ENDN: Towards an Enhanced NDN Architecture for Next Generation Internet

by

Ouassim Karrakchou

Thesis submitted to the University of Ottawa
in partial fulfillment of the requirements for the
Doctorate of Philosophy
in
Electrical and Computer Engineering

School of Electrical Engineering and Computer Science
Faculty of Engineering
University of Ottawa

Examining Committee

The following served on the Examining Committee for this thesis.

OCIECE Examiners:	Amiya Nayak
	Professor, University of Ottawa
	Marc St-Hilaire
	Professor, Carleton University
	Tet Yeap
	Professor, University of Ottawa
External Examiner:	Abdelhakim Hafid
	Professor, Université de Montréal
Supervisors:	Ahmed Karmouch
	Professor, University of Ottawa
	Nancy Samaan
	Professor, University of Ottawa

Declaration of Authorship

I hereby certify that this thesis is entirely my own original work except where otherwise indicated. I am aware of the University's regulations concerning plagiarism, including those concerning consequent disciplinary actions. Any use of the works of any other author, in any form, is properly acknowledged at their point of use.

Several sections of this thesis were adapted from my previously published Ph.D. work [1–4] (©2018-2021 IEEE), [5] (©2021 ACM), and [6], with permission from the IEEE and the ACM.

Abstract

Named data networking (NDN) is a content-centric future Internet architecture that uses routable content names instead of IP addresses to achieve location-independent forwarding. Nevertheless, NDN's design is limited to offering hosted applications a simple content pull mechanism. As a result, increased complexity is needed in developing applications that require more sophisticated content delivery functionalities (e.g., push, publish/subscribe, streaming, generalized forwarding, and dynamic content naming). This thesis introduces a novel Enhanced NDN (ENDN) architecture that offers an extensible catalog of content delivery services (e.g., adaptive forwarding, customized monitoring, and in-network caching control). More precisely, the proposed architecture allows hosted applications to associate their content namespaces with a set of services offered by ENDN.

The design of ENDN starts from the current NDN architecture that is gradually modified to meet the evolving needs of novel applications. NDN switches use several forwarding tables in the packet processing pipeline, the most important one being the Forwarding Information Base (FIB). The NDN FIBs face stringent performance requirements, especially in Internet-scale deployments. Hence, to increase the NDN data plane scalability and flexibility, we first propose FCTree, a novel FIB data structure. FCTree is a compressed FIB data structure that significantly reduces the required storage space within the NDN routers while providing fast lookup and modification operations. FCTree also offers additional lookup types that can be used as building blocks to novel network services (e.g., in-network search engine).

Second, we design a novel programmable data plane for ENDN using P4, a prominent data plane programming language. Our proposed data plane allows content namespaces to be processed by P4 functions implementing complex stateful forwarding behaviors. We thus extend existing P4 models to overcome their limitations with respect to processing string-based content names. Our proposed data plane also allows running independent P4 functions in isolation, thus enabling P4 code run-time pluggability. We further enhance

our proposed data plane by making it protocol-independent using programmable parsers to allow interfacing with IP networks.

Finally, we introduce a new control plane architecture that allows the applications to express their network requirements using intents. We employ Event-B machine (EBM) language modeling and tools to represent these intents and their semantics on an abstract model of the network. The resulting EBMs are then gradually refined to represent configurations at the programmable data plane. The Event-B method formally ensures the consistency of the different application requirements using proof obligations and verifies that the requirements of different intents do not contradict each other. Thus, the desired properties of the network or its services, as defined by the intent, are guaranteed to be satisfied by the refined EBM representing the final data-plane configurations. Experimental evaluation results demonstrate the feasibility and efficiency of our proposed architecture.

Acknowledgements

I would like to thank several people who provided me with invaluable support during my Ph.D.

First, I would like to convey my deepest gratitude to Dr. Ahmed Karmouch and Dr. Nancy Samaan. I can't express how fortunate I feel for having them as my supervisors. Their expertise, dedication, and constant support, feedback, and guidance were priceless during the course of my Ph.D. Furthermore, I am incredibly grateful for all the skills and knowledge I learned from them, which will undoubtedly prove highly valuable for my future career.

I would also like to express my sincere gratitude to my thesis committee members: Dr. Amiya Nayak, Dr. Marc St-Hilaire, Dr. Tet Yeap, and Dr. Abdelhakim Hafid. Their valuable feedback greatly improved my thesis.

In addition, I would like to thank my parents for the constant encouragement and guidance they gave me during my whole education and for being such an important source of inspiration for achieving this Ph.D. I am also very grateful for the motivation and support I received from my sister Basma, her husband Souhail, and her daughter Mayssam. I am also thankful to my grandparents and my uncle Taoufik who always believed in me and encouraged me to start the Ph.D. journey.

Finally, I would like to give a special thanks to my friends in the dance community in Ottawa and the uoSalsa club, especially Helga, Prem, James, Raphael, Molly, and Nas. I am extremely grateful for their continuous friendship that provided me with happy distractions to rest my mind outside of my research.

Dedicated to my parents.

