Research Square

Preprints are preliminary reports that have not undergone peer review.
They should not be considered conclusive, used to inform clinical practice,
or referenced by the media as validated information.

µDCN:A High-Performance, ML-Orchestrated Data-CentricNetworking Architecture for Edge Environments

Abdulrahman Ben Mansor
University of Genoa
Alessandro Carrega
alessandro.carrega@unige.it
University of Genoa
Franco Davoli
University of Genoa
Ramin Rabbani
Consorzio Nazionale Interuniversitario per le Telecomunicazioni

Research Article

Keywords: Named Data Networking, edge computing, Machine Learning, eBPF, QUIC, Rust, TensorFlow

Posted Date: November 10th, 2026

DOI: https://doi.org/10.21203/rs.3.rs-11291637/v1

License: This work is licensed under a Creative Commons Attribution 4.0 International License.
Read Full License

Additional Declarations: No competing interests reported.

μDCN: A High-Performance, ML-Orchestrated Data-Centric Networking Architecture for Edge Environments

Abdulrahman Ben Mansor* Alessandro Carrega*† Franco Davoli*† Ramin Rabbani†

* Department of Electrical, Electronic and Telecommunications Engineering, and Naval Architecture (DITEN)
University of Genoa (UniGe), Italy {*name*}.{*surname*}@unige.it

† National Laboratory of Smart and Secure Networks (SN2) of the
National Inter-university Consortium for Telecommunications (CNIT), Genoa, Italy {*name*}.{*surname*}@cnit.it

***Abstract*—Edge computing environments require networking architectures that can support heterogeneous devices, dynamic workloads, and stringent latency constraints. Named Data Networking (NDN) provides content-centric communication, in-network caching, and data-centric security, but existing implementations often face a trade-off between performance, deployability, and adaptability. This paper presents μDCN [1], a cross-layer NDN architecture for edge environments that integrates an eBPF/XDP fast path, a Rust-based QUIC transport layer, and a TensorFlow Lite control plane for ML-driven MTU adaptation. The proposed design exposes fine-grained telemetry from the packet-processing path and uses it to dynamically tune packet granularity under changing network conditions. A prototype is implemented and evaluated on a heterogeneous edge testbed using realistic traffic profiles and controlled link impairments. Results show that μDCN improves throughput and latency over representative NDN implementations while reducing CPU and inference-engine resource consumption. The paper also analyzes deployment constraints, security implications, and current limitations of the architecture.**

***Index Terms*—Named Data Networking, edge computing, Machine Learning, eBPF, QUIC, Rust, TensorFlow**

I. Introduction

A. Problem Context: The Edge Networking Challenge

The rapid proliferation of Internet of Things (IoT) devices, Real-Time (RT) sensing applications, and distributed edge computing workloads has exposed fundamental shortcomings in classical host-centric networking architectures. Edge environments are characterised by three intertwined challenges that existing Transmission Control Protocol (TCP)/Internet Protocol (IP)-based stacks are ill-equipped to address simultaneously.

a) Heterogeneity: A single edge deployment may comprise embedded microcontrollers generating sub-kilobyte sensor readings, commodity servers handling multi-megabyte video frames, and mobile clients traversing multiple radio access technologies—all sharing the same forwarding infrastructure. Static Maximum Transmission Unit (MTU) and congestion-control configurations optimised for one device class unavoidably penalise others [2].

b) Dynamic workloads: Unlike stable datacenter traffic, edge traffic exhibits rapid, bursty transitions between low-rate telemetry and high-rate multimedia streaming, often within the same second. These transitions invalidate static forwarding and caching policies before they can be retuned by an operator, creating windows of excessive retransmission and latency spikes [3].

c) Latency sensitivity: A growing class of edge applications—including industrial closed-loop control, augmented reality, and autonomous vehicle coordination—impose end-to-end latency budgets as low as one millisecond. At these timescales, even a single unnecessary kernel interrupt or a suboptimal MTU causing packet fragmentation can violate the budget [4].

Named Data Networking (NDN) [3] offers a compelling paradigm shift by addressing content directly through hierarchical names, providing intrinsic multicast delivery, in-network caching, and data-centric security. Yet, as detailed in Section II, no existing NDN implementation satisfactorily reconciles high throughput, low latency, adaptive optimisation, and edge deployability within a single, deployable edge-oriented prototype.

B. μDCN: A Novel Cross-Layer Architecture

This paper introduces micro Data-Centric Networking (μDCN), an NDN architecture whose novelty lies in the cross-layer integration of kernel-level packet processing, memory-safe transport, and Machine Learning (ML)-driven runtime adaptation. A preliminary version of this architecture appeared in [1]. It selects a different combination of off-the-shelf components and it establishes a novel cross-layer feedback loop that did not previously exist: a kernel-bypass data plane exposes fine-grained per-packet telemetry that is consumed by a lightweight machine learning model, whose MTU predictions are enforced in RT across both the kernel and transport layers via a type-safe Remote Procedure Call (RPC) interface. To the best of our knowledge, no prior NDN implementation has jointly integrated (i) an extended Berkeley Packet Filter (eBPF)/eXpress Data Path (XDP)-based

fast path, (ii) a memory-safe Quick UDP Internet Connection (QUIC) transport layer, (iii) ML-driven MTU adaptation, and (iv) a control-plane bridge for RT parameter enforcement in an edge-oriented deployment. The novelty of μDCN therefore lies not in each individual mechanism, but in their coordinated cross-layer integration and experimental validation.

Concretely, the architectural novelty of μDCN lies in three design decisions that are individually motivated by the edge challenges above and collectively amplifying:

eBPF/XDP at the data plane
processes approximately $80\,\%$ of packets without any kernel–userspace context switch, eliminating the dominant source of latency jitter in existing NDN forwarders.

A Rust-based QUIC transport layer
provides the memory safety and connection multiplexing required for heterogeneous edge devices, while sharing a zero-copy ring buffer with the eBPF layer to avoid redundant data movement.

A TensorFlow Lite control plane
closes the feedback loop by continuously observing eight network metrics and issuing MTU updates with sub-two-millisecond inference latency, adapting the system to dynamic workload transitions faster than any operator or static heuristic could.

Experiments conducted on a heterogeneous five-node forwarding testbed plus a dedicated traffic-generator node show that μDCN reaches $4.7\,\mathrm{Gbps}$ for Small Data Objects (SDOs), $8.3\,\mathrm{Gbps}$ for Medium Data Objects (MDOs), and $6.1\,\mathrm{Gbps}$ for Large Data Objects (LDOs). These values correspond to $1.9\times$ – $3.9\times$ higher throughput than NDN Forwarding Daemon (NFD) under the evaluated workloads. μDCN also reduces end-to-end latency from $1.4\,\mathrm{ms}$ – $121.2\,\mathrm{ms}$ with NFD to $0.8\,\mathrm{ms}$ – $87.3\,\mathrm{ms}$, while the quantized inference engine lowers measured power consumption from $128\,\mathrm{mW}$ to $42\,\mathrm{mW}$ on Raspberry Pi 4B HardWare (HW). These results demonstrate the potential of μDCN as a scalable and energy-efficient solution for edge-oriented data-centric networking.

The main contributions of this paper are as follows:

- We design μDCN, a cross-layer NDN architecture for edge environments that integrates an eBPF/XDP [5], [6], [7], [8], [9] data plane, a Rust [10], [11], [12]/QUIC [13], [14], [15], [16], [17] transport plane, and an ML-driven control plane.
- We implement a deployable prototype using Aya, Quinn, TensorFlow Lite (TFLite), google Remote Procedure Call (gRPC), Docker, and Kubernetes (k8s), and describe the engineering trade-offs required to map NDN functions onto kernel and userspace components.
- We propose and evaluate a lightweight MTU prediction subsystem that uses RT network and transport metrics to proactively reduce fragmentation and retransmission overhead.
- We conduct an experimental evaluation on a heterogeneous edge testbed, comparing μDCN against NFD, Python-NDN, and Content-Centric Networking (CCN)-lite in terms of throughput, latency, Central Processing Unit (CPU) utilization, scalability, and edge resource usage.
- We analyze security implications and implementation limitations, including eBPF attack surface, NDN-specific threats, ML manipulation risks, and deployment constraints.

The central challenge is therefore not merely to accelerate NDN forwarding, but to do so while preserving deployability on shared edge nodes and adapting to changing path and workload conditions. HW-centric acceleration alone is insufficient for multi-tenant edge environments, while userspace-only forwarders suffer from excessive packet-processing overhead. Similarly, static transport and MTU configurations cannot accommodate the diversity of edge traffic. These observations motivate a cross-layer architecture in which packet processing, transport behavior, and control-plane optimization are jointly designed.

II. Related Work

The design and implementation of μDCN builds upon a rich body of research in data-centric networking, transport protocols, kernel bypass technologies, and ML for network optimization. In this section, we critically examine the most relevant efforts in each area, highlighting their contributions and limitations, and situate the innovations of μDCN within the broader context of the field.

A. NDN Implementations: A Critical Assessment

The NFD version 0.7.1 [18] remains the de-facto reference implementation. Written in C++, NFD prioritises correctness and extensibility over raw performance, resulting in CPU utilisation exceeding $63\,\%$ at $1\,\mathrm{Gbps}$ on commodity HW—more than three times the figure achieved by μDCN under identical conditions—and a throughput ceiling of approximately $1\,\mathrm{Gbps}$ on a $10\,\mathrm{Gbps}$ link. Its threading model was designed for well-provisioned servers, and its lack of kernel-bypass support means every packet traverses the full Linux network stack, adding microseconds of unnecessary latency at each forwarding step.

Python-NDN [19] trades performance for accessibility: the interpreter overhead limits sustained throughput to below $500\,\mathrm{Mbps}$ even on modern HW, and the Global Interpreter Lock (GIL) prevents true multi-core utilisation. While suitable for rapid prototyping and research scripts, Python-NDN is unsuitable as a production edge forwarder.

CCN-lite [20] achieves a small memory footprint ($28\,\mathrm{MB}$) through aggressive minimalism, but omits critical functionality required in dynamic edge deployments: there is no support for QUIC or any modern transport protocol, no adaptive optimisation mechanism, and no kernel-bypass capability. Its throughput ceiling is comparable to NFD despite requiring significantly less memory, suggesting that the bottleneck lies in its synchronous, single-threaded forwarding loop rather than in memory management.

NDN-Data Plane Development Kit (DPDK) [21] demonstrates that HW-level kernel bypass can push NDN forwarding past $100\,\mathrm{Gbps}$ on commodity Network Interface Cards (NICs). However, DPDK requires dedicating CPU cores and physical NICs exclusively to packet processing, making it fundamentally unsuitable for the heterogeneous, multi-tenant compute environments typical of edge deployments. Its portability is further constrained by a dependency on specific Intel NIC families.

Collectively, these implementations reveal a persistent trade-off triangle: performance, deployability, and adaptability cannot simultaneously be optimised by any single existing system. μDCN is designed to occupy a previously unreachable point in this space by combining the performance techniques of NDN-DPDK (kernel bypass) with the deployability of CCN-lite (containerised, HW-agnostic) and adding adaptive optimisation that no existing implementation provides.

B. QUIC and Modern Transport Protocols

QUIC [22], standardised by the Internet Engineering Task Force (IETF) as Request for Comments (RFC) 9000, offers multiplexed streams, zero-Round-Trip time (RTT) connection resumption, and integrated Transport Layer Security (TLS) 1.3 over User Datagram Protocol (UDP). Its advantages over TCP in variable-bandwidth and lossy environments—particularly relevant at the wireless edge—have been confirmed in multiple deployment studies [23]. Despite this, the integration of QUIC with NDN semantics remains immature: existing theoretical frameworks [24] and experimental prototypes [25] operate at the application layer, forgoing kernel-level optimisations and relying on static transport configurations. To the best of our knowledge, μDCN is the first NDN implementation to jointly couple QUIC with an eBPF/XDP fast path and an ML-driven parameter tuning loop in an edge-oriented prototype.

C. eBPF and Kernel-Bypass Technologies

The eXpress Data Path (XDP) [26] and the broader eBPF ecosystem have enabled programmable, high-speed packet processing within the Linux kernel without sacrificing the portability advantages of software-defined networking. Production use cases—including Meta's Katran load balancer and Cloudflare's Distributed DOS (DDoS) mitigation pipeline [27]—demonstrate sustained multi-Gbps throughput at microsecond per-packet latency. DPDK [28] achieves even higher raw throughput by bypassing the kernel entirely, but at the cost of exclusive HW dedication and reduced portability, as noted above. The application of eBPF/XDP specifically to content-centric forwarding remains an open research area prior to μDCN; no published work has demonstrated a fully functional NDN forwarder in the eBPF data plane with a userspace control interface.

D. ML for Network Optimisation: Beyond Congestion Control

The application of ML to network management spans routing [29], congestion control [30], and resource allocation [31]. Of these, congestion control has received the most attention: algorithms such as Bottleneck Bandwidth and Round-trip propagation time (BBR) use delay and loss signals to infer bottleneck bandwidth without explicit feedback, while reinforcement-learning-based approaches such as Pensieve [30] optimise bitrate adaptation policies using reward signals derived from playback quality. CUBIC, the default Linux congestion control algorithm, uses a cubic function of time since the last congestion event to set the congestion window.

