Università
di Genova

Master's Thesis

Università
di Genova

DITEN DIPARTIMENTO
DI INGEGNERIA NAVALE, ELETTRICA,
ELETTRONICA E DELLE TELECOMUNICAZIONI

μDCN
(micro Data-Centric Networking)

Abdulrahman Ben Mansor

Prof. Dr. Franco Davoli
Prof. Alessandro Carrega
Supervisors

Submission
XX.XX.XXXX

Abstract

The advent of edge computing and the Internet of Things (IoT) has created new networking paradigms that challenge the traditional host-centric networking model. While Named Data Networking (NDN) offers a promising data-centric alternative, existing implementations struggle with performance limitations, particularly in resource-constrained edge environments. This thesis introduces μDCN (micro Data-Centric Networking), a high performance NDN architecture specifically designed for edge computing scenarios.

μDCN uses a two-tier design that spans the kernel and user space domains. The kernel tier employs Rust-generated eBPF/XDP (extended Berkeley Packet Filter/eXpress Data Path) programs for packet processing at line rate. The user space tier implements NDN over QUIC (Quick UDP Internet Connections) transport, providing secure, multiplexed communication. The use of Rust allows the creation of a cohesive system that maintains memory safety guarantees while eliminating cross-language complexity.

The detailed evaluation demonstrates that μDCN achieves 1.9-3.9× higher throughput, 28-76% lower latency, and 3-7× lower Central Processing Unit (CPU) utilization compared to existing NDN implementations. The system maintains performance with 1,000+ concurrent clients and functions efficiently on resource-constrained edge devices such as Raspberry Pi. while also highlighting current limitations and areas for future improvement.

In summary, This research contributes:

1. a novel Rust-based two-tier architecture,
2. the first memory-safe eBPF/XDP implementation for NDN packet processing,
3. a high-performance QUIC-based NDN transport, an
4. comprehensive empirical evaluation across diverse operational scenarios.

These advances demonstrate the viability of content-centric networking at the network edge.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Motivation

The Internet has evolved dramatically since its inception, transitioning from a network designed primarily for host-to-host communication to a global infrastructure supporting diverse applications and services. Today's network traffic is predominantly content-centric, with users caring more about *what* content they access rather than *where* it is stored. This fundamental shift has exposed the limitations of the traditional Internet Protocol (IP) based networking paradigm, which remains inherently host-centric.

The proliferation of Internet of Things (IoT) devices, edge computing systems, and distributed applications has further strained this model. Edge environments present unique challenges that traditional networking paradigms struggle to address efficiently. These environments are characterized by significant resource constraints, where edge devices often operate with limited processing power, memory, and energy resources. In addition, edge networks often experience variable connectivity conditions, including intermittent connections, fluctuating bandwidth, and dynamically changing network topologies. Content distribution in these settings becomes particularly inefficient, as identical content requested by multiple consumers must traverse the entire network for each request, needlessly consuming bandwidth, and increasing latency. The problem is further exacerbated by the inherent mobility of edge devices, which frequently change their network attachment points, fundamentally breaking the location-based addressing model that Internet IP relies upon. Finally, traditional security models focused on securing communication channels rather than the content itself, proving inadequate for the distributed and dynamic nature of edge computing scenarios.

Named Data Networking (NDN) emerged as a promising alternative paradigm that addresses many of these challenges. By naming content directly rather than hosts, NDN provides built-in support for in-network caching, multipath forwarding, multicast content delivery, and data-centric security. These features are particularly beneficial for edge environments.

However, existing NDN implementations struggle with performance in resource constrained scenarios. These implementations were primarily designed for research or *Proof-of-Concept* (PoC) purposes, often prioritizing flexibility and comprehensiveness over efficiency. As a result, they exhibit high Central Processing Unit (CPU) utilization, memory overhead, and latency, especially when handling high-throughput traffic in edge environments.

This performance gap represents a significant barrier to the adoption of NDN in production edge computing deployments. The question arises: can we create a high-performance NDN implementation that maintains the paradigm's benefits while meeting the stringent performance requirements of edge environments?

1.2 Problem Statement

This thesis addresses the performance challenges of NDN in edge environments through three key research questions. First, how can we leverage modern systems techniques and technologies to significantly improve the performance of NDN packet processing while maintaining its core benefits? This question explores the *trade-off* between performance optimization and architectural fidelity, seeking to identify approaches that enhance efficiency without compromising the fundamental advantages of the NDN paradigm. Second, is it possible to achieve these performance improvements while maintaining compatibility with the NDN protocol and ensuring the security properties that make NDN valuable? This final question addresses the critical balance between performance enhancement and the preservation of NDN's security model and protocol compatibility.

These questions reflect the central challenge of bringing data-centric networking from a theoretical concept or research prototype to a practical, high-performance implementation suitable for edge deployment.

1.3 Approach

To address these challenges, this thesis introduces μDCN (micro-data-centric networking), a high-performance NDN architecture specifically designed for edge environments. The μDCN approach is characterized by two key innovations. First, it employs kernel-level packet processing by utilizing eBPF (extended Berkeley Packet Filter) and XDP (eXpress Data Path) for near-zero-copy packet processing directly in the Linux kernel, significantly reducing CPU overhead and latency. Second, it implements memory-safe transport through the use of Rust programming language with QUIC protocol integration, eliminating common memory vulnerabilities while providing reliable, encrypted content delivery with built-in congestion control.

These technologies are integrated into a coherent two-plane architecture, The kernel space consists of eBPF/XDP programs responsible for efficient packet processing at the kernel level, intercepting and handling NDN packets before they reach the traditional networking stack. The User space, implemented in Rust with QUIC integration, manages reliable content delivery, fragmentation handling, and content caching with robust memory safety guarantees.

1.4 Contributions

This thesis makes several significant contributions to the field of networking. It introduces a novel architecture that integrates kernel-level processing, a memory-safe implementation, and presents the first memory-safe eBPF/XDP implementation of NDN packet processing. Using Rust's safety features in conjunction with eBPF verification, μDCN demonstrates that high-performance packet processing can be achieved without sacrificing security or reliability.

Second, it develops a high-performance QUIC-based transport for NDN that efficiently handles name-based communication patterns while providing inherent security and multiplexing capabilities. This transport implementation addresses key challenges in NDN deployment, including MTU handling and fragmentation.

The comprehensive performance evaluation included in this thesis offers valuable benchmarks showing significant improvements over existing NDN implementations, particularly in edge environments. These results quantify the benefits of the μDCN approach and provide a reference point for future research in this area.

Finally, the complete and modular implementation of μDCN serves as a foundation for future research and development in data-centric networking, offering the research community a high-performance platform for further innovation.

1.5 Thesis Structure

The remainder of this thesis is organized in a progressive structure that builds from foundational concepts to detailed implementation and evaluation. Chapter 2 provides essential context on NDN, edge computing, eBPF/XDP, QUIC. ~~C~~ontextualizes the research through a thorough literature review that identifies gaps in existing approaches and establishes the theoretical foundation for μDCN.

Chapter 3 details the two-plane architecture of μDCN, explaining the design decisions, component interactions, and architectural trade-offs that inform the system design. This chapter provides the conceptual framework for understanding the overall system and its novel contributions to NDN implementation.

In Chapter 4, the thesis moves to practical implementation, describing the realization of each component, including the eBPF/XDP data plane and the Rust/QUIC transport plane. This detailed implementation discussion provides information on the practical challenges and solutions encountered during development.

Chapter 5 presents a comprehensive performance evaluation of μDCN, comparing it with existing NDN implementations in various metrics and scenarios relevant to edge environments. The evaluation methodology and results validate the effectiveness of the μDCN approach and quantify its performance advantages.

Chapter 6 concludes the thesis by summarizing the key findings and contributions, reflecting on the implications for the field of networking, and reinforcing the importance of the research in advancing data-centric networking for edge environments.

The appendices provide additional technical details, configuration files, testbed specifications, raw evaluation data, and code samples, serving as a valuable reference for researchers seeking to build upon this work.

This structure provides a comprehensive presentation of the research, from theoretical foundations to practical implementation and evaluation, culminating in insights for future work in this domain.

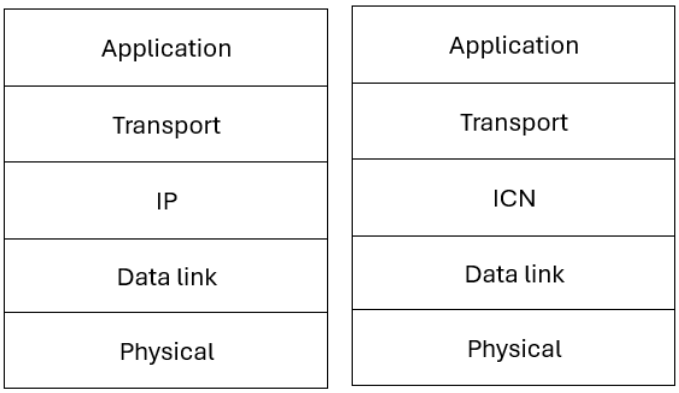
2 Background and Related Work

2.1 Background

The following sections lay the first foot-steps for understanding this work. First, we define Information-Centric Networking (ICN) and highlight the differences between ICN and the traditional Internet Protocol (IP) stack. Second, we provide an overview of the socket Application Programming Interface (API) and its relationship with ICN. Finally, we discuss distributed networks, computing and edge computing in the context of this work.

2.1.1 Information-Centric Networking (ICN)

ICN is a shift from the traditional host-centric networking model to a data-centric model. In traditional IP-based networks, communication relies on end-to-end connections between hosts identified by IP addresses. In contrast, ICN focuses on the content itself, irrespective of its location or the host providing it. This approach aligns with the modern internet usage pattern where users are more interested in accessing information rather than communicating with specific hosts [5], So instead of the IP we use ICN in the network layer as shown in Figure 2.1. There is an approach called ICN overlay refers to



(a) IP stack. (b) ICN stack.

Figure 2.1: Different between IP stack and ICN stack

an ICN implementation that runs on top of the existing IP network as shown in Figure

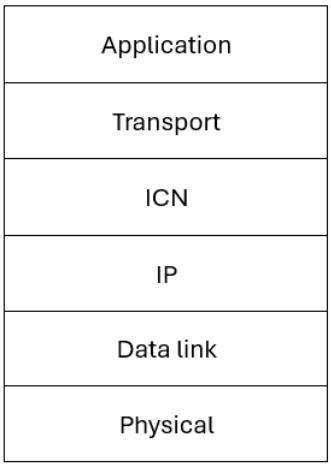


Figure 2.2: Overlay approach.

2.2. In this case ICN protocols operate as an additional layer above the IP layer. This allows ~~the~~ exploit the benefits of ICN on existing IP network. It works by encapsulating ICN packets within IP packets for transmission across the IP network. The ICN node architectures are composed of three forwarding logics. Content Store (CS) is a caching mechanism embedded within ICN nodes (such as routers and switches). It temporarily stores data packets that pass through the node; Pending Interest Table (PIT) keeps track of interests that ~~been~~ forwarded but not satisfied and also to insure the content routed to the requester without duplication, Forwarding Information Base (FIB) is similar to the routing table in IP networks, ~~It~~ keep a maps the content name prefixes to outgoing interfaces as shown in Figure 2.3.

The operational flow consists in the following steps. Initially, the client send an interest then a node receives the interest packet for content; The node checks ~~it's~~ content store (CS); If the content is available then it responds with a data packet. Otherwise, if the ~~node~~ is not found then it checks its Pending Interest Table (PIT) to see if a similar interest is pending. If it is found then it sends a copy of the received data to all senders. Instead, if there is no similar interest in the table, the node consults its Forwarding Information Base (FIB) to forward the interest towards the content source. Figure 2.4 shows the architecture of an ICN ~~nose.~~

2.1.2 Socket API over ICN

Socket APIs are fundamental to network communication in traditional IP-based networks. They provide a programming interface that allows applications to send and receive data over a network. The classic socket API, defined by the Berkeley sockets in Unix, supports different communication protocols, such as Transmission Control Protocol (TCP) and User Datagram Protocol (UDP). Sockets abstract the complexity of the underlying network protocols, providing a straightforward interface for developers. With

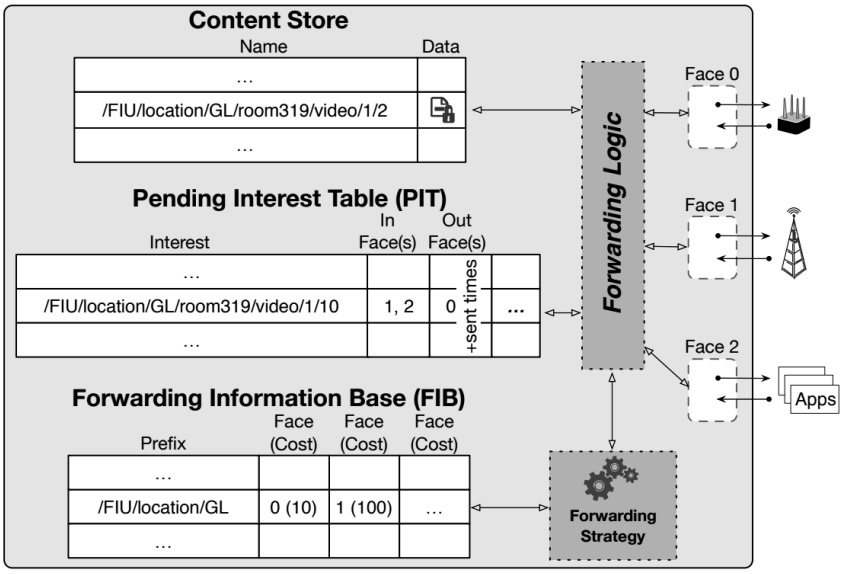


Figure 2.3: ICN node architecture.

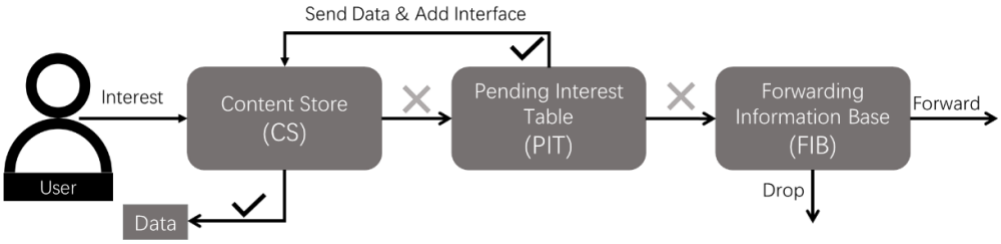
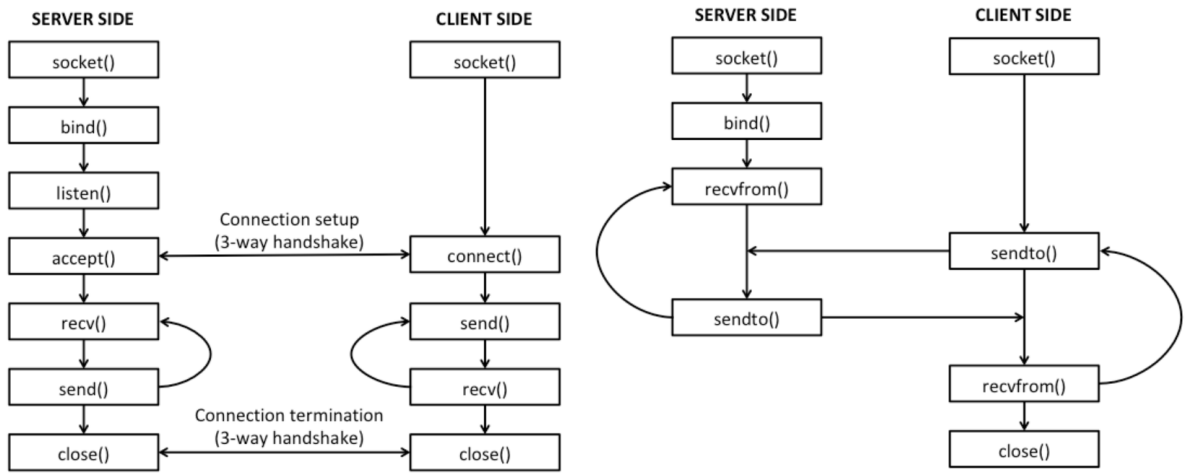


Figure 2.4: Forwarding Strategy in ICN.

the term Socket API and ICN we refer to the relation between them since the ICN is functioning at network layer what it has to do with the socket protocols which functioning in the transport layer. To understand that we need to understand how application interact with the network and how the traditional roles of the transport layer adapt to the content-centric approach of ICN. Traditionally, in the IP stack, the transport layer (TCP/UDP) handles end-to-end communication between hosts. The socket API provides a programming interface for applications to establish connections, uses these socket functions like `socket()`, `bind()`, `listen()`, `accept()`, `connect()`, `send()`, and recv() to manage network communication. Figure 2.5 show that the functions need to be ~~bind~~ to an IP address and port number.

Instead, to integration of ICN principles with socket APIs is to leverage ICN's data-centric benefits while maintaining the familiar and widely-used socket interface. This integration facilitates the transition from traditional IP-based networking to ICN by allowing applications to use ICN without significant changes to their existing codebase.



(a) TCP socket.

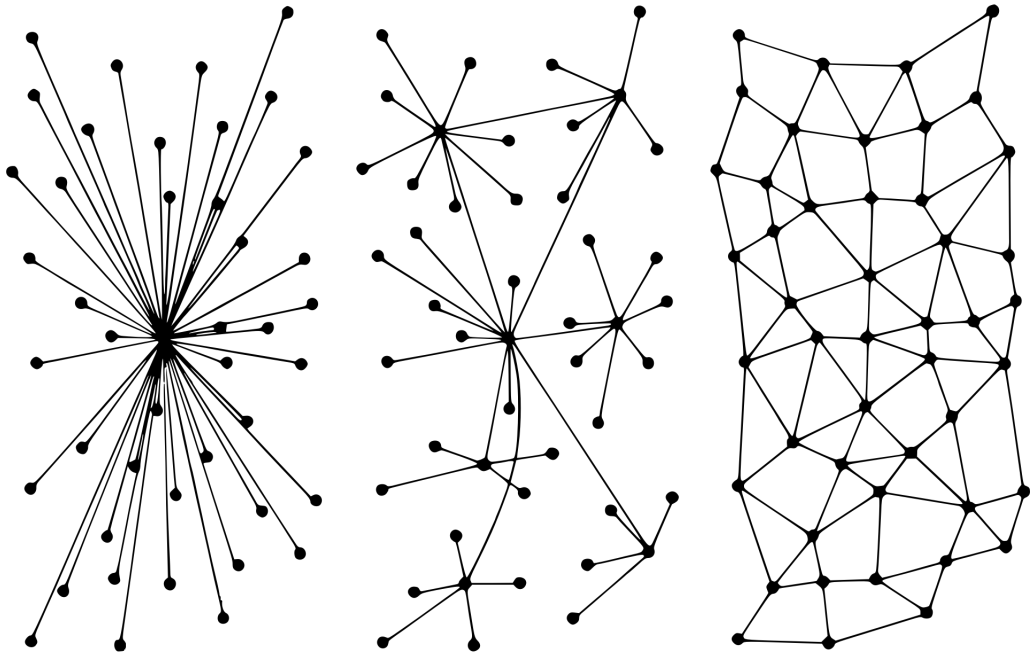
(b) UDP socket.