Table of Contents

List of Tables

List of Figures

Abbreviations

AIMD Additive-Increase/Multiplicative Decrease

API Application Programming Interface

ARP Address Resolution Protocol

ARPANET Advanced Research Projects Agency Network

AS Autonomous System

ASIC Application-Specific Integrated Circuit

BF Bloom Filter

BFC Bucket-based Front-Coding

BGP Border Gateway Protocol

BMv2 Behavioral Model version 2

CCN Content-Centric Networking

CDN Content Delivery Network

CN Content Name

CNP Content Name Prefix

CPU Central Processing Unit

CS Content Store

CT Character Trie

DHT Distributed Hash Table

DiFCTree Dictionary compressed FCTree

DNS Domain Name System

DoS Denial of Service

Dpkt Data Packet

DRAM Dynamic Random Access Memory

EBM Event-B Machine

EE Execution Environment

EFIB Enhanced FIB

ENDN Enhanced NDN

EP4 Extended-P4

EPacket Enhanced Packet

eParser Enhanced Parser

EPIT Enhanced PIT

EProcessing Enhanced packet Processing

eProcessor Extended Processor

FC Front-Coding

FCTree Front-Coded Tree

FIB Forwarding Information Base

FPGA Field-Programmable Gate Array

FS Forwarding Strategies

GPU Graphical Processing Unit

HDL Hardware Description Language

HT Hash Table

HTTP HyperText Transfer Protocol

ICMP Internet Control Message Protocol

ICN Information-Centric Networking

IDN Intent-Driven Networking

IETF Internet Engineering Task Force

IoT Internet of Things

IP Internet Protocol

Ipkt Interest Packet

IPv4 IP version 4

IPv6 IP version 6

IRTF Internet Research Task Force

ISL Intent-Service Layer

ISP Internet Service Provider

LPM Longest Prefix Match

LSA Link State Advertisements

LSDB Link State Database

MAT Match Action Table

MIB Management Information Base

ML Machine Leaning

NDN Named Data Networking

NFD NDN Forwarding Daemon

NLSR Named-data Link State Routing protocol

NPT Name Prefix Trie

ONOS Open Network Operating System

OSPF Open Shortest Path First

P2P Peer-to-Peer

P4 Programming Protocol-independent Packet Processors

PBID Processing Behavior ID

PBT Processing Behavior Table

PDP Programmable Data Plane

PHV Packet Header Vector

PIT Pending Interests Table

PSA Portable Switch Architecture

QoS Quality of Service

RBT Red-Black Tree

RIP Routing Information Protocol

RTT Round Trip Time

SDN Software-Defined Networking

SNMP Simple Network Management Protocol

SRAM Static Random Access Memory

StFCTree Statistically compressed FCTree

TCP Transmission Control Protocol

UDP User Datagram Protocol

VLAN Virtual Local Area Network

Chapter 1

Introduction

The foundation of the current Internet has emerged as part of the architecture developed by the Advanced Research Projects Agency Network (ARPANET). This architecture was designed in the seventies to interconnect several computers spanning multiple American universities. The Internet protocols still in use today (e.g., IP and TCP) were first designed and implemented by ARPANET. At that time, the ARPANET network was mainly used to send and receive emails between two end-hosts: an internal ARPA study showed that email traffic accounted for 75% of the ARPANET traffic in 1973 [13]. As a result, knowing the exact location of the destination was an essential step for networking. Consequently, the IP protocol was designed in this context with the requirement that every interconnected machine must be assigned an address. Thus, the Internet became a host-centric network that provides the ability to establish communication between any hosts based on their IP addresses.

In the nineties, with the development of the World Wide Web and the widespread usage of the Internet in residential areas, most end-users applications evolved following a host-based content retrieval approach. This is demonstrated by G. Maier et al., who showed that the Hypertext Transfer Protocol (HTTP), which retrieves web pages from pre-determined servers, and peer-to-peer (P2P) file transfers between end-hosts, accounted for the majority of the residential broadband traffic in 2009 [14].

As new applications such as media streaming, social networking, and location-based services emerged, they posed a new set of requirements different from those initially set by host-centric applications. Indeed, while successfully sending an email requires knowing the IP address of the email server, a media streaming application is only interested in the retrieved content and not from where the content was obtained.

Several technical solutions were designed in the current Internet to satisfy the requirements of content delivery applications efficiently. The traditional solution consists of following a dual-step approach; first, knowing the name of the source or destination host, an application protocol is needed to discover the exact location, represented by an IP address, of the queried content. Then, a second application uses the location to retrieve the content. The most dominant protocols to carry out the first and second steps are the Domain Name System (DNS) and HTTP protocols. Content Delivery Networks (CDN) further improve this process by duplicating contents in several servers to achieve higher availability and reduce the experienced latency. Nonetheless, CDNs rely mostly on sophisticated DNS configurations.

Another prominent technical solution to distribute content independently from their locations was developed for P2P file-sharing applications (e.g., BitTorrent). In these applications, files are decomposed into several chunks that are distributed and exchanged among end-users. Despite its efficiency, P2P applications still need to obtain an IP address of a source/destination host to initiate the exchange.

However, while all the aforementioned solutions allow for efficient content delivery in the current Internet, the host-centric nature of IP causes an increase in the application layer's complexity. Indeed, a large portion of a network application logic is dedicated to adapting the application requirements to the best-effort location-centric communication service offered by the IP protocol. As a result, several research proposals aimed to evolve the network layer to serve the application needs better. Several of these proposals fall under the family of Information-Centric Networking (ICN) [15, 16].

ICN concepts were introduced as a new clean-slate approach that aims at replacing

the host-centric IP protocol stack with a new network layer centered around information rather than hosts. One of the most prominent flavors of ICN is the Named Data Networking (NDN) architecture [8]. NDN's first proposal is to generalize the semantic of network identifiers. While IP addresses identify host interfaces, NDN introduces identifiers that can be used to name anything in the network (e.g., a video chunk, a book page, a service offered by a server, or an endpoint). NDN then proceeds with changing the semantic of the network layer service from the traditional *"deliver data to a specific destination"* to *"query the network for a specific name"*. Although relatively simple, these semantic changes allow the network to adapt to a wide variety of application requirements ranging from host-based communication to location-agnostic content delivery and real-time data transfer.

The NDN architecture performs these semantic changes by introducing two packet types: Interest (Ipkt) and Data (Dpkt) packets. A Dpkt represents a chunk of data identified by a name. Ipkts can then be sent by consumer applications to request the delivery of the Dpkt associated with the queried name. It is worth noting that NDN forces a one-to-one pull relationship between Ipkts and Dpkts: a Dpkt can only be delivered after the network has received the associated Ipkt. The NDN architecture is then implemented in a distributed manner. Every NDN switch has a Forwarding Information Base (FIB) that contains name-based routes associating name prefixes to next-hops. Hence, Ipkts are forwarded following the routes of the FIBs until they reach a data producer for the queried name. While the Ipkts are forwarded through the network, the path they follow is recorded in the switches in a special structure called the Pending Interests Table (PIT). After the data producer is reached, the requested Dpkt is generated and forwarded through the reverse path followed by the Ipkt using the PIT. Moreover, the PIT records storing the path followed by the Ipkt are deleted once the requested Dpkt is forwarded, thus freeing the state stored in the network. Finally, some switches may cache the forwarded Dpkt in a Content Store (CS) for faster delivery of the Dpkt during future requests.