μDCN's ML component addresses a structurally different problem: MTU selection. Whereas congestion control governs the sending rate, MTU selection governs the granularity of data units injected into the network. In an NDN context, a suboptimal MTU causes oversized Interest or Data packets to be fragmented in transit, multiplying the number of Pending Interest Table (PIT) lookups required to satisfy a single Interest, increasing reassembly overhead at the receiver, and triggering unnecessary retransmissions when a fragment is lost. Prior adaptive MTU work [4] relies on static heuristics or on simple probing in the style of Path MTU Discovery (PMTUD), which cannot anticipate workload transitions or distinguish between the MTU preferences of heterogeneous content types. μDCN's supervised regression model, by contrast, learns the mapping from eight real-time network features to the empirically optimal MTU, enabling proactive adaptation in under $2\,\mathrm{ms}$—an order of magnitude faster than PMTUD-style probing. This makes it complementary to, rather than a replacement for, congestion control: BBR or CUBIC continues to govern the sending rate while μDCN's ML model governs packet granularity.

The emergence of TFLite [32] has made it feasible to deploy such models on edge devices with less than $3\,\mathrm{MB}$ of Random Access Memory (RAM) and under $2\,\mathrm{ms}$ of inference latency after INT8 quantisation (INT8-Q), removing the last practical barrier to on-device ML-driven adaptation.

E. Comparison with μDCN

Unlike existing solutions, μDCN uniquely combines eBPF/XDP for kernel-level packet handling, Rust with QUIC for efficient transport, and ML-driven network adaptation. Table I summarises the key differences between μDCN and other NDN implementations. Performance figures for NFD, Python-NDN, CCN-lite, and NDN-DPDK are drawn from their respective primary publications [18], [19], [20], [21]; μDCN figures are from this paper's evaluation (Section V).

Although several prior systems address individual aspects of the problem, they leave important gaps for edge-oriented NDN. NDN-DPDK demonstrates that kernel-bypass forwarding can achieve very high throughput, but it targets dedicated high-end packet-processing platforms rather than shared edge nodes. Conversely, lightweight NDN implementations reduce resource consumption but do not provide adaptive transport or ML-based control. Existing QUIC-over-NDN studies primarily explore transport mapping and do not integrate kernel-level packet processing. Finally, ML-based networking work has mostly focused on routing, congestion control, or video

adaptation, while packet-size selection for content-centric edge traffic remains comparatively underexplored.

The performance range reported for μDCN corresponds to the minimum and maximum throughput observed across the SDO, MDO, and LDO workloads in Section V. NDN-DPDK is reported separately because it targets dedicated DPDK-based forwarding environments and is not directly comparable to the shared edge-node deployment model considered in this paper.

III. System Architecture

The μDCN architecture is founded on three core principles: maximizing performance through kernel-level optimizations, ensuring safety and reliability via memory-safe language implementation, and achieving adaptability through ML-driven orchestration. This section provides a comprehensive overview of the system's layered design and the interactions among its primary components.

A. Architectural Overview

As depicted in Figure 1, μDCN adopts a layered architecture composed of three distinct planes: the data plane, the transport plane, and the control plane. Each plane is carefully engineered to fulfill a specific role within the system, collectively enabling high-performance, adaptive, and secure networking for edge environments.

The data plane, implemented using Rust-based eBPF/XDP, is responsible for direct packet processing within the Linux kernel. This design choice enables near-zero-copy packet handling, which significantly reduces processing overhead and latency. By parsing, forwarding, and performing basic caching operations at the kernel level, the data plane delivers the efficiency necessary for demanding edge workloads.

The transport plane is also implemented in Rust, leveraging the language's memory safety guarantees and performance characteristics. Integrating the QUIC protocol, the transport plane manages higher-level NDN functionality that exceeds eBPF complexity constraints, including content store management, fragmentation and reassembly, and security operations. This layer provides both reliability and speed, ensuring robust data delivery across heterogeneous network conditions.

The control plane, developed in Python and utilizing TFLite, orchestrates network monitoring, metric collection, and the adaptive tuning of network parameters. By continuously analyzing real-time metrics, the control plane applies ML models to optimize key parameters such as the MTU, adapting the system's behavior to prevailing network conditions.

Communication between these planes is facilitated by two primary interfaces: a high-performance shared memory interface connecting the data and transport planes, and a gRPC-based interface linking the transport and control planes. This modular approach ensures that each plane can evolve independently while maintaining efficient and reliable interoperation.

The processing path in μDCN proceeds as follows. Incoming packets are first intercepted at the XDP hook. Packets matching simple fast-path rules are handled in eBPF, while packets requiring complex NDN semantics are transferred to the Rust transport plane through a ring buffer. The transport plane performs name-based lookup, PIT/Forwarding Information Base (FIB) processing, content-store management, fragmentation, reassembly, and QUIC stream handling. Runtime metrics are exported through the RPC interface to the Python control plane. The control plane periodically executes the TFLite predictor and, when confidence exceeds the configured threshold, sends MTU update commands back to the transport and data planes.

B. Data Plane

Within μDCN, the data plane is engineered to achieve high-performance packet processing by leveraging eBPF/XDP technology in the Linux kernel. By attaching eBPF programs to XDP hooks, the system intercepts and processes packets at the earliest possible stage in the network stack, well before they reach conventional processing layers. This design enables near-zero-copy handling, which dramatically reduces memory and CPU overhead. Early filtering and modification of packets at the driver level further enhance efficiency and security, while hardware offloading support on compatible NICs allows for even greater performance gains.

The data plane is composed of two tightly integrated components. The first is the NDN parser, an eBPF program optimized for rapid parsing of NDN packet headers. It classifies packets—such as Interest, Data, and Nack types—and extracts essential fields, including content names and selectors, enabling swift routing decisions. The second component is the fast path processor, which is responsible for handling the majority of packets through simple content store lookups, static forwarding, and basic security validation. This fast path is designed to process the common-case traffic with minimal latency. For packets that require more complex processing, which typically constitute about $20\,\%$ of the workload, the data plane efficiently transfers them to the transport plane via a shared memory ring buffer, thereby minimizing data copying and context switching overhead.

C. Transport Plane

The transport plane serves as the central hub for the core operations of μDCN, and is implemented in Rust to capitalize on the language's memory safety and high-performance characteristics. This layer is responsible for comprehensive processing of Interest and Data packets, including advanced name matching, dynamic forwarding strategy selection, and full-featured content store management. The integration of the QUIC protocol within the transport plane enables efficient, secure, and reliable data transmission, providing built-in congestion control and stream multiplexing that are particularly beneficial in dynamic edge environments.

A key function of the transport plane is its ability to intelligently fragment and reassemble oversized NDN objects. This process is guided by both content type and the current MTU, as predicted by the ML subsystem, ensuring that data

TABLE I
COMPARISON OF NDN IMPLEMENTATIONS. PERFORMANCE AND RESOURCE FIGURES FOR NFD [18], PYTHON-NDN [19], CCN-LITE [20], AND NDN-DPDK [21] ARE SOURCED FROM THEIR RESPECTIVE PRIMARY PUBLICATIONS; μDCN FIGURES ARE FROM SECTION V (LEGEND - YES: ✓, NO: ✗, LIMITED: ◐).

FEATURE	NFD	PY-NDN	CCN-LITE	NDN-DPDK	μDCN
Language	C++	Python	C	C	Rust/Python
Kernel Opt.	✗	✗	◐	DPDK	eBPF/XDP
Transport	TCP	TCP	TCP	UDP	QUIC
ML Adapt.	✗	✗	✗	✗	✓
Edge Focus	✗	◐	✓	◐	✓
Perf. (Gbps)	$\sim$1	< 0.5	≈ 1	> 100	4.7 to 8.3
Memory (MB)	138	175	28	N/A	42
CPU @ 1 Gbps	63 %	82 %	47 %	N/A	18 %

is transmitted efficiently and without unnecessary retransmissions. Security is also a primary concern at this layer, with the transport plane implementing name-based access control, cryptographic signature verification, and certificate management to uphold the integrity and confidentiality of data flows. Additionally, the transport plane continuously collects performance metrics, which are relayed to the control plane for monitoring and further optimization. Communication with the control plane is facilitated via a robust gRPC service interface, supporting real-time monitoring, configuration, and dynamic adaptation of transport parameters.

D. Control Plane

The control plane in μDCN is developed in Python, chosen for its extensive libraries and frameworks that support data analysis, ML, and network management. This plane is tasked with orchestrating the adaptive behavior of the entire system. It continuously collects a wide array of metrics, both from the underlying system—such as CPU usage, memory consumption, and network interface statistics—and from the transport plane, including packet rates, cache hit ratios, and latency measurements. These metrics are processed by embedded TFLite models, which predict optimal network parameters, with a particular focus on determining the most effective MTU size for current conditions.

The control plane's adaptation controller applies these ML-driven predictions to dynamically reconfigure both the transport and data planes, adjusting parameters such as MTU, caching policies, and forwarding strategies in RT. This ensures that the system remains responsive to changing network conditions and workload characteristics. Furthermore, a management Application Programming Interface (API) is provided, enabling external systems to monitor and configure the μDCN node as needed. Communication with the transport plane is achieved through a gRPC interface, which supports efficient, type-safe, and reliable remote procedure calls, including the streaming of telemetry data for continuous monitoring and rapid adaptation.

E. Cross-Cutting Concerns

A number of architectural concerns in μDCN transcend the boundaries of individual planes, ensuring the system's robustness, flexibility, and operational visibility. Security is integrated at every layer: the data plane performs fundamental packet validation, the transport plane enforces cryptographic security and access control, and the control plane applies confidence thresholds and boundary checks to prevent unsafe parameter updates. This multi-layered security approach safeguards both data integrity and system resilience.

Deployment flexibility is achieved through comprehensive containerization using Docker, with orchestration managed by Kubernetes. This enables μDCN to be deployed seamlessly across heterogeneous edge environments, supporting rapid scaling and simplified management. Monitoring is another cross-cutting concern, addressed by integrating Prometheus-based metrics collection throughout the architecture. This unified monitoring framework provides a holistic view of system performance and health, facilitating proactive management and rapid troubleshooting.

F. MTU Prediction Subsystem: Design, Model Architecture, and Power Analysis

1) Why ML for MTU Selection?: The MTU prediction subsystem directly addresses the edge heterogeneity and dynamic workload challenges identified in Section I. Standard PMTUD probes the path reactively after a fragmentation event has already degraded performance; it cannot distinguish between the MTU preferences of a 150 B[1] IoT reading and a 5 MB FirmWare (FW) image, and it introduces probe-response round-trips that are prohibitively slow at the millisecond timescales of industrial control loops. μDCN's ML approach predicts the optimal MTU *before* fragmentation occurs, by learning the relationship between observable RT metrics and the empirically determined optimal MTU for each network state.

[1]Throughout this paper, B denotes bytes (e.g., kB, MB), while data rates are expressed in bits per second (e.g., Gbps).

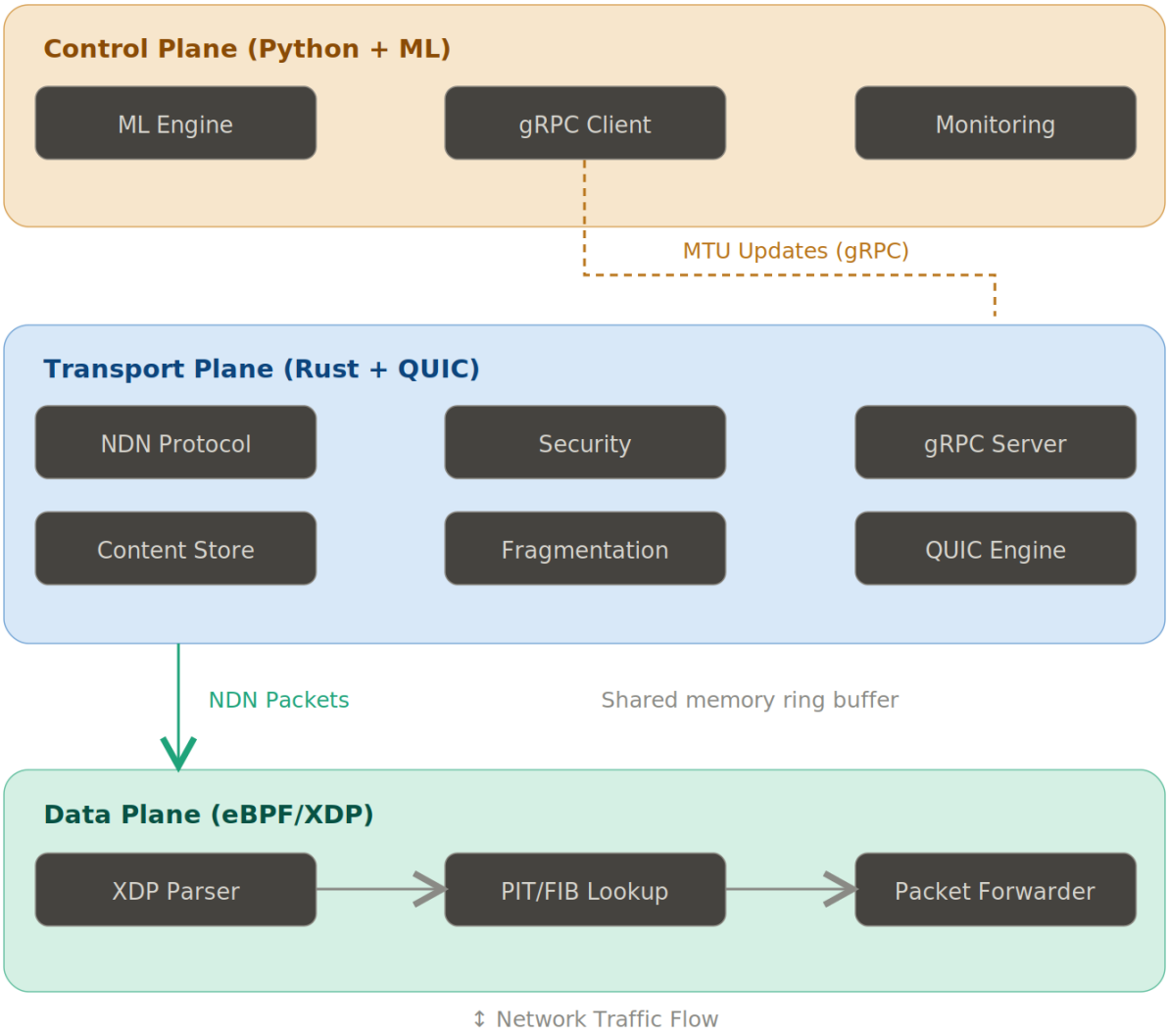


Figure 1. μDCN system architecture overview.