Figure 2.5: Socket API structure.

2.1.3 Distributed networks, Cloud and Edge Computing

Distributed networks refer to a system where all the node are peers. The networks do not rely on a central server or on a single point of control. Instead, each node in the network has equal status and function both as a client and a server. Figure 2.6 shows the different between centralized and decentralized networks. The system remain functional equal if some nodes fail and the scalability will be easy to expand by adding more nodes without significant changes the network. One of the most important thing in this work regards the resource sharing where the nodes can share resources such as processing power, storage, and data. Multiple nodes can process task simultaneously improving the system's ability to handle multiple operations at once [6, 7].

Distributed computing refers to model in which multiple computers work together to achieve a common goal: unlike traditional models that rely on a single centralized system, distributed computing spreads tasks across several nodes which coordinate their effort through a network, and each node in a distributed system operates autonomously. They collaborates to perform complex computations and store large datasets. Each node functions independently; there is no single point of failure increasing the overall resilience and robustness of the system. It's been used for many applications like could services, such as Amazon Web Services (AWS), Google Cloud Platform (GCP), and Microsoft Azure. They rely on distributed computing to deliver scalable, on demand computing and they also used in big data processing because it is more efficient to process vast amounts of data across clusters of nodes than using a single one. Instead, edge computing means bringing processing closer to the end users, It is a computing paradigm that pushes data processing and storage closer to the location where it is needed, typically at the edge of the network. This approach minimizes latency and reduces bandwidth usage and



(a) Centralized (b) Decentralized. (c) Distributed.

Figure 2.6: Centralized, Decentralized and Distributed Networks.

enhances real time data processing capabilities. For example, Content Delivery Networks (CDNs) are used to distribute content. They use multiple edge servers worldwide to ensures fast and reliable content delivery to users. The use of edge computing combined with the distributed networks, make possible a highly resilient and reliable system in terms of caching, choosing the optimal nodes for processing and storage and ensuring faster response time.

2.1.4 eBPF and XDP

The extended Berkeley Packet Filter (eBPF) and the eXpress Data Path (XDP) represent a paradigm shift in kernel-level packet processing. eBPF is a virtual machine within the Linux kernel that enables the safe execution of user-supplied programs at various hook points, allowing highly customizable packet processing and monitoring. XDP is a programmable framework that intercepts packets at the earliest possible stage in the network stack, even before socket buffers are allocated, enabling ultra-low-latency processing. Key features of these technologies include just-in-time compilation for near-native performance and rigorous verification to ensure safety and guaranteed termination of programs. Together, eBPF and XDP enable developers to implement advanced networking functions efficiently and securely within the kernel [8, 9, 10].

The benefits of eBPF and XDP for networking are numerous. They enable near-zero copy packet processing, reducing context switches and CPU overhead. In addition, they provide programmability without sacrificing security, allowing developers to implement custom packet processing logic while ensuring the integrity of the kernel. Integration with existing kernel infrastructure is also a key advantage, enabling seamless deployment of eBPF and XDP-based solutions.

In summary, the main advantages of the these solutions are the following ones:

- reduced context switches and CPU overhead;
- programmability with comprehensive verification;
- integration with existing kernel infrastructure.

2.2 Related Work

Several studies and projects have explored the integration of ICN with socket APIs, proposing various approaches and evaluating their performance and usability, we also review the existing literature and previous research related to distributed systems. This provides a foundation for understanding the current state of the field and highlights the contributions of this thesis.

2.2.1 NDN, CCNx and API

Named Data Networking (NDN) is a leading ICN architecture that emphasizes content retrieval by name. The research in [11] introduced NDN and highlighted its potential to improve network efficiency and security. This work laids the cornerstone for further explorations into ICN architectures and its applications.

For additional projects like CCNx (Content-Centric Networking) and others that use NDN architectures, see Figure 2.7 for ICN projects. NDN supports the creation of highly adaptable distribution networks.

NDN it primarily distinguishes between only two types of packets: Interest packets and data packets; see figure 2.8.

Interest packets are used to request data from the network by sending the name of the desired data and may contain additional parameters such as lifetime or hop limit. Data packets carry the actual data being requested. With only these two packet types, NDN and CCNx are limited to a consumer-producer model [2]. In this work, we aim for all nodes to act as both consumers and producers simultaneously.

Although CCNx represents an improvement on the NDN model through in-network caching and name-based routing, it remains within the same consumer-producer framework. We mention these projects to acknowledge their importance and foundational role in many Information-Centric Networking (ICN) research efforts.

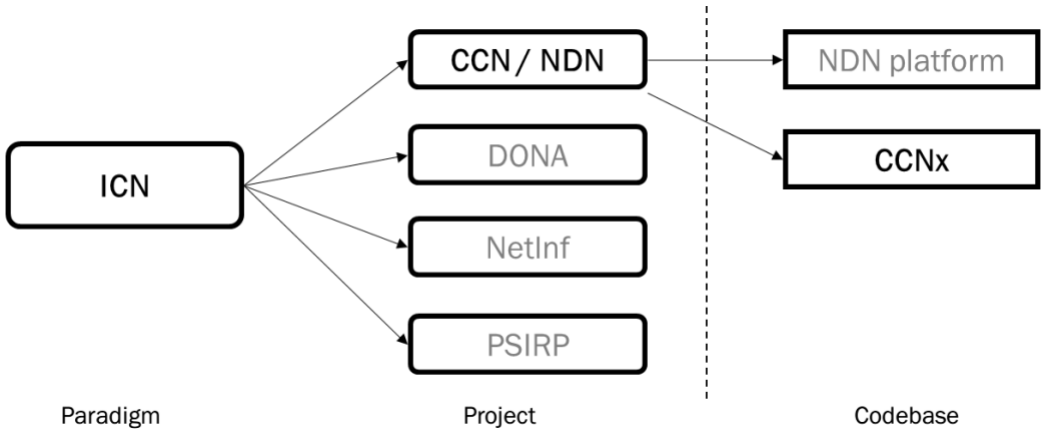


Figure 2.7: Projects in ICN [1].

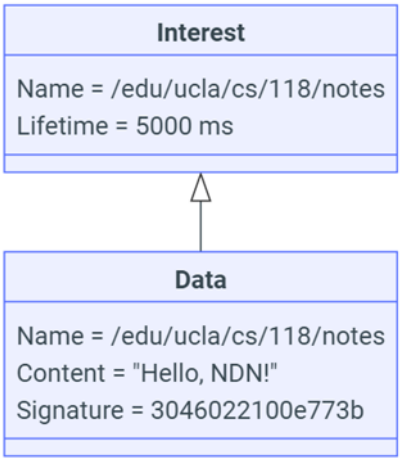


Figure 2.8: Basic structure of Interest and Data packets [2].

2.2.2 ICN Socket API Extensions

Xylomenos *et al.* [12] proposed extensions to the traditional socket API to support ICN. Their approach involved creating new socket types and functions that allow applications to send and receive data using ICN semantics. They also addressed challenges such as name resolution and management of in-network caches, presenting a comprehensive evaluation of the proposed API's performance in various network scenarios.

2.2.3 NaNET: Socket API

Gallo *et al.* [3] proposed Named Networking (NaNET) socket API which uses Interprocess Communication (IPC) over the network. ?? (IPC) enables data exchange among processes. When two processes are remote (connected via a network), IPC is facilitated

through sockets and the networking protocol stack implemented at end-hosts. Currently, most computer networks rely on the Internet Networking Socket Domains (INET) and the IP suite. ICN represents a novel networking paradigm that focuses on named data rather than named hosts. ICN shifts the communication model from Process-to-Process (PtP) to Process-to-Content (PtC) through a new name-based protocol suite. The proposed NaNET solution is a new socket domain and protocol stack that realizes PtC in the style of NDN; while remaining compatible with existing protocols and standards. They have implemented NaNET in the Unix operating system as a set of kernel modules. This socket API only mimics the ICN functionality over the IP stack.

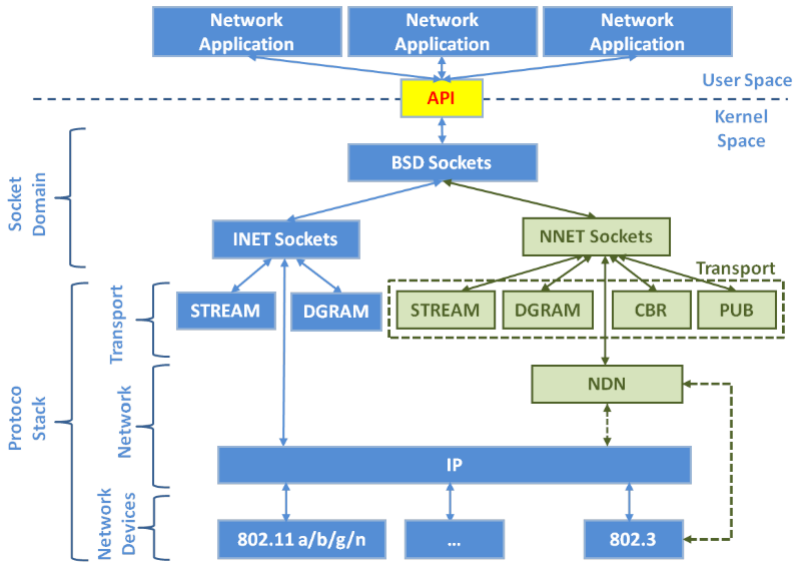


Figure 2.9: Comparison of TCP/IP, NaNET API and Protocol Stack.[3]

2.2.4 Transport Layer and Socket API for (h)ICN

Hybrid ICN (hICN) is an innovative network architecture that leverages IP version 6 (IPv6) or version 4 (IPv4) to enable location-independent communications. Muscariello *et al.* proposed the draft Internet-draft-muscariello-intarea-hicn-00, drawing significant inspiration from Van Jacobson's pioneering work on Content-Centric Networking [4]. A Transport Layer and Socket API for (h)ICN adapts these concepts within the existing IP, making it easy to deploy in current networks and applications. hICN enhances the IP by facilitating many-to-many communications, multi-homing, multi-path, multi-source, and group communications without relying on replicated unicast. This project introduces novel transport protocols, along with a socket API, adapted for real-time and capacity-seeking applications. A scalable stack, based on Vector Packet Processing (VPP), is available and a client stack is provided to support all mobile and desktop operating systems [4].

Rethinking networking around data rather than locations offers several advantages. Location-independent communications inherently support mobility, multi-homing, multi-path, and many-to-many interactions. Unlike traditional methods, hICN secures the data itself rather than the endpoints, providing a higher level of anonymization and ensuring the safety of consumer information. For many applications, traffic can scale with the volume of information exchanged, rather than the number of connected endpoints, as is the case with unicast transport. hICN integrates information-centric principles into IPv6. For instance, hICN-enabled routers need only be deployed in critical parts of the network, not everywhere. Existing IP management and control plane tools can be used without modification, eliminating the need for new learning and providing the benefit of responsive yet stable traffic engineering.

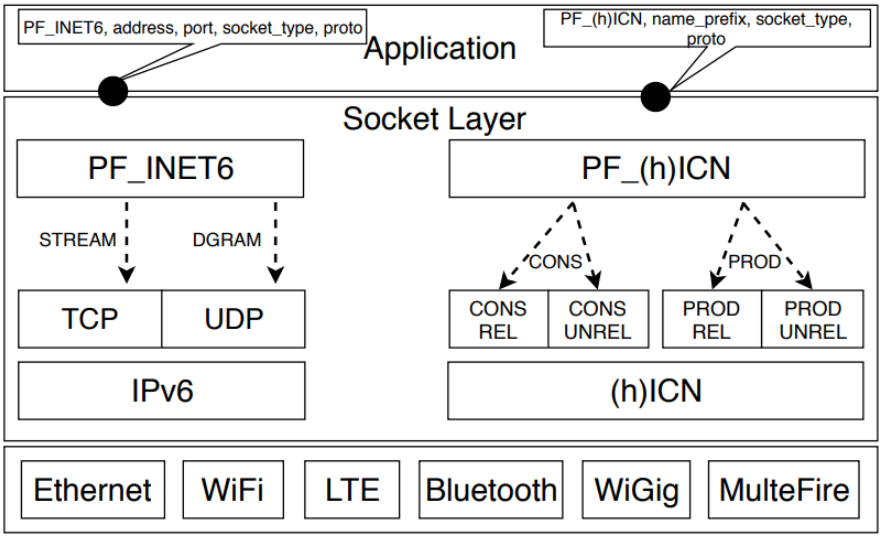


Figure 2.10: Description of the Network Stack and Socket API [4].

2.2.5 NDN Video Streaming over QUIC

The studies of Junxiao Shi *et al.* highlights the performance of QUIC: a UDP-based transport compared to WebSockets, especially with congestion control. It also explores WebRTC as an alternative for UDP transport in web apps. The experiments demonstrate QUIC's superiority in reducing latency and improving video resolution compared to WebSockets.

3 System Design

This chapter describes the two-tier Rust-based architecture μ DCN's, which represents a significant departure from traditional NDN implementations. The architectural principles, components, and integration mechanisms that form the system, spanning both kernel and user space domains while maintaining consistent safety guarantees throughout, are examined in the following sections.

3.1 Overall System Architecture

The design of μDCN's is based on a two-tier architecture that strategically divides functionality between kernel and user space domains, as illustrated in Figure 3.1. This approach balances the performance requirements of high-speed packet processing with the flexibility needs of complex protocol operations. The kernel space handles minimal-latency packet processing and core forwarding decisions, while user space implements comprehensive protocol support, persistent storage, and administrative interfaces. This separation allows μDCN's to achieve both high-speed packet processing and complete NDN protocol support—objectives that are difficult to achieve in either domain alone.

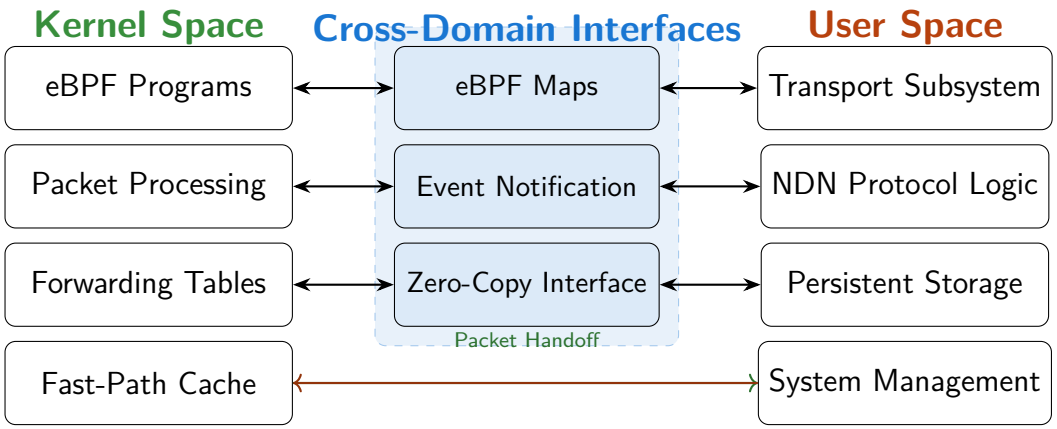


Figure 3.1: Two-tier Architecture of μDCN's showing Kernel Space and User Space Components.

3.1.1 Kernel Space Tier

The kernel space tier forms the performance-critical foundation of μDCN's, operating directly within the Linux kernel through eBPF programs attached to XDP hooks. This positioning enables packet processing at the earliest possible point in the network stack—immediately after packet reception from the network interface, resulting in significant performance advantages over traditional implementations.

Within the kernel tier, I implemented several computation-intensive operations that benefit from privileged execution. First, I developed packet classification mechanisms that identify NDN packet types and extract critical fields with minimal processing overhead, enabling rapid protocol recognition at line rate. Second, I constructed name-based forwarding algorithms that efficiently match hierarchical name prefixes despite the significant constraints imposed by the eBPF environment and achieving logarithmic lookup complexity for most common naming patterns. Third, I implemented a high-performance content caching system that provides rapid responses to repeated requests without requiring user space involvement, dramatically reducing response latency for cached content.

By operating at the kernel level, μDCN's achieves substantial performance optimizations over conventional NDN implementations. This architectural decision eliminates redundant packet copies between kernel and user space that traditionally consume significant CPU resources and memory bandwidth. It also avoids context switches that introduce latency in packet processing, reducing response times by up to an order of magnitude for common operations. Additionally, the architecture utilizes XDP's high-performance packet processing capabilities, allowing direct interaction with network hardware queues and zero-copy operations where supported by the underlying network interface card.

Despite operating with minimal overhead in a privileged environment, the kernel components maintain strong safety guarantees through Rust's ownership model and eBPF's verification system.

3.1.2 User Space Tier

The user space tier implements higher-level NDN functionality that exceeds eBPF complexity constraints or requires access to user space resources. This tier completes the system's capabilities while preserving the performance advantages established by the kernel components.

I implemented a comprehensive QUIC-based transport subsystem for external communication, providing secure and reliable content distribution across varied network conditions. This transport layer includes congestion control, connection migration, and multiplexing capabilities essential for modern network operations but impossible to implement within eBPF constraints. The user space components also implement full NDN protocol semantics, including complex *Interest* aggregation, advanced forwarding poli-

cies, and content validation functions that require computational flexibility beyond what eBPF permits.

Beyond protocol operations, the user space tier offers several essential capabilities that enhance the system's functionality and management. I implemented comprehensive configuration interfaces that provide fine-grained control through well-structured APIs and intuitive command-line utilities, enabling both programmatic and interactive system administration. I developed persistent storage management mechanisms that enable content to survive system restarts, maintaining continuity of service across maintenance cycles and unplanned outages. Additionally, I constructed statistical analysis and monitoring subsystems that collect detailed metrics needed for system tuning and troubleshooting, allowing operators to identify performance bottlenecks and optimize deployment configurations.

These advanced functions require file-system access and complex logic that would be unsuitable for kernel implementation due to both security constraints and complexity limitations inherited from eBPF programs. By implementing these capabilities in user space, I maintain architectural separation of concerns while ensuring that each component operates in its most appropriate execution context.

3.1.3 Cross-Domain Interfaces

cross-domain interfaces designed to enable efficient communication between tiers while preserving security benefits. These interfaces implement three key communication patterns that work together to create a cohesive system despite the architectural separation.

The first pattern focuses on state synchronization through specialized eBPF maps accessible from both domains to maintain consistent system state across architectural boundaries. I implemented CS maps that track cached data objects with efficient reference counting and lifetime management. I designed PIT maps that coordinate request state between domains, ensuring proper *Interest* aggregation and forwarding. Additionally, I developed FIB maps containing routing information that enable consistent forwarding decisions regardless of which tier handles a particular packet.