1.1 Motivation

The NDN architecture is especially well-suited for content delivery applications. Names identify content chunks that can be requested by consumer applications using Ipkts. Additionally, NDN natively offers several advantages like in-network caching in the CS, automatic consumer mobility management using the PIT, and simple consumer flow control due to the one-to-one pull relationship between Ipkts and Dpkts. However, similarly to IP, the design of the NDN architecture favored the needs of the content-delivery applications that were dominant at the time of its inception, at the expense of less adaptability to other applications' requirements. For instance, the NDN architecture lacks a native push mechanism critical to many applications (e.g., messaging applications).

The lack of adaptability of NDN to non-content-delivery applications seemingly contradicts the initial NDN vision that saw names as generic identifiers that could be associated with any concept in the network. In this thesis, we believe that this contradiction stems from the current design of the NDN architecture rather than an inherent limitation of the NDN principles. More precisely, the concepts of names, Ipkts, and Dpkts are general and can be used by a broad range of applications. However, the mandatory pull service provided by the current NDN architecture is specific to content-delivery applications. Hence, the services offered by the network layer are critical to the network's adaptability to various applications' needs.

In this work, we enhance the current NDN architecture to adapt it to the needs of emerging applications in the next generation Internet while maintaining the initial NDN vision. Hence, we propose our Enhanced NDN (ENDN) architecture, whose main goal is to offer network services that can satisfy a wide variety of application requirements. The next section explains how we plan to achieve this goal.

1.2 Thesis Objectives

Our ENDN architecture aims to enhance the current NDN architecture to increase the flexibility and adaptability of the network layer to the various emerging application requirements. However, the current NDN architecture still faces several challenges, especially in terms of scalability and performance, due to the use of string-based name identifiers compared to fixed-size binary IP addresses.

Additionally, NDN is a distributed network architecture. This means that the data plane, which corresponds to the fast packet forwarding equipment, and the control plane, which configures the data plane, both coexist in every switch. This distributed design complexifies the configuration and management of NDN while reducing its flexibility to support new network requirements: any new network configuration needs to be done separately in every switch. Software-Defined Networking (SDN) [11] was introduced to solve these issues by separating the control and data planes and logically centralizing the control plane in a single component: the controller. The latter has a global view of the network and can thus enforce new requirements in all the switches.

Finally, the strict pull service offered to applications limits the adaptability of the network layer to emerging application needs. From these observations, we enhanced the current NDN architecture based on the following principles:

1. *Separation of Control and Data Planes:* A significant limitation of NDN in its adaptability to novel application requirements comes from its distributed design. By following SDN principles, and hence separating the NDN control and data planes, we introduce a single configuration point: the controller. Hence, applications can specify their network requirements to the control plane that will configure the different switches in the data plane accordingly.

2. *Extensible Catalog of Network Services:* As future Internet usages are expected to result in novel network requirements, a fixed network service, as provided by

NDN, is not sufficient to make the architecture future-proof. We thus envision that the control plane should offer an extensible catalog of network services to the applications. The latter could then attach their packet flows to a chain of services translated into a data-plane configuration. Additionally, this service catalog must be extensible with new services each time a new network behavior is needed.

3. *Data Plane Programmability:* While the current NDN data plane was optimized for the pull service, our ENDN data plane needs to support various new network services. As a result, our proposed data plane needs to be programmable to offer enough flexibility for implementing the different services in the catalog. Additionally, as applications may have competing network requirements, the data plane needs to provide isolation between the network services processing different application flows.

4. *Intent-Driven Networking:* Allowing more network flexibility and programmability may result in an increased difficulty to manage and configure the network. Additionally, as our proposed architecture aims at adapting to the evolving application needs, we require a way for application developers to express their application service requirement in a simple abstract manner. Intent-Driven Networking (IDN) solves these issues by allowing operators to configure the network using intents that represent high level abstract operational goals rather than low-level technical implementation details.

5. *Data Plane Performance and Scalability:* The introduction of string-based names as network identifiers instead of binary IP addresses increases the complexity of the different structures and algorithms used in the packet processing pipeline. This leads to stringent scalability requirements on the NDN switch forwarding tables. Additionally, as name-based routes are expected to be more dynamic than host-based routes, the NDN FIB data structure needs to support efficient insertion and update operations. Finally, the implementation of new services in the data plane