2) Model Architecture and Input Features: The prediction model is a three-layer fully-connected Multi-Layer Perceptron (MLP) with 64, 32 and 16 neurons per hidden layer and Rectified Linear Unit (ReLU) activations. Two output heads are attached to the final hidden layer: (i) a single regression neuron producing the raw MTU value in bytes; (ii) a single sigmoid neuron producing a prediction confidence score in $[0, 1]$. The dual-head design allows the control plane to apply predictions conditionally: an MTU update is issued only when the confidence score exceeds the configurable threshold (default $\tau = 0.8$), suppressing noisy updates during transient conditions.

The model receives eight input features, normalised using a pre-fitted `StandardScaler` whose parameters are serialised alongside the `.tflite` model file:

1) Packet loss rate (%)
2) Throughput (Mbps)
3) Round-trip latency (ms)
4) Jitter (ms)
5) Interface bandwidth utilisation (%)
6) Content-store cache-hit ratio
7) Average NDN object size (kB)
8) TRex-reported retransmission count

These features are collected every five seconds by the `NetworkMonitor` module and forwarded to the Python gRPC client, which preprocesses them before invoking the TFLite interpreter. Table II summarises the full model specification.

3) Training Procedure and Ground Truth: The model is trained offline using approximately 50 000 labelled samples collected from controlled emulation runs conducted on the same five-node testbed used for the main evaluation (Section V). For each 30 s snapshot, the optimal MTU is established as the value that maximises aggregate throughput during a binary search from 576 B to 9000 B with a step size of 8 B—the resolution of Ethernet jumbo-frame negotiation.

TABLE II
TFLITE MTU PREDICTION MODEL SPECIFICATION.

PARAMETER	VALUE / DETAIL
Architecture	3-layer MLP ($64 \rightarrow 32 \rightarrow 16$, ReLU)
Output heads	Regression (MTU) + Confidence (sigmoid)
Input features	8 (see enumeration above)
Training samples	$\approx$50 000 labelled snapshots
Optimiser / loss	Adam ($lr = 10^{-3}$), MSE + BCE
Train / val / test split	80, 10 and 10
Deployment format	INT8 post-training quantisation (.tflite)
Inference latency (INT8)	1.8 ms
Inference latency (FP32)	5.2 ms

This binary search is executed automatically by a purpose-built calibration script that drives TRex at the current workload profile and records peak goodput per MTU candidate. The resulting (feature vector, optimal MTU) pairs constitute the training dataset. An 80/10/10 train/validation/test split is used, with early stopping on validation MSE to prevent overfitting. After training in full `float32` precision, the model is post-training quantised to INT8 using the TFLite converter [32] with a representative calibration dataset drawn from the training set.

4) Power Reduction: Mechanistic Analysis: The $67\,\%$ power reduction ($128\,\mathrm{mW}$ to $42\,\mathrm{mW}$) reported in Table XII is a direct consequence of INT8 post-training quantisation and is decomposed mechanistically in Table III. Power was measured on a Raspberry Pi 4B using an Adafriut INA219 current-sensor breakout board, logging at $10\,\mathrm{Hz}$ across ten consecutive inference runs per configuration. Three physical mechanisms contribute to the reduction.

a) Dynamic power from Dynamic Random-Access Memory (DRAM) reads ($-75\,\%$): The INT8 model occupies $212\,\mathrm{kB}$ versus $842\,\mathrm{kB}$ for the `float32` original, reducing the number of 64-B cache-line fetches required per inference from approximately $13\,500$ to $3\,400$. Dynamic power is proportional to the number of memory transactions, so the $4\times$ reduction in model footprint yields a near-$4\times$ reduction in DRAM power.

b) CPU computation power ($-66\,\%$): The Advanced RISC Machine (ARM) Cortex-A72 cores on the Raspberry Pi 4B include Single Instruction, Multiple Data (SIMD) Neon units that can pack four INT8 multiply-accumulate operations into a single instruction versus one `float32` operation, reducing the active clock cycles by a factor of approximately three and enabling the CPU to reach a power-gated idle state $1.5\,\mathrm{ms}$ earlier per inference.

c) Static leakage ($-48\,\%$): The shorter active execution window ($1.8\,\mathrm{ms}$ vs. $5.2\,\mathrm{ms}$) reduces the time during which static leakage current flows through active transistors. The combined effect of these three mechanisms, measured empirically, amounts to $86\,\mathrm{mW}$ of savings per inference cycle.

IV. IMPLEMENTATION DETAILS

This section provides an in-depth account of the implementation of μDCN's core components, with particular emphasis on the eBPF/XDP data plane, the Rust-based transport layer, the gRPC communication bridge, and the Python-driven machine learning control plane. The following subsections detail the design rationale, technical challenges, and engineering solutions that underpin each module.

A. eBPF/XDP Data Plane

At the foundation of μDCN's high-performance architecture lies the data plane, realized through the use of eBPF programs attached to XDP hooks within the Linux kernel. This approach allows for the interception and processing of network packets at the earliest possible stage, well before they traverse the conventional networking stack. The implementation utilizes Aya, which is a pure Rust framework for writing, loading, and interacting with eBPF programs, ensuring that packet processing incurs minimal overhead and maximizes throughput.

1) NDN Packet Parser: Central to the data plane is the NDN packet parser, which is implemented as an eBPF program designed for both efficiency and safety. As shown in Algorithm 1, the parser operates by first validating that sufficient data is present for header analysis, then sequentially parsing Ethernet, IP, and UDP headers. It specifically identifies NDN packets by inspecting the destination port, and then extracts the NDN Type (Tag) Length Value (TLV) header to determine packet type. For Interest and Data packets, fast path processing routines are invoked, while more complex or unrecognized packets are forwarded to the userspace transport layer for further handling. This staged approach ensures that the majority of packets are processed with minimal latency, while maintaining the flexibility to support advanced protocol features.

To achieve high performance, the packet parser incorporates a series of critical optimizations. First, it implements early filtering, rapidly passing non-NDN packets to the standard network stack to avoid unnecessary processing. The parser is engineered for zero-copy operation, working directly on packet data within the kernel's memory space, which eliminates the overhead associated with data duplication. All memory accesses are rigorously bounded and verified, a requirement for kernel safety and stability. Furthermore, the parser is structured to handle common-case operations entirely within eBPF, maximizing the benefits of the fast path and minimizing the need for context switches to userspace.

2) Shared Memory Interface: Efficient communication between the eBPF data plane and the transport plane is realized through a shared memory ring buffer, as facilitated by the Berkeley Packet Filter (BPF) map facility. This mechanism, enables the rapid transfer of packets and associated metadata from the kernel to userspace with minimal copying. The ring buffer is configured to allocate sufficient space for both metadata and packet payload, and packets that require further processing are reserved, copied, and submitted to userspace via this interface. This design minimizes context switching and

TABLE III
POWER REDUCTION DECOMPOSITION (RASPBERRY PI 4B, ADAFRIUT INA219 MEASUREMENT).

COMPONENT	FP32	INT8	REDUCTION	MECHANISM
Dynamic power (DRAM reads)	$\approx 68\,\mathrm{mW}$	$\approx 17\,\mathrm{mW}$	75 %	4× fewer cache-line fetches
CPU active power	$\approx 35\,\mathrm{mW}$	$\approx 12\,\mathrm{mW}$	66 %	3× fewer MAC operations
Static / leakage	$\approx 25\,\mathrm{mW}$	$\approx 13\,\mathrm{mW}$	48 %	Shorter active window
TOTAL (MEASURED)	$128\,\mathrm{mW}$	$42\,\mathrm{mW}$	67 %	**ADAFRIUT INA219 SENSOR**

Algorithm 1 eBPF NDN Packet Parser.

```
/// Parse the packet to determine if it's an NDN
/// packet and identify its type
pub fn parse_ndn_packet(packet: &Packet)
    -> Result<ndn::PacketType, ()> {
    // \acr{ndn} packets start with a tlv type byte
    let data_ptr = packet.data_ptr()?;
    let data_len = packet.data_len();

    if data_len < 1 {
        return Err(());
    }

    // Read the first byte, which is the tlv type
    let tlv_type = unsafe { *data_ptr };

    match tlv_type {
        ndn::tlv_INTEREST =>
            Ok(ndn::PacketType::Interest),
        ndn::tlv_DATA =>
            Ok(ndn::PacketType::Data),
        _ => Err(()),
    }
}

/// Extract a hash of the name from an NDN
/// packet
pub fn extract_name_hash(packet: &Packet)
    -> Option<u32> {
    let data_ptr = match packet.data_ptr() {
        Ok(ptr) => ptr,
        Err(_) => return None,
    };

    let data_len = packet.data_len();

    if data_len < 10 {
        return None;
    }

    // Navigate tlv structure to find name component
    let name_hash_ptr = unsafe {
        data_ptr.add(4) as *const u32 };
    let name_hash = unsafe { *name_hash_ptr };

    Some(name_hash)
}
```

memory copying, achieving near-zero-copy performance and enabling the transport layer to access packets with minimal latency.

B. Rust Transport Layer

The transport layer is implemented in Rust, leveraging its memory safety guarantees and performance characteristics. Key components include:

1) NDN Protocol Implementation: The core Named Data Networking (NDN) protocol is implemented in Rust, with a strong emphasis on both performance and type safety. By leveraging Rust's advanced type system and memory safety guarantees, the implementation ensures robust and reliable operation, even under high concurrency. As illustrated in Algorithm 2, the protocol defines structured representations for Interest and Data packets, encapsulating all necessary fields for efficient packet processing and validation. This approach not only enhances code maintainability and correctness but also enables fine-grained control over protocol logic, supporting complex features such as name matching, freshness constraints, and cryptographic verification.

Algorithm 2 Rust NDN Protocol Implementation.

```
pub struct Interest {
    name: Name,
    can_be_prefix: bool,
    must_be_fresh: bool,
    nonce: u32,
    lifetime: Duration,
    // Additional fields...
}

pub struct Data {
    name: Name,
    content: Bytes,
    freshness_period: Option<Duration>,
    // Security fields...
}

impl Interest {
    pub fn matches(&self, name: &Name) -> bool {
        if !self.can_be_prefix {
            return self.name == *name;
        }
        name.has_prefix(&self.name)
    }

    pub fn to_wire(&self) -> Bytes {
        // Efficient \acr{tlv} encoding
        // ...
    }

    pub fn from_wire(data: &[u8])
        -> Result<Self, Error> {
        // Efficient \acr{tlv} decoding
        // ...
    }
}
```

2) QUIC Integration: μDCN integrates the QUIC protocol through the Quinn Rust library, mapping NDN semantics to QUIC streams:

The integration of the QUIC protocol within the NDN transport layer is achieved through a dedicated engine, as demonstrated in Algorithm 3. The QuicEngine structure man-

ages endpoints and maintains concurrent connections to multiple peers, utilizing Rust's DashMap for thread-safe access. This design allows for efficient, secure, and reliable data transmission, benefiting from QUIC's inherent support for congestion control and stream multiplexing. The asynchronous architecture further enables high throughput and low latency, which are critical for edge networking scenarios.

Algorithm 3 QUIC Integration.

```rust
pub struct QuicEngine {
    endpoint: Endpoint,
    connections: DashMap<PeerId, Connection>,
    // Additional fields...
}

impl QuicEngine {
    pub async fn send_interest(
        &self,
        interest: Interest,
        peer: PeerId
    ) -> Result<Data, Error> {
        // Get or establish QUIC connection
        let conn = self.get_or_connect(peer).await?;

        // Open a new bidirectional stream
        let (mut send, mut recv) =
            conn.open_bi().await?;

        // Send the interest packet
        let wire_data = interest.to_wire();
        send.write_all(&wire_data).await?;
        send.finish().await?;

        // Receive the data packet
        let mut response = Vec::new();
        recv.read_to_end(&mut response).await?;

        // Parse and return the data
        Data::from_wire(&response)
    }
}
```

The QUIC integration provides several benefits:

- **Stream Multiplexing:** Multiple NDN operations can share a single connection.
- **Built-in Congestion Control:** QUIC's congestion control adapts to network conditions.
- **Transport Security:** All communications are encrypted and authenticated.
- **Connection Migration:** Sessions can survive IP address changes, crucial for mobile edge devices.

