROM and RAM footprints of GNRC and FC implementations, both written in C, using the LLVM compiler on the FIT IoT-LAB M3 node. Footprints were analyzed with Cosy[2]. As shown in Fig. 3, the FC implementation increases the ROM footprint by 2.49% compared to GNRC. This increase is due to the installation of the FC engine and new modules for packet processing and interaction with RIOT. On the other hand, the RAM footprint of the FC implementation shows a reduction in RAM usage by almost 5.7% compared to GNRC. While the FC engine slightly increases the RAM footprint, this is offset by the removal of the continuously running thread for the UDP layer and its dedicated stack in RAM. Overall, this adjustment compensates for the slight increase and results in a reduced overall RAM footprint.

[2] https://github.com/haukepetersen/cosy

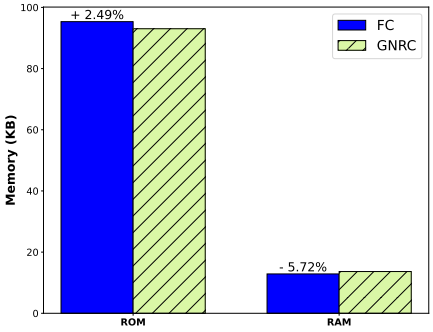

Fig. 3 Memory Footprint

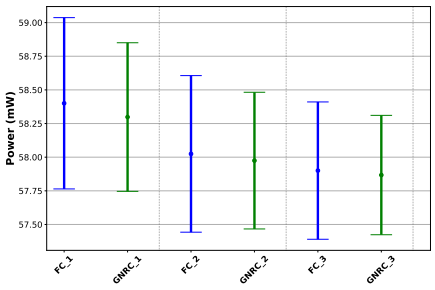

Fig. 4 Power Consumption

4.3 Power Consumption

To measure the power consumption of the FC and GNRC implementations, we disabled the radio transceiver of one FIT IoT-LAB M3 node to isolate its power consumption contribution. Subsequently, we ran the UDP sender and UDP receiver together on this node to measure the power consumption resulting from both implementations. The communication scenario involved sending 1000 packets from the UDP sender to the UDP receiver using the loopback interface. We varied the transmission intervals between 1-second, 2-seconds, and 3-seconds to assess power consumption under different operational conditions. We used the INA226 hardware component provided by FIT IoT-LAB to measure power consumption, taking periodic measurements every 588 μs with an averaging count of 512.

By observing the results in Fig. 4, we can see that both implementations have comparable power consumption. This demonstrates that our proposed architecture, leveraging on lightweight virtualization, maintains low power consumption—a critical factor for LLWN devices—despite the utilization of virtualization. Our architecture is energy efficient for packet processing, but it still requires further investigation for low-level management tasks that manipulate the radio.

4.4 Execution Time

We measured the execution time needed to send or receive a packet at the UDP layer to compare the performance of the FC and GNRC implementations. To conduct this measurement, we sent 1000 packets at 1-second intervals from a UDP sender on one M3 node to a UDP receiver on another M3 node. Additionally, to demonstrate interoperability, we measured the execution time for scenarios where packets were exchanged between two nodes, with one node running the FC implementation and the other running the GNRC implementation.

Fig. 5 and Fig. 6 show the execution time for each packet, with the sequence number indicated on the X-axis. The results show almost constant execution times for the transmission and reception of UDP packets over time for both implementations.

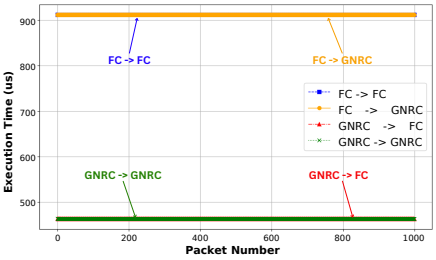

Fig. 5 Transmission Execution Time

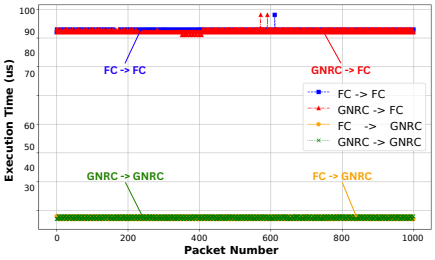

Fig. 6 Reception Execution Time

The longer execution time for transmission compared to reception in both implementations is due to a *while loop* in the code, which increases processing overhead. Fig. 5 indicates that the FC implementation takes approximately 1.97 times longer than the GNRC for transmission, while Fig. 6 shows that FC increases reception time by about 3.3 times compared to GNRC, due to virtualization overhead. Despite this, FC's execution time remains in the microsecond range, which is acceptable for LLWN networks. This is the trade-off for achieving a fully programmable data plane in LLWN using virtualization. However, using FC for synchronous protocols that require precise timings may be challenging, a topic we will explore further in future research.