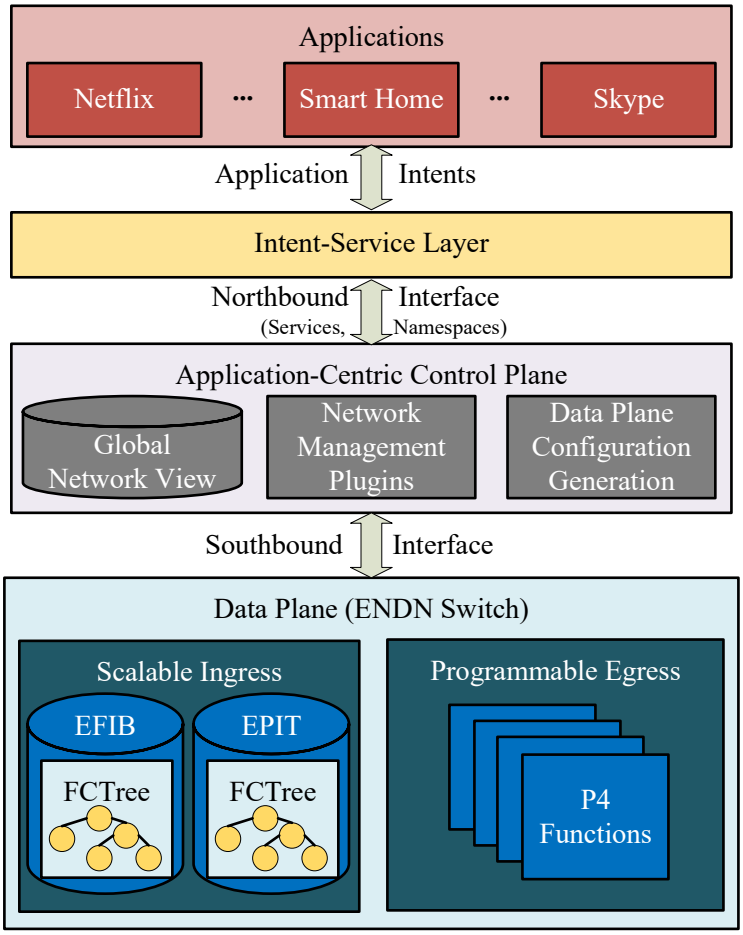


Figure 1.1: Overview of the proposed architecture.

should not incur any switch downtime to avoid interference with the processing of existing application flows.

1.3 Overview of ENDN

Fig. 1.1 shows an overview of the proposed architecture. Following the principles of SDN, ENDN separates the control and data planes. The main goal of the control plane of ENDN is to provide the applications with a catalog of content delivery services. These services are then translated into data plane configurations that are installed in the network switches. As shown in Fig. 1.1, applications use intents to express their custom

network service requirements. The intent-service layer translates these intents into a list of network services, selected from the ENDN service catalog, that is associated to the application's namespaces, each representing a collection of related content names using the NDN hierarchical naming structure (e.g., */com/netflix*).

1.3.1 ENDN Services

The ENDN services are completely programmable and extensible. Examples of categories of these services are:

1. *Content Delivery Pattern services:* Describe which pattern is used to deliver the contents in a namespace. For example, an application may request that certain contents are only delivered after receiving an Ipkt (pull pattern), while other contents are part of a stream of Dpkts that are delivered automatically to any end-point that subscribes to the stream by sending a unique Ipkt (publish/subscribe pattern).

2. *Content Name Rewrite services:* Allow the application to request that the network changes the content name of packets at specific network locations (e.g., at the edge of certain network regions). For instance, the network could flag packets coming from a specific network region by inserting a region ID component in the content name. The region ID component can then be used by the application to adapt the content to that region (e.g., translate it to the region's language). Another service in this category is content name aliasing, where multiple namespaces can be considered equivalent (i.e., associated with the same content).

3. *Adaptive Forwarding services:* Define how the next-hops where packets are forwarded are selected at particular network locations. The next-hop choice can be based on the value of a name component, face attributes (e.g., broadcast capable, link capacity, and usage charging), measured traffic statistics (e.g., RTT), throughput thresholds, or internal states. For example, these services can be part of im-

plementing more complex functionalities such as load balancers, stateful firewalls, as well as congestion detection and avoidance mechanisms.

4. *Customized Monitoring services:* Allows the application to request the network to notify it whenever a particular event occurs. For instance, a video streaming application may request a notification in case of congestion in order to reduce the bitrate of the streamed video.

5. *In-Network Cache Control services:* Specify when and which Dpkts of a namespace can be cached in the network and for how long. It can be useful for real-time applications who want to ensure that the received Dpkts are always up-to-date.

6. *Namespace Traffic Management services:* Specify the requirements of a namespace in terms of packet prioritization and throughput, delay, and packet loss. ENDN should also offer a cheap best-effort delivery that is suitable for low priority content delivery applications. These services allow the control plane to optimize the use of the network resources among the different flows.

Several services can be combined to create complex network behaviors. For instance, geo-gating can be implemented with content name rewrite and adaptive forwarding services: the network flags the Ipkts with a region ID at the edge of the network. The region ID is then used in the core of the network to forward the Ipkts to specific content producers that contain only the restricted content.

1.3.2 ENDN Control Plane

The control plane generates the corresponding data plane configuration after the application intents are translated into a set of services associated to namespaces. As shown in Fig. 1.1, the control plane has a global network view, which contains the different switches, their capabilities, their links, and the locations of the content producers. The

control plane uses this global network view along with the set of namespaces and associated services as an input to generate data plane configurations consisting of two parts: table configuration (i.e., rules in the ENDN data plane tables) and custom functions written in P4, a prominent data-plane programming language [17]. Content delivery pattern services can be directly expressed in the data plane as a set of table rules, while other services like content name rewrite and adaptive forwarding result in the generation of P4 code.

1.3.3 ENDN Data Plane

The data plane contains two main pipelines, a scalable table-based ingress, and a programmable egress. The scalable ingress pipeline corresponds to the fast processing common to all applications. Its goal is to handle name-based routing using Enhanced FIB (EFIB) and PIT (EPIT) tables implemented using FCTree, our proposed scalable data structure. The EFIB and EPIT tables are used according to the content delivery pattern associated to the application namespace. For instance, the pull pattern would result in the standard NDN operations on the EFIB and EPIT, while the push would only use the EPIT as a Dpkt forwarding table. Finally, the EFIB and EPIT can also associate an application namespace with a P4 function implementing a custom forwarding logic.

P4 functions are executed in isolated P4 targets in the programmable egress pipeline. As a result, P4 functions can be updated in run-time by the control plane without impacting the other switch functionalities.

