PENSIL: ProgrammablE Network Stack for low-power lossy IoT networks using Lightweight-virtualization

Ahmad Mahmod[a,*], Julien Montavont[a], Thomas Noel[a]

[a]*ICube lab (UMR CNRS 7357), University of Strasbourg, Strasbourg, 67000, France*

Abstract

Low-Power and Lossy Wireless Networks (LLWNs) form the foundation of the Internet of Things (IoT), connecting billions of constrained devices across diverse domains. Despite their critical role, the design of LLWN devices is strongly constrained by limited memory, processing power, and energy supply. These limitations have historically led to the adoption of monolithic network stacks, where protocol logic is tightly integrated and bound at compile time. As a result, even minor changes require a full firmware update, making protocol evolution costly and impractical. Because LLWN deployments face diverse and evolving conditions, a single static stack design or fixed configuration is insufficient. In this paper, we propose PENSIL, a network architecture featuring a programmable and modular network stack for LLWN that enables selective updates of protocol functions, combined with a central orchestrator that manages device stacks. PENSIL enables dynamic and semantic reconfiguration, from parameter tuning to network configuration swapping, allowing networks to adapt without downtime. A proof-of-concept implementation on real hardware demonstrates that our architecture enhances performance through fast, lightweight and secure updates while respecting the stringent memory, energy, and processing constraints of LLWN devices, ultimately bridging the gap between programmability and efficiency.

Keywords:
Internet of Things (IoT), Low-power Lossy Wireless Network (LLWN), Programmable networks, Lightweight virtualization, Over-the-Air update.

1. Introduction

The Internet of Things (IoT) aims to connect every "thing" or "object" around us, enabling them to collect data, process it, and make decisions automatically [1]. Low-Power and Lossy Wireless Network (LLWN) is a core component of IoT, supporting applications ranging from healthcare and environmental monitoring to smart manufacturing and smart cities. LLWN devices function as sensors or actuators connected wirelessly, operating under strict resource constraints—including limited energy, memory, and processing capacity—in order to remain low-cost [2]. These limitations result in constrained wireless communication in terms of range and data rate, often leading to lossy links.

With the continuous evolution of IoT networks, software updates have become essential to ensure long-term sustainability, maintain connectivity, address security vulnerabilities, and adapt to evolving application requirements. Updates also reduce electronic waste by extending device lifetimes and avoiding frequent replacements or physical interventions. Over-the-Air (OTA) mechanisms provide a practical solution, enabling remote updates without physical access and lowering operational costs [3].

Most LLWN operating systems hardcode the network stack into the system, necessitating full-image firmware updates even for minor reconfigurations (e.g., tuning a single parameter) [4]. Although simple, full-image OTA updates are inefficient, as they transmit large binaries over bandwidth-limited LLWN and require a reboot, which results in the loss of all network states. The device must then rerun the necessary protocols to reconverge, leading to long connectivity interruptions [5].

[*]Corresponding author
Email address: `mahmod@unistra.fr` (Ahmad Mahmod)

We propose PENSIL, a network architecture that introduces a modular, programmable, and lightweight network stack for LLWNs. PENSIL relies on lightweight Virtual Functions (VFs), each implementing a specific network function. By chaining these VFs, complete network protocols can be flexibly composed and dynamically reconfigured. Each VF operates independently of the device firmware and can be updated over-the-air (OTA). PENSIL thus enables a wide range of programmability, from fine-grained parameter tuning (e.g., adjusting the retransmission limit at the MAC layer) to full protocol or function replacement (e.g., switching from hop-by-hop to source routing).

PENSIL includes a central orchestrator that manages the network stacks of connected devices and supports any control-plane deployment model. The orchestration process is **semantic-aware**, extending beyond conventional binary image management to interpret protocol semantics and state dependencies. This enables the orchestrator to precisely determine *what* to update and *how* to apply those updates within the network stack. By reasoning about the functional context of each update, the orchestrator maintains continuous connectivity and prevents service disruptions that typically result from naive, low-level modifications to running protocols. For example, updating a routing protocol often leads to the loss of forwarding tables and temporary disconnection. In PENSIL, the orchestrator can centrally store each node's neighbor and routing information, automatically restoring forwarding tables after the update to prevent connectivity interruptions and reconvergence delays, as demonstrated in our evaluation. Through this semantic-aware orchestration, PENSIL achieves safe, context-preserving updates across protocol layers, enabling dynamic reconfiguration without compromising network stability or performance.

The main contributions of this paper are as follows:

1. A programmable and highly modular network stack for LLWNs that enables flexible, fine-grained, and selective run-time updates.

2. An over-the-air (OTA) management framework that supports semantic-aware network stack reconfiguration and multiple control-plane deployment modes.

3. A proof-of-concept implementation and experimental evaluation on real hardware, validating the practicality and efficiency of the proposed architecture.

To encourage other researchers to reproduce and expand our results, we will release the implementation of PENSIL and the evaluation scripts used in this paper. The remainder of this paper is organized as follows: Section 2 reviews existing approaches for enabling runtime updates, focusing on both dynamic code updates and SDN-based techniques. Section 3 details the proposed architecture, Section 4 presents the proof-of-concept study using the UDP protocol, and Section 5 concludes the paper and discusses future work.

2. Background and Related Works

2.1. Dynamic Code Update Frameworks in LLWN

Many runtime code update approaches have been proposed to update code after deployment in constrained LLWN devices. These approaches primarily aim to reduce update size and time and avoid the reboot required after full-image firmware updates. Most of them focus on updating applications, while far less attention has been devoted to updating the network stack.

The **scripting approach** interprets script logic at runtime before execution. This runtime interpretation allows scripts to be updated after deployment. Several well-known scripting languages, including JavaScript, have been ported to constrained IoT devices in frameworks such as RIOTjs [6] and JerryScript [7]. However, scripting approaches require a significant memory footprint and processing capacity to interpret scripts at runtime, making them heavy for constrained LLWN devices.

The **module-based approach** decomposes firmware into separate modules that can be independently developed and compiled, establishing clear boundaries between system components and enabling updates to target specific modules rather than the entire firmware. The *symbol table* relocates the addresses of functions and data in each module to their physical memory addresses, enabling linking between modules. The binding model between already installed modules and new ones defines two categories of the module-based approach: strict and loosely-coupled.

In *strict binding*, the base image exposes a symbol table generated at build time and fixed at deployment, as exemplified by Contiki [8] and TinyOS [9]. Because symbol tables are non-extensible at runtime, updates are typically confined to application-level modules layered above the base image (kernel). By contrast, *loosely-coupled binding* allows runtime interaction between modules, enabling the symbol table to be modified after deployment and expanding programmability toward kernel components but introduces post-update tasks (re-binding, allocation, relocation) that can increase complexity and operational risk. Systems such as Remoware [10], Lorien [11], and GITAR [12] adopt loosely-coupled binding. However, only GITAR has demonstrated updates to network stack components, while the others focus solely on the application layer.

The **virtualization approach** encapsulates functionality within independent virtual machines that operate separately from the device firmware. This enables strong isolation between services and allows components to be added, replaced, or reused without impacting the rest of the system, in a plug-and-play manner. Examples include WebAssembly [13], adapted Java VM [14], and rBPF [15]. While this approach achieves high modularity and reusability, it introduces non-negligible execution overhead, which may limit its suitability for constrained IoT devices.

All previous frameworks using these approaches lack a remote OTA update mechanism—except TinyOS, which is inherently very limited in programmability. This creates a fundamental gap: while these approaches are modular in concept, they fail to translate modularity into practice, since updates still require physical intervention. Such intervention is disruptive, costly, and impractical in large-scale LLWN. On the other hand, GITAR is the only effort in this field that has attempted to introduce a form of modularity to the network stack. However, beside lacking OTA updates, it lacks several key features that make it impractical and inefficient for real network programmability. By packaging each protocol as a single monolithic module rather than decomposing it into fine-grained networking components, the system treats protocols as generic software units instead of networking entities. This obscures essential semantics (e.g., state machines, timers) and leads to coarse-grained, inefficient updates.

Recently, a **lightweight virtualization approach** has emerged, designed to minimize overhead and resource usage. One example is the *Femto Container (FC)*, a middleware framework for deploying lightweight virtual functions on resource-constrained devices. These virtual functions maintain a minimal memory footprint and low processing overhead while being hardware-agnostic, enabling operation across diverse hardware platforms and boards. The architecture follows an event-driven execution model, triggered by specific events integrated basically at the application layer. FCs can be launched and updated transparently to the operating system as independent elements, without requiring full firmware updates [16].