5 Conclusion and Future Work

A programmable protocol suite for LLWNs provides essential adaptability to dynamic wireless environments and varying application QoS requirements. This flexibility ensures optimized performance and enhances resilience in LLWN. In this article, we have reviewed and compared several network programming technologies and studied their feasibility for LLWN. We then proposed a lightweight, programmable and modular architecture that respects the constraints of LLWN devices and responds to the dynamic changes of the environment. We have also validated the feasibility of using lightweight virtualization to define the data plane through a proof-of-concept implementation of the UDP protocol and checksum update using Femto Containers (FCs), comparing it with the GNRC implementation in RIOT operating system across the FIT IoT-LAB testbed.

The proof-of-concept study demonstrated a **98.95%** reduction in the update size required to enable checksum functionality in UDP using FC implementation when compared to the GNRC implementation using OTA firmware updates. This significant reduction underscores the remarkable efficiency of our proposed architecture in minimizing update overhead, a critical factor for constrained LLWNs. System-level results showed that our proposal balances a slight increase in ROM with a corresponding reduction in RAM usage. Moreover, both implementations present similar power consumption profiles. Finally, our implementation showed a slight increase in packet processing delay, but remains in the microsecond range which is accept-

able in LLWN communications. This point will be further investigated, especially when we will consider synchronous protocol.

For future work, we aim to implement the entire network stack of LLWN devices in FCs, including low-level protocols such as MAC protocols. Our initial choice of UDP was driven by its simplicity, serving as a first step to validate the feasibility of using FCs for implementing network protocols. We will also develop an easy-update mechanism for the installed FCs and integrate it with an SDN controller to manage the distribution of FCs.

Acknowledgements This work was funded by ANR, Grant ANR-23-CE25-0008.

References

1. Ko, J., et al. (2011). Connecting low-power and lossy networks to the internet. In: IEEE Communications Magazine.
2. Omar, A., et al. (2023). A comprehensive survey on detection of sinkhole attack in routing over low power and Lossy network for internet of things. In: Internet of Things 22.
3. Djidi, N. E. H., et al. (2022). The revenge of asynchronous protocols: Wake-up Radio-based Multi-hop Multi-channel MAC protocol for WSN. In: IEEE WCNC.
4. P. H. Isolani, et al. (2019). A Survey on the Programmability of Wireless MAC Protocols. In: IEEE Communications Surveys & Tutorials.
5. C. Vallati, et al. (2019). Improving Network Formation in 6TiSCH Networks. In IEEE Transactions on Mobile Computing.
6. E. Baccelli, et al. (2018). RIOT: An Open Source Operating System for Low-End Embedded Devices in the IoT. In: IEEE Internet of Things Journal.
7. W. Xia, et al. (2015). A Survey on Software-Defined Networking. In: IEEE Communications Surveys & Tutorials.
8. L. Galluccio, et al. (2015). SDN-WISE: Design, prototyping and experimentation of a stateful SDN solution for WIreless SEnsor networks. In: INFOCOM.
9. Ouhab, A., et al. (2020). Energy-efficient clustering and routing algorithm for large-scale SDN-based IoT monitoring. In: IEEE-ICC.
10. Veisi, F., et al. (2023). Enabling centralized scheduling using software defined networking in industrial wireless sensor networks. In: IEEE Internet of Things Journal.
11. Hauser, F., et al. (2023). A survey on data plane programming with p4: Fundamentals, advances, and applied research. In: Journal of Network and Computer Applications.
12. Zanna, P., et al. (2020). WP4: A P4 Programmable IEEE 802.11 Data Plane. In: 30th International Telecommunication Networks and Applications Conference (ITNAC).
13. Vieira, M. A., et al. (2020). Fast packet processing with ebpf and xdp: Concepts, code, challenges, and applications. In: ACM Computing Surveys (CSUR).
14. Tran, V. H., et al. (2019). Beyond socket options: making the Linux TCP stack truly extensible. In: 2019 IFIP Networking Conference.
15. Zandberg, K., et al. (2022). Femto-containers: lightweight virtualization and fault isolation for small software functions on low-power IoT microcontrollers. In: Proceedings of the 23rd ACM/IFIP International Middleware Conference.
16. Adjih, C., et al. (2015). FIT IoT-LAB: A large scale open experimental IoT testbed. In: IEEE 2nd World Forum on Internet of Things (WF-IoT).