1.4 Thesis Contributions

This thesis starts with the current NDN architecture and gradually enhances it to achieve our goal: adaptability of the network services to current and future application needs. We proceed as follows:

1. First, we start with the data plane design, especially the design of the FIB and PIT data structures. The current FIB and PIT data structures have been designed in the NDN standard pull service context. As a result, during Ipkt and Dpkt forwarding, the current FIB only performs longest name prefix match operations while the PIT performs insertion, deletion, and exact name match operations. However, in the general case, the FIB and PIT tables may be used differently by different services. For instance, a Dpkt push service would use the PIT as a Dpkt forwarding table and would thus require longest name prefix match operations. Furthermore, the FIB and PIT data structures may face stringent scalability requirements depending on the semantic of the names used by the applications. For example, host-based names would incur similar scalability requirements compared to IP addresses. In contrast, content-based names would lead to more rigorous scalability requirements as the number of contents is orders of magnitude higher than hostnames. Our first contribution is thus to design a novel scalable FIB/PIT data structure called FCTree that can provide a varied set of operations. FCTree uses compression techniques to achieve low memory utilization while providing scalable logarithmic time operations. This work led to the publication of a conference paper [2], that was extended to a published journal paper [3].

2. Second, we enhance the NDN architecture by introducing programmability to the data plane using P4 [17]. P4 is a prominent protocol-independent programming language used to instruct switches to execute programs at line rate. Each P4 program defines a pipeline of match action tables (MATs) to control the packets' forwarding behavior. In order to increase the flexibility of the data plane to adapt to potentially competing application needs, we allow application packet flows to be associated with a custom P4 program. Our second contribution is thus to design a novel P4-based data-plane architecture that allows isolated custom P4 functions to be attached to application flows. The programmable ENDN data plane thus offers an extensible catalog of custom network services to applications using these isolated

P4 functions. Our third contribution is to separate the NDN processing pipeline into two consecutive pipelines. The first is a fixed ingress pipeline that implements various content delivery patterns (i.e., pull, push, and publish/subscribe) using FCTree enhanced FIB and PIT tables. This ingress pipeline also associates with packet flows a custom P4 function executed in the second programmable egress pipeline. Finally, we also enhance the data plane with programmable stateful parsers to make it protocol-independent and thus allow interfacing with other types of networks. This work led to the publication of two conference papers [4, 5], as well as an invited poster presentation at the 2020 NDN community meeting in NIST [18].

3. Finally, our last step in the design of the ENDN architecture is to have a full separation of the control and data planes following SDN principles [11] and allow operators to express the application requirements using intents [19]. Traditional SDN architectures separate the control and data planes of the network by centralizing the control plane into a controller entity. The controller has a global view of the network and runs SDN applications to manage the network efficiently. Our final contribution is to use the SDN control plane to capture the application requirements using intents, check their consistency using proof-based formal methods, and enforce them in the data plane. The complete ENDN architecture thus allows operators to specify the applications' custom network service needs to the control plane that generates the required programmable data plane configuration. As a result, the complete ENDN architecture is no longer content-centric: it is now application-centric. This work led to the publication of a journal paper [6].

The rest of this thesis is organized as follows. Chapter 2 contains a background study on different subjects related to this work, especially NDN, SDN, IDN, and programmable data planes. Chapter 3 introduces our novel FCTree FIB/PIT data structure. Chapter 4 adds programmability to the data plane. Chapter 5 introduces the ENDN control plane and enhances the data plane to be protocol-independent using programmable parsers.

Chapter 6 adds the intent-service layer to the control plane and explains how application intents are mapped to data-plane configurations. Finally, Chapter 7 concludes this thesis and presents potential future research areas.

Chapter 2

Background

Our proposed ENDN architecture is based on four main research areas: NDN, SDN, programmable data planes, and IDN. In this chapter, we provide a detailed background study on these four research fields.

2.1 Overview of Named Data Networking

The NDN architecture [8] is derived from the Content-Centric Networking (CCN) project [20]. CCN was introduced as a clean-slate ICN approach to solving the content delivery problem. CCN proposes to directly query the network for contents using only their names, regardless of their physical location. Following the concepts introduced by CCN, the NDN architecture uses data names as its network identifiers. Thus, every NDN name is associated with a single chunk of data that the network can deliver in the form of an NDN Data packet (Dpkt). NDN names follow a hierarchical format. They consist of several name components separated by a '/' character (e.g., */netflix/video/1*). This hierarchical scheme has several advantages. First, it allows applications to provide meaningful context to named content. For instance, a movie streaming application may organize its movies by categories (e.g., */streaming/action/movie4* and */streaming/horror/movie8*).

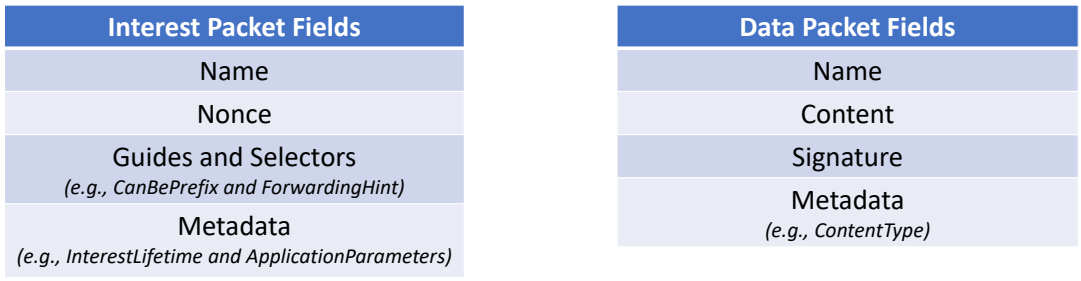


Figure 2.1: The NDN packets format (adapted from [7]).

Second, a hierarchical naming scheme increases name forwarding scalability. For example, all the content names starting with */youtube/* can be retrieved from the YouTube content producers. Hence, a single forwarding rule could be used by the NDN network to retrieve all contents prefixed by */youtube/*.

In NDN, the retrieval of Dpkts is initiated by applications using a simple pull communication service: a content consumer requests the delivery of a Dpkt by sending an associated Interest packet (Ipkt) containing the requested Dpkt name. Fig. 2.1 shows the structure of Ipkts and Dpkts [7]. A notable field in Dpkts is the signature field which securely binds names to contents using the content producer cryptographic signature. Additionally, we see that Ipkts contain several optional selectors and guides to help the network layer retrieve the correct Dpkt. Thus, the main goal of the NDN network layer is to forward an Ipkt to a content producer that can answer with the requested Dpkt. This goal is achieved in NDN switches using several components in the Ipkt and Dpkt forwarding pipelines.