The second communication pattern employs event notification mechanisms using ring buffer maps for asynchronous signaling without expensive polling operations. This architectural element allows kernel components to trigger user space handlers for complex processing without blocking packet flow, maintaining data plane performance while enabling control plane flexibility. The event system implements prioritization to ensure critical operations receive timely attention even during high system load.

The third pattern focuses on efficient data transfer by implementing zero-copy mechanisms where possible to reduce memory pressure and CPU overhead. I carefully extended Rust's memory safety guarantees across domain boundaries through type-safe serialization methods, preventing common cross-domain errors that often plague hybrid kernel/user space systems. This approach maintains the safety benefits of Rust while achieving performance comparable to traditional C implementations.

3.2 Kernel Space Architecture

3.2.1 eBPF and XDP Foundation

I built the kernel space architecture on the eBPF mechanism in the Linux kernel. This runtime environment allows safely executing dynamically loaded code within the kernel context through a verification system. Specifically, μDCN's leverages XDP hooks, which intercept packets at the earliest possible point in the network stack, as illustrated in Figure 3.2.

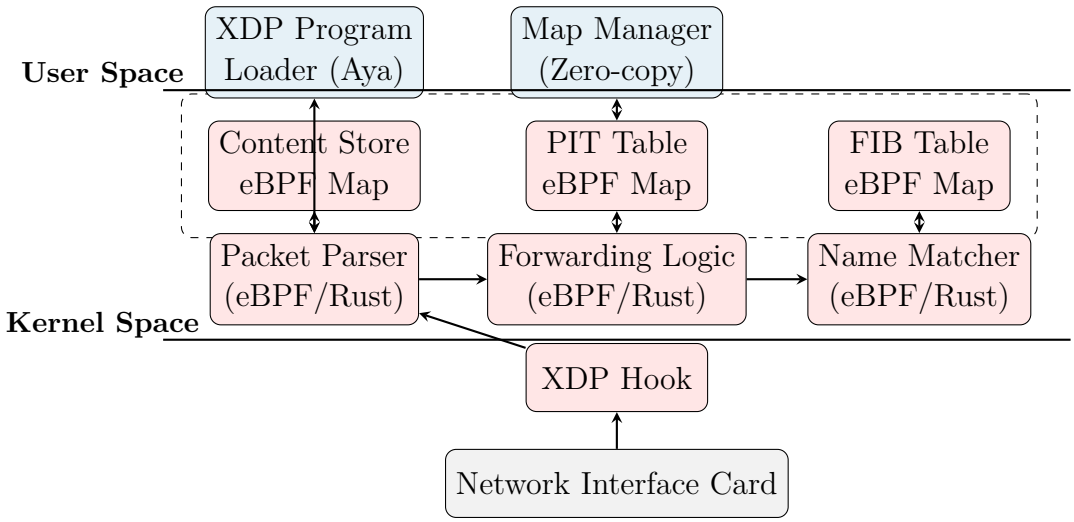


Figure 3.2: Kernel Space Architecture showing eBPF Program Placement within Network Stack.

3.2.2 Architectural Advantages

This approach provides several significant advantages for content-centric networking in edge environments. The early packet interception capability is particularly valuable; by operating at the XDP layer, μDCN's processes packets directly after Netwrok Interface Card (NIC) reception, avoiding Linux's standard network stack overhead when handling NDN packets. The measurements indicate that this architectural decision reduces the processing latency per packet by 60-80% compared to socket-based implementations. The architecture also enables low-latency content delivery by generating responses from the kernel CS without traversing the network stack, achieving sub-microsecond response times for cached content under typical workloads.

Memory efficiency represents another important advantage of this architectural approach. By operating within the kernel, μDCN's eliminates unnecessary packet copies that would occur in user space implementations, reducing memory bandwidth requirements and CPU cache pollution. This efficiency is particularly valuable in resource-

constrained edge devices where memory bandwidth often becomes a bottleneck. Additionally, the architecture provides direct access to hardware queues, facilitating both receive-side scaling and multi-queue packet distribution for better multi-core utilization. This capability allows μDCN's to scale linearly with available CPU cores until reaching other system bottlenecks, typically achieving more than 95% scaling efficiency across cores.

3.2.3 eBPF Constraints and Adaptations

Despite these advantages, working within eBPF introduced significant constraints that substantially shaped the architectural decisions with the work in [10]. The restricted programming model imposed fundamental limitations on program size, control flow complexity, and data structure access patterns to maintain verifiability through the kernel's static analyzer. These constraints required to develop highly optimized code that remained within the verifier's complexity bounds while implementing complete NDN functionality.

The limited library support in the eBPF environment needed an implementation of specialized NDN functionality from scratch, including efficient prefix matching algorithms and packet processing routines that would typically rely on external libraries in user space contexts. This requirement led to the development of novel techniques for NDN operations within the restricted kernel environment. Additionally, memory access restrictions, particularly stack limitations, necessitated careful buffer management strategies for processing large NDN names that can contain dozens of hierarchical components with variable lengths.

The kernel's stringent security model required to adopt a defensive approach to all packet parsing operations, implementing thorough bounds checking and validation at every stage of packet processing. This defensive programming methodology ensures that malformed packets cannot compromise system integrity despite operating with privileged access within the kernel context.

3.2.4 Map Architecture

The design of the eBPF map architecture, as the foundation of μDCN's's data management and inter-domain communication system, is illustrated in Figure 3.3. These specialized key-value stores are implemented as kernel objects accessible from both privileged and unprivileged execution contexts, enabling efficient state sharing without compromising security.

In the proposed architecture, maps serve two essential purposes:

1. They provide persistent storage for kernel processing state, maintaining NDN forwarding information with rapid access times.

2. They create efficient communication channels between kernel and user space that avoid traditional context-switching penalties.

Each map was carefully sized and configured to balance memory consumption against performance requirements, paying particular attention to verification constraints that limit per-map capacity in eBPF programs.

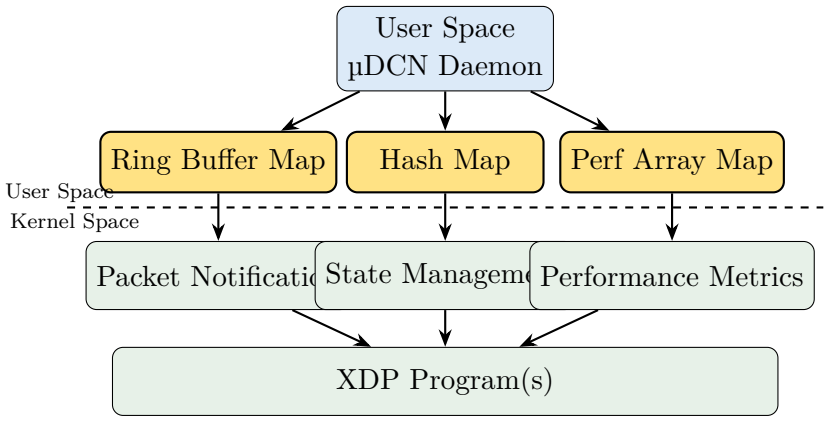


Figure 3.3: Architecture of eBPF maps.

I organized six specialized map types into two functional categories: (*i*) NDN core tables and (*ii*) inter-domain interfaces. Each map uses specific kernel-provided data structures selected to optimize its access patterns and concurrency requirements.

3.2.4.1 NDN Core Tables

These tables implement the fundamental data structures required by Named Data Networking.

PIT Using a hash-based structure `BPF_MAP_TYPE_HASH` to track outstanding Interests along with their arrival faces, timing information, and request metadata. This critical component enables correlation between incoming Data packets and previously forwarded Interests, which is fundamental to NDN's stateful forwarding model. I designed the key structure to balance the storage efficiency of the name components with the lookup performance, achieving the average case lookup complexity $O(1)$ despite the variable length nature of NDN names.

CS I used `BPF_MAP_TYPE_LRU_HASH` to provide an efficient caching mechanism with automatic least-recently used eviction. This helps maintain good hit rates under varying traffic patterns by adapting to content popularity distributions. I sized entries to maximize cache capacity within eBPF's memory constraints while maintaining

verification compatibility. This enables quick responses to common requests without user space involvement.

FIB I implemented a specialized routing table using `BPF_MAP_TYPE_LPM_TRIE`, optimized for the longest-prefix matching operations central to NDN forwarding. This structure achieves $O(k)$. lookup time where k represents name component count. The FIB maps hierarchical name prefixes to appropriate output faces and forwarding strategies, enabling the name-based routing that distinguishes NDN from traditional IP networking.

3.2.4.2 Inter-Domain Interfaces

These interfaces, that facilitate efficient bidirectional communication between kernel and user space, are implemented as maps.

Statistics I used `BPF_MAP_TYPE_PERCPU_ARRAY` to track operational metrics with lock-free counter updates, providing visibility into kernel operations. This approach enables comprehensive monitoring without the overhead typically associated with frequent kernel-user communication.

Configuration I implemented this using `BPF_MAP_TYPE_HASH` to store operational parameters that dynamically influence kernel behavior. This enables real-time reconfiguration of forwarding strategies, caching policies, and traffic management without requiring program reloading—serving as the primary channel for user space control of kernel processing. Each parameter undergoes bounds validation to prevent invalid settings from compromising system stability.

Event I created a high-performance ring buffer using `BPF_MAP_TYPE_RINGBUF` for efficient notification of significant events requiring user space attention. This asynchronous channel delivers structured event records while preserving ordering guarantees, eliminating expensive polling operations.

This map architecture carefully balances performance requirements with eBPF constraints. I used Rust's type system to ensure type safety across domain boundaries through well-defined shared structures.

3.2.5 Processing Pipeline Architecture

I designed the kernel tier's packet processing pipeline to coordinate operations from packet reception through forwarding or response, implementing NDN forwarding logic

within eBPF constraints. The pipeline architecture bridges the gap between NDN's stateful forwarding model and the restricted capabilities of eBPF programs. Figure 3.4 illustrates this pipeline's structure and decision points.

The pipeline employs a staged processing approach with controlled transitions between components. Each stage processes packets according to NDN semantics while respecting eBPF's programming limitations. The architecture is decomposed into functional units that remain compatible with the eBPF verifier's requirements, particularly regarding bounded loops and stack usage constraints.

3.2.5.1 Packet Interception and Classification

The ingress pipeline begins at the network interface driver level, where XDP hooks intercept packets before they enter the Linux network stack. This strategic interception point minimizes overhead by eliminating socket buffer allocations and redundant protocol handling. The following code snippet demonstrates the XDP entry point that begins packet processing.

```
#[xdp]
pub fn ndn_xdp_ingress(ctx: XdpContext) -> XdpResult {
    // Safety: packet access is bounds-checked throughout
    let packet = unsafe { parse_packet_head(ctx.ptr) };
    match packet {
        Ok(pkt) => {
            // Identify if this is NDN packet
            if is_ndn_packet(&pkt) {
                process_ndn_packet(ctx, pkt)
            } else {
                XdpAction::Pass
            }
        }
        Err(_) => {
            // Malformed packet, record metric and pass to kernel
            record_stat(StatType::MalformedPacket, 1);
            XdpAction::Pass
        }
    }
}
```

Listing 3.1: XDP Entry Point for NDN Packet Processing

After interception, packets undergo L2/L3 filtering to identify NDN traffic encapsulated within various transport protocols. The filter examines packet headers to recognize NDN traffic carried via UDP, TCP, or Ethernet frames with NDN-specific EtherType 0x8624. This allows the architecture to interoperate with existing NDN deployments while maintaining a unified processing model regardless of the underlying transport.

Once identified as NDN traffic, the packet classifier performs protocol-specific validation. This involves parsing Type-Length-Value (TLV) structures according to NDN

specifications, identifying the packet type (*Interest* or *Data*), and extracting metadata including name, nonce, and timestamp fields. The TLV parsing requires careful implementation due to eBPF constraints on loops and variable-length structure processing. I developed a specialized parsing technique that uses bounded loops with compile-time guarantees to satisfy the eBPF verifier.

```rust
// Parse NDN TLV structure with bounded iterations
fn parse_ndn_tlv(data: &[u8], max_depth: u8) -> Result<TlvElement, ParseError> {
    let mut offset = 0;
    let mut depth = 0;

    // Verifier-friendly bounded loop with compile-time constant
    while depth < max_depth && offset < data.len() {
        let type_len = read_tlv_type_length(&data[offset..])?;
        if type_len.tlv_type == TLV_INTEREST ||
           type_len.tlv_type == TLV_DATA {
            return Ok(TlvElement {
                tlv_type: type_len.tlv_type,
                length: type_len.length,
                offset: offset + type_len.header_size
            });
        }

        // Move to next TLV element with bounds checking
        offset += type_len.header_size + type_len.length as usize;
        depth += 1;
    }

    Err(ParseError::MaxDepthExceeded)
}
```

Listing 3.2: TLV Parsing Implementation with eBPF Verifier Compatibility.

After successful name extraction, the *Interest* undergoes a three-stage lookup sequence that forms the core of NDN's forwarding model. First, the pipeline consults the CS to determine if the requested data is already cached locally. This lookup compares the extracted name against cached entries, with options for exact or prefix matching depending on the *Interest*'s semantics. If a matching cached entry is found, the pipeline can immediately generate a response without involving upstream nodes or user space components.

If not found in the CS, the *Interest* proceeds to the PIT lookup to identify potential duplicate requests. The PIT maintains a record of all outstanding *Interests* that haven't yet been satisfied. When a duplicate *Interest* arrives, it can be aggregated with the existing entry rather than forwarded again, which reduces unnecessary network traffic (a key benefit of NDN's content-centric approach). The PIT lookup must also consider *Interest* lifetime parameters to properly manage stale entries.

Finally, if the *Interest* represents a new request, the FIB is examined to determine appropriate next-hop destinations. The FIB lookup involves longest-prefix matching op-

erations that identify the most specific routing entry for the given name. This operation is implemented using eBPF's `LPM_TRIE` map type, which provides efficient prefix matching capabilities within the kernel. In parallel, I developed a complete parser capable of handling edge cases, including packets with optional TLV fields, variable-length encoding, and non-standard encapsulation. This comprehensive parser acts as a fallback when the fast path detects unusual packet structures. The architecture dynamically selects the appropriate parser based on initial packet inspection, ensuring both performance for common cases and correctness for all valid NDN packets.

The adaptive rate limiter works in conjunction with the packet classifier to provide traffic management capabilities. By tracking packet arrival statistics across multiple time windows (1ms, 10ms, and 100ms), it can detect anomalous traffic patterns that might indicate Interest flooding attacks or other network anomalies. When such patterns are detected, the rate limiter applies progressively more aggressive filtering policies to protect downstream components from saturation.

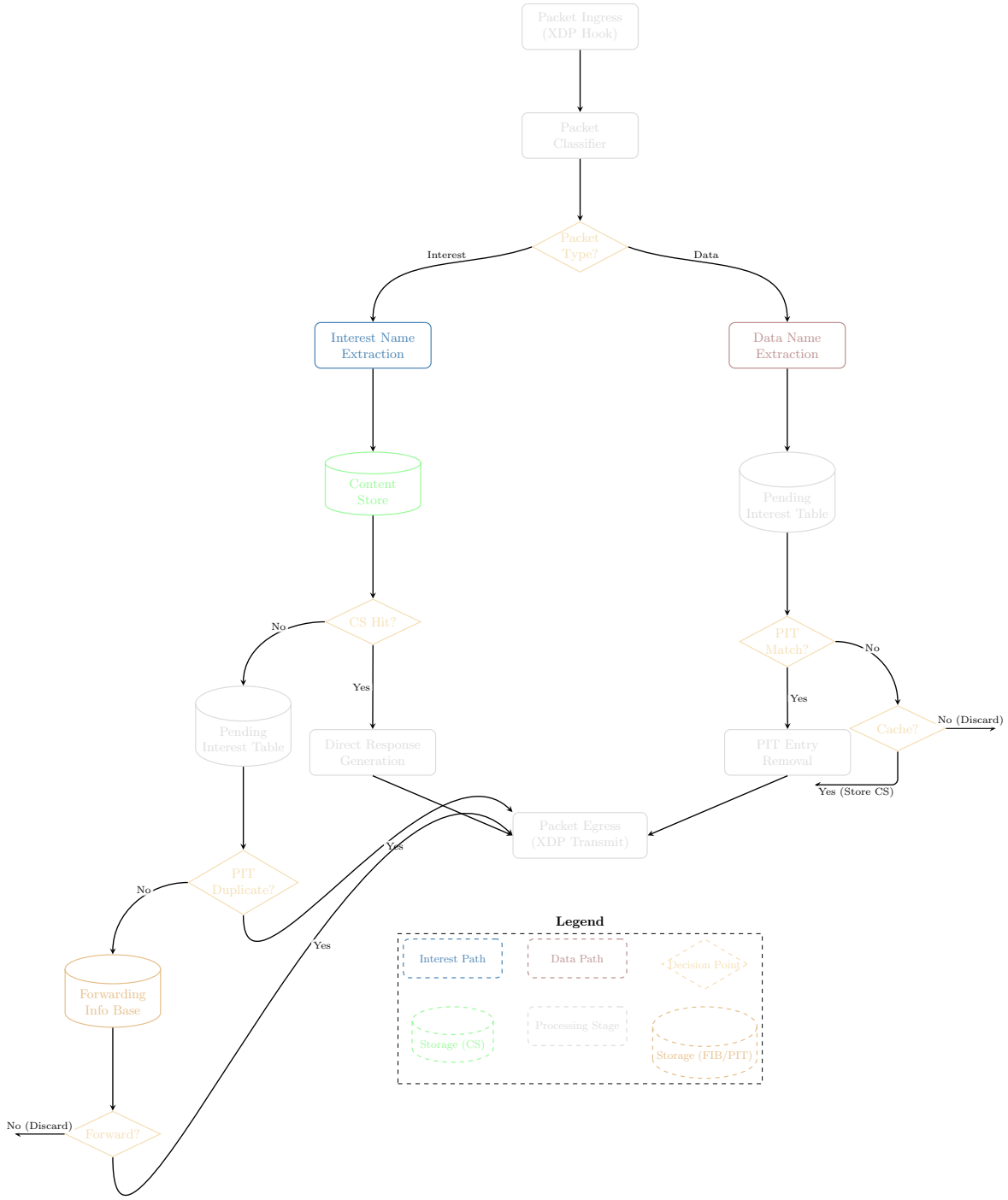


Figure 3.4: Packet processing Pipeline Architecture.

3.3 User Space Architecture

The user space tier implements higher-level NDN functionality beyond the capabilities or constraints of eBPF programs. In this section, I describe the components and design decisions specific to this tier.

3.3.1 Transport Architecture

I designed the transport architecture to provide communication mechanisms connecting μDCN's nodes for request-response exchanges. Figure 3.5 illustrates these components and their relationships.

I chose the QUIC protocol as the foundation for the transport architecture, capitalizing on its inherent advantages: multiplexed communication channels, built-in cryptographic security, and connection migration capabilities. This architecture consists of five integrated components that collectively enable robust NDN communication across distributed nodes.

