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

Ahmad Mahmod, Julien Montavont, Thomas Noel

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

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

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

Thomas Noel
ICube, University of Strasbourg
Pole API, 67412 Illkirch, France
noel@unistra.fr

***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 programmable communication protocol suite 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 conducted a proof-of-concept validation to demonstrate the feasibility of our proposed architecture on the FIT IoT-LAB testbed. The results show that our architecture successfully meets the memory and power consumption constraints of LLWN devices, although this comes at the cost of a slightly increased packet processing delay.**

***Index Terms*—Low-power Lossy Wireless Network, Software Defined Networking, Programmable Data Plane, Lightweight Virtualization, Femto Container**

I. Introduction

The Internet of Things (IoT) aims to connect physical objects in the real world to the Internet [1]. A Low-power Lossy Wireless Network (LLWN) is a branch of IoT consisting of a large number of embedded devices connected using lossy wireless communication links. LLWN include sensors, actuators, and gateways which are advantageous where the installation of infrastructure-based networks (e.g., 5G) is not possible or is prohibitively expensive. The application areas of LLWN include environmental monitoring, such as the collection of climate temperature and humidity data over large areas; healthcare, for the collection and transmission of vital signs (e.g., heart rate); industrial automation, to monitor machinery status and performance; and many other areas [2].

LLWN devices are typically constrained by limited battery power, as well as limited processing and memory capacity which results in short-range, low data rate and possibly multi-hops communications [3]. 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 application Quality-of-Service (QoS) requirements in the network, render the use of a single protocol suite inefficient. This underscores the need for *network programmability*: the ability to reconfigure the protocol suite according to varying conditions to achieve optimal performance. This reconfiguration may involve adjusting parameters of specific protocols or replacing the entire protocol suite. Three main programmability levels are defined in [4]:

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

Currently, some solutions for network programmability involve hardcoding network protocols into the operating system. Implementing a monolithic solution by provisioning the operating system with numerous concurrent protocols is impractical due to constraints in memory and processing capacity. Over-the-Air (OTA) updates enable the replacement of running firmware with a new version that includes updated protocols. However, this solution disrupts service due to the required reboot and place additional strain on the network with the new firmware image. Moreover, it significantly increases power consumption due to the transmission of the new firmware

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

image, the reboot process, and the need to reestablish communication (resulting in the loss of all states due to the reboot). Nevertheless, this monolithic solution eliminates the need to provision the operating system with multiple concurrent protocols. Some operating systems offer a parametric approach using Application Programming Interfaces (APIs) to modify specific parameters of the network stack, such as RIOT [5]. However, this solution provides limited configuration options.

We can observe that the available solutions are insufficient for LLWN-constrained devices, as they do not respect the constraints of LLWN or provide only a limited level of programmability. For this reason, there is a pressing need for a new architecture that provides high programmability of the protocol suite, including low-level functions crucial for wireless communications, while also respecting the constraints of LLWN devices.

In this article, we propose an architecture that respects the constraints of LLWN devices and provides a high level of programmability for both the control plane and data plane. To the best of our knowledge, we are the first to exploit virtualization techniques to implement the data plane in constrained LLWN environments. The contributions of this article are threefold:

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 lightweight virtualization.
3) Validating and evaluating the proposed architecture by providing a proof-of-concept implementation.

II. 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 handle 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.

A. Control Plane Programmability

The Software Defined Networking (SDN) paradigm redefines network architecture by separating the control plane from the data plane [6]. 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 IoT 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 IoT 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 [7] replaces the packet processing pipeline of devices with Match-Action flow tables managed by the controller. Each packet that matches a rule in these tables triggers a predefined action, such as forwarding the packet to a specific neighbor. What sets SDN-WISE from other solutions is that the flow tables include node state information, enabling nodes to make forwarding decisions based on the state of other nodes. 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 [8].

Other solutions have been developed to manage the scheduling of time-slotted networks. SDN-WISE has been enhanced in [9] by scheduling the flows according to QoS indicators. SDN-TSCH [10] proposed a novel SDN-based scheduling that isolates flows and provides a reliable control plane using dedicated slots.

We observe that the majority of SDN-based works in IoT focuses on configuring data forwarding rules or managing the scheduling of time-slotted MAC protocols. In this article, our objective is to expand on this contribution by advocating for the comprehensive management of the entire communication protocol suite.