Lightweight VFs form the core of our network architecture. We extend FC beyond the application layer to additional layers of the network stack and employ it to evaluate our architecture, as detailed in next sections.

2.2. SDN-Based Data Plane Programmability

Early LLWN deployments relied on a distributed control plane, with control functionality co-located with the data plane on network nodes. Examples include distributed routing protocols such as RPL [17] and contention-based MAC scheduling such as IEEE 802.15.4 CSMA/CA [18]. This approach provides local autonomy and minimizes control traffic but may suffer from slow convergence, sub-optimal global performance, and typically requires full firmware updates to modify network behavior—an especially challenging task in LLWNs [19].

More recently, lightweight SDN adaptations have been proposed to address LLWN constraints [20]. Most of these approaches focus on routing management at the SDN controller, dynamically pushing forwarding rules to nodes [21, 22, 23], while others target the MAC layer, managing TDMA resource allocations [24, 25]. SDN has also been applied to enhance security by detecting and mitigating abnormal behavior [26, 27]. Nevertheless, establishing and maintaining the management plane—the link between nodes and the SDN controller used to collect network metrics and push updates—remains challenging, as investigated in [28].

Overall, the programmability provided by these solutions is largely restricted to updating match-action rules, as in conventional wired SDN networks. This limitation underscores the need for a more flexible and fine-grained approach, where entire network functions can be selectively updated at runtime, as proposed with PENSIL.

3. The PENSIL Architecture

In our architecture, Programmable Network Stack for low-power lossy IoT networks using Lightweight-virtualization (PENSIL), network protocols are implemented as sets of lightweight Virtual Functions (VFs). These lightweight VFs minimize overhead, making the approach suitable for constrained devices, while their modular independence from

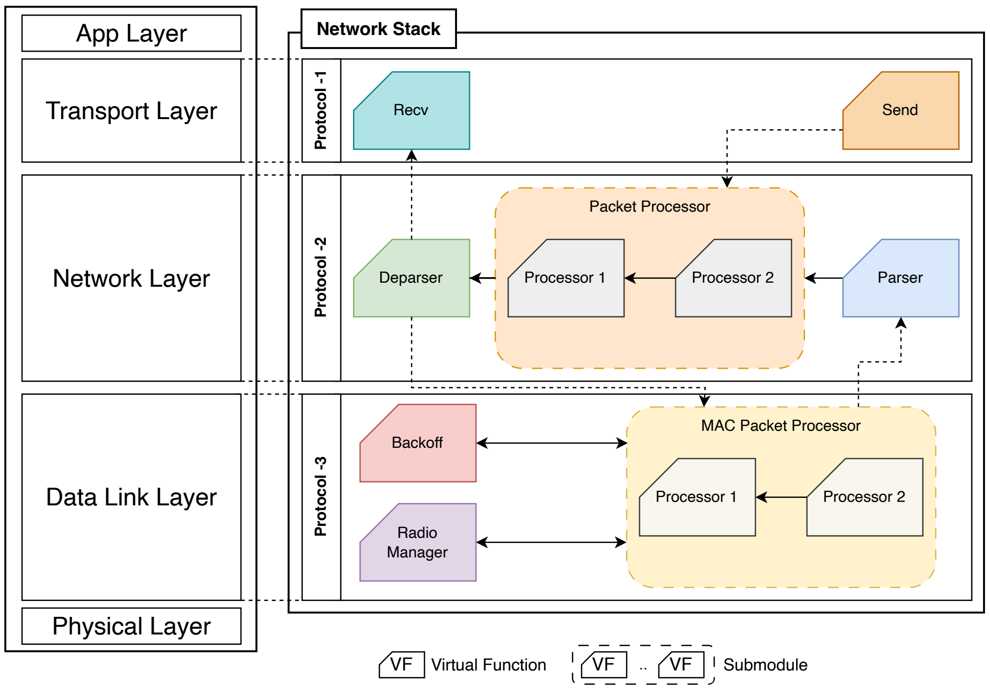


Figure 1: Network Stack on LLWN device

both the operating system and other functions enhances security, isolation, and reliability. A central orchestrator manages the network stacks of all nodes in the network, ensuring semantically consistent and safe configurations. By combining fine-grained modularity with semantic orchestration, PENSIL supports flexible control-plane deployment models, including hybrid approaches that overcome the limitations of purely centralized or distributed modes and thereby enable exceptional programmability. Finally, its function-level modularity allows remote over-the-air (OTA) updates, enabling fast, selective, and efficient reconfiguration of each node's network stack. Femto Containers, or any other lightweight virtualization technique, can be adopted to serve these VFs.

3.1. Network Stack

We build our network stack using **Virtual Functions (VFs)**, minimal building blocks that each implement a specific network function through well-defined interfaces. A **submodule** represents a logical unit formed by composing two or more VFs to realize a concrete capability or feature within the network stack. A complete network protocol can thus be constructed by combining VFs and submodules, enabling the implementation of any communication protocol across both high and low layers of the stack, as illustrated in Fig. 1. For instance, at the Network Layer, a *Parser* VF may parse a received packet and extract its header fields before processing by a *Packet Processing* submodule composed of multiple VFs. Subsequently, another VF acts as a *Deparser* to prepare the packet for transmission. VFs, either individually or grouped into submodules, can implement a wide range of mechanisms, including backoff algorithms, packet scheduling, link-quality estimation, routing decisions, and neighbor discovery. The primary goal of the network stack design is protocol agnosticism, allowing any protocol, whether part of the data plane or the control plane, to be implemented entirely as VFs. By decomposing protocols into independent VFs or submodules, PENSIL achieves fine-grained, function-level control, enabling semantic, selective, and faster updates at runtime so that changes can be applied to a single VF. In contrast, existing approaches often treat the stack as monolithic binaries and typically require full protocol updates. Additionally, the stack promotes reusability, as individual VFs or submodules can be leveraged across different protocols and layers.

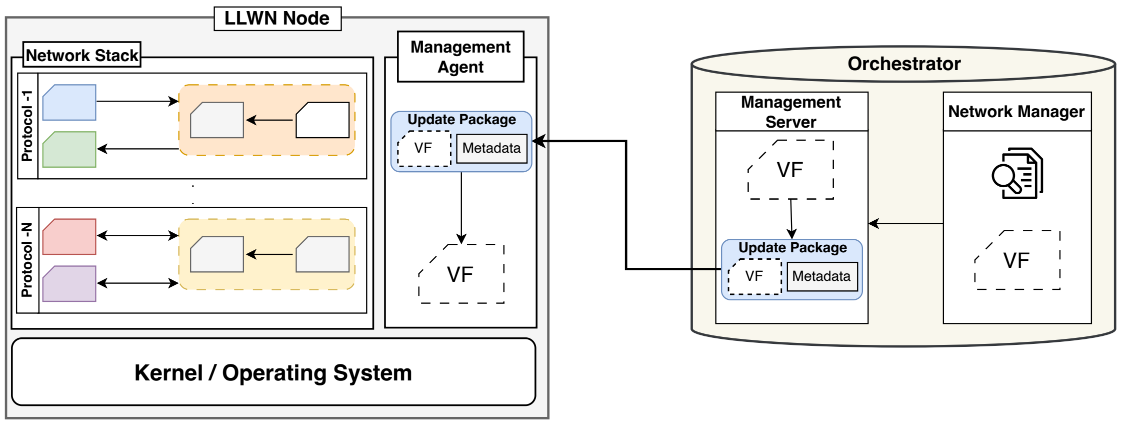


Figure 2: PENSIL Architecture

3.2. Network Stack Management

To update the protocol stack in real time during network exploitation, PENSIL includes a central orchestrator responsible for managing over-the-air the network stacks of LLWN devices. Fig. 2 illustrates the complete PENSIL architecture.

The orchestrator consists of two main elements: (i) *Network Manager*: It analyzes network performance and determines when the network stack should be reconfigured. When updates are needed, it generates new versions of the relevant VF(s) and applies them after compiling and testing. (ii) *Management Server*: Distributes new versions of VF(s) to selected devices in the LLWN. Each VF is packaged together with the necessary metadata to ensure secure and reliable installation, forming an *Update Package*. The metadata guarantees both safety and security: security metadata includes authentication information and a hash code for integrity verification, while installation metadata specifies the version number for version control, the target device class, the VF size, and the location of the new VF within the network stack.