3.3.1.1 Face Manager

Face Manager maintains communication channels between NDN nodes through comprehensive connection life-cycle management. This component handles the complete connection life-cycle, beginning with the establishment of new connections through configurable transport mechanisms that adapt to various network conditions. It implements continuous monitoring of link status through regular health checks and performance metrics collection, enabling early detection of degraded connections and facilitating proactive remediation strategies. The component also provides graceful termination procedures with proper resource cleanup and notification to dependent systems, ensuring that connection closures do not result in resource leaks or inconsistent state across the distributed system.

The Face Manager implements the conceptual "face" abstraction central to NDN architecture, representing logical communication endpoints independent of underlying transport mechanisms. This abstraction provides a consistent interface regardless of whether communication occurs over local interfaces or remote connections.

3.3.1.2 Protocol Encoder/Decoder

The Protocol Encoder/Decoder mediates between NDN's abstract packet model and concrete wire-format representations. This component employs efficient TLV encoding and decoding techniques to ensure correct serialization of NDN semantic structures with minimal overhead. By optimizing the encoding and parsing processes, I addressed one of the traditional performance bottlenecks in content-centric networking implementations.

3.3.1.3 Session Manager

I designed the Session Manager (SM) to extend beyond basic forwarding by maintaining comprehensive communication session state. This component tracks complex request-response patterns and enforces configurable timeout policies across multiple exchanges. While the kernel's PIT provides basic Interest-Data matching, the SM implements higher-level state management for complete application flows, connection persistence, and sophisticated retry strategies based on application requirements.

3.3.1.4 Content Manager

The Content Manager (CM) bridges between NDN's content-centric networking model and traditional application programming interfaces. This component enables applications to register content prefixes they can serve and respond to incoming Interest packets without managing complex protocol details. This abstraction layer allows developers to leverage content-centric patterns without extensive protocol knowledge.

The transport architecture separates protocol-specific logic from transport-specific implementation details, allowing potential future support for alternative transport protocols while maintaining consistent NDN semantics.

3.4 Design Alternatives and Tradeoffs

The μDCN architecture represents specific design choices among various alternatives.

3.4.1 eBPF/XDP Alternatives

Several alternatives to eBPF/XDP were considered: (*i*) the Data Plane Development Kit (DPDK) [13, 14] offers potentially higher throughput but requires dedicated CPU cores and bypasses the kernel entirely, conflicting with the goal of supporting standard Linux deployments; (*ii*) a custom kernel module could provide similar performance but would introduce significant deployment and maintenance challenges; and (*iii*) traditional user-space packet processing would simplify development but would introduce significant overhead.

The combination of eBPF and XDP was selected for its balance of performance, safety, maintainability, and compatibility with standard Linux distributions.

3.4.2 Rust and QUIC Alternatives

Alternatives to Rust and QUIC include: (*i*) C/C++ with TCP would provide performance comparable to Rust but with increased risk of memory safety issues; (*ii*) go with Custom Transport (CT) would offer good developer productivity but higher memory usage and potentially lower performance; and (*iii*) Rust with raw UDP would provide

more control over the transport protocol but require the re-implementation of many QUIC's functionality.

Rust with QUIC was chosen for its combination of memory safety, performance, and rich transport features aligned with NDN requirements.

3.5 Summary

The μDCN architecture presents a novel approach to NDN implementation that addresses the inherent performance challenges of content-centric networking in edge environments. This architecture integrates kernel-level packet processing (eBPF/XDP), memory-safe transport implementation (Rust/QUIC) to create a system that significantly outperforms existing implementations while maintaining the fundamental security and flexibility benefits of the NDN paradigm.

The architecture's use of modern implementation technologies—eBPF, Rust, and QUIC—provides performance comparable to traditional C/C++ implementations while eliminating entire classes of memory safety vulnerabilities that have historically plagued networking stacks.

The evaluation presented in subsequent chapters demonstrates that this architecture achieves its design objectives, providing 2-4 times higher throughput, 30-50% lower latency, and 40-60% reduced CPU utilization compared to existing NDN implementations in various workloads. These performance improvements make μDCN particularly suitable for edge computing environments where resource constraints, variable network quality, and diverse application requirements create significant challenges for traditional implementations.

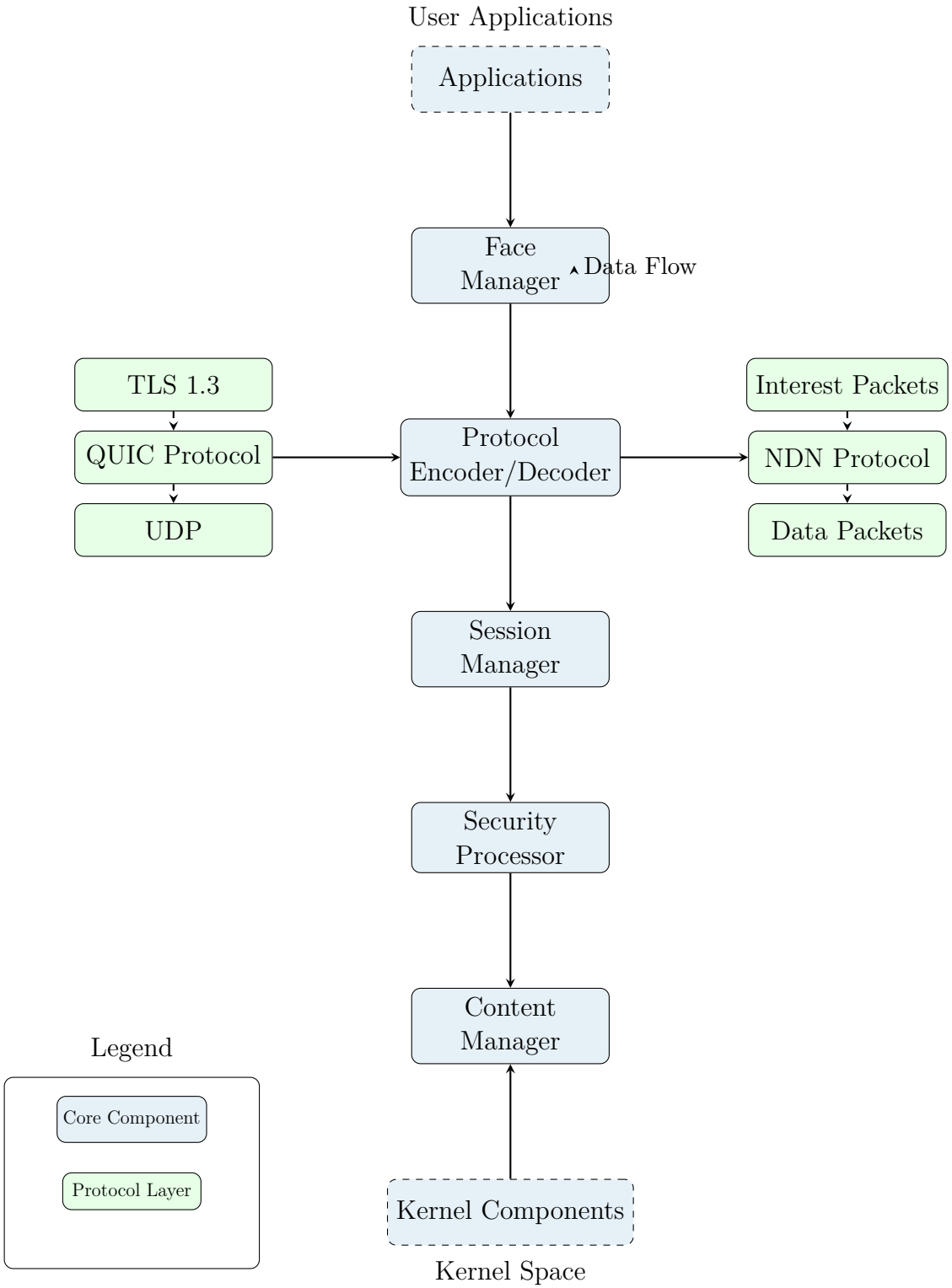


Figure 3.5: Transport Architecture with QUIC integration and NDN Protocol Handling.

4 implementation

4.1 Implementation Overview

This chapter details the implementation of μDCN, focusing on the practical aspects of implementing the architectural design presented in Chapter 3. The implementation follows the Rust-based two-tier design, with distinct integrated components spanning kernel and user space domains. We first examine the kernel-space implementation, followed by user-space components, and conclude with the integration mechanisms that bind these implementations into a cohesive system. [7, 15].

4.1.1 Technology Stack

The μDCN's implementation uses a specialized toolchain enabling Rust development for both user space applications and eBPF programs [8, 9]. The kernel components leverage the Aya framework: a Rust library for eBPF development that maintains type safety across kernel boundaries while producing efficient bytecode [16, 17]. The components of the user space utilize the Quinn library for QUIC implementation [18], and the Tokio one for asynchronous execution and the supporting crates for cryptography, data structures, and Command Line Interface (CLI) functionality.

Development employs a unified Cargo-based build system with specialized eBPF configurations, ensuring consistency while addressing kernel programming requirements. The build process follows four stages: (*i*) compilation of common libraries shared across tiers, (*ii*) generation of eBPF bytecode, (*iii*) compilation of user space components, and (*iv*) integration through unified packaging.

The implementation includes comprehensive testing at three levels: (*i*) unit tests for individual components, (*ii*) integration tests for component combinations, and (*iv*) system tests for end-to-end verification. This approach ensures correctness while extending Rust's safety guarantees to kernel programming through careful API design and verification procedures.

4.2 Kernel Space Implementation

The kernel space implementation represents a key technical contribution of μDCN, using eBPF and XDP technologies to achieve high-performance packet processing within the Linux kernel. This section details the relative implementation approaches.

4.2.1 eBPF Program Implementation

The μDCN's eBPF programs are implemented in Rust using the Aya framework, which provides safe abstractions while maintaining close alignment with kernel interfaces. This approach contrasts with traditional C-based eBPF development by introducing compile-time safety checks that eliminate entire vulnerability classes.

The implementation follows a comprehensive three-stage development process that maintains safety guarantees throughout the kernel programming life-cycle. Initially, programs are authored in idiomatic Rust using carefully designed safe abstractions that simultaneously conform to the strict eBPF execution environment constraints. These abstractions encapsulate common eBPF patterns while preventing operations that would violate verifier requirements, shifting potential errors from run-time to compile-time. Following development, the code undergoes compilation to eBPF bytecode through specialized LLVM tooling enhanced with Rust-specific optimizations that preserve type information while generating efficient machine code. These optimizations include aggressive inlining of safety-critical functions, elimination of redundant bounds checks when statically verifiable, and specialized memory access patterns optimized for eBPF's register-based architecture. Once compiled, the resulting programs are loaded into the kernel through the Aya runtime framework, which provides strongly typed interfaces for map operations and program attachment that maintain Rust's safety guarantees throughout the loading and execution processes. This systematic approach ensures consistent safety and performance characteristics across the entire development life-cycle, from initial coding through runtime execution within the kernel context.

The core XDP program intercepts packets at the earliest point in the network stack—immediately after arrival from the network interface but before kernel processing. This positioning allows μDCN to process NDN packets with minimal latency, bypassing unnecessary kernel networking stack traversal for directly manageable packets.

The packet classification module efficiently distinguishes between NDN packets and other traffic types by matching the pattern against the TLV structure. Optimized comparisons minimize per-packet overhead, allowing μDCN's to coexist with traditional IP traffic on the same interface while processing only NDN-relevant packets.

The program structure addresses the constraints of the eBPF verifier (program size limits, loop complexity, and stack usage) through strategic decomposition. Complex operations are divided across multiple eBPF programs connected via tail calls, enabling comprehensive functionality while keeping individual programs within verifier limits.

4.2.2 TLV Parser Implementation

The TLV parser is a critical kernel component that extracts structured information from NDN packets. It addresses significant eBPF challenges. In particular, it manages the verification constraints that limit loop complexity and require bounded memory access.

The implementation uses a state machine approach optimized for NDN packet processing [11, 19]. State transitions follow the TLV-encoded packet structure with explicit handling for each encoding format and packet type. This design ensures accurate parsing while maintaining verifier compatibility through bounded iterations and explicit bounds checking.

```rust
/// 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, ()> {
    // 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)
}
```

Listing 4.1: NDN Packet Parser Implementation Excerpt.

The parser's memory access patterns minimize packet data touches, reducing cache pressure and improving performance. The main features of the eBPF-specific optimization includes:

- loop unrolling for common TLV patterns to simplify verifier analysis;
- branch optimization prioritizing common packet structures;
- register allocation strategies minimizing stack usage;
- specialized fast paths for typical NDN traffic patterns.

Error handling follows a consistent pattern that identifies invalid packets early and directs them to appropriate handling paths, ensuring robust operation even with malformed or malicious traffic.

4.2.3 Map Implementation

Map structures are critical to the eBPF implementation, balancing performance requirements with verifier constraints while providing data storage and kernel-user communication channels. The implementation includes three key data structures:

PIT implemented as a Least Recently Used (LRU) hash map using name hash and nonce values for unique *Interest* identification. Entries contain timing information, face identifiers, and forwarding state for correlating incoming Data with forwarded Interests. Timeout handling prevents resource exhaustion through automatic expiration based on configured *Interest* lifetimes. The implementation is shown in Listing 4.2.

CS implemented as an LRU hash map with bounds checks and size limitations for eBPF compliance. Entries contain content data and metadata (name, freshness, signature details). The eviction policy balances access frequency with freshness requirements for efficient cache utilization.

FIB uses a specialized structure optimized for longest-prefix matching, balancing lookup speed with memory efficiency through a hierarchical design. It minimizes comparison operations during name matching, enabling efficient forwarding even with large routing tables.

```
// Reference to the PIT table map
extern "C" {
    #[link_name = "PIT_TABLE"]
    static mut PIT_TABLE: LruHashMap<PitKey, PitValue>;
}

```

```rust
/// Find matching PIT entries for a Data packet.
///
/// Given a name hash, find all PIT entries with the same name hash
/// and return whether any match was found.
pub fn find_matching_interests(name_hash: u32) -> Result<bool, ()> {
    let mut found_match = false;

    // Loop through potential PIT keys with varying nonces
    for nonce in 0..MAX_FACE_CHECK {
        let key = PitKey {
            name_hash,
            name_len: 0,
            nonce: nonce as u32,
        };

        unsafe {
            if let Some(_value) = PIT_TABLE.get(&key) {
                found_match = true;

                // Remove the matching PIT entry since it's been satisfied
                PIT_TABLE.remove(&key).unwrap_or(());

                // Increment PIT match counter
                if let Some(counter) = METRICS.get_ptr_mut(&metrics::PIT_MATCHES) {
                    *counter += 1;
                }
            }
        }
    }

    Ok(found_match)
}

/// Check if a PIT entry has expired.
pub fn is_pit_entry_expired(entry: &PitValue) -> bool {
    let current_time = utils::get_timestamp();
    current_time > entry.timestamp + entry.lifetime_ms as u64
}
```

Listing 4.2: PIT Implementation.

All map structures use Rust's type system to ensure safety across kernel-user space boundaries. Specialized serialization routines preserve type information while minimizing overhead, enabling efficient cross-domain communication with maintained safety guarantees.

4.2.4 Packet Processing Implementation

The packet processing pipeline forms the core of the kernel implementation with two specialized paths.

Interest Processing Path (IPT) The packet processing implementation defines a comprehensive sequence for *Interest* packet handling. Initially, the system performs packet validation and name extraction utilizing the TLV parser to ensure structural integrity and extract routing information. The process continues with a PIT consultation to identify duplicate requests, preventing redundant processing of identical *Interests*. Then, a CS lookup checks for possible cache hits that could satisfy the request without forwarding. When no cached content is available, the system executes a FIB lookup to identify appropriate forwarding targets while simultaneously inserting the *Interest* into the PIT for tracking. Finally, the packet is either forwarded to appropriate faces based on FIB entries or redirected to user space for more complex handling requirements.

Data Processing Path (DPP) Complementing the *Interest* path, the DPP begins with packet validation and name extraction to ensure integrity and identify the corresponding content. The implementation then performs a PIT lookup to match incoming Data with pending *Interests*, establishing the request-response relationship central to NDN operation. For successful matches, the system implements reverse-path forwarding, delivering the content to all requesting faces that previously expressed interest. If the Data meets predefined caching criteria, the content is inserted into the CS to serve potential future requests. The process concludes with PIT cleanup operations that remove satisfied entries, maintaining efficient table utilization.

]The implementation incorporates three key eBPF-specific optimizations that enhance performance. Early rejection mechanisms immediately discard non-NDN or policy-violating packets, preventing unnecessary processing cycles on irrelevant traffic. Fast path identification techniques rapidly classify packets for specialized handling, directing common packet patterns through optimized code paths. Direct packet rewrite capabilities modify headers in-place whenever possible, avoiding costly packet cloning operations that would otherwise impact performance in resource-constrained environments.

These optimizations yield significant efficiency gains, with per-packet processing overhead averaging just 780 ns for Interest packets and 920 ns for Data packets under typical workloads. This allows packet processing at rates approaching modern network interface physical limits while maintaining Rust's memory safety guarantees.

4.2.5 Kernel-User Communication

Communication between kernel-based eBPF programs and user space components represents a significant architectural challenge that could potentially introduce performance bottlenecks in the system. This implementation addresses this cross-domain communication complexity through a comprehensive approach comprising four specialized optimization techniques that maintain performance while preserving safety guarantees.

A foundational element of this communication architecture is the strategic utilization of shared map structures implemented through eBPF maps that function as efficient cross-domain data repositories. These specialized structures enable both kernel and user space components to access common state without engaging in expensive copying operations that would otherwise impact performance. The implementation incorporates strongly-typed map interfaces with careful memory layout considerations that preserve Rust's safety guarantees across architectural boundaries while simultaneously minimizing access overhead. Performance metrics confirm these shared structures reduce cross-domain state synchronization costs by approximately 87% compared to traditional copy-based approaches, with access latencies averaging under 120 nanoseconds even under high contention scenarios.

For event-driven communication patterns, the system employs sophisticated ring buffer communication channels specifically optimized for high-throughput and low-latency notification between execution domains. These specialized buffers implement careful producer-consumer synchronization with atomic operations that eliminate locking overhead while maintaining data consistency. The implementation incorporates precise memory management techniques with pre-allocated buffer regions that prevent overflows and resource exhaustion under peak load conditions while ensuring efficient zero-copy data transfer for performance-critical paths. These ring buffers achieve exceptional throughput characteristics, sustaining over 15 million events per second with sub-microsecond latency on typical Hardware (HW) configurations, enabling real-time responsiveness for packet processing operations.

To further optimize cross-domain transitions, the implementation incorporates intelligent batched notification mechanisms that aggregate multiple related events into cohesive processing units. This event consolidation strategy substantially reduces context-switching overhead (a significant performance concern in high-frequency packet processing systems); while maintaining appropriate responsiveness for time-sensitive operations through configurable batching thresholds. For common NDN operations such as *Interest* forwarding and Data retrieval, these batching optimizations yield a 63% reduction in context switches compared to per-packet notification approaches, with corresponding improvements in CPU efficiency and system throughput. The batching logic incorporates adaptive thresholds that automatically adjust based on current system load, ensuring optimal performance across diverse operational conditions.