B. 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.

*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) **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

TABLE I
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 IoT 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

has many applications in networking, such as extending the TCP stack with new arbitrary options [14].

*3) **Femto Container**:* Femto Container (FC) is a new middleware that enables the deployment of lightweight virtual machines on resource-constrained IoT devices [15]. This technology extends the eBPF virtual machine to Real-Time Operating Systems (RTOS) used in IoT 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 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. For security and isolation, FC performs memory access checks and uses pre-flight verification to ensure the safety of FC applications before execution. 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.

*4) **Comparison**:* LLWN devices face significant constraints in power, communication, memory, and processing, in addition to a dynamic wireless environment. These factors necessitate a dynamic configuration of the protocol suite to respond to changing conditions. This configuration should encompass both high-level processing protocols and crucial low-level functions such as frequency switching and duty cycling, which are essential for optimizing efficiency in LLWNs.

Table I 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.

By 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 Real-Time Operating Systems (RTOS) exists. However, as a novel technology, there is potential for FCs to expand support to additional operating systems in the future.

III. PROPOSED ARCHITECTURE

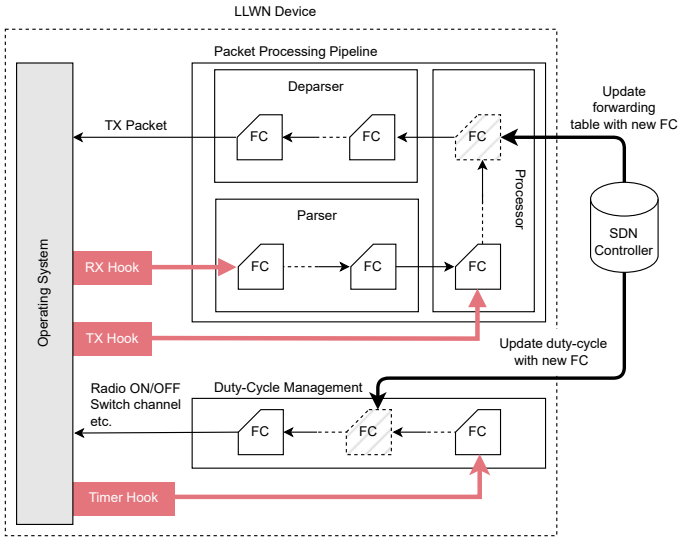

Fig. 1. Proposed Architecture

For programming LLWN, we propose an architecture that integrates the SDN paradigm, featuring an SDN controller that serves as the central manager of the network and runs the control plane, leaving only the data plane 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 suite 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.

A. Control Plane

The SDN controller continuously receives updates on environmental conditions from LLWN devices, including metrics such as Packet Delivery Rate (PDR) and interference level. Based on the evaluation of these conditions and performance targets, the controller defines the appropriate protocols in the form of FC chains and distributes them to the devices. The proposed modifications can range from adjusting specific protocol parameters to updating entire protocol or individual functions as needed.

Implementing a centralized SDN architecture in LLWNs poses significant challenges, primarily due to unreliable links and network contentions that can potentially disrupt control traffic. Ensuring successful updates and fast convergence require the reliable and timely transmission of modifications from the SDN controller to LLWN devices. This is crucial as all devices should promptly apply the modifications to restore their communication capabilities. One potential solution, as suggested in [10], involves allocating dedicated time-frequency blocks for control traffic. This approach aims to establish a reliable control plane by removing contention and ensuring that control messages reach devices effectively.

B. Data Plane

The data plane is distributed in all LLWN devices and consists of a sequence of FCs, each responsible for fundamental functions such as medium access control and packet processing. This is achieved using a wide range of hooks that can be installed at different layers within the operating system.

For instance, consider the implementation of a simple forwarding protocol using FCs (Fig. 1). Upon receiving a message from the radio, 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 pre-reception functions related to the radio using specific timing hooks, such as duty cycling (Fig. 1). These functionalities are crucial and cannot be achieved using P4 or eBPF.

C. Architecture Features

Our architecture is adaptive and features a programmable control plane and data plane. The SDN paradigm in the control plane enables the definition of network protocols tailored to specific requirements and conditions. Additionally, FC lightweight virtualization in the data plane offers a flexible solution to accommodate dynamic updates deployed by the control plane.