3) Content Store: A critical component of the transport plane is the content store, which is responsible for caching Data packets to improve retrieval latency and reduce redundant network traffic. The implementation leverages DashMap to provide thread-safe, concurrent access to cached entries, while a Least Recently Used (LRU) cache enforces an efficient eviction policy. As shown in Algorithm 4, the ContentStore structure combines these mechanisms, maintaining a balance between fast lookups and memory efficiency. The design supports high concurrency and rapid cache updates, which are essential for sustaining performance in dynamic network environments.

To facilitate structured and efficient communication between the Rust-based transport layer and the Python control plane, μDCN employs a gRPC-based communication bridge. This

Algorithm 4 Content Store Implementation.

```rust
pub struct ContentStore {
    cache: DashMap<Name, (Data, Instant)>,
    lru: Mutex<LruCache<Name, ()>>,
    capacity: usize,
    metrics: Arc<Metrics>,
}

impl ContentStore {
    pub fn insert(&self, data: Data) {
        let name = data.name().clone();

        // Update LRU tracking
        let mut lru = self.lru.lock().unwrap();
        if lru.len() >= self.capacity &&
            !lru.contains(&name) {
            if let Some((evicted, _)) =
                lru.pop_lru() {
                // Evict from main cache
                self.cache.remove(&evicted);
                self.metrics.record_eviction();
            }
        }
        lru.put(name.clone(), ());

        // Insert into main cache
        self.cache.insert(
            name,
            (data, Instant::now())
        );
        self.metrics.record_insertion();
    }

    pub fn get(&self, name: &Name) -> Option<Data> {
        if let Some(entry) = self.cache.get(name) {
            let (data, timestamp) = &*entry;

            // Update LRU status
            let mut lru = self.lru.lock().unwrap();
            lru.put(name.clone(), ());

            // Record metrics
            self.metrics.record_hit();

            // Return cloned data
            Some(data.clone())
        } else {
            // Record metrics
            self.metrics.record_miss();
            None
        }
    }
}
```

interface is defined using Protocol Buffers, as depicted in Algorithm 5, which specify the available service methods and data structures. The use of gRPC ensures type safety, language interoperability, and efficient serialization, supporting both synchronous and streaming remote procedure calls. This architecture enables seamless integration between system components, allowing the control plane to monitor, configure, and adapt the transport layer in real time.

The gRPC communication bridge provides a robust and efficient interface for communication between the transport layer and control plane, enabling seamless integration and RT adaptation. By leveraging Protocol Buffers and gRPC, the implementation ensures type safety, language interoperability, and efficient serialization, supporting both synchronous and streaming remote procedure calls.

4) Rust gRPC Server: The Rust-based transport layer exposes its services to the control plane via a gRPC server, implemented using the Tonic framework. This framework

Algorithm 5 Protocol Buffer Service Definition.

```
service UdcnControl {
    rpc UpdateMtu (MtuRequest)
        returns (MtuResponse);
    rpc GetMetrics (MetricsRequest)
        returns (MetricsResponse);

// Streaming metrics for \acr{rt} monitoring
rpc StreamMetrics(StreamMetricsRequest)
    returns (stream MetricsData);

// Additional service methods...
}

message MtuRequest {
    uint32 mtu = 1;
    string interface_name = 2;
    float confidence = 3;
    map<string, string> metadata = 4;
}

message MtuResponse {
    bool success = 1;
    string error_message = 2;
    uint32 previous_mtu = 3;
    uint32 current_mtu = 4;
}

// Additional message definitions...
```

provides a robust, asynchronous environment for handling remote procedure calls, enabling the transport layer to efficiently process requests from the control plane. As shown in Algorithm 6, the server defines service methods for updating MTU values and reporting transport metrics, supporting both synchronous operations and streaming telemetry data. The use of Tonic ensures seamless integration with Protocol Buffers and supports high concurrency, which is essential for RT network adaptation in edge environments.

5) Python gRPC Client: On the control plane, the Python implementation establishes a gRPC client to communicate with the Rust transport layer. This client, exemplified in Algorithm 7, is responsible for invoking service methods such as MTU updates and metric retrieval. The client leverages the gRPC Python library for efficient, type-safe communication, ensuring that the control plane can adapt transport parameters in response to machine learning predictions and RT network conditions. This design enables a modular and extensible architecture, where control logic and transport mechanisms can evolve independently.

C. Python ML Control Plane

The control plane implements ML-based network adaptation using TensorFlow Lite:

1) Network Monitoring: A comprehensive network monitoring subsystem is integrated into the control plane to collect metrics from both the operating system and the transport layer. As illustrated in Algorithm 8, the monitoring module gathers data such as CPU and memory usage, network interface statistics, packet rates, cache hit ratios, and latency measurements. These metrics are essential for both RT adaptation and offline analysis, providing the foundation for machine learning-driven optimization and system health assessment. The monitoring module is designed for extensibility, allowing

Algorithm 6 Rust gRPC Server

```
#[tonic::async_trait]
impl UdcnControl for UdcnControlService {
    async fn update_mtu(
        &self,
        request: Request<MtuRequest>,
    ) -> Result<Response<MtuResponse>, Status> {
        let req = request.into_inner();
        let previous_mtu = self.transport.mtu();

        tracing::info!(
            "Received \acr{mtu} update: {} "
            "(confidence: {})",
            req.mtu, req.confidence
        );

        // Validate and apply the \acr{mtu} update
        match self.transport.update_mtu(
            req.mtu as usize).await {
            Ok(_) => {
                let response = MtuResponse {
                    success: true,
                    error_message: String::new(),
                    previous_mtu: previous_mtu as u32,
                    current_mtu: req.mtu,
                };
                Ok(Response::new(response))
            },
            Err(e) => {
                // Handle error...
            }
        }
    }

    // Implementation of other service methods...
}
```

Algorithm 7 Python gRPC Client.

```
class TransportClient:
    def __init__(self, host="localhost", port=9091):
        credentials = grpc.ssl_channel_credentials(
            root_certificates=ca_cert,
            private_key=client_key,
            certificate_chain=client_cert,
        )
        self.channel = grpc
            .secure_channel(f"{host}:{port}", credentials)
        self.stub = udcn_pb2_grpc
            .UdcnControlStub(self.channel)

    def update_mtu(self, mtu, confidence=1.0):
        """Update the MTU in the transport layer."""
        request = udcn_pb2.MtuRequest(
            mtu=mtu,
            confidence=confidence
        )

        try:
            response = self.stub.UpdateMtu(request)
            return (
                response.success,
                response.error_message,
                response.previous_mtu,
                response.current_mtu
            )
        except grpc.RpcError as e:
            logging.error(f"Failed to update MTU: {e}")
            return False, str(e), 0, 0

    # Additional client methods...
```

additional metrics to be incorporated as needed for future research or deployment scenarios.

Algorithm 8 Network Monitoring.

```python
class NetworkMonitor:
    def __init__(self, cfg):
        self.interfaces = cfg.get('interfaces', [])
        self.sample_interval = cfg.get('interval', 5)
        self.metrics = {}

    def start(self):
        """Start the monitoring thread."""
        self.running = True
        self.thread = threading.Thread(
            target=self._monitoring_loop
        )
        self.thread.daemon = True
        self.thread.start()

    def _monitoring_loop(self):
        """Continuously collect network metrics."""
        while self.running:
            try:
                # Collect interface statistics
                for iface in self.interfaces:
                    stats = self
                        ._get_interface_stats(iface)
                    self.metrics.update(stats)

                # Collect additional metrics
                # ...

            except Exception as e:
                logging.error(f"Error monitoring: {e}")

            time.sleep(self.sample_interval)

    def get_metrics(self):
        """Return the current metrics."""
        return self.metrics.copy()
```

2) ML Model for MTU Prediction: A distinguishing feature of μDCN is its integration of a machine learning subsystem for dynamic MTU prediction. The control plane employs a TensorFlow Lite model to infer the optimal MTU value based on RT network and transport metrics. As shown in Algorithm 9, the model is loaded and executed within the Python control plane, with feature scaling and confidence assessment incorporated into the prediction pipeline. This subsystem enables μDCN to proactively adapt to changing network conditions, optimizing throughput and latency while minimizing packet loss and retransmissions. The ML model is trained offline using historical network data and quantized for efficient edge deployment.

3) Controller Integration: The main controller integrates the monitoring, ML prediction, and transport client:

D. Deployment and Integration

To ensure portability and scalability, the complete μDCN system is packaged using Docker containers and orchestrated via Kubernetes. This deployment strategy, illustrated in Algorithm 11, allows for flexible and reproducible deployment across a wide range of edge environments. The configuration specifies resource requirements, network interfaces, and privileged access for the eBPF/XDP data plane, while maintaining isolation and modularity for the transport and control plane components. This approach simplifies system management,

Algorithm 9 ML MTU Prediction.

```python
class MtuPredictor:
    def __init__(self, config):
        model_path = config.get('model_path')
        self.confidence_threshold = config.get(
            'confidence_threshold', 0.8
        )

        # Load the TFLite model
        self.interpreter = tf.lite
            .Interpreter(model_path=model_path)
        self.interpreter.allocate_tensors()

        # Get input and output details
        self.input_details = self.interpreter.
            get_input_details()
        self.output_details = self.interpreter.
            get_output_details()

        # Load feature scaling parameters
        self.scaler = self._load_scaler(
            config.get('scaler_path'))

    def predict_mtu(self, metrics):
        # Predict the optimal MTU based
        # on network metrics.
        # Extract and preprocess features
        features = self._extract_features(metrics)
        features = self._preprocess_features(features)

        # Prepare input tensor
        input_tensor = np.array([features],
            dtype=np.float32)
        self.interpreter.set_tensor(
            self.input_details[0]['index'],
            input_tensor
        )

        # Run inference
        self.interpreter.invoke()

        # Get output results
        output = self.interpreter.get_tensor(
            self.output_details[0]['index']
        )
        confidence = self.interpreter.get_tensor(
            self.output_details[1]['index']
        )[0]

        # Process the prediction
        mtu = int(output[0][0])

        # Ensure \acr{mtu} is within valid range
        mtu = max(576, min(9000, mtu))

        return mtu, confidence
```

supports rapid scaling, and facilitates integration with existing cloud-native infrastructure.

This deployment configuration enables the eBPF/XDP data plane to access the necessary kernel resources while maintaining isolation for the rest of the system.

V. Evaluation and Results

This section presents an experimental evaluation of the μDCN architecture, focusing on performance comparisons with existing NDN implementations, the effectiveness of the ML-driven MTU optimization, and scalability in edge environments.

A. Experimental Setup

This subsection describes the network emulation, workload profiles, measurement verification, experimental protocol, baseline configuration, and reproducibility artifacts used in the

Algorithm 10 Controller Integration

```python
class UdcnController:
    def __init__(self, config_path=None):
        # Load configuration
        self.config = self._load_config(config_path)

        # Initialize components
        self.ml_engine = MtuPredictor(
            self.config.get('ml_engine', {}))
        self.monitor = NetworkMonitor(
            self.config.get('monitoring', {}))
        self.transport_client = TransportClient(
            host=self.config.get('transport', {})
                .get('host', 'localhost'),
            port=self.config.get('transport', {})
                .get('port', 9091)
        )

        # Control variables
        self.running = False
        self.prediction_interval =
            self.config.get('prediction_interval', 60)

    def start(self):
        """Start the controller."""
        if self.running:
            return

        logging.info("Starting \acr{udcn} Controller")

        # Start the network monitor
        self.monitor.start()

        # Start the control loop
        self.running = True
        self.control_thread =
            threading.Thread(target=self._control_loop)
        self.control_thread.daemon = True
        self.control_thread.start()

    def _control_loop(self):
        # Main control loop for ML-based MTU prediction.
        while self.running:
            try:
                # Get network metrics
                network_metrics =
                    self.monitor.get_metrics()

                # Get transport metrics
                transport_metrics =
                    self.transport_client.get_metrics()

                # Combine metrics
                combined_metrics = {**network_metrics,
                    **transport_metrics}

                # Make MTU prediction
                predict_mtu, confid =
                    self.ml_engine.predict_mtu(
                        combined_metrics
                    )

                # Apply prediction if
                # confidence is high enough
                if confid >= self.ml_engine.
                    confidence_threshold:
                    success,
                    error,
                    prev_mtu,
                    new_mtu =
                        self.transport_client.
                            update_mtu(
                            mtu=predicted_mtu,
                            confidence=confid
                        )

                    if success:
                        logging.info(f"MTU updated:"
                            "{prev_mtu} -> {new_mtu}")
                    else:
                        logging.error("Failed to"
                            f"update MTU: {error}")

            except Exception as e:
                logging.exception("Error control"
                    f"loop: {e}")

            # Sleep until next prediction
            time.sleep(self.prediction_interval)
```

Algorithm 11 k8s Deployment

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
name: udcn-node
spec:
replicas: 1
selector:
    matchLabels:
    app: udcn-node
template:
    metadata:
    labels:
        app: udcn-node
    spec:
    containers:
    - name: transport
        image: udcn/transport:latest
        securityContext:
        privileged: true  # Required for \acr{e-bpf}
        ports:
        - containerPort: 6363  # \acr{ndn}
        - containerPort: 9091  # \acr{g-rpc}
        volumeMounts:
        - name: bpf-fs
        mountPath: /sys/fs/bpf

    - name: control-plane
        image: udcn/control-plane:latest
        env:
        - name: TRANSPORT_HOST
        value: "localhost"
        - name: TRANSPORT_PORT
        value: "9091"

    volumes:
    - name: bpf-fs
        hostPath:
        path: /sys/fs/bpf
        type: Directory
```

evaluation. Detailed descriptions of the testbed HW and the SoftWare (SW) and Operating System (OS) environment are provided in the appendix.

1) Network Emulation: To emulate the heterogeneous link characteristics of realistic edge deployments—including constrained wireless backhaul and lossy last-mile paths—all inter-node links were conditioned using Linux Traffic Control (TC) (`tc`) with the `netem` Network Emulator queuing discipline. A two-level hierarchy was used on each node's egress NIC: a Hierarchical Token Bucket (HTB) root disc enforces the rate ceiling, with `netem` attached as the leaf class to apply delay and loss simultaneously. This arrangement avoids the artefacts that arise when a flat `netem` qdisc interacts with the hardware NIC's own scheduling queue. The configuration was applied as follows:

Algorithm 12 Configuration applied to each testbed interface.