Fig. 2.2 shows the Ipkt forwarding pipeline inside every NDN switch. The most crucial component in the Ipkt forwarding pipeline is the Forwarding Information Base (FIB). The NDN FIB is a routing table containing routes associating a list of next-hops to a content name prefix (e.g., */netflix/**). The FIB is queried using the longest prefix match operation on the Ipkt content name field: among the routes associated with a prefix to the Ipkt content name, only the route with the longest content name prefix is retrieved (cf., Fig. 2.3). It is worth noting that a prefix is understood in terms of

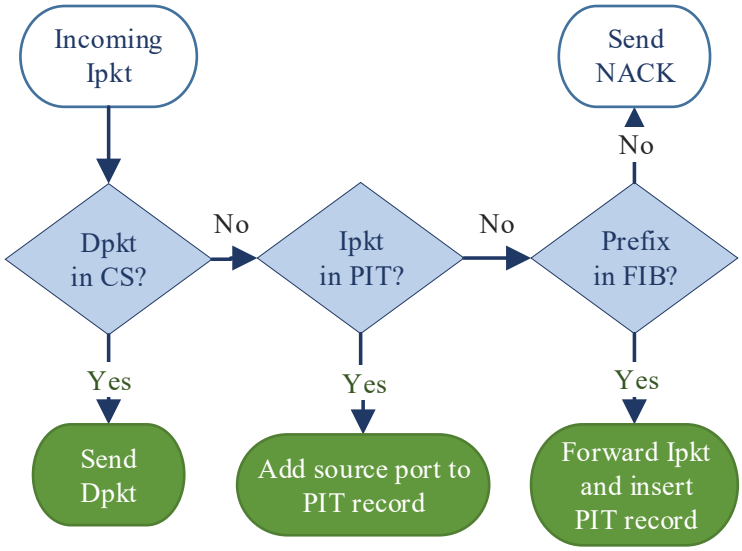


Figure 2.2: The Ipkt forwarding pipeline in an NDN router (adapted from [8]).

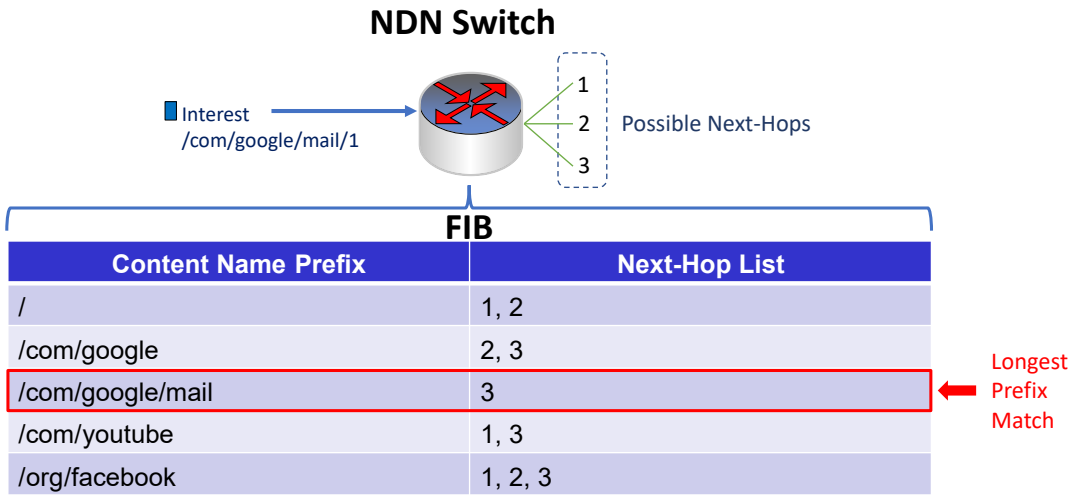


Figure 2.3: Example of an NDN FIB.

components and not in terms of characters (e.g., */netflix* is a content name prefix of */netflix/video/1*, but */net* is not a prefix). When the FIB route is retrieved, the Ipkt is forwarded using a forwarding strategy that can decide which of the specified next-hops of the FIB route to use. For instance, the forwarding strategy can detect congested next-hops and thus forward Ipkts through uncongested paths [21]. Finally, when an Ipkt is successfully forwarded, a new Pending Interest Table (PIT) entry is created.

The PIT stores all the Ipkts that the NDN switch has forwarded until the corresponding Dpkt is received, or the Ipkts expire. Alongside the forwarded Ipkt, the PIT entry also stores the source port from where the Ipkt was received. Hence, the PITs of the different switches store the path that a certain Ipkt followed. This path can then be followed in reverse when the Dpkt is retrieved. Fig. 2.4 illustrates the different paths taken by Ipkts and Dpkts in a simple NDN network. First, Consumer 1 sends an Ipkt for */youtube/video1*. This Ipkt is forwarded using the FIBs in the different switches towards the Producer containing the requested Dpkt. This Dpkt is then forwarded in the reverse path followed by the Ipkt using the PIT entries until it reaches Consumer 1. Another significant usage of the PIT can also be seen in Fig. 2.4 when Consumer 2 sends an Ipkt requesting the same content as Consumer 1. Suppose the second Ipkt reaches a switch that already forwarded an Ipkt for the same content before the associated Dpkt is received (i.e., a switch containing a PIT entry for the same content name). In that case, the second Ipkt is not forwarded to the Producer, and the PIT entry is updated to add the source port from where the second Ipkt was received (cf., Fig. 2.2). Hence, native multicasting is achieved by forwarding the requested Dpkt to both Consumer 1 and 2 using the same PIT entry. Another notable advantage of the PIT consists in the consumer mobility management mechanism it provides. Indeed, as the forwarding of Dpkts always follows the reverse path used by the associated Ipkts, no predefined routes are needed for the consumers. As a result, even if a consumer moves and connects to different edge nodes, no handover procedure is needed as newly sent Ipkts create a temporary Dpkt route back to the consumer.