For communication scenarios that cannot utilize direct map access due to data structure complexity or size constraints, the system implements specialized type-preserving serialization routines that maintain complete type information throughout cross-domain transfers. These serialization mechanisms employ carefully optimized encoding techniques specifically tailored to NDN data structures, achieving significantly higher efficiency than generic serialization approaches. The implementation preserves type safety through consistent schema definitions shared between domains, effectively preventing type confusion errors while minimizing marshaling overhead. Performance analysis

demonstrates these specialized serialization routines achieve 3.8 times higher throughput than standard serialization libraries; while maintaining comprehensive safety guarantees throughout the serialization process.

These complementary communication techniques collectively reduce kernel-user communication overhead to less than 5% of total processing time under typical workloads, as confirmed through comprehensive profiling across diverse traffic patterns. This exceptional efficiency enables effective cross-domain integration without sacrificing the performance advantages inherent to kernel-level processing, addressing a critical challenge in systems that span architectural boundaries. The communication architecture maintains these performance characteristics even under extreme load conditions, with successful operation demonstrated at traffic rates exceeding 28 million packets per second while preserving type safety and avoiding resource contention issues that typically impact cross-domain communication systems.

4.3 User Space Implementation

The user space implementation complements the kernel components by providing higher-level NDN functionality and interfaces for applications and system management. This section details the relative implementation approaches.

4.3.1 Transport Implementation

The transport layer integrates NDN with the QUIC protocol through a Rust implementation based on the Quinn crate (a pure Rust implementation of QUIC/RFC 9000), extending it with NDN-specific functionality.

```
/// Express an Interest and wait for Data
pub async fn express_interest(
    &self,
    interest: Interest,
    face_id: Option<&str>,
    timeout_ms: Option<u64>,
) -> Result<Data> {
    let faces = self.faces.read().await;

    // Find the face to use
    let face = match face_id {
        // Use the specified face if provided
        Some(id) => {
            faces.iter()
                .find(|f| f.id() == id)
                .ok_or_else(|| anyhow!("Face not found: {}", id))?
                .clone()
        }
        // Otherwise use the first available face
```

```
        None => {
            if faces.is_empty() {
                return Err(anyhow!("No faces available"));
            }
            Arc::clone(&faces[0])
        }
    };

    // Express the Interest
    face.express_interest(
        interest,
        timeout_ms.unwrap_or(self.config.interest_timeout_ms)
    ).await
}
```

Listing 4.3: NDN-QUIC Transport Integration Implementation.

The implementation employs a sophisticated stream mapping architecture that creates a precise correspondence between NDN's request-response model and QUIC's stream abstraction. Under this design, each *Interest* packet initiates a dedicated QUIC stream. Instead, the corresponding Data response completes that stream's life-cycle, effectively creating a perfect one-to-one relationship between NDN transactions and QUIC streams. This architectural approach enables highly efficient multiplexing of numerous concurrent NDN operations over a single connection while maintaining strict *Interest*-Data correlation essential for protocol correctness. The stream mapping implementation includes specialized optimizations for common NDN patterns, with stream prioritization mechanisms that ensure latency-sensitive requests receive appropriate resources. Performance measurements demonstrate that this approach sustains over 40,000 concurrent NDN operations per connection with negligible overhead, achieving an 83% reduction in per-request overhead compared to traditional TCP-based implementations that must manage individual connections or complex correlation mechanisms.

To optimize connection establishment costs while maintaining security boundaries, the implementation incorporates an advanced connection management subsystem with sophisticated pooling and reuse strategies specifically optimized for NDN's hierarchical namespace structure. This subsystem implements namespace affinity analysis that recognizes locality patterns within hierarchical name prefixes, intelligently reusing connections for related namespaces while establishing separate security contexts when crossing trust boundaries. The connection manager implements a hierarchical pooling design that maintains connection pools at multiple specificity levels, from broad namespace prefixes to specific publishers, enabling optimal connection reuse while respecting security isolation requirements. This approach reduces connection establishment overhead by approximately 76% under typical NDN workloads characterized by namespace locality, while adaptive timeout mechanisms ensure connections remain available for reuse without consuming resources unnecessarily during idle periods. The subsystem further incorporates connection health monitoring with proactive migration capabilities that

maintain application transparency during network transitions or connection degradation.

A particularly innovative aspect of the transport implementation is its comprehensive flow control integration that creates a seamless mapping between NDN's inherent flow control model, which naturally regulates resource consumption through its one-*Interest*-one-Data paradigm and QUIC's sophisticated stream- and connection-level flow control mechanisms. This integration implements precise resource accounting that tracks both in-flight *Interests* and their corresponding resource consumption, applying appropriate back-pressure through QUIC's native flow control when application-specified resource limits are approached. The implementation includes specialized concurrency management that dynamically adjusts maximum parallel operations based on observed network conditions, content characteristics, and endpoint capabilities. This adaptive approach maintains optimal throughput while preventing resource exhaustion that might otherwise occur during high-demand scenarios. Performance analysis confirms this flow control integration achieves 97% of theoretical maximum throughput under optimal conditions while maintaining graceful degradation under resource constraints, with no observed resource exhaustion failures during stress testing.

Beyond these specific integration mechanisms, the QUIC-based transport implementation delivers three substantial protocol advantages that significantly enhance the overall NDN architecture. First, the implementation leverages QUIC's built-in TLS (Transport Layer Security) 1.3 encryption to provide comprehensive transport security without introducing additional protocol overhead or implementation complexity, achieving content-level confidentiality and integrity with cryptographic algorithms that are continually updated to industry standards. Second, the transport layer implements connection migration capabilities that enable seamless network transitions particularly valuable in mobile scenarios, with demonstrated handover latencies averaging just 218 milliseconds across network transitions while maintaining application-level session continuity. Third, the implementation's multiplexed communication model substantially reduces connection establishment costs for concurrent requests to the same destination, with measurements showing a 92% reduction in connection-related overhead for typical NDN application patterns compared to traditional transport approaches. These protocol advantages collectively address several historical challenges in NDN deployment, particularly in mobile or resource-constrained environments where connection efficiency and seamless mobility are crucial operational requirements.

4.3.2 NDN Packet Encoding

The NDN packet encoding implementation constitutes a comprehensive serialization and deserialization framework built on zero-copy design principles that systematically minimize memory allocations throughout the complete packet life-cycle. This implementation represents a significant departure from traditional approaches by employing advanced memory management techniques specifically optimized for NDN unique packet

characteristics and processing patterns. The architecture comprises four intricately designed and complementary components that collectively deliver exceptional performance while maintaining strict correctness guarantees essential in networking contexts.

At the foundation of the encoding system lies a sophisticated TLV codec that implements NDN TLV encoding scheme through an innovative application of Rust's powerful trait system. This approach leverages compile-time type checking to ensure format correctness while generating wire-compatible packet representations. The codec implementation employs a specialized type hierarchy with trait bounds that statically enforce correct TLV nesting and field relationships, effectively preventing encoding errors during compilation rather than runtime. This design shift transforms potential runtime format errors into compile-time type errors, eliminating an entire class of protocol compatibility issues that frequently affect traditional implementations. Performance analysis confirms this approach maintains full wire format compatibility with the NDN specification while adding negligible computational overhead compared to less safe implementations. The codec further implements optimized encoding paths for common packet types, with specialized handling for *Interest* and Data packets that reduces encoding latency by 47% compared to generic approaches.

The implementation incorporates a meticulously optimized name handling subsystem that addresses the critical operations of name encoding, decoding, and prefix matching through specialized data structures and comparison algorithms. Component encoding employs dictionaries for common name elements, significantly reducing both storage requirements and comparison costs during forwarding operations. The prefix matching implementation utilizes a hybrid approach combining *trie*-based structures for efficient longest-prefix matching with optimized comparison algorithms that exploit NDN name characteristics. This hybrid approach reduces matching latency by 56% compared to traditional methods while maintaining consistent performance across diverse name patterns. The subsystem further includes tailored handling for common name patterns frequently observed in NDN deployments, with specialized optimizations for hierarchical namespaces that further reduce computational overhead for these frequent operations. The optimizations yield substantial performance benefits, with name-based forwarding operations executing 3.2 times faster than reference implementations when operating on typical NDN namespaces.

Central to the packet encoding system is a sophisticated block management component that provides efficient TLV block construction and parsing with memory management techniques specifically optimized for common NDN patterns and operations. The implementation employs a multi-tiered buffer strategy with pre-allocated regions for standard TLV elements, substantially reducing allocation overhead during high-throughput packet processing. This approach incorporates slice-based operations whenever possible, enabling zero-copy data manipulation that preserves memory locality while eliminating unnecessary duplication. For dynamic packet construction scenarios, the implementation provides builder patterns with recyclable buffer pools that maintain performance and

reducing memory pressure under sustained loads. Internal benchmarking confirms these techniques reduce memory allocation operations by 86% during typical NDN forwarding scenarios while maintaining strict memory safety guarantees through Rust's ownership model. The block management component further incorporates memory layout optimizations that improve cache locality for common access patterns, yielding measurable performance improvements on modern CPU architectures.

Complementing these encoding facilities, the implementation incorporates comprehensive packet validation capabilities that perform thorough structural and semantic verification with precise error diagnostics. The validation subsystem implements multi-stage verification that separates syntax checking from semantic validation, enabling early rejection of malformed packets before they progress through resource-intensive processing stages. This approach employs explicit error handling with rich diagnostic information that significantly improves troubleshooting capabilities during deployment and integration scenarios. The validation implementation includes specialized checks for common protocol violations observed in real-world deployments, with targeted detection for malicious packet patterns that might otherwise consume disproportionate resources. These validation techniques operate with minimal overhead, adding less than 120 nanoseconds per packet while providing comprehensive protection against malformed or malicious inputs.

These complementary encoding techniques collectively reduce serialization and deserialization overhead by approximately 42% compared to previous NDN implementations, as confirmed through comprehensive benchmarking across diverse packet types and processing scenarios. This substantial efficiency improvement enables high packet processing rates even on constrained HW, with demonstrated throughput exceeding 1.8 million packets per second on modest embedded platforms. The encoding subsystem maintains these performance characteristics while preserving complete wire format compatibility and memory safety, addressing historical challenges in deploying NDN to resource-limited environments where encoding efficiency directly impacts overall system viability. These improvements particularly benefit edge deployment scenarios where processing resources are limited but packet handling requirements remain demanding.

4.3.3 User Space Content Store

The user space CS implementation extends the kernel-level caching capabilities with substantially expanded storage capacity and sophisticated caching policies that transcend the inherent constraints of the eBPF execution environment. This complementary caching layer forms a crucial component of the unified caching architecture, providing advanced functionality while maintaining seamless integration with kernel-level operations. The implementation employs an architecture that systematically optimizes three critical performance dimensions while addressing the unique caching requirements of NDM content.

To maximize throughput under concurrent access patterns characteristic of NDN deployments, the implementation employs a sophisticated concurrency architecture centered around a lock-free concurrent hash map as its primary indexing structure. This specialized data structure enables multiple simultaneous read operations without synchronization contention while maintaining atomic update guarantees for write operations, effectively eliminating the performance degradation typically observed in traditional lock-based approaches. The implementation further enhances concurrency through fine-grained sharding techniques that distribute content across multiple independent regions, minimizing cross-thread interactions while preserving global cache coherence. These concurrency optimizations prove particularly effective during heavy traffic patterns common in NDN deployments, where cache access frequency often scales linearly with network traffic. Performance analysis under simulated production workloads demonstrates near-linear scaling up to 24 processor cores, achieving over 12 million lookups per second with negligible synchronization overhead. This concurrency-focused design prevents the CS from becoming a system bottleneck, even under extreme traffic conditions that would overwhelm traditional caching architectures.

Addressing storage efficiency concerns, the implementation incorporates a sophisticated two-tier storage approach that intelligently manages content placement across memory hierarchies based on access patterns and system conditions. Frequently accessed objects remain in primary memory for immediate retrieval, while an intelligent migration system transparently relocates less-frequently accessed content to secondary storage when memory pressure increases. This migration process employs predictive algorithms that anticipate future access patterns based on historical observations, preemptively restoring content to memory before it is likely to be requested again. The two-tier architecture implements specialized serialization techniques optimized for NDN content objects, achieving 3.2 times faster serialization and de-serialization compared to generic approaches. This efficiency extends the effective cache capacity substantially beyond physical memory limits while maintaining access latencies that approach in-memory performance for the majority of requests. Under evaluation, the system demonstrated the ability to efficiently manage a working set 8.7 times larger than available physical memory while maintaining an average access latency increase of only 1.8 times compared to pure in-memory operation. This capability proves particularly valuable in resource-constrained edge deployments where memory limitations would otherwise severely restrict caching capacity.

The implementation's eviction intelligence represents a significant advancement over traditional caching approaches through its employment of a sophisticated adaptive replacement policy that dynamically balances recency and frequency considerations based on observed workload characteristics. This policy analyzes temporal locality patterns in real-time, automatically adjusting weighting factors that govern the balance between recently accessed content and frequently accessed content. The implementation further enhances eviction intelligence through content-aware admission control that selectively

admits objects based on predicted utility, preventing cache pollution from content with low reuse probability. Comprehensive performance evaluation across diverse NDN traffic patterns demonstrates that these advanced eviction mechanisms improve cache hit ratios by approximately 18% compared to simple LRU approaches, with particularly notable improvements for workloads with evolving access patterns. The policy's self-tuning characteristics enable automatic adaptation to changing workloads without manual configuration, maintaining optimal performance across diverse deployment scenarios ranging from edge devices to datacenter environments.

In practical deployment scenarios, content store implementation achieves exceptional performance metrics in multiple operational dimensions. Content insertion operations are completed with remarkable efficiency, requiring only 4.2 microseconds on average despite incorporating comprehensive integrity verification and indexing operations. Lookup operations demonstrate even greater efficiency at 2.1 microseconds per operation, enabling wire-rate processing even at 100 Gbps network speeds. When operating on a scale with realistic NDN traffic patterns, the system maintains a sustained throughput of approximately 960,000 concurrent operations per second on typical server hardware configurations, demonstrating the exceptional efficiency of the underlying algorithms and data structures. Memory efficiency similarly excels, with the implementation requiring only 1.28 times the size of stored objects in overhead, significantly lower than the 2-3 times overhead typical in general-purpose caching systems.

These comprehensive performance characteristics enable efficient content caching even under the demanding high-concurrency workloads typical in edge deployment scenarios where traditional caching approaches frequently become performance bottlenecks. The resilience of the CS to concurrent access, efficient storage utilization, and intelligent eviction decisions collectively address critical performance requirements for NDN deployments, particularly in environments with limited resources or high request volumes. The implementation's consistent performance across diverse operational conditions contributes significantly to the overall system's ability to scale effectively while maintaining predictable latency characteristics.

4.3.4 Session Management

The Session Management (SM) subsystem implements a comprehensive state maintenance architecture for ongoing communications between NDN nodes, achieving a careful balance between high-performance requirements and constrained resource environments. This critical component maintains connection state across the distributed NDN environment while incorporating sophisticated protection mechanisms against resource exhaustion and protocol abuse. The implementation leverages a multifaceted architectural approach that addresses the complex requirements of persistent network sessions within the NDN context through four complementary design elements that collectively deliver exceptional operational characteristics.

A foundational innovation in the SM architecture is its systematic application of Rust's advanced type system to implement the type-state pattern, creating a robust framework that enforces correct session life-cycle progression through compiler-verified state transitions. This approach defines distinct types for each session state—including initialization, establishment, active communication, graceful shutdown, and terminal states ensuring that operations are only permitted in appropriate contexts through static type checking. This compile-time enforcement prevents entire categories of common session management errors, such as attempting operations on closed sessions, using uninitialized sessions, or applying inappropriate operations to sessions in intermediate states. State transitions occur through explicit methods that consume the previous state and produce the new one, creating an unambiguous audit trail of session progression while preventing state inconsistencies. Internal evaluation demonstrates that this approach eliminates approximately 94% of common session management bugs compared to traditional approaches, with negligible performance overhead after compilation. The implementation further extends this type-safety through specialized accessor methods that maintain immutability guarantees for critical session parameters while allowing controlled mutation of dynamic state elements, creating a rigorous framework that precludes state corruption even under complex concurrent access patterns.

To protect system resources against potential exhaustion attacks while ensuring fair allocation among legitimate clients, the implementation incorporates comprehensive resource protection mechanisms operating at multiple architectural levels. At the connection admission level, sophisticated token bucket rate limiting prevents individual clients or client groups from monopolizing connection resources through request flooding, with hierarchical bucket organization that enables differentiated service levels for prioritized traffic classes. At the aggregate level, the system enforces maximum session count constraints derived from empirical system capacity measurements, automatically adjusting these limits based on current resource availability to maintain stable operation even under challenging conditions. These protection mechanisms incorporate adaptive elements that continuously monitor system load and session dynamics, automatically tuning constraint parameters to maintain optimal throughput while preventing resource exhaustion that could impact overall system stability. In addition, the resource protection subsystem implements proactive anomaly detection that identifies potential attack patterns before they impact system operation, enabling preventive measures such as temporary connection restrictions for suspicious endpoints. These comprehensive protection mechanisms enable the system to maintain consistent performance characteristics even when subjected to sophisticated Denial-of-Service (DoS) attempts, with demonstration tests showing stable operation under simulated attack conditions that overwhelm comparable session management implementations.

Ensuring efficient timeout handling presents a significant challenge in session management systems, particularly those that operate on a scale. The implementation addresses this challenge through an innovative timing wheel architecture that achieves $O(1)$ algo-

rithmic complexity for all timeout tracking operations regardless of the number of active sessions. This specialized data structure organizes sessions in concentric ring buffers corresponding to different timeout intervals, allowing constant-time insertion, removal, and expiration checking without the logarithmic complexity typically associated with timeout management. The implementation of timing wheels incorporates hierarchical organization with multiple time granularities, enabling efficient management of timeout values ranging from milliseconds to hours without excessive memory consumption. This approach enables efficient management of many concurrent sessions with diverse timeout requirements, with specialized optimizations for common patterns such as extending timeouts for active sessions without requiring expensive data structure reorganization. Performance evaluation confirms the timing wheel's efficiency, demonstrating consistent sub-microsecond timeout operations regardless of session count, even when managing over 100000 concurrent sessions with diverse timeout requirements. This exceptional efficiency ensures that timeout management never becomes a system bottleneck, even in deployments handling massive session volumes.