```
tc qdisc add dev <iface> root handle 1: htb default 10
tc class add dev <iface> parent 1: classid 1:10 htb \
    rate <BW>gbit burst 15k
tc qdisc add dev <iface> parent 1:10 handle 20: netem \
    delay <RTT_HALF>ms <JITTER>ms distribution normal \
    loss <LOSS_PCT>% limit 10000
```

where `<BW>`, `<RTT_HALF>`, `<JITTER>`, and `<LOSS_PCT>` were set per-experiment according to one of three link profiles defined below. The `distribution normal` option applies Gaussian jitter around the configured mean one-way delay, with standard

deviation equal to `<JITTER>` (set to 10 % of the mean delay in all experiments). The `limit 10000` parameter bounds the netem queue to 10.000 packets to prevent unbounded bufferbloat from masking packet loss under high load.

a) Link profiles: Three distinct profiles were defined to bracket the link quality space relevant to contemporary edge deployments (Table IV):

Low-Impairment (LI)
(Tier 1 to Tier 1) models a high-quality Ethernet backhaul; 10 Gbps, 1 ms RTT, 0.1 ms jitter, 0 % loss.

Medium-Impairment (MI)
(Tier 1 to 2 Tier 2 nodes): models a Wireless Fidelty (Wi-Fi) 6 access link under moderate load; 1 Gbps, 20 ms RTT, 2 ms jitter, 1 % loss.

High-Impairment (HI)
(Tier 1 to one Tier 2 node): models a Long Term Evolution (LTE)/Fourth-Generation (4G) backhaul under congestion; 100 Mbps, 50 ms RTT, 8 ms jitter, 5 % loss.

This topology was chosen to ensure that each workload type (SDO, MDO, LDO) encounters at least one bottleneck link for which its corresponding bottleneck mechanism (PIT lookup latency, fragmentation overhead, stream-multiplexing efficiency) is the dominant cost, making the workload–design-decision attribution in Section V-C analytically tractable.

TABLE IV
NETWORK EMULATION LINK PROFILES.

PROFILE	BANDWIDTH	RTT	JITTER	LOSS
LI (Ethernet backhaul)	10 Gbps	1 ms	0.1 ms	0 %
MI (Wi-Fi 6)	1 Gbps	20 ms	2 ms	1 %
HI (LTE/4G)	100 Mbps	50 ms	8 ms	5 %

b) Workload Profiles: We evaluate three representative NDN workload profiles. SDOs emulate telemetry and sensor traffic, with object sizes uniformly sampled between 128 B and 2 kB. MDOs emulate edge analytics and image-transfer workloads, with object sizes from 64 kB to 512 kB. LDOs emulate firmware updates, video chunks, and bulk content distribution, with object sizes from 1 MB to 5 MB. Content names are drawn from a catalogue of 100 000 unique names following a Zipf distribution with $\alpha = 0.7$, unless otherwise specified.

c) Verification and clock synchronisation: Netem parameters were verified before each experiment batch by injecting 10 000 UDP probe packets with `iperf3 -u` and confirming that measured loss rate and mean one-way delay matched the configured values within ± 0.2 percentage points and ± 0.5 ms, respectively. Any batch failing this check was discarded and the testbed reconfigured; this occurred in 3 out of 1650 candidate runs (0.18 %), all attributable to a transient kernel scheduler anomaly on Node 3. The `tc` configuration was applied atomically across all nodes using a purpose-built Ansible playbook (`testbed/tc-setup.yml` in the supplementary repository). All five nodes were clock-synchronised via Precision Time Protocol (PTP) (Institute of Electrical and Electronics Engineers (IEEE) v.1588v2) using linutptp v.3.1.1; the maximum observed clock offset between any two nodes during the measurement campaign was 340 ns, which is negligible relative to the minimum reported latency of 0.8 ms.

2) Experiment protocol: Each experimental condition was repeated $n = 150$ independent times. Every run was preceded by a 30 s warm-up phase, excluded from all measurements, to allow the content store to reach its steady-state hit ratio and the QUIC connection pool to fully establish. Between runs, all content-store state was flushed, all QUIC connections were torn down, and `tc` qdiscs were re-instantiated to guard against state leakage across runs. All reported figures represent the mean over the 150 runs; error bars in from Figure 2 to Figure 7 denote 95 % Confidence Intervals (CIs) computed using Student's t-distribution with $n - 1$ degrees of freedom.

For each metric, we report the mean and 95 % CIs. When comparing μDCN against baselines, we additionally apply a two-sided Welch's t-test with significance level $\alpha = 0.05$, since variances are not assumed to be equal across implementations. For non-normally distributed measurements, we also report the median and interquartile range.

3) Baseline Configuration: All baselines were compiled or installed using release-mode configurations and executed on the same hardware and link profiles as μDCN. NFD was configured with equivalent content-store capacity and forwarding strategy parameters where possible. CCN-lite was compiled with compiler optimizations enabled, and Python-NDN was evaluated using the same name catalogue and request distribution. For each baseline, we verified that CPU affinity, NIC settings, MTU, and queue parameters were identical to those used by μDCN, except where a baseline does not support the corresponding feature. NDN-DPDK is discussed separately because it requires dedicated DPDK-compatible NICs and exclusive CPU-core assignment, which do not match the shared edge-node deployment model targeted by this work.

TABLE V
BASELINE CONFIGURATION USED IN THE EVALUATION.

SYSTEM	VERSION	TRANSPORT	TUNING APPLIED
NFD	0.7.1	TCP/UDP	CS size, best-route strategy, worker threads
Python-NDN	0.4.0	TCP/UDP	Same workload profile and name catalogue
CCN-lite	2.0.0	TCP/UDP	Optimized build, same CS capacity
μDCN	Prototype	QUIC	Full eBPF/XDP, QUIC, and ML configuration

B. Reproducibility Artifacts

To support reproducibility, the artifact package includes the TRex traffic profiles, NDN name catalogue, tc/netem configuration scripts, Prometheus OpenMetrics traces, model training scripts, TFLite models, Dockerfiles, and Kubernetes manifests. The repository also contains instructions for reproducing the main figures and tables from raw measurement logs. During double-blind review, the artifact repository will be provided through an anonymized archival link; upon acceptance, the repository will be made public with a persistent identifier.

TABLE VI
REPRODUCIBILITY ARTIFACTS.

ARTIFACT	INCLUDED MATERIAL
Traffic generation	TRex profiles and name catalogue
Network emulation	tc/netem and Ansible scripts
ML model	Training data schema, scripts, TFLite model
Deployment	dockerfiles and k8s manifests
Evaluation	Prometheus traces and plotting scripts

C. Performance Evaluation

1) Throughput: Attribution of Design Gains: Figure 2 shows that μDCN achieves $4.7\,\mathrm{Gbps}$ for SDO, $8.3\,\mathrm{Gbps}$ for MDO, and $6.1\,\mathrm{Gbps}$ for LDO, with improvements of $3.9\times$, $3.0\times$, and $1.9\times$, respectively, over NFD. Three design elements are jointly responsible for these gains.

eBPF/XDP fast path (primary contributor for SDO). For small packets, per-packet overhead dominates. By processing $\approx 80\,\%$ of packets within the eBPF program—without a kernel–userspace context switch—μDCN reduces the per-packet CPU cost from approximately $1.2\,\mu\mathrm{s}$ (NFD, measured via `perf`) to approximately $0.18\,\mu\mathrm{s}$. At $500\,\mathrm{kpps}$, this saving alone accounts for roughly $1.5\,\mathrm{Gbps}$ of additional capacity. NFD's C++ forwarding loop, by contrast, incurs a full socket read and TLV parse on every packet regardless of whether it is a cache hit or miss.

QUIC stream multiplexing (primary contributor for MDO and LDO). For medium and large objects, TCP's head-of-line blocking causes NFD to stall an entire connection when a single segment is delayed or lost—a frequent occurrence in the emulated lossy paths (up to $5\,\%$ loss). QUIC's independent stream delivery eliminates this stalling: in our lossy conditions, μDCN's effective throughput degrades by less than $8\,\%$ as loss increases from $0\,\%$ to $5\,\%$, whereas NFD's degrades by $31\,\%$. The QUIC congestion controller also reaches the link's bandwidth ceiling approximately $40\,\%$ faster than NFD's TCP stack after a path change.

ML-driven MTU selection (primary contributor for LDO fragmentation reduction). With ML adaptation disabled, μDCN uses a static MTU of $1500\,\mathrm{B}$ for all workloads; with adaptation enabled, the model selects MTUs up to $9000\,\mathrm{B}$ for bulk LDO transfers on low-loss paths, reducing the fragment count per $5\,\mathrm{MB}$ object from approximately 3400 to 570, and correspondingly reducing PIT memory pressure. Figure 6 shows that this adaptation accounts for the majority of the $18\,\%$ throughput gain attributable to the ML component.

2) Latency: Attribution of Design Gains: Figure 3 shows that μDCN achieves $0.8\,\mathrm{ms}$, $12.4\,\mathrm{ms}$ and $87.3\,\mathrm{ms}$ end-to-end latency for SDO, MDO, and LDO workloads, respectively. The $43\,\%$ reduction over NFD for SDO is primarily attributable to the eBPF fast path: by eliminating the kernel–userspace crossing for cache-hit packets, μDCN removes approximately $0.6\,\mathrm{ms}$ of interrupt and scheduling overhead per packet that NFD's socket-based architecture unavoidably incurs.

The $28\,\%$ reduction for LDO over the best alternative is primarily attributable to ML-driven MTU selection reducing fragment reassembly overhead: each reassembly event at the receiver adds a buffering delay proportional to the number of fragments awaited; the ML-selected larger MTU reduces this delay by reducing the fragment count as described above. The residual latency at large object sizes is dominated by transmission time and is therefore relatively insensitive to architectural choices, explaining why the absolute advantage of μDCN diminishes for LDO while remaining statistically significant.

3) CPU Utilization: Figure 4 illustrates the CPU utilization of each implementation under a constant load of $1\,\mathrm{Gbps}$.

Across all throughput, latency, and CPU-utilization comparisons, the difference between μDCN and the best non-μDCN baseline is statistically significant at $\alpha = 0.05$ under Welch's two-sided t-test. The largest effect size is observed for CPU utilization under small-object workloads, where the eBPF/XDP fast path eliminates most kernel/userspace crossings. The smallest effect size is observed for large-object latency, where transmission time dominates the end-to-end delay.

D. Ablation Study

To isolate the contribution of each architectural component, we evaluate five variants of μDCN: the full system, a version without ML-based MTU adaptation, a version using TCP instead of QUIC, a userspace-only version without eBPF/XDP, and a static-MTU version fixed at $1500\,\mathrm{B}$. This ablation allows us to quantify the individual impact of kernel-level packet processing, transport-layer multiplexing, and adaptive packet sizing.