In terms of modularity, our architecture has a double-modular data plane. The first level of modularity operates between protocols (services), enabling the replacement of one protocol with another without affecting the others. For example, updating the Deparser does not impact the Processor or the Parser. The second level of modularity exists within each protocol itself, allowing individual FCs (micro-services) to be updated independently of the others. For instance, specific FCs within the Processor can be updated while the others remain unchanged, as illustrated in Fig. 1.

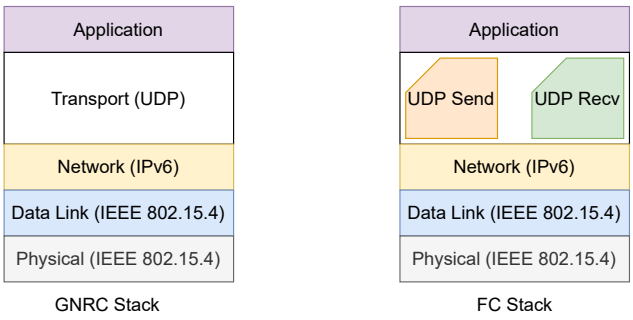

Fig. 2. GNRC and FC Stack Implementations

IV. Evaluation

To validate our proposed architecture, we implemented the UDP protocol using Femto-Containers in RIOT as a proof-of-concept. This open-source implementation[1] was compared to the default GNRC IP stack 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. By 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 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 (IEEE 802.15.4) radio transceiver, 256KB of ROM, and 64KB of RAM.

We compared the FC and GNRC implementations on three metrics: memory footprint, power consumption, and execution time across various scenarios. The raw results and processing scripts are available in the Git repository[1].

A. Memory

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 the cosy tool [17]. 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.

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

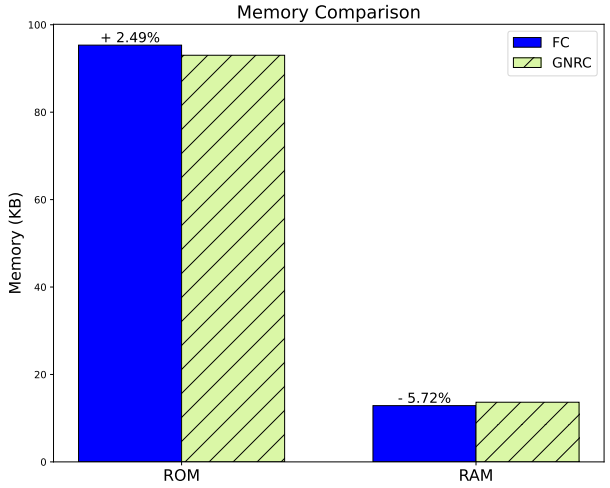

Fig. 3. Memory Comparison

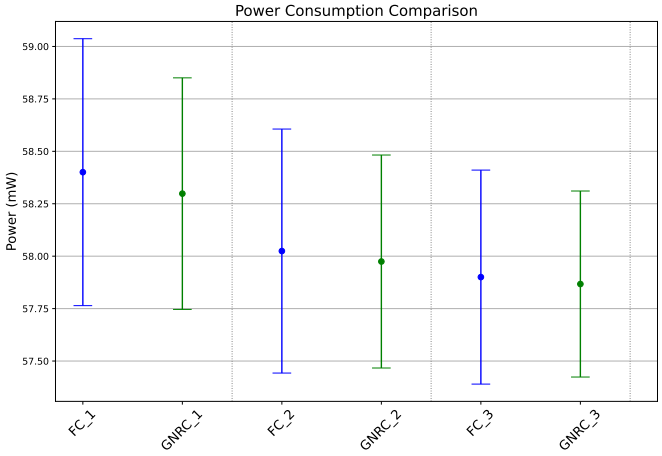

Fig. 4. Power Consumption Comparison

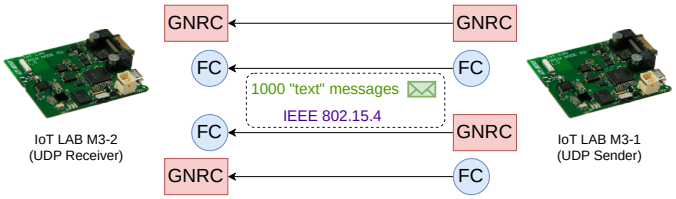

Fig. 5. Execution Time Scenarios

B. 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 both the UDP sender and UDP receiver on this node to measure the power consumption resulting from both implementations. This setup involved triggering FCs for handling UDP packets or running the GNRC UDP thread.