On the LLWN devices, in addition to the modular network stack, a *Management Agent* is required to interact with the Management Server to download updates. These agents implement predefined methods for downloading, parsing metadata, and properly installing the new VF(s). After downloading the Update Package, the Management Agent parses the metadata to perform security checks, extracts the installation parameters, and then waits for a trigger from the Management Server to install the new VF(s).

Our implementation of PENSIL leverages CoAP [29] to realize the management components, with the Management Server implemented as a CoAP server and the Management Agent as a CoAP client. We also rely on the IETF SUIT standard [30] to construct the Update Package, allowing VFs and their metadata to be transmitted in a standardized and secure manner. The use of metadata guarantees safe, authenticated, and verifiable updates, reducing the risk of corruption or malicious injection. The mechanisms by which the Network Manager collects metrics and determines when to trigger VF updates will be explored in future work. The resulting management plane stack is composed of CoAP and SUIT running over UDP and IPv6.

3.3. Update Process

The update process is illustrated in Fig. 3. It begins when the Network Manager decides to reconfigure a running network protocol, a submodule or a single network function, and generates the corresponding configuration as a new VF or a set of VFs (1), which are then sent internally to the Management Server. The Management Server then creates an Update Package for each new VF (2) and notifies the relevant devices about the updates (3).

LLWN devices, through their Management Agents, begin downloading the new VF indicated by the Management Server (4). Upon completing the download, the Management Agent parses the received Update Package and extracts the new VF along with the metadata required for installation in the network stack (5). The Management Agent then sends an acknowledgment to the Management Server to confirm the successful download (6) and waits for

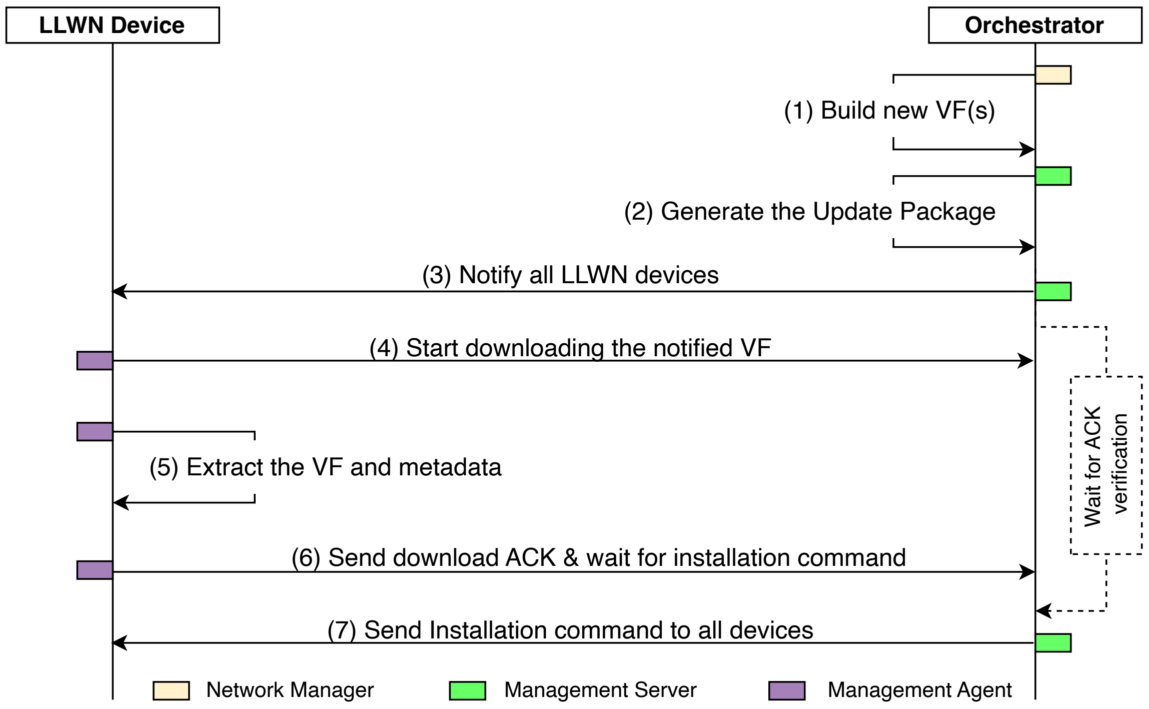

LLWN Device
Orchestrator
(1) Build new VF(s)
(2) Generate the Update Package
(3) Notify all LLWN devices
(4) Start downloading the notified VF
(5) Extract the VF and metadata
(6) Send download ACK & wait for installation command
(7) Send Installation command to all devices
Wait for ACK verification
Network Manager
Management Server
Management Agent

Figure 3: Update Process

the installation command, which is issued only after the Management Server verifies that all required devices have confirmed completion.

If a download process fails and no acknowledgment is received within a specified *Timeout*, the Management Server retries up to the *Maximum Number of Retries*. Devices that still fail to complete the download are marked as inaccessible. Once the *Minimum Number of Updated Devices* threshold is reached, the Management Server issues an installation command, allowing devices to install the new VF and integrate it into the network stack (7). This procedure ensures that all updated devices run the same version of the VF, avoiding inconsistencies among devices that have completed the update. Future work will investigate optimal retransmission parameters and strategies for handling inaccessible devices.

3.4. PENSIL's Control-plane Deployments

PENSIL supports all conventional control-plane deployment models, including distributed and centralized approaches, while also enabling innovative hybrid models that combine the advantages of both, as illustrated in Fig. 4.

Distributed Control Plane Deployment: In this model, both the control and data planes are distributed across LLWN devices, enabling decentralized decision-making through mechanisms such as routing protocols and MAC scheduling. This approach is particularly effective for fast, local decisions and provides inherent redundancy and resilience to errors. However, it can suffer from slow, suboptimal convergence and may become trapped in local optima. Updating the control plane in this mode is complex, as it requires delivering modifications to multiple nodes across potentially multi-hop paths, which increases the risk of failures and inconsistencies. PENSIL supports this mode, as illustrated in Fig. 4-(a). Thanks to PENSIL's modular design, updates can be very small and targeted to individual functions, reducing the amount of data transmitted and thereby lowering the likelihood of update failures.

Centralized Control Plane Deployment: In this model, the control plane is centralized in a dedicated node—referred to as the controller in SDN terminology—which makes global decisions on behalf of the nodes regarding routing, packet processing, and resource allocation. The data plane can be seen as a set of matching-rule tables, where the

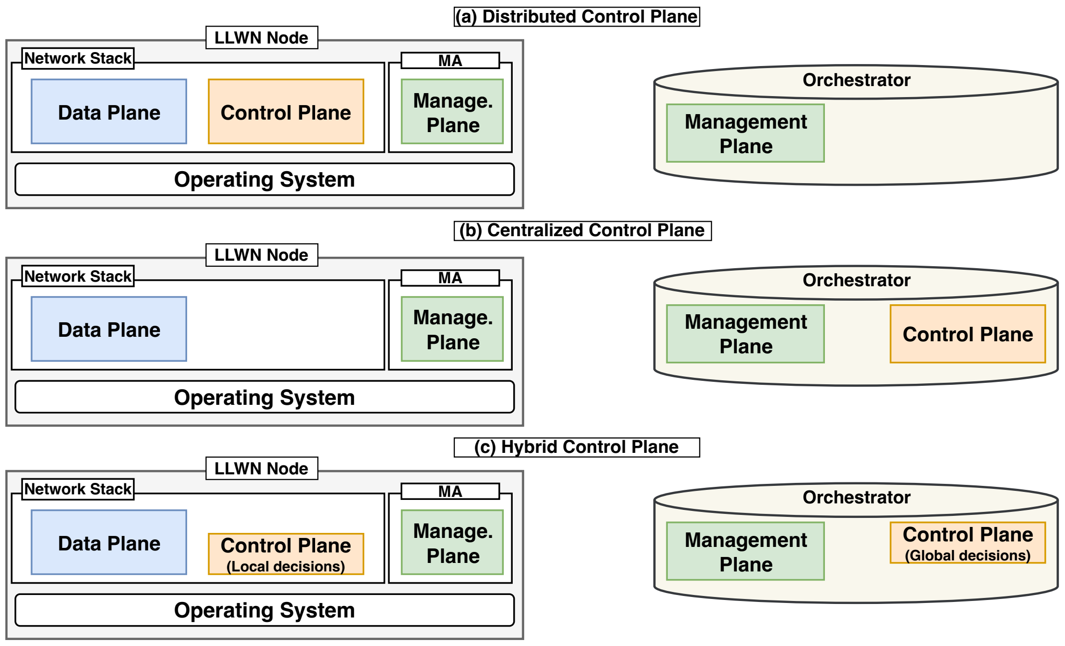


Figure 4: Control Plane Distribution Flexibility: (a) Distributed, (b) Centralized, (c) Hybrid