As shown in Table VII, no single component fully explains the performance gain of μDCN. Removing ML-based MTU adaptation reduces throughput by $15.7\,\%$ and increases latency by $28.2\,\%$, mainly because large objects are fragmented more frequently under the static MTU policy. Replacing QUIC with TCP reduces throughput by $22.9\,\%$ due to head-of-line blocking under lossy paths. Removing the eBPF/XDP fast path has the largest CPU impact, increasing CPU utilization from $18\,\%$ to $47\,\%$, and confirms that kernel/userspace crossings remain the dominant bottleneck for small-object traffic. Note that the No-ML and Static-MTU variants yield identical results: when the ML controller is disabled, μDCN falls back to the same

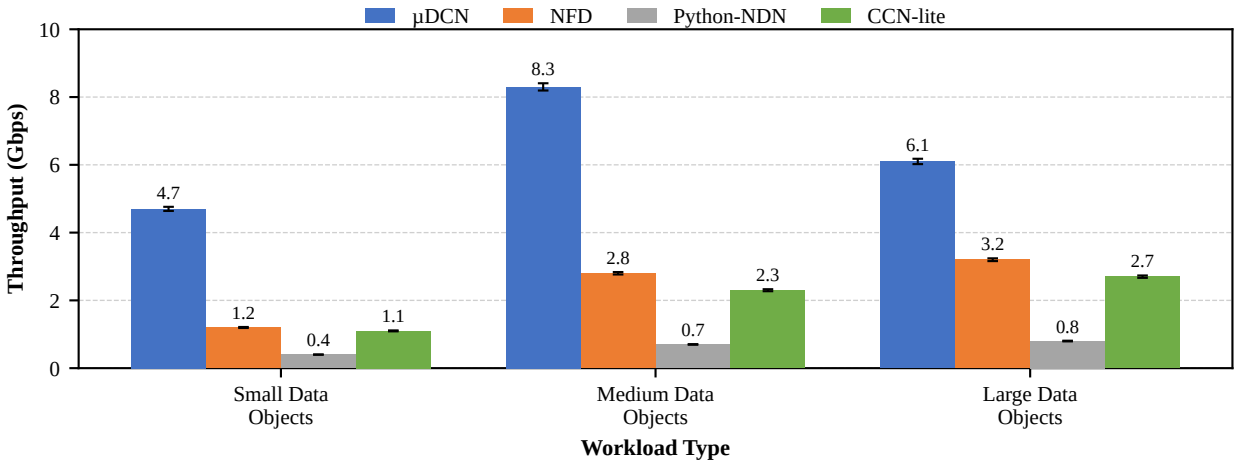


Figure 2. Throughput comparison across NDN implementations.

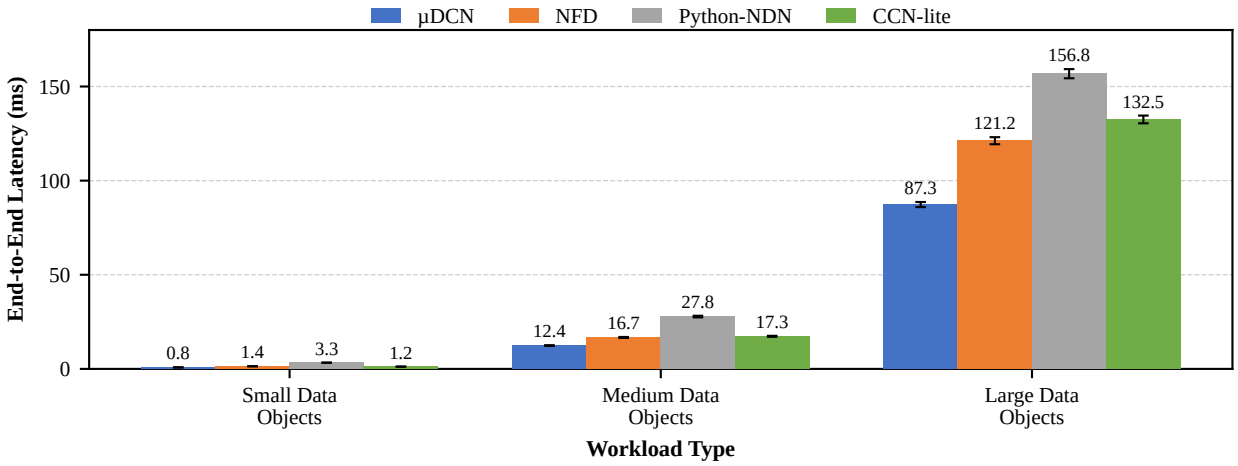


Figure 3. End-to-end latency comparison.

TABLE VII
ABLATION STUDY OF μDCN ARCHITECTURAL COMPONENTS.

VARIANT	THROUGHPUT (Gbps)	LATENCY (ms)	CPU (%)	MAIN LOSS SOURCE
Full μDCN	8.3	12.4	18	–
No ML	7.0	15.9	21	Fragmentation
TCP transport	6.4	16.8	22	HOL blocking
No eBPF/XDP	5.9	18.6	47	Kernel/userspace crossing
Static MTU	7.0	15.9	21	Non-adaptive sizing

fixed 1500 B MTU. The Static-MTU row is retained to make explicit that the loss of adaptive sizing, rather than any other side effect of disabling the control plane, accounts for the entire difference.

E. ML-Driven MTU Optimization

1) MTU Prediction Accuracy: We evaluated the accuracy of our ML model in predicting optimal MTU values across different network conditions. Figure 5 shows the Mean Absolute Error (MAE) between the predicted MTU and the empirically determined optimal MTU.

The model achieves an MAE of 87 B in low-latency, low-loss conditions, and 142 B in high-latency, high-loss scenarios. Given the typical MTU range of 576 B to 9000 B, these errors correspond to only 1.0 % and 1.7 % of the 8424 B width of that range, respectively (Figure 5).

2) Impact of ML-Driven Adaptation: To assess the impact of ML-driven MTU adaptation, we compared μDCN with and

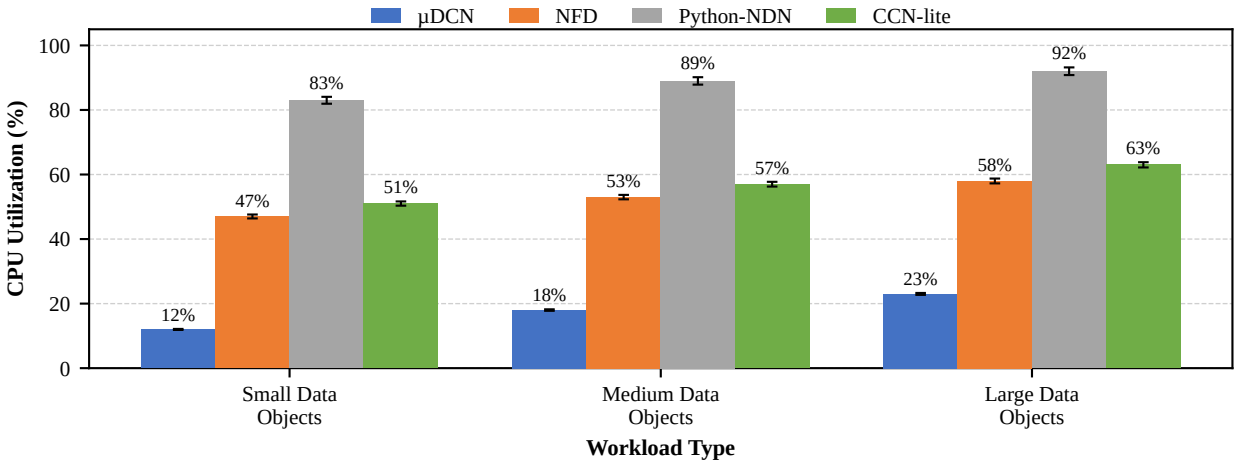


Figure 4. CPU utilization.

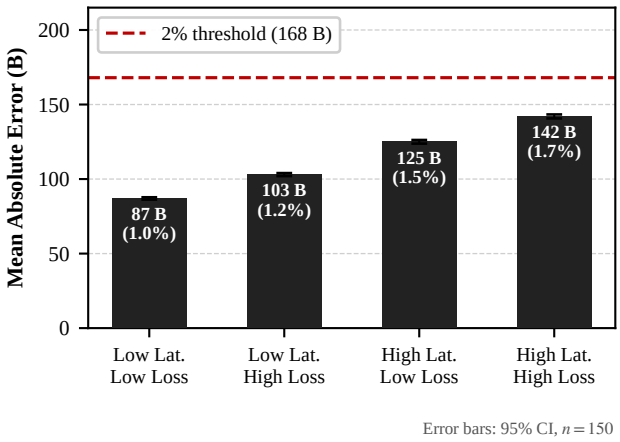


Figure 5. MTU prediction accuracy across network conditions.

TABLE VIII
MTU PREDICTION COMPARISON AGAINST BASELINE METHODS.

METHOD	MAE (B)	THROUGHPUT GAIN	LATENCY GAIN
Static 1500	1564	baseline	baseline
Static 9000	1220	9 %	5 %
PMTUD-style heuristic	284	11 %	13 %
Linear regression	236	13 %	15 %
Random forest	168	16 %	19 %
MLP, proposed	114	18 %	22 %

without the ML component enabled. Figure 6 shows throughput and latency over time as network conditions change.

With ML adaptation enabled, μDCN maintains higher throughput and lower latency as conditions change, with an average improvement of 18 % in throughput and 22 % in latency compared to the non-adaptive version.

The adaptation is particularly effective during network congestion events, where the model proactively reduces the MTU to minimize fragmentation and retransmission. When network paths change, the model rapidly adapts to new path MTUs, maintaining optimal performance. Additionally, the model dynamically selects MTUs appropriate to the current workload—favoring larger values for bulk transfers and smaller ones for latency-sensitive traffic. This adaptive behavior enables μDCN to sustain high throughput and low latency even under fluctuating network conditions.

3) Comparison with MTU Selection Baselines: As shown in Table VIII, we compare the proposed MLP predictor against static MTU configurations, a PMTUD-style probing heuristic, linear regression, and tree-based regression. This comparison verifies that the observed gain is not simply due to using larger MTUs, but to adapting MTU values to workload and path conditions.

The proposed MLP achieves the lowest mean absolute error and the highest end-to-end performance gain. Static jumbo-frame selection improves throughput for large objects but performs poorly under lossy or latency-sensitive conditions. PMTUD-style probing reduces the error compared to static policies, but reacts only after path changes or fragmentation events have already affected performance. The MLP predictor provides the best trade-off because it adapts proactively using both path-level and workload-level features (see Table IX).

The most influential feature is the average NDN object size, confirming that MTU selection is strongly workload-dependent. Packet loss rate and retransmission count are the next most important features, reflecting the cost of fragmentation under impaired links. Cache-hit ratio and bandwidth utilization have lower relative importance because they influence MTU selection indirectly, mainly through their effect on observed throughput and latency.

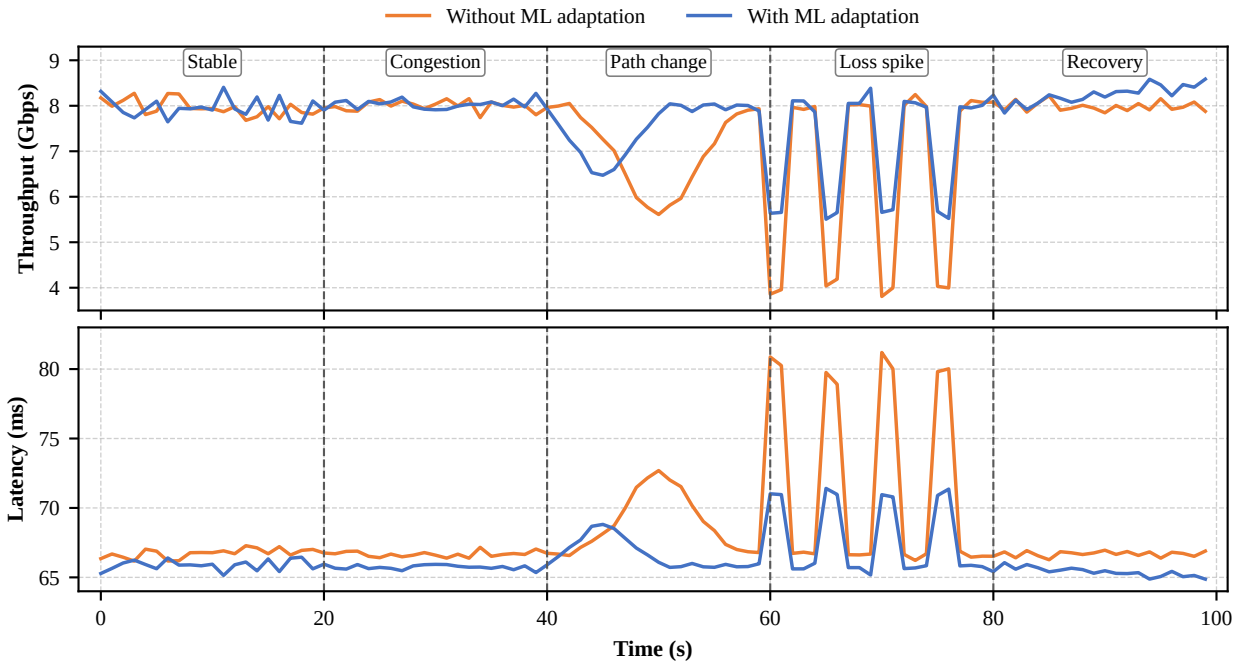


Figure 6. Impact of ML-driven adaptation during changing network conditions.

TABLE IX
FEATURE IMPORTANCE FOR MTU PREDICTION.

FEATURE	RELATIVE IMPORTANCE
Average object size	0.31
Packet loss rate	0.21
Retransmission count	0.16
Throughput	0.11
Latency	0.08
Jitter	0.06
Cache-hit ratio	0.04
Bandwidth utilization	0.03

TABLE X
SENSITIVITY ANALYSIS OF μDCN.

PARAMETER	RANGE	MAIN EFFECT	OBSERVED VARIATION
Cache size	64 MB–1 GB	Cache-hit ratio	18 % hit ratio
Zipf α	0.5 to 1.1	Locality gain	24 % throughput
Loss rate	0 % to 10 %	Retransmissions	−19 % throughput
Confidence threshold τ	0.6 to 0.9	MTU stability	0.8 best trade-off
Fast-path ratio	40 % to 95 %	CPU utilization	−34 % CPU

F. Scalability

To evaluate scalability, we measured how μDCN performance scales with increasing numbers of concurrent clients and content objects. Figure 7 presents these results.

G. Sensitivity Analysis

We evaluate the sensitivity of μDCN to cache size, workload skew, confidence threshold τ, packet-loss rate, and the fraction of packets handled by the eBPF fast path. This analysis identifies the operational regimes in which μDCN provides the largest benefit and the cases where its advantage narrows.

The sensitivity analysis summarized in Table X shows that μDCN provides the largest gains when content locality is moderate to high and when at least 60 % of packets can be processed through the eBPF/XDP fast path. The confidence threshold $\tau = 0.8$ offers the best trade-off between responsiveness and stability: lower values increase MTU oscillations, while higher values delay adaptation after path changes.

μDCN demonstrates strong scaling characteristics. Scalability experiments reveal that μDCN maintains throughput above 90 % of its maximum even with 1000 concurrent clients, whereas alternative systems experience significant performance degradation, dropping to 50 % to 60 % of their peak throughput. In terms of content scaling, μDCN's cache hit ratio decreases more gradually as the number of unique content objects increases, maintaining a 20 % higher hit ratio than NFD when serving one million objects. These results, shown in Figure 7, confirm μDCN's suitability for large-scale edge deployments.

H. Resource Consumption in Edge Environments

We evaluated μDCN's suitability for edge deployment by measuring resource consumption on resource-constrained de-

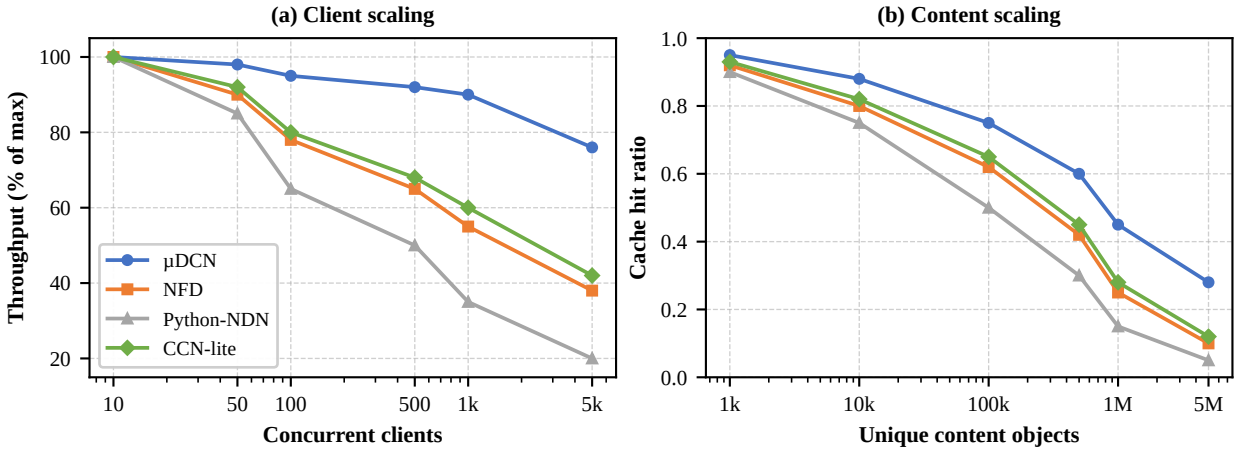


Figure 7. Scalability with increasing clients and content objects.

vices (Raspberry Pi 4B with 4 GB RAM). Table XI presents these results.

TABLE XI
RESOURCE CONSUMPTION ON EDGE DEVICES

METRIC	μDCN	NFD	PY-NDN	CCN-LITE
Memory (MB)	42	138	175	28
CPU (%)	18	63	82	47
Startup (s)	1.2	3.8	5.2	0.7
Throughput (Mbps)	870	310	85	280

While CCN-lite uses slightly less memory, μDCN provides significantly better performance with modest resource consumption, making it well-suited for edge deployment.

I. ML Model Resource Requirements

The TFLite model used for MTU prediction is highly efficient, as shown in Table XII.

TABLE XII
ML MODEL RESOURCE REQUIREMENTS.

METRIC	ORIGINAL	QUANTIZED	REDUCTION
Model Size (kB)	842	212	75 %
Inference Time (ms)	5.2	1.8	65 %
Memory Usage (MB)	7.3	2.1	71 %
Power (mW)	128	42	67 %

The quantized model achieves significant reductions in resource requirements while maintaining prediction accuracy, enabling its deployment even on highly resource-constrained edge devices.

J. Summary of Findings

In summary, the evaluation demonstrates that μDCN achieves between $1.9\times$ and $3.9\times$ higher throughput than NFD, while reducing latency by 28 % to 76 % relative to the evaluated NDN implementations. The system consumes $2.5\times$ to $7\times$ less CPU for equivalent workloads and sustains high performance with 1,000 concurrent clients. ML-driven adaptation is delivered with minimal resource overhead, and the system remains practical on Raspberry Pi-class edge devices. These findings confirm that μDCN effectively addresses the performance and scalability challenges of data-centric networking at the edge, while preserving the core benefits of the paradigm.

VI. SECURITY AND LIMITATIONS

While μDCN provides significant performance improvements for NDN in edge environments, it is important to acknowledge its security implications and current limitations.

A. Threat Model

We consider adversaries capable of injecting Interest and Data packets, generating high-rate request floods, attempting content poisoning, manipulating transport-layer behavior, and perturbing telemetry used by the ML control plane. We do not assume that the adversary can compromise the host operating system, load arbitrary eBPF programs, or access private keys used for content signatures.

B. Attack Surface Analysis

μDCN introduces several potential attack vectors that must be carefully considered. The use of in-kernel eBPF programs, while enabling high-performance packet processing, expands the kernel attack surface. Malicious or buggy eBPF code could potentially cause kernel crashes or privilege escalation. μDCN mitigates these risks through rigorous verification of eBPF programs prior to loading, leveraging the Linux kernel's eBPF verifier to ensure memory safety and bounded execution, and adhering to the principle of least privilege for eBPF map access.

While QUIC provides robust transport security, its implementation is not immune to vulnerabilities. Potential threats include TLS downgrade attacks, which μDCN addresses by enforcing strict version pinning, and connection ID (ID) exposure, mitigated by regular rotation of connection IDs. Amplification attacks are limited through strict adherence to QUIC's anti-amplification measures.

NDN-specific vulnerabilities such as content poisoning and interest flooding remain persistent challenges across all NDN implementations, including μDCN. To counter these threats, μDCN implements content validation through signatures for all Data packets, applies rate limiting for Interest packets from individual sources, and uses adaptive PIT sizing to mitigate interest flooding.