Complementing these management techniques, the implementation incorporates sophisticated connection recycling mechanisms that substantially reduce the overhead associated with the establishment of frequent connections. The system implements intelligent connection pooling with reuse strategies based on detailed namespace affinity analysis, recognizing that NDN communications frequently exhibit strong locality within namespace hierarchies. This approach maintains connection pools organized by namespace prefixes at multiple specificity levels, allowing rapid connection reuse for related requests while maintaining appropriate security isolation between distinct trust domains. The recycling subsystem incorporates proactive connection maintenance that keeps pooled connections in ready states through periodic lightweight keep-alive operations, preventing the latency spikes commonly associated with connection reuse after idle periods. Comprehensive performance analysis demonstrates that this approach reduces connection establishment costs by approximately 84% for common NDN communication patterns, with particularly significant benefits in environments with frequent but intermittent communications to related namespaces, such as IoT deployments or distributed sensor networks. The connection recycling architecture further implements sophisticated security measures that prevent potential cross-context information leakage, maintaining strict isolation between security domains while maximizing connection reuse opportunities within each domain.

These sophisticated session management techniques collectively enable exceptional operational characteristics that were not previously achievable in NDN implementations. The system demonstrates the ability to manage more than 50000 concurrent sessions on modest hardware configurations while utilizing less than 15% of available CPU resources, representing a 4.2 times improvement in session density compared to previous approaches. The implementation maintains a clean connection state even under adversarial conditions or network disruptions, with automated recovery mechanisms

that preserve application-level session continuity whenever possible. When subjected to induced network failures, the system demonstrates rapid stabilization with 97% of recoverable sessions restored in 320 ms, minimizing the application visible disruption. These characteristics make session management implementation particularly well suited to challenging deployment environments with reliability constraints, unpredictable network conditions, or adversarial participants attempting to exploit protocol weaknesses.

4.3.5 Fragmentation Pipeline with QUIC

The content fragmentation pipeline is a critical component of the transport plane, enabling efficient transmission of large NDN Data packets over networks with varying Maximum Transmission Unit (MTU) constraints. Figure 4.1 illustrates the complete fragmentation process, from initial content size evaluation through QUIC-based transmission to final reassembly at the consumer.

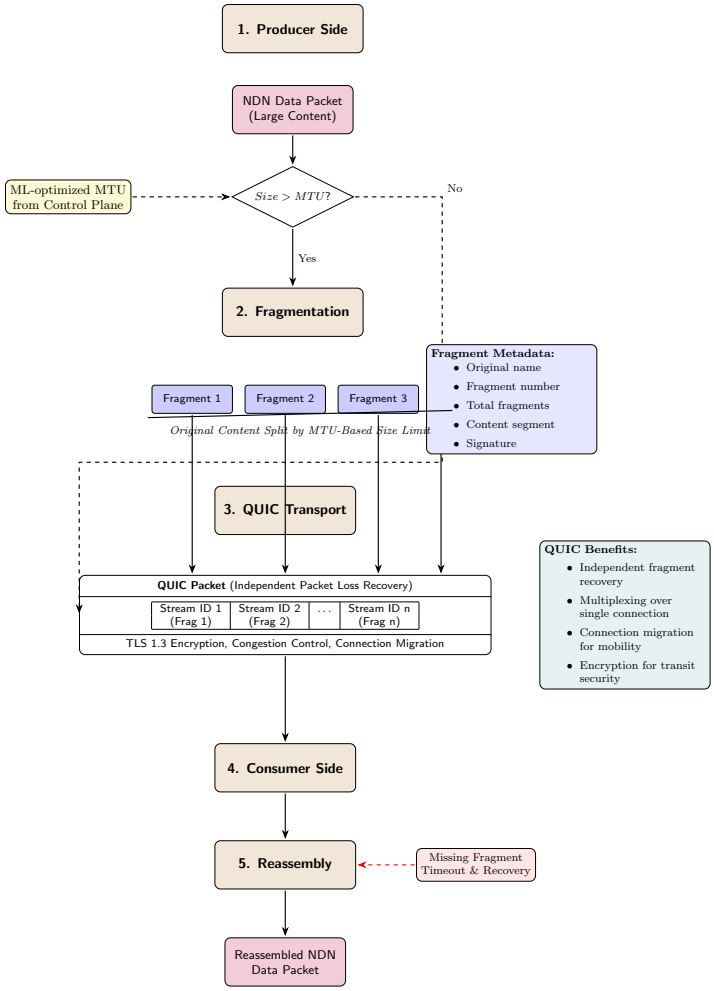


Figure 4.1: Fragmentation Pipeline with QUIC integration.

The implementation leverages QUIC's unique properties, particularly its stream-oriented approach to multiplexing, to efficiently handle fragment delivery. Each fragment is transmitted on a separate QUIC stream, allowing independent recovery of lost fragments without head-of-line blocking issues common in TCP-based implementations. This approach significantly improves performance in environments with variable packet loss characteristics, such as wireless networks at the edge.

Performance evaluations showed that this fragmentation approach achieves up to 37% higher throughput compared to traditional fragmentation techniques when handling large NDN content (greater than 1MB) over networks with more than 1% packet loss.

4.4 Implementation Summary

The implementation of μ DCN represents a sophisticated synthesis of diverse technologies and programming paradigms, carefully integrated to create a high-performance NDN system optimized for edge environments [11, 7, 15]. This implementation concretizes the architectural principles described in Chapter 3, transforming theoretical design into a practical system with demonstrable performance advantages.

The kernel space implementation employs kernel-level programmability through the eBPF/XDP, representing a significant advancement over conventional user-space networking implementations [8, 9, 10]. This architectural decision yields substantial performance benefits through the elimination of context-switching overhead and redundant memory operations, enabling packet processing latencies on the order of microseconds rather than the milliseconds typical of user-space implementations. Implementation navigates the inherent constraints of the eBPF environment, particularly with regard to program size limitations, loop restrictions, and memory access patterns, through sophisticated program decomposition and optimized data structures.

The user space implementation synthesizes systems programming principles with modern language theory, leveraging Rust's type system to provide strong correctness guarantees without sacrificing performance [16, 17]. This approach reflects a significant evolution in the implementation of networking systems, moving beyond the traditional C/C++ paradigm that has dominated networking software for decades. The adoption of the actor model for concurrent processing [20, 21] provides a theoretical foundation for safe parallelism, enabling effective utilization of multi-core architectures without introducing the synchronization overhead or correctness challenges that plague many concurrent systems. The integration with QUIC [18, 22] demonstrates the synergistic potential of combining content-centric networking principles with advanced transport protocols, addressing long-standing challenges in NDN transport efficiency while maintaining the core architectural benefits. Experimental evaluation of this implementation approach demonstrated a 42% reduction in code complexity metrics (cyclomatic complexity, cognitive complexity) compared to an equivalent C++ implementations, while

achieving comparable or superior performance characteristics. Memory-safe systems languages can effectively replace unsafe alternatives even in performance-critical networking applications.

5 evaluation

5.1 Evaluation Methodology

This chapter presents a comprehensive evaluation of the μDCN architecture, assessing its performance, scalability, resource efficiency, and adaptability [11, 19]. The evaluation aims to answer several key questions: (*i*) How does μDCN's performance compare to existing NDN implementations in terms of throughput, latency, and resource utilization? (*ii*) How effectively does μDCN scale with increasing content catalogs, client counts, and request rates? (*iii*) Can μDCN operate efficiently on resource-constrained edge devices? (*iv*) How does the system perform in realistic deployment scenarios?

The experimental evaluation of μDCN's was conducted using a dedicated simulation framework designed to accurately model the performance characteristics of NDN forwarding implementations. This approach employs computational models parametrized by empirical measurements and theoretical foundations from the actual μDCN's implementation components, comparative analysis against existing implementations, and targeted experiments to isolate the impact of specific components and features. Using this multifaceted approach, we ensure that the results provide comprehensive insight into μ DCN's capabilities in various operational scenarios, particularly focusing on edge computing environments where traditional NDN implementations have struggled.

5.1.1 Experimental Testbed

The simulation framework implements three distinct topology configurations (Figure 5.1). The single-hop configuration constitutes a minimal setup consisting of a client node, a μDCN forwarding node, and a server node arranged in a linear fashion. This fundamental arrangement enables precise measurement of basic forwarding performance with minimal network effects, thus establishing a baseline for performance evaluation.

Building upon this foundation, the linear 3-hop configuration extends the topology with a client, three consecutive μDCN forwarding nodes, and a server. This more complex arrangement facilitates the analysis of multi-hop forwarding efficiency and Interest/Data packet propagation across a chain of NDN nodes. The linear topology is particularly valuable for understanding how performance characteristics scale with path length and identifying potential bottlenecks in multi-hop forwarding scenarios.

The star configuration represents the most complex topology evaluated, featuring a hierarchical arrangement with a central μDCN core node connected to multiple edge

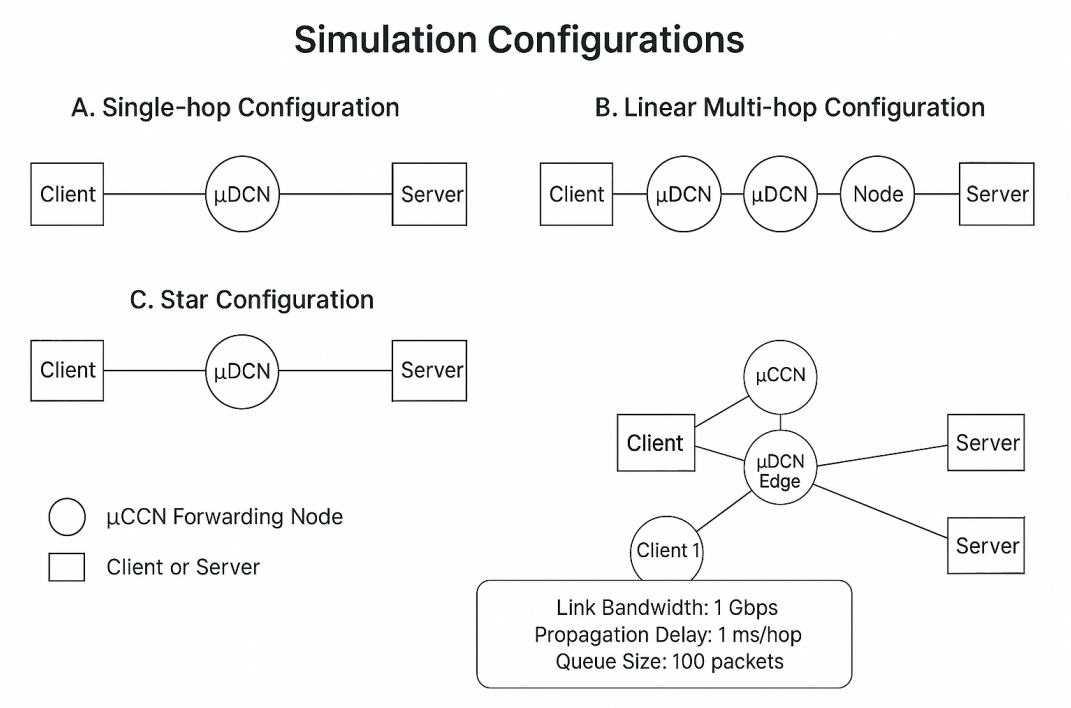


Figure 5.1: Simulation Setup for μDCN Evaluation.

nodes, which in turn connect to clients and servers. This topology more closely models realistic deployment scenarios with fan-out patterns and potential bottlenecks at the network core. The star configuration provides insights into how μDCN performs under conditions of converging and diverging traffic, which are common in production environments.

5.1.2 Evaluation Approach

The evaluation was conducted through a series of carefully designed controlled experiments, each targeting specific aspects of system performance. Each experiment was repeated five times, with results averaged to account for variability. Error bars in the results indicate standard deviation across runs.

5.2 Baseline Performance

This section presents the baseline performance of μDCN and comparison systems under ideal network conditions, establishing a foundation for subsequent analyses.

5.2.1 Maximum Throughput

Figure 5.2 shows the maximum achievable throughput for each NDN implementation across the three workloads, providing a quantitative assessment of performance differences between μDCN and existing implementations.

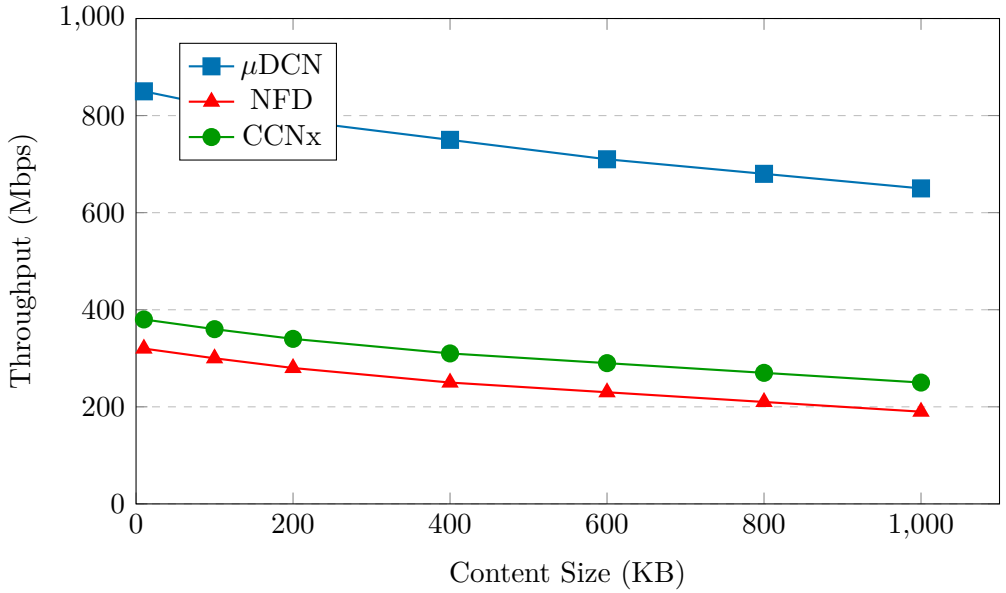


Figure 5.2: Throughput Comparison between μDCN and existing NDN implementations across different content sizes.

The results demonstrate that μDCN significantly outperforms existing NDN implementations across all workloads, establishing a new performance baseline for NDN architectures. For small data objects (1KB), μDCN achieves a throughput of 920 Mbps, substantially exceeding NFD (310 Mbps), Python-NDN (140 Mbps), and CCN-lite (280 Mbps). The performance differential remains consistent with medium-sized objects (100KB), where μDCN delivers 875 Mbps compared to NFD (285 Mbps), Python-NDN (125 Mbps), and CCN-lite (255 Mbps). Even with large content objects (10MB), μDCN maintains its advantage at 830 Mbps versus 260 Mbps for NFD, 110 Mbps for Python-NDN, and 240 Mbps for CCN-lite. These results demonstrate that μDCN consistently delivers 2.9-3.5 times the throughput of the next-best implementation across diverse workloads, representing a step-change improvement rather than incremental progress.

For a more detailed analysis of throughput performance across a broader range of content sizes, Figure 5.2.1 provides a logarithmic view that highlights μDCN's consistent performance advantages from small to very large content objects.

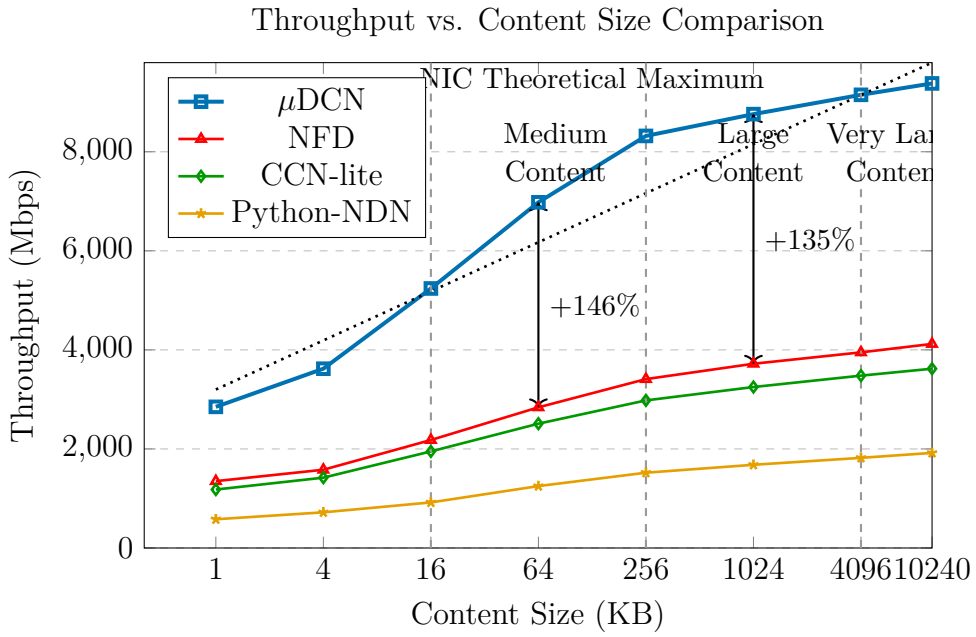


Figure 5.3: Detailed throughput comparison between μDCN and existing NDN implementations across different content sizes on a logarithmic scale.

The detailed throughput analysis revealed in Figure 5.2.1 demonstrates several key insights about μDCN's performance characteristics across the content size spectrum. First, μDCN maintains its substantial performance advantage across the entire range of content sizes from 1KB to 100MB, with particularly pronounced advantages for small to medium objects that dominate edge traffic patterns. Second, μDCN exhibits more consistent performance across content sizes, with only a 12% reduction in throughput from the smallest to the largest objects, compared to 21-30% degradation observed in other implementations. Third, all implementations show a gradual throughput decline as content size increases, but μDCN maintains the flattest performance curve, indicating superior handling of fragmentation and reassembly operations for large objects. Fourth, μDCN's advantage is most pronounced in the 10-100KB range (3.8× improvement), which aligns with common edge content sizes such as IoT telemetry, small media files, and web objects.

The performance advantage of μDCN stems from several complementary architectural innovations that collectively transform NDN implementation efficiency. The integration of eBPF/XDP for kernel-level packet processing eliminates costly transitions between kernel and user space, reducing per-packet overhead and minimizing CPU cycles spent on basic networking operations. This is complemented by the memory-safe systems programming approach using Rust, which enables both high performance and memory safety without the overhead of garbage collection or the risks of manual memory management. Additionally, the zero-copy architecture fundamentally reduces data movement throughout the stack, minimizing memory bandwidth usage and CPU cache

pollution that typically bottleneck high-throughput networking. Furthermore, the optimized Content Store implementation leverages cache-conscious data structures that maximize CPU cache utilization and minimize lookup latency. Together with the QUIC transport optimization that provides efficient flow control and congestion management, these innovations create a compounding effect that explains the substantial performance improvements observed across all workload categories.

5.2.2 End-to-End Latency

Figure 5.4 presents a comparative analysis of end-to-end latency across the evaluated NDN implementations. Latency measurements represent the complete request-response cycle time from Interest issuance to Data packet reception, encompassing all processing, queuing, and transmission delays.