control plane pushes, removes, or modifies rules. This approach simplifies LLWN devices and enables optimal, network-wide decisions. However, collecting network metrics over potentially multi-hop paths is complex and can lead to delayed or biased views, potentially resulting in suboptimal decisions by the controller. PENSIL supports this mode by leveraging its central orchestrator as the controller, hosting the control plane and pushing rules to device data planes via VFs, as illustrated in Fig. 4-(b). Because VF logic can be added or updated at runtime, new rule types can be introduced without rebuilding device software. Moreover, if the orchestrator predicts network degradation, it can proactively switch to a distributed control-plane mode by pushing the corresponding VFs before the degradation occurs, ensuring continuity and robustness in network operation.

Hybrid Control Plane Deployment: Neither a fully distributed nor a fully centralized control plane is optimal under all conditions. By splitting the control plane between the orchestrator and the devices, PENSIL enables a hybrid mode, as illustrated in Fig. 4-(c), which addresses the limitations of each approach. With its global view, the orchestrator can push convergence information instantly, reducing neighbor exchanges and enabling faster—or even near-instant—convergence with optimal decisions. Meanwhile, local control plane components can make immediate decisions, allowing the network to react quickly to dynamic conditions and maintain continuous operation even in lossy or intermittent environments where connectivity to the orchestrator is unstable.

In conclusion, PENSIL's modular architecture makes it straightforward to adjust the division of control decisions and seamlessly switch between modes at runtime, providing enhanced programmability and adaptability for LLWN across diverse operating conditions.

4. Evaluation

This section presents a proof-of-concept implementation on real hardware platforms to experimentally validate the proposed PENSIL architecture. Specifically, the User Datagram Protocol (UDP)—a lightweight, connectionless transport protocol—is decomposed into two VFs: UDP-Send and UDP-Recv. This protocol is chosen for its simplicity, making it an ideal candidate for an initial validation of the framework. To demonstrate PENSIL's flexibility, UDP is then extended to support reliability through the addition of sequence numbers, acknowledgment messages (ACKs),

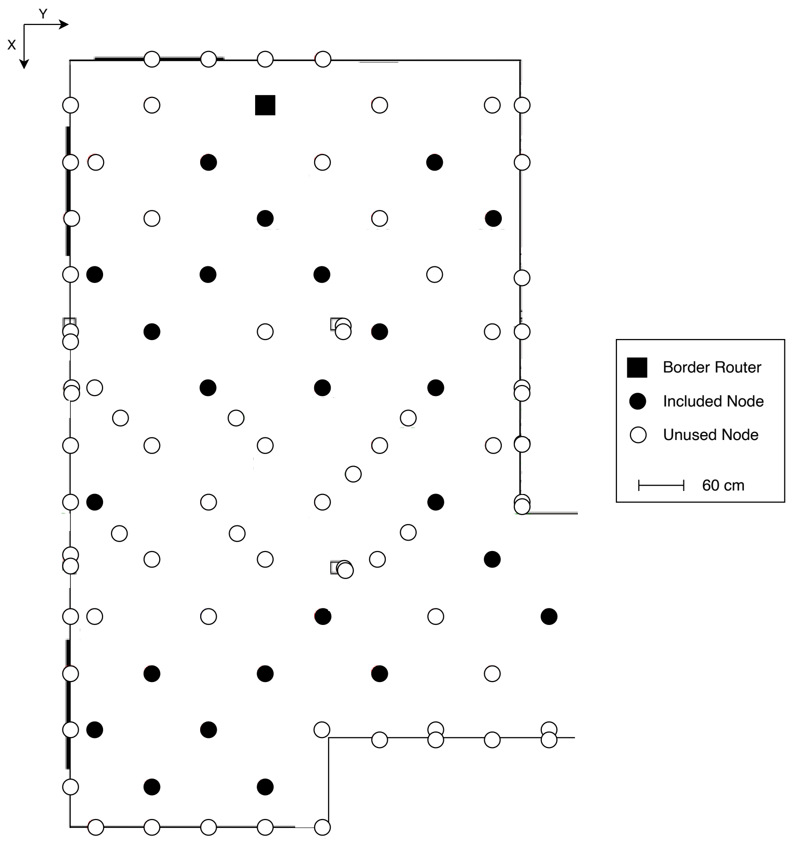


Figure 5: Network Nodes on FIT IoT-LAB Testbed [31]

and a retransmission queue. Future work will apply the same modular approach to more complex protocols across different layers of the network stack.

All experiments are conducted on the FIT IoT-LAB testbed [31], using IoT-LAB M3 nodes representative of typical LLWN devices. Each node features 256 KB of Flash memory, 64 KB of RAM, an ARM Cortex-M3 processor, and a 2.4 GHz transceiver. To ensure reproducibility, the complete implementation of PENSIL is publicly available on GitHub [1].

For benchmarking, the PENSIL UDP implementation is compared with the native UDP module of the RIOT operating system [32], which requires retransmission of the entire firmware image for any update or modification of the network stack. The employed network stack in our experiments follows the IETF protocol suite (UDP/IPv6-RPL/6LoWPAN/IEEE 802.15.4), although PENSIL remains protocol-agnostic and can support alternative data-plane or control-plane protocols.

As shown in Fig. 5, the experimental topology consists of 25 nodes physically deployed across the FIT IoT-LAB testbed. One node operates as the Border Router (BR), providing end-to-end connectivity to the Orchestrator, while the remaining nodes form a realistic multi-hop topology using the RPL routing protocol [17]. Depending on their physical placement, the network depth extends up to three hops from the BR. Each node runs a temperature-sensing application that periodically transmits a 1-byte UDP packet containing the measured temperature to the BR every

[1]https://github.com/ahmahmod/pensil

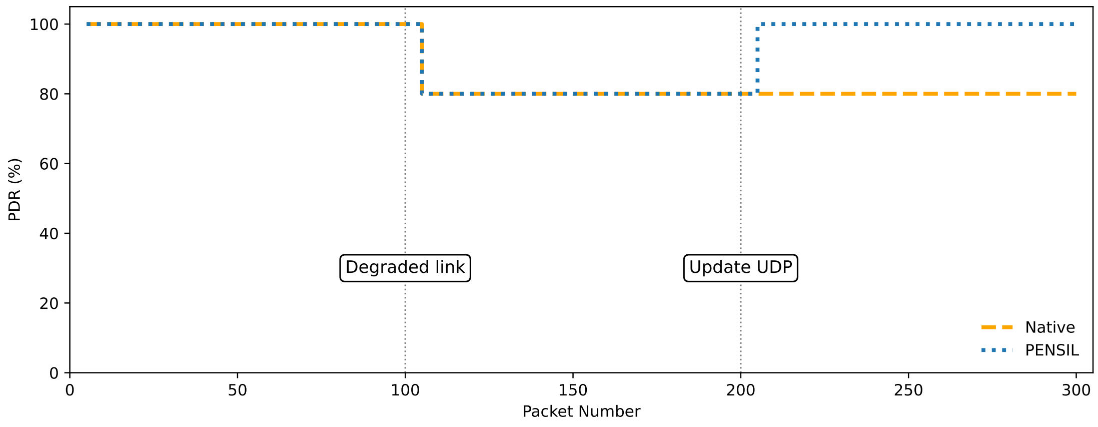


Figure 6: Packet Delivery Ratio

40 seconds, with staggered transmission intervals to avoid collisions. To emulate realistic operating conditions, the GoMacH duty-cycling protocol [33] is integrated to reduce power consumption. GoMacH employs a virtual TDMA scheme to coordinate transmissions within a superframe of 10 slots, each lasting 1 ms and capable of carrying a single packet. The evaluation focuses on three key aspects: (i) the impact of updates on network performance, (ii) update performance in terms of size, duration, security and recovery, and (iii) system-level such as resource utilization and energy efficiency.