Finally, NDN switches can also cache forwarded Dpkts in a Content Store (CS). The caching decision is made by a caching strategy [22, 23]. The retrieval latency of Dpkts can thus be reduced by checking the local CS at the beginning of the Ipkt forwarding pipeline (cf., Fig. 2.2).

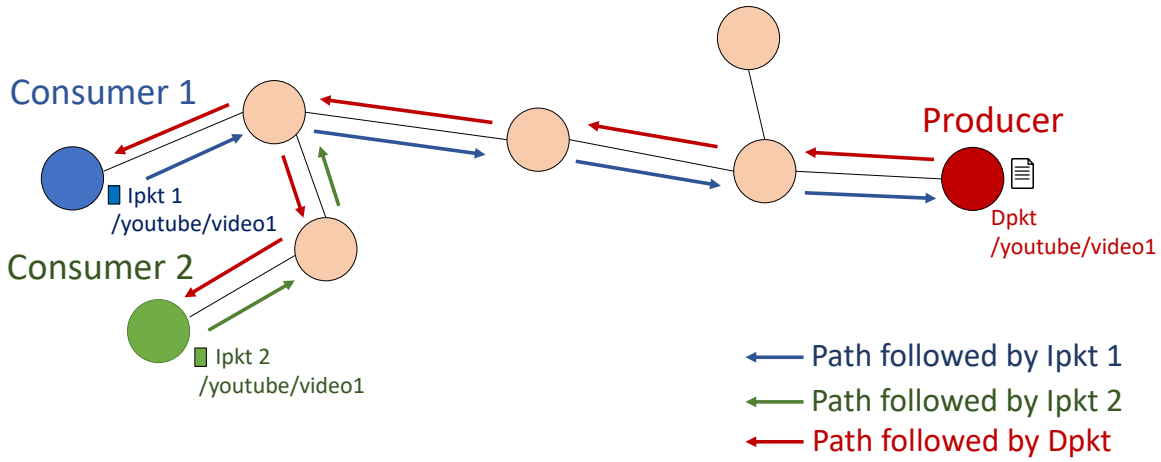


Figure 2.4: Example of an NDN network.

2.2 NDN Management Functionalities

While NDN networks have several interesting features (e.g., location independence, native multicasting), their realization needs to be supported by various network management functionalities. The two main NDN network management tasks are performed by forwarding strategies and routing protocols:

1. *Forwarding strategies:* NDN gives the ability to have multiple next hops per content prefix in the FIB. The forwarding strategy run by the NDN switch is responsible for efficiently using these multiple next hops to implement various network behaviors (e.g., adaptive load balancing, congestion control, and security features like prevention of Denial of Service attacks). While several forwarding strategies were proposed [24,25], the design of efficient and simple forwarding strategies constitutes a nascent domain and thus an open research area.

2. *Routing Protocols:* The routing protocols aim at populating the FIB by ideally providing multiple routes for each destination in the network. The introduction of

content names as network identifiers instead of location-based IP addresses incurs additional complexity and scalability requirements for designing the NDN routing protocols. Moreover, the possibility of caching contents in the CS makes the content routes more dynamic. As a result, NDN routing protocols need to execute faster and more frequently than in traditional IP networks. The design of efficient NDN routing protocols is thus an important research area.

In the next section, we focus on the routing challenges specific to NDN and ICN networks. We will first present the routing techniques used in the current Internet before analyzing key existing name-based routing methods.

2.3 Routing in ICN

The Internet topology consists of a network of different networks, each managed by a different entity. A network managed by a single entity is termed an Autonomous System (AS). The different types of agreements under which two different autonomous systems connect play an important role in routing. Indeed, two different autonomous systems can either be connected through a free peering link, or one of them will pay the other for forwarding its traffic. This introduces a hierarchy among the autonomous systems:

1. *Tier-1 AS:* A small number of ASs that can reach every host in the Internet by using only peering links.

2. *Tier-2 AS:* These ASs can reach several parts of the Internet using peering links but need to pay other ASs to reach the remaining parts of the Internet.

3. *Tier-3 AS:* These are the remaining ASs that have no peering links and only pay other higher level ASs to reach other parts of the Internet.

Economic considerations of different ASs result in different routing requirements, depending on whether a packet is forwarded inside an autonomous system or between

autonomous systems. Indeed, the network inside an AS is completely managed by a unique entity optimizing therefore the network utilization. This is usually achieved by providing each node of the network with the shortest path to each destination in the AS. On the other hand, in the context of inter-AS communication, ASs usually favor forwarding packets to other ASs with which they have a peering link over forwarding them to ASs that require a payment for transit. This could be the case even if the path to reach the destination will be longer when the peering link is used.

These diverse requirements led to the creation of two types of routing protocols:

1. *Intra-domain routing:* These are the protocols that are executed inside an AS. They find shortest-path routes among the nodes in the network domain. They can either be distributed or centralized.

2. *Inter-domain routing:* These are the protocols that are executed at the ASs level. They aim at making every part of the global network reachable. They thus need to handle the scalability issues while creating routes that satisfy the different ASs policies in order to accommodate economic considerations. These protocols are usually distributed.

It is realistic to assume that this hierarchical structure of the current Internet and the related economic considerations will remain in place even if an Internet-scale ICN network is deployed. As a result, ICN routing protocols must be developed for both intra- and inter-domain communication.

2.3.1 Flooding-based Approaches

The first routing approaches that were implemented in ICN networks were methods where no real routing algorithm is used. These are termed flooding approaches.

A prominent instance of these approaches is the default method used in the CCN architecture [26]. An Ipkt is simply broadcasted over the network until a node storing

the requested content is found. The FIB is then enriched when the corresponding Dpkt is received and used to reply to future requests for the same content.