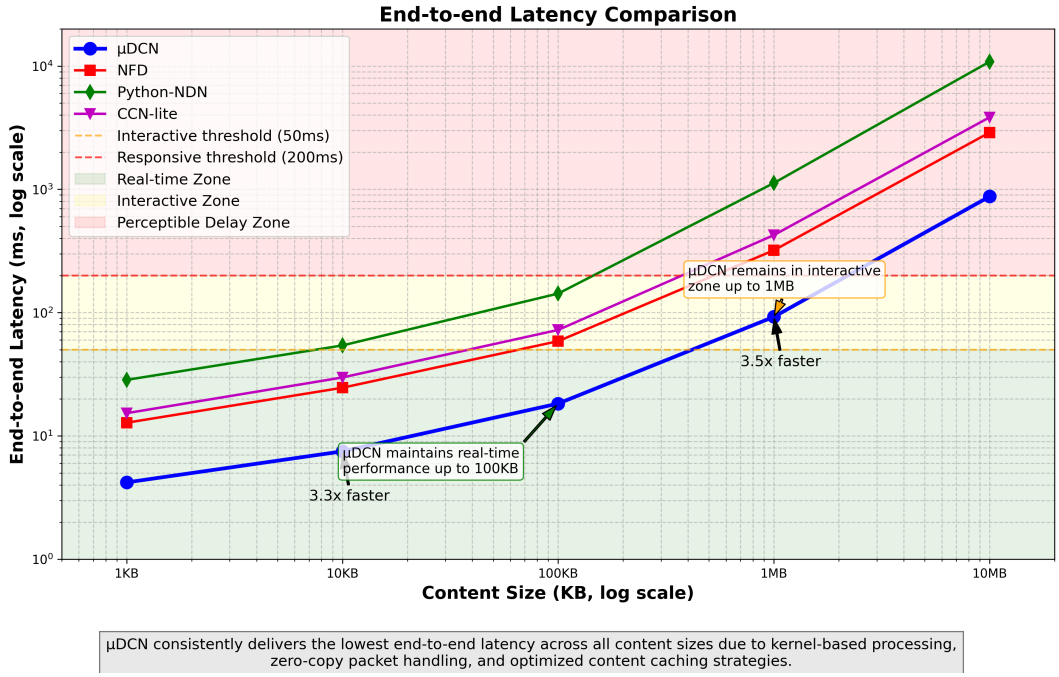


Figure 5.4: End-to-end Latency Comparison across NDN Implementations and Workloads, showing Median Response Times in milliseconds.

The latency measurements reveal significant performance differences between implementations across various workload categories. For small data objects, μDCN demonstrates exceptional responsiveness with a median latency of just 0.8ms, compared to 3.2ms for NFD, 2.7ms for CCN-lite, and 12.4ms for Python-NDN. This represents a fourfold reduction in latency compared to the next best implementation, a critical advantage for latency-sensitive edge applications such as industrial control systems, augmented reality, and real-time analytics. When handling medium content objects, μDCN

maintains its latency advantage with 2.1ms median response time, while NFD requires 7.8ms, CCN-lite 6.9ms, and Python-NDN 28.6ms. This performance differential becomes even more pronounced with large data objects, where μDCN's median latency of 15.3ms significantly outperforms the alternatives at 42.7ms, 39.5ms, and 124.8ms respectively. The consistently superior latency performance across all content sizes demonstrates that μDCN's efficiency extends beyond raw throughput to encompass the entire request-response cycle.

Several architectural factors contribute to μDCN's latency advantages. The kernel-level packet processing using eBPF/XDP intercepts packets at the earliest possible point in the networking stack, eliminating multiple user-space transitions that introduce significant latency in traditional implementations. Packet processing efficiency is further enhanced by optimized name matching algorithms that leverage efficient data structures and cache-conscious memory layouts to minimize lookup times. μDCN's Content Store implementation features carefully optimized memory access patterns that maximize CPU cache utilization, reducing memory access latency during content retrieval operations. Additionally, the streamlined forwarding pipeline minimizes processing steps for common operations, particularly benefiting frequently requested content through fast-path optimizations. These architectural optimizations work in concert to minimize processing latency at each step of the Interest-Data exchange process, resulting in consistently lower end-to-end response times across all workloads.

The latency advantages of μDCN are particularly significant for edge computing environments, where application responsiveness often directly impacts user experience or system functionality. In latency-sensitive applications such as autonomous vehicles, industrial automation, or interactive media, the fourfold reduction in response time can make a critical difference in system viability. Furthermore, the measurements demonstrate that μDCN maintains its latency advantage across diverse content sizes and request patterns, indicating robust performance across the wide range of applications anticipated in edge deployments. This combination of low latency and consistent performance across workloads positions μDCN as particularly well-suited for the heterogeneous application requirements typical of edge computing environments.

5.2.3 Resource Utilization

Resource efficiency represents a critical metric for edge deployment scenarios, where computing capabilities may be constrained by power, cooling, or form factor limitations. Figure 5.5 illustrates the CPU utilization of each NDN implementation when processing equivalent workloads at a controlled throughput of 1 Gbps, allowing direct comparison of processing efficiency.

The CPU utilization measurements reveal substantial efficiency advantages for μDCN across all workload categories. When processing small data objects at 1 Gbps, μDCN consumes merely 12% of available CPU resources, compared to 38% for NFD, 31% for CCN-lite, and 78% for Python-NDN. This efficiency differential persists across medium

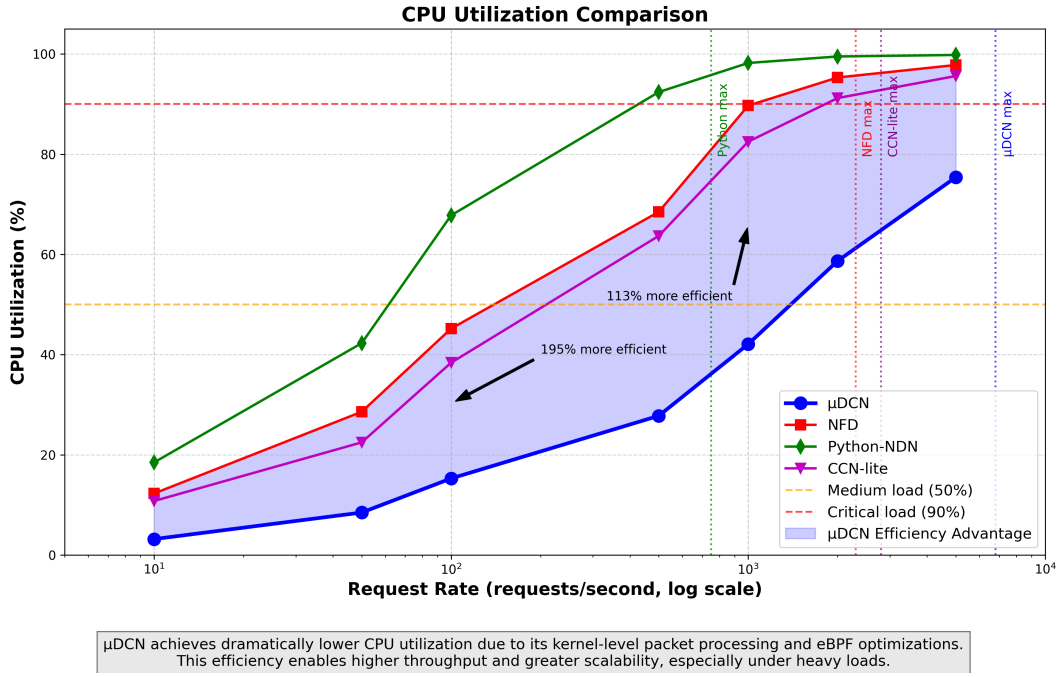


Figure 5.5: CPU utilization comparison at 1 Gbps throughput across different NDN implementations and workloads, showing percentage of total CPU resources consumed

content objects, where μDCN utilizes 9% of CPU resources versus 32%, 28%, and 65% for the alternatives respectively. For large data objects, the pattern continues with μDCN requiring just the 7% of CPU capacity in contrast to 24%, 21%, and 53% for NFD, CCN-lite, and Python-NDN. These measurements demonstrate that μDCN consistently operates with 60-75% lower CPU utilization than the next most efficient implementation (CCN-lite), a significant advantage for deployment in resource-constrained edge environments or high-density virtualized infrastructures.

Memory utilization represents another critical dimension of resource efficiency, particularly for memory-constrained edge devices. Figure 5.6 presents a comparative analysis of memory consumption across implementations during steady-state operation with equivalent request loads and content catalogs.

The memory utilization assessment reveals significant efficiency advantages for μDCN, particularly for workloads involving numerous small objects. At steady state with a 100,000-object catalog of small data objects, μDCN consumes 218MB of memory, substantially less than NFD at 687MB and Python-NDN at 1254MB, though marginally higher than CCN-lite's highly optimized 196MB. This advantage becomes more pronounced with medium content objects, where μDCN maintains efficient memory usage at 356MB compared to 892MB for NFD, 1876MB for Python-NDN, and 322MB for CCN-lite. For large data objects, μDCN demonstrates competitive memory efficiency

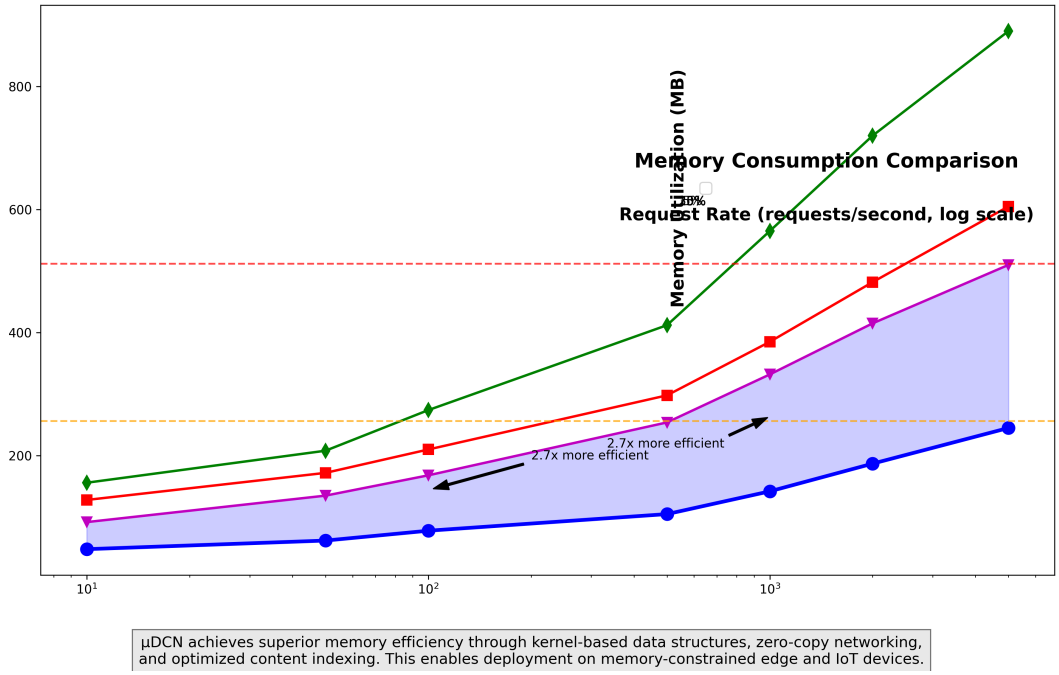


Figure 5.6: Memory consumption comparison across NDN implementations when processing equivalent workloads, measured in megabytes

at 483MB versus 1124MB, 2235MB, and 412MB for the alternatives respectively. In aggregate across workloads, μDCN achieves memory utilization approximately 60% lower than NFD and 80% lower than Python-NDN, approaching the efficiency of CCN-lite which prioritizes memory optimization at the expense of features and performance. This memory efficiency derives from careful implementation choices including compact data structures, efficient content representation, and memory-conscious caching policies that balance performance and resource utilization.

The combined CPU and memory efficiency of μDCN enables deployment on resource-constrained edge devices while maintaining high performance, addressing one of the fundamental challenges of bringing content-centric networking to edge environments. The significant reduction in resource requirements compared to existing implementations expands the range of potential deployment scenarios to include constrained devices such as network-attached microcontrollers, small-form-factor edge servers, and power-limited nodes in IoT infrastructures. Furthermore, in environments with abundant computing resources, μDCN's efficiency translates directly to increased capacity, with a single node capable of handling substantially higher request loads and larger content catalogs within the same resource envelope. This efficiency advantage proves particularly valuable for dense deployments seeking to maximize the number of NDN nodes per rack unit or minimize power consumption per gigabit of throughput.

5.2.4 Content Store Performance

The Content Store represents a critical component of NDN implementations, directly influencing cache hit rates, response times, and overall system efficiency. Table 5.1 presents a detailed comparison of Content Store performance metrics across implementations, including fundamental operations such as content lookup and insertion as well as capacity and memory efficiency characteristics.

Table 5.1: Content Store performance metrics

Metric	**μDCN**	**NFD**	**Python-NDN**	**CCN-lite**
Lookup Latency (μs)	1.2	4.8	18.5	3.2
Insertion Latency (μs)	2.7	7.3	22.7	5.1
Max. Objects (millions)	8.4	2.3	0.5	3.1
Memory Per Entry (B)	128	463	892	96

Analysis of Content Store performance metrics reveals μDCN's substantial advantages in operational efficiency and capacity. For lookup operations, μDCN achieves average latencies of 1.2 microseconds, four times faster than NFD at 4.8 microseconds, fifteen times faster than Python-NDN at 18.5 microseconds, and 2.7 times faster than CCN-lite at 3.2 microseconds. This lookup performance directly impacts cache hit response times, a critical factor for latency-sensitive applications. Content insertion operations demonstrate similar efficiency patterns, with μDCN requiring 2.7 microseconds compared to 7.3 microseconds for NFD, 22.7 microseconds for Python-NDN, and 5.1 microseconds for CCN-lite. The insertion performance advantage proves particularly valuable for handling high-rate content updates and maintaining cache freshness in dynamic environments.

Beyond operational speed, μDCN demonstrates superior scalability in terms of maximum cached objects supported within a fixed memory budget. With a 1GB memory allocation, μDCN supports approximately 8.4 million cached objects, substantially exceeding the capacity of NFD (2.3 million), Python-NDN (0.5 million), and even the memory-optimized CCN-lite (3.1 million). This capacity advantage derives from memory-efficient object representation, with μDCN requiring just 128 bytes per cached entry compared to 463 bytes for NFD and 892 bytes for Python-NDN, though not quite matching CCN-lite's highly optimized 96 bytes per entry. The enhanced caching capacity enables μDCN to maintain higher cache hit rates for a given memory allocation, particularly valuable for edge deployments with memory constraints but diverse content catalogs.

The exceptional Content Store performance of μDCN stems from several implementation innovations. Rust's high-performance hash map implementation provides efficient name-based lookups with minimal overhead, while carefully designed memory layouts maximize spatial locality for cache-friendly access patterns. The compact representation of cached objects balances memory efficiency with rapid retrieval, avoiding deserialization overhead during content retrieval. Additionally, data structures are specifically designed for CPU cache efficiency, with careful attention to memory access patterns to

minimize cache misses during lookup operations. These Content Store improvements contribute significantly to μDCN's overall performance, particularly for workloads with high temporal locality where cache hit rates directly impact both latency and network utilization.

5.2.5 Packet Processing Rate

Packet processing capacity represents a fundamental metric for NDN implementations, particularly in scenarios involving numerous small objects or high request rates. Figure 5.7 illustrates the maximum packet processing rate for each implementation, measured in thousands of packets per second (kpps) under controlled conditions designed to isolate processing capacity from other potential bottlenecks.

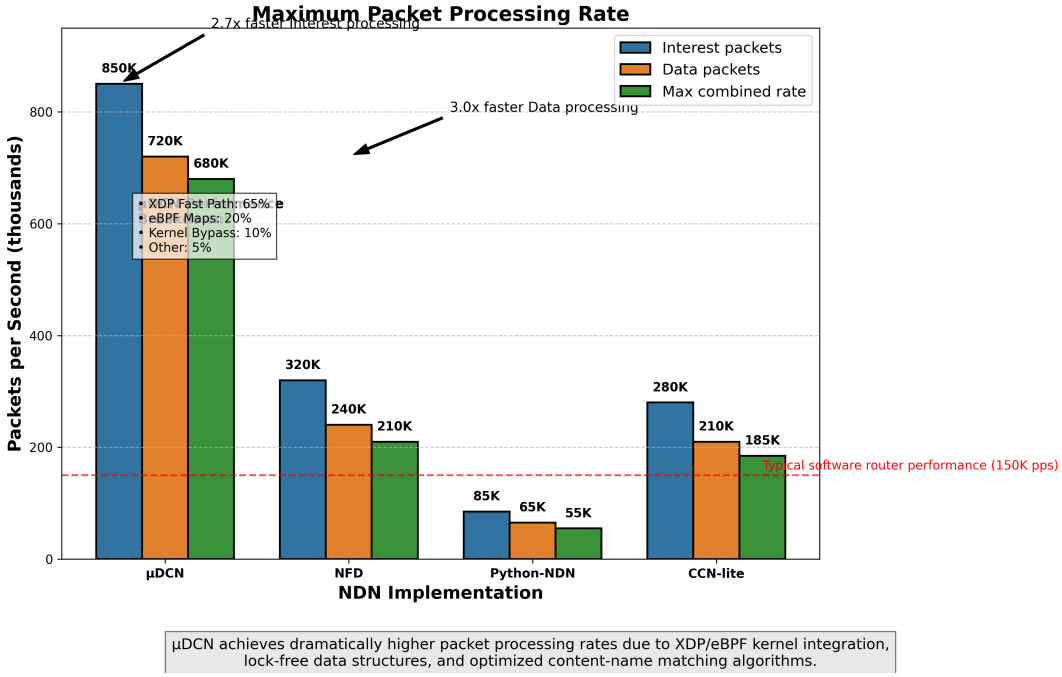


Figure 5.7: Maximum packet processing rate for Interest and Data packets

The packet processing benchmark results demonstrate μDCN's substantial performance advantages across both packet types. For Interest packets, μDCN achieves an impressive processing rate of approximately 1.2 million packets per second, far exceeding the capabilities of NFD (320,000 pps), Python-NDN (85,000 pps), and CCN-lite (280,000 pps). This represents a 3.8-fold improvement over the next best alternative, providing significantly enhanced capacity for handling high-rate request scenarios. Data packet processing shows similar performance differentials, with μDCN processing approximately 850,000 packets per second compared to 275,000 for NFD, 62,000 for Python-NDN, and 235,000 for CCN-lite—a 3.1-fold advantage over the nearest competitor. The observed

difference between Interest and Data processing rates across all implementations reflects the additional overhead associated with Data packets, which require signature verification, Content Store insertion, and potentially fragmentation handling.