4.1. Packet Delivery Ratio (PDR)

To demonstrate the benefits of the proposed UDP update, we design a controlled scenario in which link quality degradation renders the connection unreliable and causes packet loss. To mitigate this issue, PENSIL dynamically updates the running UDP module to a reliable variant by integrating sequence numbers, acknowledgments, and retransmissions upon timeout.

In this scenario, both the native and PENSIL implementations of UDP are evaluated. Two neighboring nodes from the topology, denoted as node A and node B, are selected. Node A transmits 300 UDP packets to node B in three successive stages of 100 packets each. In the first stage, both implementations use the baseline unreliable UDP achieveing a 100% Packet Delivery Ratio (PDR). In the second stage, node B is configured to drop every fifth packet, emulating link degradation and resulting in a 20% packet loss for both implementations. In the third stage, UDP is updated in PENSIL to the reliable variant, which retransmits lost packets at the next available transmission opportunity. Consequently, PENSIL restores the PDR to 100%, while the native implementation remains at 80%. The evolution of the PDR across the three stages is depicted in Fig. 6.

This experiment highlights the importance of fast and selective network stack reconfiguration to maintain the Quality of Service (QoS) required by diverse LLWN applications. Although the native implementation could also achieve reliability by updating UDP, doing so would require a full-image OTA update involving the transmission of a large binary image, longer update duration, and a reboot that interrupts communication. In contrast, PENSIL performs an in-place update of only the targeted UDP function, drastically reducing transfer size and update time while enabling immediate recovery without requiring system reboot, as further analyzed in the following sections.

4.2. Update Size and Time

Installing reliable UDP extensions in the native implementation requires modifying the operating system's UDP module and transmitting the full firmware image to all nodes. In contrast, PENSIL updates and transmits only the two VFs that implement the UDP protocol, namely UDP-Send and UDP-Recv, rather than the full firmware. This subsection presents a comparison of update size and duration between the two approaches.

Table 1 summarizes the update size transmitted to each node. The native full-image update produces a substantially larger payload (109.6 KB) compared to PENSIL (5.54 KB). This reduction directly translates into fewer packets

	Native	**PENSIL**	
Updated Files	Full Firmware	VF1 (UDP-Send)	VF2 (UDP-Recv)
Size (KB)	109.6	2.67	2.87
Total Size (KB)	109.6	5.54	

Table 1: Update Size Comparison

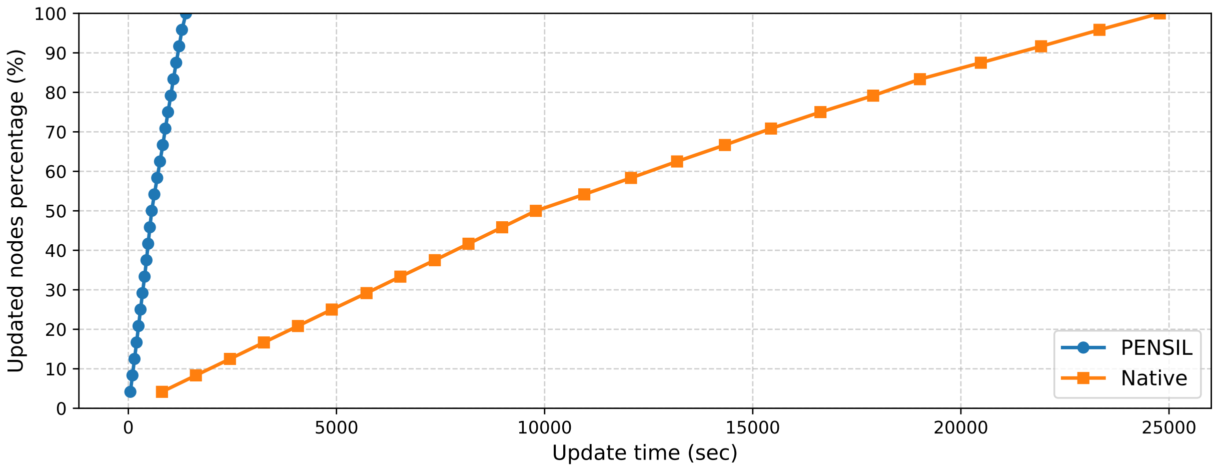


Figure 7: Update Time

transmitted from the orchestrator to the nodes. Assuming a 64-byte payload per packet, the native update requires approximately 1,713 packets, whereas the PENSIL-based update needs only 87 packets (42 + 45), representing a 94.9% reduction in update size.

The corresponding update duration scales critically with network size. As shown in Fig. 7, PENSIL reduces the total update time by 94.4% compared to the full-image approach. The native update experiences substantial delays due to multi-hop communication, duty-cycling effects, and concurrent application traffic, all of which increase collision probability and packet loss. Under the same conditions, PENSIL completes the update in less than 1,385 seconds thanks to its significantly smaller update size, whereas the native update exceeds 24,779 seconds. The measured update time also includes retransmissions to recover from packet loss at both the MAC layer (GoMacH) and the application layer (CoAP).

Overall, PENSIL demonstrates superior scalability and efficiency, particularly in dense, multi-hop, and lossy networks where minimizing update size is crucial for rapid convergence. Larger updates introduce higher packet overhead, increased loss probability, more retransmissions, and elevated network contention, all of which degrade network performance. While PENSIL substantially mitigates these limitations, further optimization, such as employing multicast dissemination for update blocks, could improve scalability and reduce update latency even further.

4.3. Post-deployment Convergence

One major limitation of full-image firmware updates is the loss of volatile network state caused by the mandatory reboot that follows installation. This reboot interrupts network connectivity, forcing nodes to re-establish communication and rebuild routing tables. In contrast, PENSIL allows selective updates of VFs without requiring a device reboot, preserving active connections and eliminating the need for time-consuming reconvergence.

The reconvergence time after the update is illustrated in Fig. 8. Here, convergence time refers to the duration required for the Border Router (BR) to re-register a node in its forwarding table and restore end-to-end communication. In the native implementation, this process takes over 60 seconds and increases with hop distance from the BR, as nodes sequentially exchange routing messages to rebuild the topology. In contrast, PENSIL maintains the existing network state during the update, allowing nodes to remain fully operational and eliminating reconvergence delays.

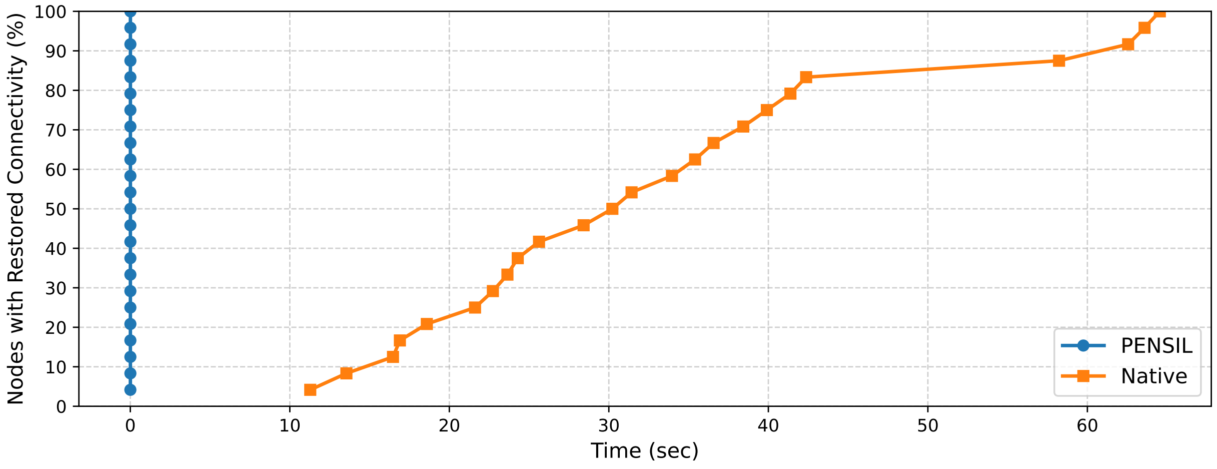


Figure 8: Convergence Time after Update

	Native	**PENSIL**	
		VF1 (UDP-Send)	VF2 (UDP-Recv)
Signature Verification (msec)	2649	2649	2649
Other Verifications & Validations (msec)	1040	1040	1040
Total (msec)	3689	7378	