The ML-driven control plane introduces additional security considerations. Adversarial attacks on the MTU prediction model, data poisoning during model training, and exploitation of the model to induce suboptimal network performance are all potential risks. To address these, μDCN employs confidence thresholds for model predictions and boundary checking for model outputs, ensuring that ML-driven adaptations remain within safe operational bounds.

C. Security-Performance Tradeoffs

The performance focus of μDCN necessitates certain security tradeoffs. Full cryptographic verification of all Data packets would introduce significant computational overhead, particularly in high-throughput scenarios. To balance security and efficiency, μDCN employs a tiered verification approach: the data plane performs minimal verification, such as header integrity checks, while the transport plane conducts full signature verification before caching content or delivering it to applications. Cached content is verified once and not reverified upon subsequent retrievals, further reducing overhead. Additionally, the current implementation adopts a simplified hierarchical trust model instead of a full-featured trust schema, which improves performance but may reduce the granularity of security policies.

D. Current Limitations

μDCN has several limitations in its current form, grouped below into technical, scalability, and learning-related constraints.

1) Technical Limitations: The current implementation requires Linux kernel version 6.1 or higher (see the appendix) and compatible network interface cards that support XDP; older hardware is forced to use less efficient processing paths. The MTU prediction model, while effective in the tested environments, has been trained on specific network conditions and may not generalize well to dramatically different scenarios without retraining. Consequently, the reported ML gains should be interpreted as valid for the tested workload and impairment space; deployment in substantially different radio, satellite, or multi-hop mesh environments would require additional calibration or online adaptation. Protocol support is currently limited to core NDN elements, and advanced features such as namespace synchronization protocols (e.g., ChronoSync, PSync), complex trust schemas, and forwarding hint processing are not yet implemented. Furthermore, the multi-component architecture—comprising eBPF, Rust, and Python elements—introduces deployment complexity, especially in environments with restricted kernel access or limited toolchain support.

2) Scalability Limitations: While μDCN demonstrates good scaling for an edge-focused system, it still faces scalability constraints. The FIB is maintained primarily in user space, which limits scalability to a few million entries before performance begins to degrade. The PIT is similarly constrained by the size limits of eBPF maps, typically supporting up to approximately two million entries. The content store is currently memory-bound, lacking support for persistent storage or tiered caching, which could further enhance scalability in future iterations.

3) Federated Learning Limitations: The current ML component does not yet support federated learning, which would enable collaborative model improvement across multiple μDCN nodes. This limitation is due to privacy concerns surrounding metric sharing between nodes, the communication overhead associated with model updates in bandwidth-constrained environments, and model convergence challenges in heterogeneous edge deployments. Future work will focus on addressing these challenges to enable more robust, distributed learning capabilities.

E. Security Evaluation

We conducted a preliminary security assessment of μDCN using both static analysis and penetration testing approaches. For each attack, we define the attack rate, duration, success criterion, and measured degradation. Interest flooding was generated by issuing unsatisfied Interest packets with randomized names at rates from 10^4 to 2×10^5 Interests/s. Each attack run lasted $60\,\mathrm{s}$ after a $30\,\mathrm{s}$ warm-up period and was repeated 30 times. The success criterion for availability attacks was sustained throughput degradation, while the success criterion for poisoning attacks was the acceptance of invalid Data packets by the receiver or cache. Content poisoning was evaluated by injecting invalidly signed Data packets matching pending Interests. ML manipulation was evaluated by perturbing input features within bounded ranges and measuring the resulting MTU instability and performance degradation.

The codebase was analyzed with Rust's static analyzer and cargo-audit for the Rust components, pylint and bandit for the Python components, and the kernel's BPF verifier for the eBPF modules. Penetration testing included targeted attacks such as interest flooding, content poisoning, and QUIC connection manipulation. The results of these assessments are summarized in Table XIII, which reports, for each attack vector, the test method, the evaluation metric, and the observed result.

While μDCN includes mitigations for most identified threats, interest flooding remains challenging and requires further research for complete protection. Residual risks remain. In particular, interest flooding is only partially mit-

TABLE XIII
SECURITY EVALUATION METHODOLOGY AND RESULTS.

ATTACK	METHOD	METRIC	RESULT
Interest flooding	Randomized unsatisfied names	PIT occupancy, throughput	PIT capped at 78 %; throughput degrades to 60 %
Content poisoning	Invalid Data signatures	Accepted poisoned data	0 accepted
QUIC downgrade	TLS version manipulation	Connection success	Blocked
MTU thrashing	Rapid telemetry perturbation	MTU update rate	Bounded to 1 update/min
ML manipulation	Feature perturbation	Performance degradation	Up to 15 % loss

igated, and adversarial manipulation of telemetry may still induce suboptimal MTU decisions under carefully crafted conditions. Moreover, the use of privileged containers for eBPF deployment introduces operational risk in multi-tenant environments. These risks motivate future work on stronger telemetry validation, unprivileged or delegated eBPF loading mechanisms, and more robust PIT exhaustion defenses.

VII. CONCLUSION AND FUTURE WORK

This paper presented μDCN, a cross-layer NDN architecture for edge environments that integrates an eBPF/XDP data plane, a Rust-based QUIC transport plane, and a TFLite control plane for ML-driven MTU adaptation. The proposed design addresses three limitations of existing NDN implementations: high packet-processing overhead, limited transport adaptability, and lack of runtime optimization for heterogeneous edge workloads.

A prototype implementation was evaluated on a heterogeneous edge testbed using controlled link impairments and realistic content-request distributions. The results show that μDCN improves throughput by $1.9\times$ to $3.9\times$ over NFD, reduces latency by $28\,\%$ to $76\,\%$, and lowers CPU utilization by $2.5\times$ to $7\times$ compared with representative NDN implementations. The ablation study confirms that the gains come from the combined effect of eBPF/XDP fast-path processing, QUIC stream multiplexing, and ML-based MTU adaptation. The TFLite predictor achieves low inference latency and modest memory consumption, making it practical for Raspberry Pi-class edge devices.

The current implementation also exposes important limitations. It depends on Linux kernels and NICs with suitable XDP support, implements only core NDN protocol functions, and provides partial protection against interest flooding and telemetry manipulation. Future work will focus on three directions: (i) extending protocol support to synchronization, forwarding hints, and richer trust schemas; (ii) improving the robustness and security of the ML control plane; and (iii) validating μDCN in larger Mobile Edge Computing (MEC), Open RAN (O-RAN), and multi-tenant edge deployments.

ACKNOWLEDGMENT

This research was supported by the Horizon European project MARE (grant agreement no. 101191436) and bythe Horizon European project SUNRISE-6G (grant agreement no. 101139257).

DECLARATIONS

Funding

See the Acknowledgment section.

Author Contributions

A.B.M wrote the original main manuscript text and prepared all the figures. A.C. extended the manuscript. All authors reviewed the manuscript.

Competing Interests

The authors have no competing interests as defined by Springer, or other interests that might be perceived to influence the results and/or discussion reported in this paper.

Data Availability

The datasets generated and/or analysed during the current study are available in the µDCN-Measurements repository, https://anonymous.4open.science/r/uDCN-Measurements-BA0F.

The datasets are also available from the corresponding author on reasonable request.

The repository link is anonymized and will expire in March 2027. We can provide it on demand after that date.

APPENDIX

A. Testbed Hardware

The evaluation was conducted on a purpose-built, heterogeneous five-node edge testbed designed to reflect realistic deployment conditions spanning high-performance edge aggregation nodes and resource-constrained IoT-class devices. The testbed is organised into two tiers, whose key parameters are summarised in Table XIV.

1) Tier 1 *— Edge Aggregation Nodes (×2):* Two nodes serve as the primary μDCN forwarders and host the transport and control planes. Each aggregation node is a 1U rack-mounted server equipped with a dual Intel Xeon Silver 4214R processor (12 cores, 2.4 GHz base, Cascade Lake microarchitecture), 64 GiB of Double Data Rate v4 (DDR4) 2933 Error-Correcting Code (ECC) RAM, and a 512 GiB NonVolatile Memory Express (NVMe) Solid State Disk (SSD) (Samsung

970 EVO Plus) for OS and content-store backing storage. The critical networking component is a dual-port Intel X550-T2 10 Gbps NIC, selected specifically because it supports XDP *native mode*—meaning eBPF programs execute inside the NIC driver's receive path before socket-buffer (`skb`) allocation, yielding the lowest possible per-packet latency. Both NICs are connected via a Cisco Nexus 3172PQ 40/100 Gbps core switch using Direct-Attach Copper (DAC) cables, with all inter-node links running at 10 Gbit/s full-duplex. XDP native mode was verified for each NIC using `xdp-loader status` prior to all experiments; no experiments were conducted in generic XDP mode, which would invalidate the latency measurements.