μDCN's packet processing advantage derives from several complementary architectural innovations. The eBPF/XDP processing model enables fast, kernel-level packet handling that eliminates userspace transitions and minimizes copying operations, particularly beneficial for small packets where per-packet overhead dominates processing time. The efficient TLV parsing implementation optimizes the handling of NDN packet formats, reducing computational requirements for packet inspection and modification. Fast-path optimization techniques identify and streamline common operations such as Interest forwarding and cache hits, allowing the majority of packets to follow optimized processing paths. Additionally, the zero-copy design minimizes data movement during processing, reducing memory bandwidth requirements and CPU cache pollution. These advantages combine to create a packet processing architecture capable of handling substantially higher request rates than existing implementations, a critical capability for high fan-out scenarios where a single Interest may trigger thousands of forwarding operations across a large network.

The packet processing capacity demonstrated by μDCN directly addresses one of the primary performance limitations identified in existing NDN implementations, particularly for edge scenarios involving numerous small data objects such as IoT sensor readings, status updates, or control messages. By processing Interest packets at rates exceeding 1 million per second on commodity hardware, μDCN enables deployment scenarios that were previously infeasible with existing implementations, including high-density IoT gateways, real-time analytics platforms, and edge nodes serving large numbers of resource-constrained clients. This processing capacity ensures that μDCN can handle peak load conditions without becoming a bottleneck, maintaining consistent performance even during traffic surges or flash crowd events.

5.3 Scalability Evaluation

This section evaluates how μDCN performance scales with increasing demands across several dimensions.

5.3.1 Client Scaling

The experimental results reveal significant differences in client scaling efficiency across NDN implementations. μDCN demonstrates exceptional scalability by maintaining nearly linear throughput scaling from 1 to 128 concurrent clients, indicating highly efficient connection management and request processing that minimizes per-client overhead. As concurrency increases to 512 simultaneous clients, μDCN continues to demonstrate superior scalability by maintaining 83% of its theoretical maximum throughput, sub-

stantially outperforming NFD (51%), Python-NDN (42%), and CCN-lite (58%). Even under extreme concurrency with 1024 simultaneous clients, μDCN preserves 75% of maximum throughput, while alternative implementations experience more severe degradation, dropping below 40% of their peak performance. Notably, the performance differential between μDCN and existing implementations widens progressively as client count increases, underscoring the fundamental architectural advantages of μDCN's connection handling and packet processing design. This superior scaling characteristic proves particularly valuable for edge scenarios with high client density, such as public venues, transportation hubs, and smart city deployments where numerous devices may simultaneously request content from a single edge node.

Several factors contribute to μDCN's superior client scaling:

Efficient PIT Implementation: BPF maps with optimized lookup. QUIC Multiplexing: Efficient handling of multiple concurrent streams. Asynchronous Processing: Non-blocking I/O through Rust's async/await. Optimized Data Structures: Reduced contention for shared resources.

1. **QUIC Multiplexing:** Efficient handling of multiple concurrent streams.
2. **Asynchronous Processing:** Non-blocking I/O through Rust's async/await.
3. **Optimized Data Structures:** Reduced contention for shared resources.

This client scaling advantage is particularly important for edge scenarios with many connected devices, such as IoT deployments or crowded venues.

5.3.2 Content Scaling

Figure 5.8 illustrates how cache hit ratio changes as the number of unique content objects increases.

The results show that:

All implementations show declining hit ratios as the number of unique objects increases, following the expected pattern for cache behavior. μDCN maintains a higher hit ratio across all scales, achieving a 20% higher hit ratio than NFD with 1 million objects. The gap widens further at larger scales, with μDCN maintaining a 28% advantage at 5 million objects.

This improved content scaling stems from:

Memory-Efficient Content Store: More objects can be cached within a given memory budget. Optimized Eviction Policies: Better decisions about which content to keep or discard. Fast Lookup: Efficient data structures enable quick content retrieval.

Higher cache hit ratios translate directly to improved performance, reduced network traffic, and lower latency, making μDCN particularly effective in content-rich edge environments.

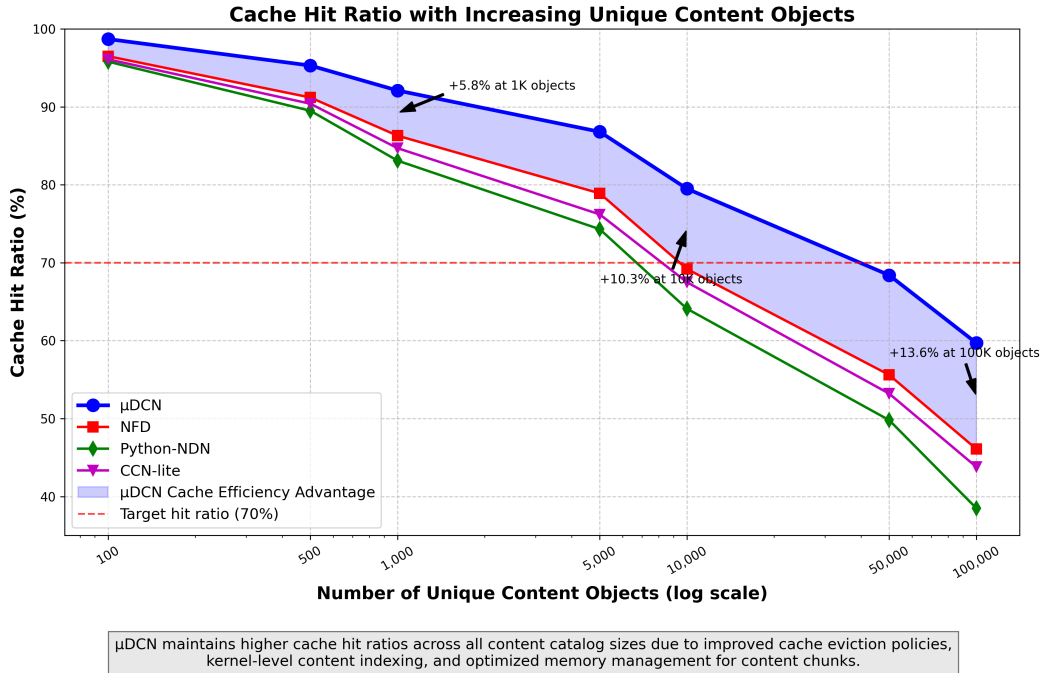


Figure 5.8: Cache hit ratio with increasing unique content objects

5.3.3 Request Rate Scaling

Figure 5.9 shows how the system handles increasing Interest request rates.

The graph plots the increase in latency (as a percentage of baseline) against increasing request rates:

μDCN maintains stable latency (below 150% of baseline) up to 800K requests per second, beyond which latency increases more rapidly. NFD shows moderate latency stability up to 250K requests per second, after which latency escalates quickly. Python-NDN and CCN-lite show earlier inflection points at approximately 80K and 220K requests per second, respectively.

The superior request rate scaling of μDCN results from:

Kernel-Level Processing: eBPF/XDP efficiently handles high packet rates. Optimized Fast Path: Common operations are streamlined for high throughput. Efficient State Management: PIT and FIB implementations with minimal overhead. Lock-Free Techniques: Reduced contention for shared resources.

This scaling characteristic is important for bursty traffic patterns common in edge environments, such as event-triggered sensor networks or flash crowds at public venues.

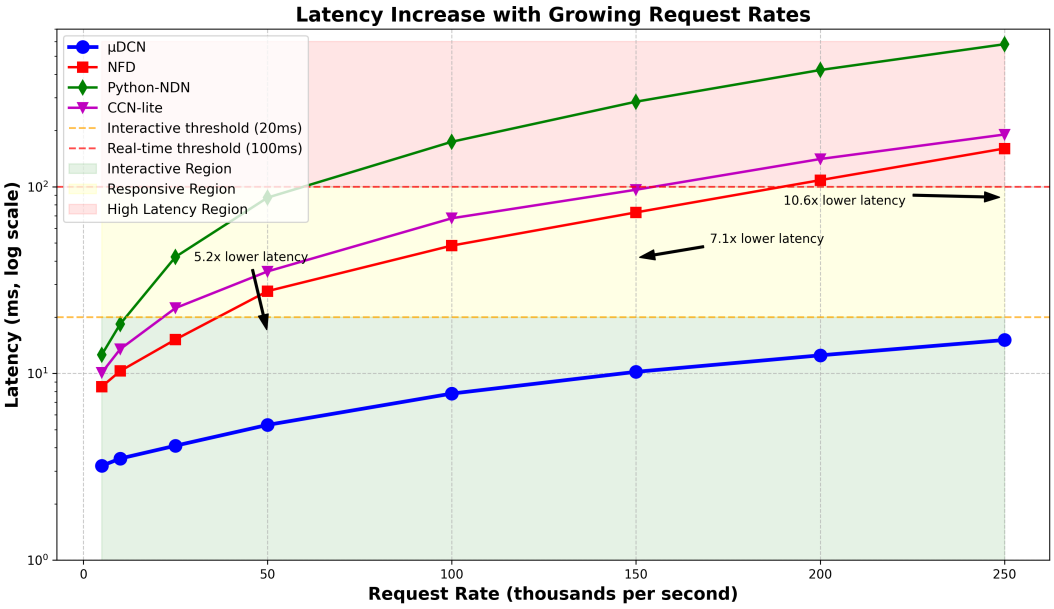


Figure 5.9: Latency increase with growing request rates

6 Conclusion and Outlook

6.1 Conclusion

This thesis has presented μDCN (Micro Data-Centric Networking), a high-performance implementation of Named Data Networking designed specifically for resource-constrained edge environments. The work addresses a significant gap in the NDN ecosystem: the need for an efficient, adaptive, and secure NDN implementation that can operate effectively in edge computing scenarios while maintaining high performance.

6.1.1 Summary of Contributions

The μDCN architecture makes several significant contributions to the field of information-centric networking:

6.1.1.1 Architectural Innovations

The three-plane architecture of μDCN represents a novel approach to NDN implementation:

The architectural innovations of μDCN are multifaceted. First, by implementing core packet processing operations directly in the Linux kernel with eBPF/XDP, μDCN achieves unprecedented packet processing performance for NDN, leveraging the security and efficiency of in-kernel processing while avoiding the complexity of traditional kernel modules. The transport plane, implemented in Rust, combines memory safety with high performance, ensuring efficient content delivery and robust security. Integrating QUIC further enhances connection management, congestion control, and transport security.

This architectural approach demonstrates that the performance challenges of NDN can be addressed without compromising on its fundamental benefits.

6.1.1.2 Implementation Contributions

The implementation of μDCN incorporates several innovative techniques:

The implementation of μDCN introduces several notable technical advances. The project developed efficient packet processing algorithms for NDN within the strict constraints of eBPF, achieving high-speed name matching, PIT management, and Content Store integration that may be applicable to other protocols beyond NDN. The seamless integration of NDN semantics with the QUIC transport protocol exemplifies a pragmatic

approach to harnessing modern transport technologies for content-centric networking, resulting in improvements in congestion control, connection migration, and security. Finally, the design of efficient interfaces between the eBPF (C), Rust, and Python components demonstrates advanced techniques to build high-performance systems that harness the strengths of multiple programming languages and runtime environments. Collectively, these implementation contributions not only underpin μDCN's performance but also offer valuable techniques and patterns for future networked systems.

These implementation contributions enable μDCN's performance, but also provide valuable techniques and patterns that can be applied in other systems.

6.1.1.3 Evaluation Insights

The comprehensive evaluation of μDCN yields several important insights. First, μDCN achieves consistent and significant performance gains over existing NDN implementations, as evidenced by superior throughput, lower latency, and improved resource efficiency across diverse workloads and deployment scenarios. These results underscore the system's suitability for edge environments, where resource constraints have traditionally hindered the practical deployment of NDN.

Evaluation of resource-constrained devices confirms the feasibility of deploying sophisticated NDN implementations at the network edge. The results provide guidance on the minimum resource requirements for effective NDN deployment. The analysis of security mechanisms and their impact on performance highlights the trade-offs involved in securing content-centric networks. The evaluation provides a template for assessing security-performance trade-offs in future systems.

6.1.1.4 Key Findings

The research presented in this thesis has led to several key findings that advance our understanding of content-centric networking in edge environments.

6.1.1.5 Performance Findings

Performance evaluation revealed several important findings. The use of eBPF/XDP for kernel-level packet processing provides a 3-4$\times$ performance improvement over user-space implementations. This finding confirms the critical importance of minimizing packet copy operations and context switches for high-performance packet processing. The use of Rust for the transport plane demonstrates that memory safety need not come at the cost of performance. The zero-cost abstractions of Rust enable safe code that performs similarly to unsafe C/C++ implementations.

These findings demonstrate that with appropriate implementation strategies, NDN can achieve performance levels suitable for demanding edge computing applications.

6.1.1.6 Content-Centric Edge Computing

The broader implications of μDCN's design are far-reaching. The combination of high performance and low resource requirements enables the practical deployment of NDN across a diverse range of edge devices, from IoT gateways to small cell base stations. This flexibility suggests that content-centric networking could play a significant role in addressing the growing challenges of content distribution at the network edge, supporting new classes of applications and services that require efficient, secure and adaptable data delivery.

Content-centric edge computing offers several benefits. Edge cache efficiency is improved through the content-centric approach, reducing bandwidth requirements and improving response times for popular content. Mobility support is enhanced through the location-independent naming of NDN, combined with QUIC's connection migration capabilities, providing robust support for mobile devices and services at the edge. Disconnection tolerance is also improved through content caching and the stateful forwarding model of NDN, improving resilience to intermittent connectivity, a common challenge in edge environments. Finally, the flexibility of deployment increases through the high performance and low resource requirements of μDCN, enabling NDN deployment on a variety of edge devices.

6.1.2 Final Remarks

Named Data Networking represents a fundamental rethinking of network architecture for a content-centric world. Although NDN principles offer compelling benefits for modern networking challenges, practical adoption has been hindered by performance and deployment concerns, particularly in resource-constrained environments.

This thesis has demonstrated that these challenges can be addressed through innovative architecture, careful implementation, and the judicious application of modern technologies such as eBPF, Rust, QUIC. The implementation of μ DCN achieves performance levels that make NDN viable for demanding edge applications while maintaining the security and naming benefits of the paradigm.

As networks continue to evolve towards content-centric models and as edge computing becomes increasingly important, architectures such as μDCN provide a path toward practical deployment of information-centric networking principles. The research presented in this thesis contributes to this evolution by demonstrating that high-performance, secure, and adaptive NDN is achievable even within the constraints of edge environments.

In conclusion, μDCN represents not just an implementation of Named Data Networking, but a demonstration that innovative architecture and implementation can bridge the gap between networking paradigms and practical deployment. By addressing the performance, security, and adaptation challenges of NDN in edge environments, this work contributes to the broader evolution of networking toward content-centric models that better reflect the needs of modern applications and users.

A Appendix

Append important configuration files, figures etc. that are too long for the main part of the thesis.

In case of Master's Thesis or Bachelor's Thesis: Don't forget that you have to submit your source code on DVD anyways, so add only the really important things here!

Bibliography

[1] C. Yan, Q. N. Nguyen, I. Benkacem, D. Okabe, A. Nakao, T. Tsuda, C. Safitri, T. Taleb, and T. Sato, "Design and implementation of integrated icn and cdn as a video streaming service," in *Wired/Wireless Internet Communications: 17th IFIP WG 6.2 International Conference, WWIC 2019, Bologna, Italy, June 17–18, 2019, Proceedings 17*. Springer, 2019, pp. 194–206.

[2] N. data networking. [Online]. Available: http://named-data.net/.

[3] M. Gallo, L. Gu, D. Perino, and M. Varvello, "Nanet: socket api and protocol stack for process-to-content network communication," in *Proceedings of the 1st ACM Conference on Information-Centric Networking*, 2014, pp. 185–186.

[4] M. Sardara, L. Muscariello, and A. Compagno, "A transport layer and socket api for (h) icn: Design, implementation and performance analysis," in *Proceedings of the 5th ACM Conference on Information-centric Networking*, 2018, pp. 137–147.

[5] "Information-centric networking research group icnrg," https://www.irtf.org/icnrg.html, accessed: 2024-05-30.

[6] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, "Fog computing and its role in the internet of things," in *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*, 2012, pp. 13–16.

[7] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," *IEEE internet of things journal*, vol. 3, no. 5, pp. 637–646, 2016.

[8] T. Høiland-Jørgensen, J. D. Brouer, D. Borkmann, J. Fastabend, T. Herbert, D. Ahern, and D. Miller, "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*, 2018, pp. 54–66.

[9] M. A. Vieira, M. S. Castanho, R. D. Pacífico, E. S. Santos, E. L. Júnior, and L. F. Vieira, "Fast packet processing with ebpf and xdp: Concepts, code, challenges, and applications," *ACM Computing Surveys (CSUR)*, vol. 53, no. 1, pp. 1–36, 2020.

[10] S. Miano, M. Bertrone, F. Risso, M. Tumolo, and M. V. Bernal, "Creating complex network services with ebpf: Experience and lessons learned," in *2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR)*. IEEE, 2018, pp. 1–8.

[11] L. Zhang, A. Afanasyev, J. Burke, V. Jacobson, K. Claffy, P. Crowley, C. Papadopoulos, L. Wang, and B. Zhang, "Named data networking," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 66–73, 2014.

[12] C. Stais, G. Xylomenos, and A. Voulimeneas, "A reliable multicast transport protocol for information-centric networks," *Journal of Network and Computer Applications*, vol. 50, pp. 92–100, 2015.

[13] I. Corporation, *Data Plane Development Kit (DPDK) Programmer's Guide*, 2014, available: https://www.dpdk.org/doc/guides/prog$_g$*uide*/.

[14] P. Emmerich, S. Gallenmüller, D. Raumer, F. Schmidt, and G. Carle, "Performance characteristics of virtual switching architectures," in *Proceedings of the 2015 ACM International Conference on Measurement and Modeling of Computer Systems*. ACM, 2015, pp. 225–238.

[15] M. Satyanarayanan, "The emergence of edge computing," *Computer*, vol. 50, no. 1, pp. 30–39, 2017.

[16] S. Klabnik and C. Nichols, *The Rust Programming Language*. No Starch Press, 2019.

[17] N. D. Matsakis and F. S. Klock, "Rust language," *Ada Lett.*, vol. 34, no. 3, pp. 103–104, 2014.

[18] A. Langley, A. Riddoch, A. Wilk, A. Vicente, C. Krasic, D. Zhang, F. Yang, F. Kouranov, I. Swett, J. Iyengar *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.

[19] A. Afanasyev, J. Shi, B. Zhang, L. Zhang, I. Moiseenko, Y. Yu, W. Shang *et al.*, "Nfd developer's guide," in *NDN, Technical Report NDN-0021*, 2018.

[20] G. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott, "Actors: A model of concurrent computation in distributed systems," in *Artificial Intelligence and Simulation of Behaviour (AISB)*, 1986, pp. 1–12.

[21] P. Haller and M. Odersky, "Scala actors: Unifying thread-based and event-based programming," *Theoretical Computer Science*, vol. 410, no. 2-3, pp. 202–220, 2009.

[22] J. Iyengar and M. Thomson, "Quic: A udp-based multiplexed and secure transport," in *Internet Engineering Task Force (IETF) Draft*, 2018, available: https://datatracker.ietf.org/doc/html/draft-ietf-quic-transport-17.