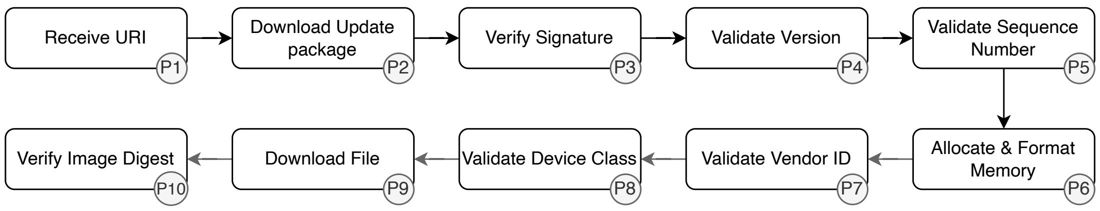
Table 2: Security Cost

Whereas native updates cause state loss and require a reconvergence process, PENSIL maintains connectivity and needs no post-update stabilization. This preserves ongoing application sessions from service interruptions and scales gracefully with network depth, demonstrating that fine-grained VF updates can achieve seamless operation while enabling rapid and reliable network reconfiguration.

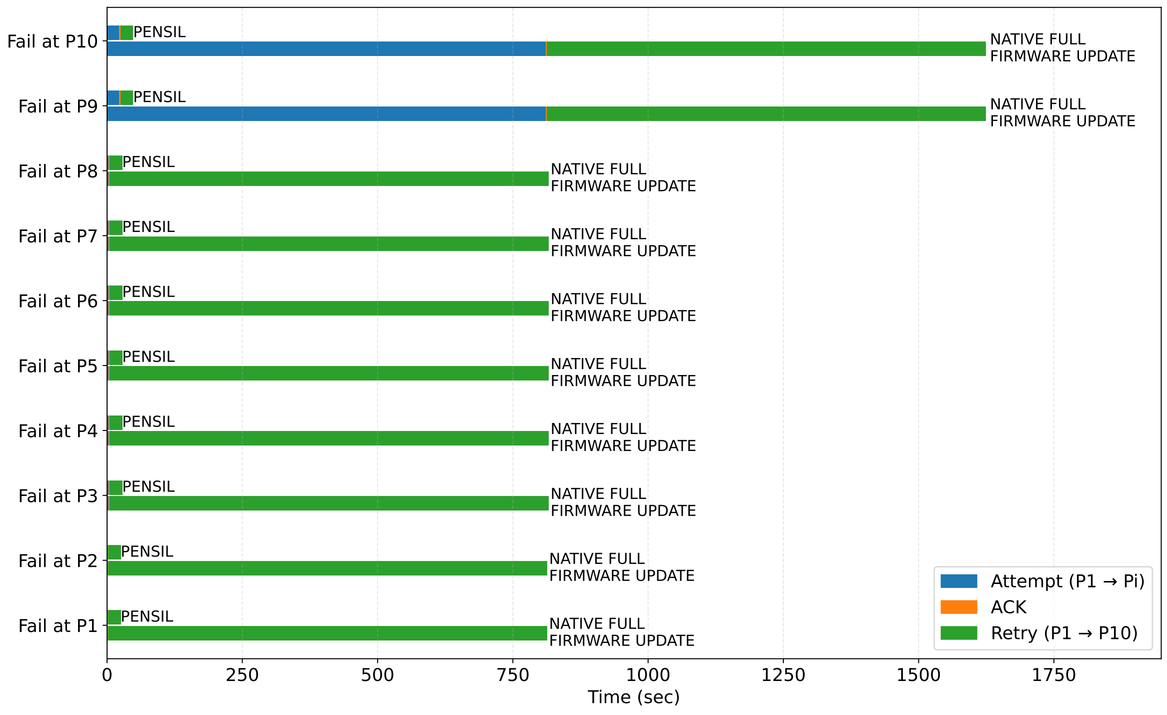
4.4. Security and Recovery Cost

PENSIL includes multiple security mechanisms to ensure the authenticity, integrity, and reliability of the update process. Fig. 9a illustrates the sequence of update phases together with the security checks, including signature and digest verification, sequence number and version validation, and vendor and device class identification. Among these, signature verification represents the majority of computational cost, as detailed in Table 2, accounting for more than 70% of the total verification and validation overhead. Each VF included in a PENSIL update undergoes independent verification and validation, unlike the native full-image approach, which performs these checks only once per update. Consequently, when two VFs are updated (UDP-Send and UDP-Recv), the total security verification time in PENSIL roughly doubles that of the native update. In general, the security cost scales linearly with the number of updated VFs, approximating N times the verification cost of a single image, where N denotes the number of updated VFs. Nevertheless, this additional cost remains marginal compared to the substantial reduction in update time achieved by PENSIL's compact update size.

PENSIL also includes recovery mechanisms to maintain robustness in case of update failures. If any phase of the update process fails, the device notifies the orchestrator, which triggers a recovery procedure to retransmit the corrected update. When same update sequence is applied, both PENSIL and native full-image updates incur similar additional recovery time if the failure occurs before the download phase (P9). However, as shown previously, the download time for full-image updates is much longer than for PENSIL, leading to a costly recovery when failures occur during or after the download phase. Fig. 9b compares the recovery delays in PENSIL and native full firmware image following an update failure of the UDP-Recv VF and the firmware image respectively to the same node, with delays measured relative to the phase of the update process in which the failure occurs. The update process in the



(a)



(b)

Figure 9: Update and Recovery: (a) Update Phases, (b) Recovery Cost on Every Phase

native implementation is consistently longer than in PENSIL due to the need to transmit the entire firmware. The highest recovery cost, in both implementations, arises when a failure occurs during or after the download phase when all retransmission retries fail in MAC layer and CoAP, as the node must restart the process after partially or fully downloading the update. Conversely, failures occurring before the download phase result in lower recovery times.

Despite these variations, the overall recovery latency in PENSIL remains considerably lower than that of the native full-image update, particularly for failures during the long download phase, confirming PENSIL's resilience and efficiency under adverse conditions.

4.5. Memory Footprint

Enabling PENSIL in the RIOT operating system requires adding specific modules. Fig. 10 compares the flash and RAM memory requirements of PENSIL with those of the native RIOT implementation for the UDP protocol.

Regarding flash memory, PENSIL introduces an additional footprint composed of two parts: (i) Architecture-related: includes the VF engine that enable virtual functions in the operating system, as well as the management plane components that interface between the orchestrator and the LLWN devices. These elements are required only once, regardless of the number of implemented protocols. (ii) Protocol-related: contains the UDP functionalities. In PENSIL, the UDP code size is nearly twice that of the native implementation due to the decomposition of UDP