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. The communication scenario involved sending 1000 packets from the UDP sender to the UDP receiver, both located on the same node (using the loop-back interface). We varied the transmission intervals between 1-second, 2-seconds, and 3-seconds to assess power consumption under different operational conditions.

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. The comparable power consumption results from the nature of the FC implementation, which is event-triggered and calls an FC only when a packet needs to be sent or received. Our architecture is energy efficient for packet processing, but it still requires further investigation for low-level management tasks that manipulate the radio.

C. 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. The execution time for the FC implementation was obtained from debugging information, while a timer was used for the GNRC implementation. 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 for both FC and GNRC implementations. 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 illustrates the different scenarios.

Fig. 6 and 7 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. 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. 6 indicates that the FC implementation takes approximately 1.97 times longer than the GNRC for transmission, while Fig. 7 shows that FC increases reception time by about 3.3 times compared to GNRC, due to virtualization overhead. Some execution time outliers may occur because of high-priority interrupts, such as radio acknowledgments and retries, which extend processing time.

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 requiring precise timings may be challenging, a topic we will explore further in future research.

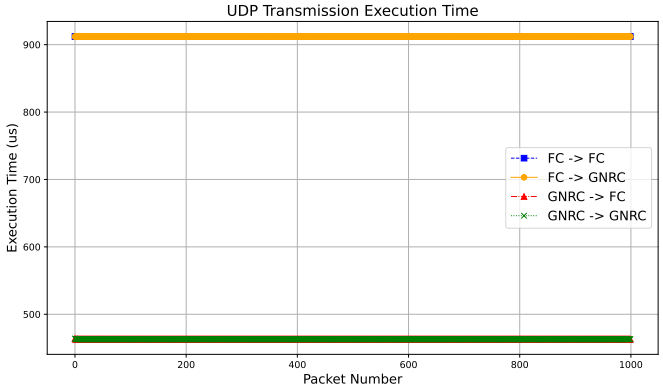

Fig. 6. Transmission Execution Time Comparison

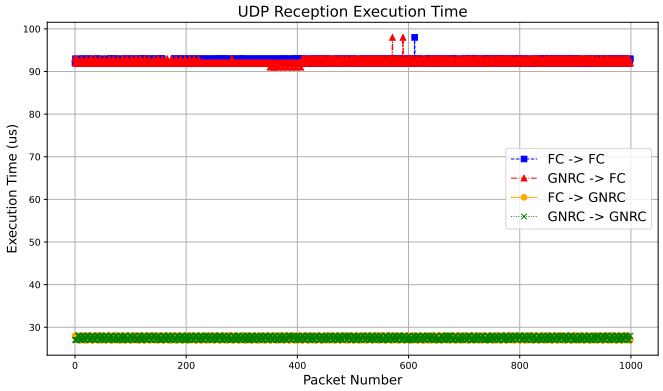

Fig. 7. Reception Execution Time Comparison

V. Conclusion and Future Works

A programmable protocol suite for LLWNs offers crucial adaptability to dynamically changing wireless environments, ensuring optimized performance and resilience against environmental fluctuations. In this article, we reviewed and compared several network programming technologies and studied their feasibility for LLWN. We then proposed a double-programmable and a double-modular architecture that respects the constraints of LLWN devices and responds to the dynamic changes of the environment.

We validated our architecture through a proof-of-concept implementation of the UDP protocol using Femto Containers (FCs), comparing it with the GNRC implementation in RIOT operating system across the FIT IoT-LAB testbed. 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. The event-based nature of FCs effectively manages power consumption associated with virtualization and RAM footprint by spawning FCs only when required, thus eliminating the need for continuously running thread as found in GNRC. Finally, our implementation showed a slight increase in packet processing delay, but remains in the microsecond range. 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 the low-level protocols, such as MAC 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. Finally, machine learning may be employed in the SDN controller to determine the best protocol suite based on the running application's needs.