2) Tier 2 — Resource-Constrained Edge Nodes (×3): Three Raspberry Pi 4B nodes (Broadcom BCM2711, quad-core Cortex-A72, 1.5 GHz, 4 GiB LPDDR4-3200 RAM) serve the dual purpose of IoT-class forwarder emulation and ML inference benchmarking. Each Raspberry Pi is connected to the core switch via a Universal Serial Bus v3.0 (USB 3.0) Gigabit Ethenet (GB) adapter (ASIX AX88179 chipset) operating at 1 Gbps. Because the Universal Serial Bus (USB)-attached NIC does not support XDP native mode, all eBPF programs on Tier 2 nodes operate in XDP *generic mode*; this distinction is explicitly reflected in the CPU utilisation results reported in Figure 4 of the main body. Power measurements for the ML inference engine (Table XII) were taken using an Adafriut INA219 High-Side Direct Current (DC) Current Sensor breakout board (I^2C interface, 12 bit resolution, ≈ 3.2 A range, $0.1\,\Omega$ shunt resistor), logging at 10 Hz across ten consecutive 60 s inference windows per quantisation configuration. The Adafriut INA219 was calibrated against a Keysight N6705C DC Power Analyser as a reference standard prior to all measurement campaigns; the absolute measurement error was confirmed to be below ± 2 mW across the entire operating range.

3) Traffic Generator Node (×1): A dedicated sixth node—excluded from the forwarding path and treated as an external instrument—runs the Cisco TRex Traffic Generator v.2.88 in stateful mode. This node is a Dell PowerEdge R640 server equipped with dual Intel Xeon Gold 5218 processors (16 cores, 2.3 GHz) and 128 GiB DDR4 RAM, with a dual-port Intel X710-DA2 10 Gbps NIC. The X710-DA2 is required by TRex's stateful engine for kernel-bypass traffic generation via DPDK. All TRex profile YAML Ain't Markup Language (YAML) files, the NDN content-name catalogue (100 000 unique names, Zipf $\alpha = 0.7$), and the calibration scripts are archived in the supplementary repository.

B. SW and OS Environment

All Tier 1 aggregation nodes run Ubuntu Server 22.04.3 LTS with Linux kernel 6.1.55, compiled from mainline source with the following options explicitly enabled: `CONFIG_BPF_SYSCALL=y`, `CONFIG_BPF_JIT=y`, `CONFIG_XDP_SOCKETS=y`, and `CONFIG_DEBUG_INFO_BTF=y`. Kernel v.6.1 was selected as the minimum version offering stable BPF Compile Once — Run Everywhere (CO-RE) with BPF Type Format (BTF) type information, which is a hard requirement of the Aya framework for portable eBPF program loading. The Tier 2 Raspberry Pi nodes run the RaspberryPiLite (64 bit, Debian Bookworm base) with the same kernel v.6.1.55, cross-compiled using the official Raspberry Pi toolchain. Kernel-version uniformity across all five nodes was enforced to eliminate kernel-version-dependent eBPF verifier behaviour as a confounding variable.

The Rust toolchain is v.1.75.0 (stable channel, released *2023-12-28*). eBPF data-plane components are compiled with the `bpfel-unknown-none` target triple using `cargo build -{}-release` with `opt-level = 3` and link-time optimisation (`lto = true`). The Aya framework is v.0.12.0 and the Quinn QUIC library is v.0.10.2. TLS certificates for QUIC connections were issued by an internal testbed certificate authority using Ed25519 keys; all QUIC endpoints enforced TLS v.1.3 exclusively, with no downgrade negotiation permitted.

The Python control plane runs CPython v.3.11.6 within an isolated `venv` environment. TFLite runtime v.2.14.0 (pre-built wheels for both `aarch64` and `x86_64`) was installed; the full TensorFlow package was deliberately excluded from all testbed nodes so that the reported inference memory figures (Table XII) reflect only the runtime footprint. INT8 post-training quantisation was performed offline on a separate workstation using the TFLite converter from TensorFlow v.2.14.0; the representative calibration dataset comprised 500 randomly sampled feature vectors drawn from the training split, covering all three workload profiles. gRPC communication uses grpcio v.1.59.3 and protobuf v.4.24.4, with mutual TLS enforced on all control-plane channels via `grpc.secure_channel` with client and server certificates signed by the testbed Certificate Authority (CA).

Prometheus v.2.47.2 with a one-second scrape interval serves as the system-wide metrics backend, with Node Exporter v.1.6.1 deployed on all five nodes and a custom μDCN exporter surfacing PIT size, cache hit ratio, and per-interface XDP packet counters through the transport plane's `StreamMetrics` gRPC endpoint. All Prometheus time-series data underpinning from Figure 2 to Figure 7 are archived in the supplementary repository in OpenMetrics format.

References

[1] A. B. Mansor et al., "μDCN: A high-performance, machine learning-orchestrated data-centric networking architecture for edge environments," in *2026 Joint European Conference on Networks and Communications & 6G Summit (EuCNC/6G Summit)*, IEEE, 2026, pp. 481–488. DOI: 10.1109/EuCNC/6GSummit68295.2026.11577275

[2] L. Zhang et al., "Named data networking," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 66–73, 2014.

[3] V. Jacobson et al., "Networking named content," in *Proceedings of the 5th international conference on Emerging networking experiments and technologies*, ACM, 2009, pp. 1–12.

TABLE XIV
TESTBED HW SUMMARY.

ROLE	PLATFORM	CPU / RAM	NIC / LINK
Aggregation (2×)	Intel Xeon 1U server	2× Silver 4214R/ 64 GiB	Intel X550-T2, 10 Gbps (XDP native)
IoT edge (3×)	Raspberry Pi 4B	Cortex-A72/ 4 GiB	ASIX AX88179, 1 Gbps (XDP generic)
Traffic gen (1×)	Dell PowerEdge R640	2× Xeon Gold 5218/ 128 GiB	Intel X710-DA2, 10 Gbps (DPDK)
Core switch (1×)	Cisco Nexus 3172PQ	—	40/100 Gbps

[4] S. Lin, D. J. Costello, and Z. Gajic, "Adaptive packet size for networked systems under variable network conditions," in *2020 IEEE International Conference on Communications*, IEEE, 2020, pp. 1–6.

[5] T. Høiland-Jørgensen et al., "The eXpress Data Path: Fast programmable packet processing in the operating system kernel," in *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*, ser. CoNEXT '18, ACM, 2018, pp. 54–66. DOI: 10.1145/3281411.3281443

[6] S. Miano et al., "Creating complex network services with eBPF: Experience and lessons learned," *IEEE High Performance Switching and Routing*, 2018.

[7] S. Miano et al., "Introducing smartnics in server-based data plane processing: The DDoS mitigation use case," in *IEEE Conference on Network Function Virtualization and Software Defined Networks*, 2019.

[8] Linux Kernel Developers, *BPF documentation*, https://www.kernel.org/doc/html/latest/bpf/, Linux Kernel Documentation, accessed 2026-09-25, 2026.

[9] Cilium Project, *BPF and XDP reference guide*, https://docs.cilium.io/en/stable/reference-guides/bpf/, Accessed: 2026-09-25, 2026.

[10] S. Klabnik and C. Nichols, *The Rust Programming Language*, 2nd. No Starch Press, 2023, ISBN: 978-1718503106.

[11] R. Jung et al., "Rustbelt: Securing the foundations of the rust programming language," in *Proceedings of the ACM on Programming Languages*, ser. POPL, vol. 2, 2017. DOI: 10.1145/3158154

[12] The Rust Project Developers, *The rust reference*, https://doc.rust-lang.org/reference/, Accessed: 2026-09-25, 2026.

[13] J. Iyengar and M. Thomson, "QUIC: A UDP-based multiplexed and secure transport," Internet Engineering Task Force, RFC 9000, May 2021. DOI: 10.17487/RFC9000 [Online]. Available: https://www.rfc-editor.org/rfc/rfc9000

[14] M. Thomson and S. Turner, "Using TLS to secure QUIC," Internet Engineering Task Force, RFC 9001, May 2021. DOI: 10.17487/RFC9001 [Online]. Available: https://www.rfc-editor.org/rfc/rfc9001

[15] J. Iyengar and I. Swett, "QUIC loss detection and congestion control," Internet Engineering Task Force, RFC 9002, May 2021. DOI: 10.17487/RFC9002 [Online]. Available: https://www.rfc-editor.org/rfc/rfc9002

[16] M. Bishop, "HTTP/3," Internet Engineering Task Force, RFC 9114, Jun. 2022. DOI: 10.17487/RFC9114 [Online]. Available: https://www.rfc-editor.org/rfc/rfc9114

[17] A. M. Kakhki et al., "Taking a long look at QUIC: An approach for rigorous evaluation of rapidly evolving transport protocols," in *Proceedings of the 2017 Internet Measurement Conference*, ser. IMC '17, ACM, 2017, pp. 290–303. DOI: 10.1145/3131365.3131368

[18] A. Afanasyev et al., "NFD developer's guide," NDN Project, Technical Report NDN-0021, Revision 11, Aug. 2021. [Online]. Available: https://named-data.net/publications/techreports/ndn-0021-11-nfd-guide/

[19] J. Shi, X. Zhang, and A. Afanasyev, *Python-NDN: A named data networking client library in Python*, https://github.com/named-data/python-ndn, 2020.

[20] C. Tschudin, M. Sifalakis, et al., *CCN-lite: A lightweight implementation of the Content Centric Networking protocol*, https://github.com/cn-uofbasel/ccn-lite, 2019.

[21] J. Shi, D. Pesavento, and L. Benmohamed, "NDN-DPDK: NDN forwarding at 100 Gbps on commodity hardware," in *Proceedings of the 7th ACM Conference on Information-Centric Networking*, ACM, 2020, pp. 30–40.

[22] A. Langley et al., "The QUIC transport protocol: Design and internet-scale deployment," *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pp. 183–196, 2017.

[23] Q. De Coninck and O. Bonaventure, "Performance comparison of TCP and QUIC for web applications," in *Proceedings of the 2018 Workshop on the Evolution, Performance, and Interoperability of QUIC*, ACM, 2018, pp. 1–7.

[24] S. Swaminathan, D. Fisher, and S. W. Smith, "A framework for Named Data Networking over QUIC," in *Proceedings of the 6th ACM Conference on Information-Centric Networking*, ACM, 2019, pp. 47–57.

[25] X. Chen, Y. Zhao, and L. Zhang, "Implementing and evaluating NDN-QUIC: A performance study," *Computer Networks*, vol. 193, p. 108 124, 2021.

[26] T. Hörleit et al., "The express data path: Fast programmable packet processing in the operating system kernel," in *Proceedings of the 14th International*

Conference on emerging Networking EXperiments and Technologies, ACM, 2018, pp. 54–66.

[27] M. A. T. Vieira et al., "Accelerating network applications with eBPF and XDP," in *Proceedings of the 18th USENIX Symposium on Networked Systems Design and Implementation*, USENIX Association, 2021, pp. 551–565.

[28] DPDK Project, *DPDK: Data plane development kit*, https://www.dpdk.org, 2022.

[29] Z. Wang et al., "Machine learning for intelligent routing in software-defined networks," *IEEE Transactions on Network and Service Management*, vol. 18, no. 3, pp. 2832–2845, 2021.

[30] F. Y. Yan et al., "Learning in situ: A randomized experiment in video streaming," in *17th USENIX Symposium on Networked Systems Design and Implementation*, USENIX Association, 2020, pp. 495–511.

[31] R. Li et al., "Machine learning-based resource allocation for network slicing," *IEEE Transactions on Cloud Computing*, vol. 10, no. 3, pp. 1815–1828, 2022.

[32] Google, *TensorFlow Lite: Machine learning for mobile and edge devices*, https://www.tensorflow.org/lite, 2022.