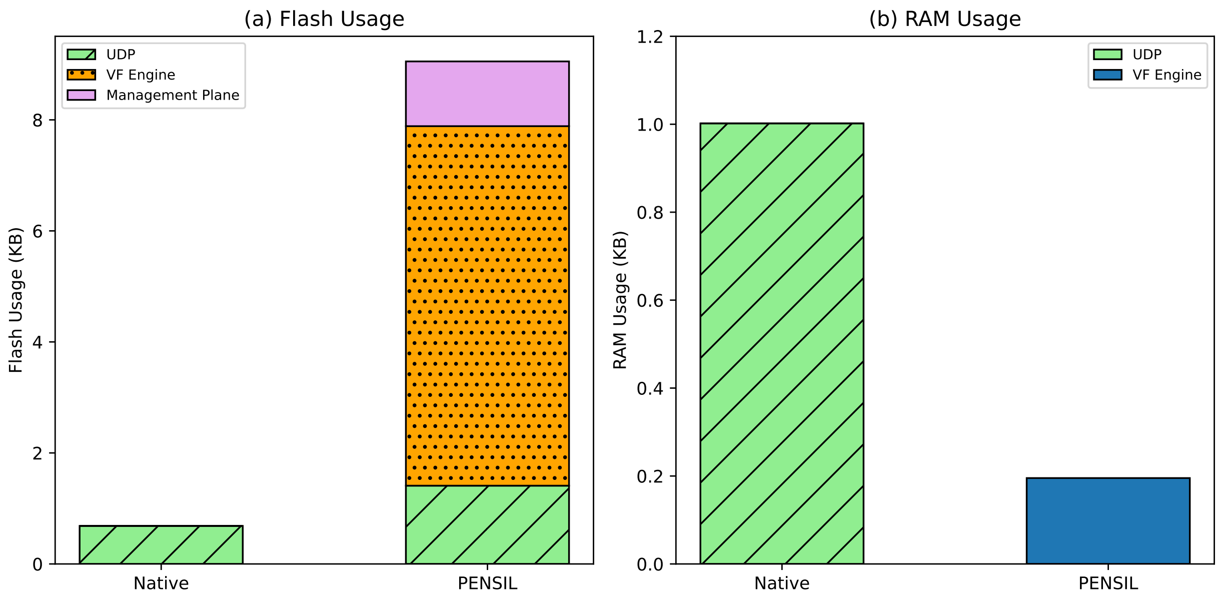


Figure 10: Memory Footprint (a) Flash Memory, (b) RAM Memory

functions into two VFs, which introduces minor code duplication, additional dependencies, and reduced compiler optimization. However, the overall increase in flash memory remains small and acceptable, totaling 8.57 KB, which corresponds to 7.82 % of the native RIOT firmware's total size.

Regarding RAM memory, the native RIOT implementation allocates a dedicated thread for each protocol, meaning that the UDP protocol alone requires a 1 KB RAM stack. In contrast, PENSIL invokes the UDP-Send or UDP-Recv VFs directly from the triggering thread, eliminating the need for an additional dedicated thread. Although it introduces an additional 200 Bytes of RAM for the VF engine, PENSIL reduces the overall RAM usage by 800 Bytes.

Another important aspect is the update slot, the memory space required to temporarily store new updates during download. In the native implementation, which relies on full-image OTA updates, a flash memory slot equal to the size of the new firmware image is required. Such large memory space may not always be available on constrained devices, representing a critical limitation. By contrast, PENSIL requires only a RAM slot at least as large as the largest VF to be updated. This slot can be configured according to the maximum possible size of a VF. In our experiment, updating UDP requires 110.6 KB of flash memory in the native implementation, whereas PENSIL needs only 3 KB of RAM, overcoming one of the major limitations of constrained devices.

The variation in memory usage introduced by PENSIL is negligible, demonstrating that it operates well within the memory constraints typical of LLWNs and is therefore suitable for deployment in such networks.

4.6. Power Consumption

Energy is one of the most constrained resources in LLWNs, making it crucial to ensure that PENSIL does not significantly increase power consumption. To compare the power consumption of PENSIL-based UDP with the native UDP implementation in RIOT, 1,000 packets were transmitted locally using the loopback interface on one node. This setup allows us to measure the combined transmission and reception power consumption on the same node while excluding the contribution of the radio transceiver. Measurements were performed using the INA226 hardware monitor available on the FIT IoT-LAB testbed. Power samples were collected every 588 μs, with an averaging count of 512. Fig. 11 presents the distribution of the collected power consumption samples. Results show that the power consumption of PENSIL is comparable to that of the native implementation, confirming that it introduces no significant overhead. This can be attributed to the lightweight design of PENSIL's VFs.

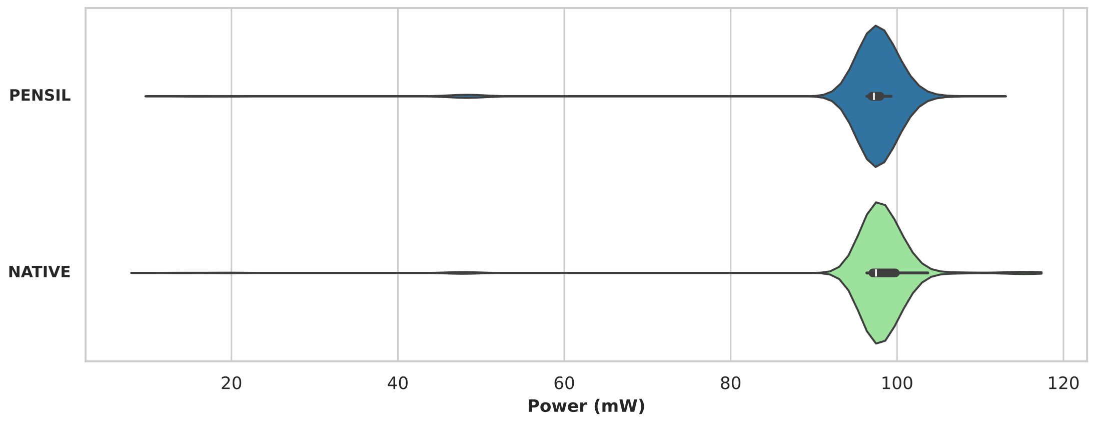


Figure 11: Power Consumption

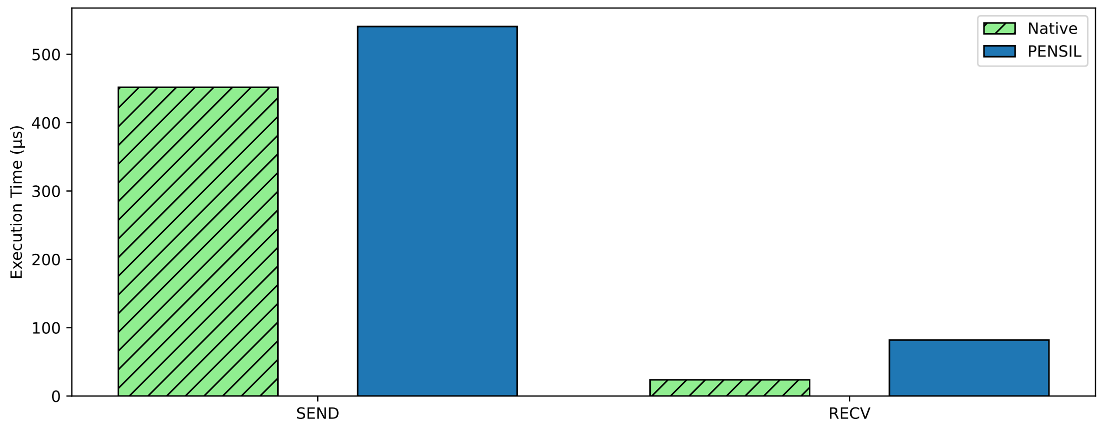


Figure 12: Execution Time

4.7. *Execution Time*

This section evaluates the processing time, a key factor in LLWNs when stringent timing constraints may apply, such as in TDMA-based MAC protocols. To this end, we measured in-device processing time required to send and receive a packet at the UDP layer—referred to as the *execution time*—and compared it with that of the native RIOT implementation. A total of 1,000 messages were sent between two nodes, node A to node B, measuring the send time on node A and the receive time on node B. Fig. 12 shows the average send and receive times over the 1,000 transmissions. First, the send time is longer than the receive time in both implementations due to implementation details, as the send process includes a *while* loop. Second, PENSIL introduces a small execution time overhead in both sending and receiving due to virtualization with VFs, amounting to 88.86 μs for sending and 85.29 μs for receiving. This additional delay mainly results from the interactions between the UDP VFs (UDP-Send and UDP-Recv) and the operating system through APIs. Nevertheless, the observed overhead—on the order of tens of microseconds—is negligible compared to typical 10 ms time slot duration of TDMA-based MAC protocols, ensuring that it does not compromise timing requirements. Future work will confirm this through practical evaluation with a TDMA-based MAC implementation.

5. Conclusion and Future Work

Low-Power and Lossy Wireless Networks (LLWNs) operate in highly dynamic environments, where fixed and rigid communication protocol stacks struggle to meet evolving requirements. This motivates the need for a fully programmable network stack, capable of fine-grained adaptation in both the control and data planes, and able to evolve seamlessly over time.

This paper introduces PENSIL, a network architecture that represents a shift from hardware-centric to software-driven networking, paving the way for fully programmable wireless networks. PENSIL leverages lightweight virtual functions that can be chained into protocols, updated over-the-air, and semantically managed by a central orchestrator. For example, a routing protocol could be dynamically extended to include a new metric for path selection, or a MAC protocol could be updated to implement a more efficient backoff strategy—without requiring device reboots or full firmware updates. The architecture supports distributed, centralized, and hybrid control-plane deployments, overcoming the limitations of existing approaches while providing unprecedented flexibility.