References

[1] K. Kritsis, G. Z. Papadopoulos, A. Gallais, P. Chatzimisios and F. Théoleyre, "A Tutorial on Performance Evaluation and Validation Methodology for Low-Power and Lossy Networks," in IEEE Communications Surveys & Tutorials, vol. 20, no. 3, pp. 1799-1825, thirdquarter 2018.

[2] J. Ko, A. Terzis, S. Dawson-Haggerty, D. E. Culler, J. W. Hui and P. Levis, "Connecting low-power and lossy networks to the internet," in IEEE Communications Magazine, vol. 49, no. 4, pp. 96-101, April 2011.

[3] H. Almutairi and N. Zhang, 'A Survey on Routing Solutions for Low-Power and Lossy Networks: Toward A Reliable Path-Finding', Network, 2024.

[4] P. H. Isolani, M. Claeys, C. Donato, L. Z. Granville and S. Latré, "A Survey on the Programmability of Wireless MAC Protocols," in IEEE Communications Surveys & Tutorials, vol. 21, no. 2, pp. 1064-1092, Second quarter 2019.

[5] E. Baccelli et al., "RIOT: An Open Source Operating System for Low-End Embedded Devices in the IoT," in IEEE Internet of Things Journal, vol. 5, no. 6, pp. 4428-4440, Dec. 2018.

[6] W. Xia, Y. Wen, C. H. Foh, D. Niyato and H. Xie, "A Survey on Software-Defined Networking," in IEEE Communications Surveys & Tutorials, vol. 17, no. 1, pp. 27-51, Firstquarter 2015.

[7] L. Galluccio, S. Milardo, G. Morabito and S. Palazzo, "SDN-WISE: Design, prototyping and experimentation of a stateful SDN solution for WIreless SEnsor networks," 2015 IEEE Conference on Computer Communications (INFOCOM), Hong Kong, China, 2015, pp. 513-521.

[8] A. Ouhab, T. Abreu, H. Slimani and A. Mellouk, "Energy-efficient clustering and routing algorithm for large-scale SDN-based IoT monitoring," ICC 2020 - 2020 IEEE International Conference on Communications (ICC), Dublin, Ireland, 2020, pp. 1-6.

[9] F. Orozco-Santos, V. Sempere-Payá, T. Albero-Albero, and J. Silvestre-Blanes, 'Enhancing SDN WISE with Slicing Over TSCH', Sensors, vol. 21, no. 4, 2021.

[10] F. Veisi, J. Montavont and F. Théoleyre, "Enabling Centralized Scheduling Using Software Defined Networking in Industrial Wireless Sensor Networks," in IEEE Internet of Things Journal, vol. 10, no. 23, pp. 20675-20685, 1 Dec, 2023.

[11] F. Hauser et al., 'A Survey on Data Plane Programming with P4: Fundamentals, Advances, and Applied Research', CoRR, vol. abs/2101.10632, 2021.

[12] P. Zanna, P. Radcliffe and D. Kumar, "WP4: A P4 Programmable IEEE 802.11 Data Plane," 2020 30th International Telecommunication Networks and Applications Conference (ITNAC), Melbourne, VIC, Australia, 2020, pp. 1-6.

[13] M. Vieira, M. Castanho, R. Pacífico, E. Santos, E. Júnior, and L. Vieira, 'Fast Packet Processing with eBPF and XDP: Concepts, Code, Challenges, and Applications', ACM Comput. Surv., vol. 53, no. 1, Feb. 2020.

[14] V. -H. Tran and O. Bonaventure, "Beyond socket options: making the linux TCP stack truly extensible," 2019 IFIP Networking Conference (IFIP Networking), Warsaw, Poland, 2019, pp. 1-9.

[15] K. Zandberg, E. Baccelli, S. Yuan, F. Besson, and J.-P. Talpin, '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, Quebec, Canada, 2022, pp. 161–173.

[16] C. Adjih et al., "FIT IoT-LAB: A large scale open experimental IoT testbed," 2015 IEEE 2nd World Forum on Internet of Things (WF-IoT), Milan, Italy, 2015, pp. 459-464.

[17] Cosy Tool. https://github.com/haukepetersen/cosy