We validated PENSIL through a proof-of-concept implementation on real hardware. The standard unreliable UDP protocol was implemented and dynamically updated to add reliability. UDP was chosen for its simplicity, as the entire protocol can be represented using only two VFs (UDP-Send and UDP-Recv), making it ideal for a first demonstration of PENSIL's capabilities. Even in this simple scenario, PENSIL drastically reduced update size and duration while maintaining seamless, reboot-free operation. These results suggest that PENSIL can achieve similar or greater benefits with more complex protocols such as TCP or RPL, where fine-grained programmability could enhance adaptability, robustness, and efficiency.

We further quantified the impact of traditional full-image updates, which require device rebooting and lead to reconvergence delays exceeding 60 seconds along with service interruptions. PENSIL fully mitigates these limitations while keeping security verification overhead modest and recovery latency consistently lower than full-image updates. Resource usage remains practical: flash memory increases slightly due to the VF engine (below 10% of native firmware), RAM usage is reduced, power consumption is comparable, and processing overhead is negligible, confirming suitability for time-critical LLWN applications.

Future work will extend PENSIL validation to additional protocol layers and different control-plane deployments, particularly MAC protocols with strict timing requirements such as TDMA. We also plan to design and evaluate robust update-distribution algorithms to ensure reliable delivery across the network. Beyond this, exploring automated adaptation strategies responsive to dynamic network conditions will further establish PENSIL as a foundation for fully software-driven, programmable wireless networks.

Acknowledgments

This work was funded by ANR, Grant ANR-23-CE25-0008. For the purpose of Open Access, a CC-BY public copyright licence has been applied by the authors to the present document and will be applied to all subsequent versions up to the Author Accepted Manuscript arising from this submission.

References

[1] Y.-K. Chen, Challenges and opportunities of internet of things, in: 17th Asia and South Pacific design automation conference, IEEE, 2012, pp. 383–388.

[2] J. W. Hui, D. E. Culler, Extending ip to low-power, wireless personal area networks, IEEE Internet Computing 12 (4) (2008) 37–45.

[3] P. Ruckebusch, et al., Modelling the energy consumption for over-the-air software updates in LPWAN networks: SigFox, LoRa and IEEE 802.15.4g, Elsevier Internet of Things 3 (2018).

[4] K. Zandberg, et al., Secure Firmware Updates for Constrained IoT Devices Using Open Standards: A Reality Check, IEEE Access 7 (2019).

[5] H. Petersen, et al., Old wine in new skins? revisiting the software architecture for ip network stacks on constrained iot devices, in: Proceedings of the 2015 Workshop on IoT challenges in Mobile and Industrial Systems, 2015, pp. 31–35.

[6] E. Baccelli, et al., Scripting over-the-air: towards containers on low-end devices in the internet of things, in: 2018 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops), IEEE, 2018, pp. 504–507.

[7] E. Gavrin, et al., Ultra lightweight javascript engine for internet of things, in: Companion Proceedings of the 2015 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity, Association for Computing Machinery, New York, USA, 2015, p. 19–20.

[8] A. Dunkels, et al., Run-time dynamic linking for reprogramming wireless sensor networks, in: proc. of the 4th International Conference on Embedded Networked Sensor Systems (SenSys), 2006, pp. 15–28.

[9] W. Munawar, et al., Dynamic TinyOS: Modular and Transparent Incremental Code-Updates for Sensor Networks, in: proc. of the IEEE International Conference on Communications (ICC), 2010, pp. 1–6.
[10] A. Taherkordi, et al., Optimizing sensor network reprogramming via in situ reconfigurable components, ACM Transactions on Sensor Networks 9 (2) (2013) 1–33.
[11] B. Porter, et al., Type-safe updating for modular wsn software, in: proc. of the IEEE International Conference on Distributed Computing in Sensor Systems and Workshops (DCOSS), 2011, pp. 1–8.
[12] P. Ruckebusch, et al., Gitar: Generic extension for internet-of-things architectures enabling dynamic updates of network and application modules, Ad Hoc Networks 36 (2016) 127–151.
[13] V. Shymanskyy, Wasm3: A high performance webassembly interpreter written in c, `https://github.com/wasm3/wasm3` (2020).
[14] N. Brouwers, et al., Darjeeling, a feature-rich VM for the resource poor, in: proc. of the 7th ACM Conference on Embedded Networked Sensor Systems (SenSys), 2009, pp. 169–182.
[15] K. Zandberg, et al., Minimal virtual machines on IoT microcontrollers: The case of Berkeley Packet Filters with rBPF, in: proc. of the 9th IFIP International Conference on Performance Evaluation and Modeling in Wireless Networks (PEWMN), 2020, pp. 1–6.
[16] K. Zandberg, et al., 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, 2022, pp. 161–173.
[17] R. Alexander, et al., RPL: IPv6 Routing Protocol for Low-Power and Lossy Networks, RFC 6550 (2012). doi:10.17487/RFC6550.
URL `https://www.rfc-editor.org/info/rfc6550`
[18] Ieee standard for local and metropolitan area networks–part 15.4: Low-rate wireless personal area networks (lr-wpans), IEEE Std 802.15.4-2011 (Revision of IEEE Std 802.15.4-2006) (2011) 1–314.
[19] I. Rabet, et al., Sdmob: Sdn-based mobility management for iot networks, Journal of Sensor and Actuator Networks 11 (1) (2022) 8.
[20] M. Baddeley, et al., Evolving sdn for low-power iot networks, in: 2018 4th IEEE Conference on Network Softwarization and Workshops (NetSoft), IEEE, 2018, pp. 71–79.
[21] L. Galluccio, et al., Sdn-wise: Design, prototyping and experimentation of a stateful sdn solution for wireless sensor networks, in: 2015 IEEE conference on computer communications (INFOCOM), IEEE, 2015, pp. 513–521.
[22] R. Samadi, et al., Intelligent Energy-Aware Routing Protocol in Mobile IoT Networks Based on SDN, IEEE Transactions on Green Communications and Networking 7 (4) (2023).
[23] A. Ouhab, et al., Energy-efficient clustering and routing algorithm for large-scale sdn-based iot monitoring, in: ICC 2020-2020 IEEE International Conference on Communications (ICC), IEEE, 2020, pp. 1–6.
[24] F. Veisi, et al., Enabling Centralized Scheduling Using Software Defined Networking in Industrial Wireless Sensor Networks, IEEE Internet of Things Journal 10 (2023).
[25] M. Baddeley, et al., Isolating sdn control traffic with layer-2 slicing in 6tisch industrial iot networks, in: 2017 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN), IEEE, 2017, pp. 247–251.
[26] Y. Zhou, et al., An SDN-Enabled Proactive Defense Framework for DDoS Mitigation in IoT Networks, IEEE Transactions on Information Forensics and Security 16 (2021).
[27] U. Garba, et al., SDN-based detection and mitigation of DDoS attacks on smart homes, Computer Communications (2024).
[28] F. Veisi, et al., Link quality estimation in wireless software defined network with a reliable control plane, in: 2023 IEEE 9th International Conference on Network Softwarization (NetSoft), IEEE, 2023.
[29] C. Bormann, A. P. Castellani, Z. Shelby, Coap: An application protocol for billions of tiny internet nodes, IEEE Internet Computing 16 (2) (2012).
[30] M. Brendan, et al., A Firmware Update Architecture for Internet of Things, `https://datatracker.ietf.org/doc/html/rfc9019`, rFC 9019 (2021).
[31] C. Adjih, et al., Fit iot-lab: A large scale open experimental iot testbed, in: 2015 IEEE 2nd World Forum on Internet of Things (WF-IoT), IEEE, 2015, pp. 459–464.
[32] E. Baccelli, et al., Riot: An open source operating system for low-end embedded devices in the iot, IEEE Internet of Things Journal 5 (6) (2018) 4428–4440.
[33] S. Zhuo, Y.-Q. Song, Gomach: A traffic adaptive multi-channel mac protocol for iot, in: 2017 IEEE 42nd Conference on Local Computer Networks (LCN), IEEE, 2017, pp. 489–497.