Despite its simplicity, flooding has several advantages. Indeed, in small networks, it can have an acceptable performance with a low complexity. Also, it was shown in [27] that content requests have very strong temporal (i.e., a requested content will be requested again in the future with a high probability) and spatial localities (i.e., it is very likely that requested contents are found in nearby nodes). As a result, a good performance can be experienced by coupling flooding with caching. Furthermore, it was shown in [28] that even when the radius of flooding is restricted, the performance of this mechanism can still be acceptable. These observations led Wang et al. [28] to introduce Pro-Diluvian, a method to dynamically calculate the optimal radius of flooding based on inferred a-posteriori content availabilities.

Unfortunately, flooding approaches suffer from several drawbacks, the main one being their scalability. Indeed, in large networks where contents can potentially be located far from the nodes requesting them, flooding results in a very high bandwidth utilization and congestion where only a small fraction of the broadcasted requests may reach a node containing the requested content. As a result, a large amount of network resources is wasted. A second disadvantage of flooding approaches is that they do not offer lookup guarantee: if the requested content is not stored in any node of the network, the requester will not know it in advance and the request will still be broadcasted in the entire network wasting resources.

These disadvantages led to the introduction of more efficient name-based routing approaches for ICN networks in the form of intra- and inter-domain routing protocols.

2.3.2 Intra-Domain Routing Protocols

In this section, key intra-domain routing protocols for both the current Internet and ICN networks are presented.

2.3.2.1 Intra-Domain Routing Protocols in the Current Internet

Two types of routing protocols were developed and used in the current Internet, namely, distance-vector and link-state. The routing algorithm that was used in ARPANET was a distance-vector routing protocol that was later referred to as the Routing Information Protocol (RIP) [29]. However, in order to solve some of the limitations of RIP, the Open Shortest Path First (OSPF) [30] link-state protocol was developed and is now the most dominant intra-domain routing protocol.

The objective of the two types is to calculate, for a given network topology, the shortest path between every pair of routers in the network. The network topology is represented by a graph where each router corresponds to a vertex and each link between two routers corresponds to an edge with an associated cost assigned arbitrarily by the network administrator.

Distance-vector protocols rely on the Bellman-Ford equation that calculates the shortest path cost to a certain destination using the cost from the current node to its neighbors, and the cost of the shortest path from these neighbors to the destination:

$$d_x(y) = min_n(c(x,n) + d_n(y)) \tag{2.1}$$

In this equation, x is the current node, y the destination node, $c(x,n)$ the cost from the current node to the neighbor node n, and $d_x(y)$ and $d_n(y)$ are the costs of the shortest path from x to y, and from n to y, respectively.

The Bellman-Ford equation states that the shortest path to a destination will go through the neighbor for which the total cost to reach the destination from it is minimal over all neighbors. To use this equation, every router stores a distance vector for itself and each of its neighboring routers. A distance vector stores for each destination the cost of the shortest path to it from a specific node. The router will first initialize its distance vector with the costs of the links to its neighbors. The neighboring routers will then share their distance vectors, and the Bellman-Ford equation will be used by each

of them in order to update their distance vectors. The algorithm will terminate when applying the Bellman-Ford equation does not induce any change to the distance vectors. The distance-vectors are finally used to infer the next hop to use for every destination.

One of the main advantages of distance-vector routing protocols is their locality that is each router only needs to know the information available in its neighboring routers, and not the full topology of the network. This makes them quite scalable. However, they also have several limitations; first, their convergence is slow (i.e., several iterations of distance-vector updates are needed before the algorithm terminates). Secondly, they may suffer from some problems such as routing loops and count to infinity (i.e., when the cost of a link is increased, routing loops appear during several distance vector updates before the algorithm terminates) when link costs are updated.

Link-state routing protocols solve these issues by having each router discover the full topology of the network and then running a graph shortest path algorithm like the Dijkstra's algorithm in order to compute the next hops to each destination. To achieve this, every router will broadcast in the network Link State Advertisements (LSA) containing the list of all its neighbors. The set of LSAs from all the routers is kept locally by every router in a structure called the Link State Database (LSDB). The LSDB is then used to infer the graph of the full network topology from the knowledge of the neighbors of every router contained in the LSAs.

The main disadvantage of link-state routing protocols is their scalability. Indeed, every router needs to know the full topology of the network and then run a shortest path algorithm on it which may be computationally intensive for large networks. However, this is usually mitigated in large autonomous systems by dividing the network topology into several parts (called areas in OSPF) and then having each part run its own link state routing algorithm.

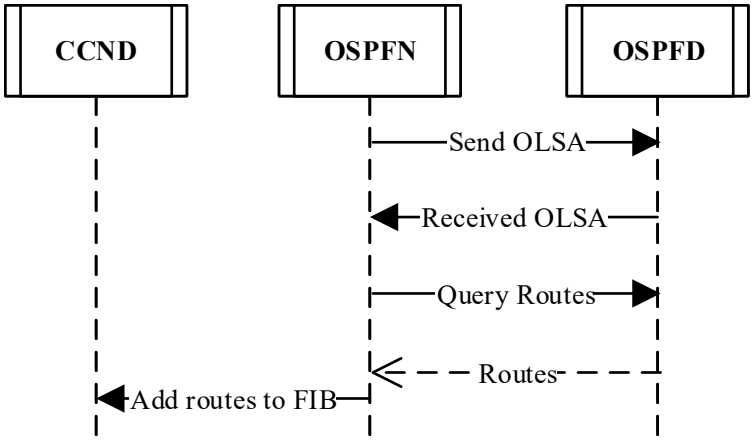


Figure 2.5: Messages between OSPFN and the OSPF daemon (adapted from [9]).

2.3.2.2 Name-based Link State Routing Protocols

The first intra-domain routing protocol that was designed for NDN is OSPFN [9]. OSPFN is essentially an extension of OSPF where special LSAs (termed Opaque LSAs or OLSAs) are used to advertise the contents stored or directly accessible by a router. As a result, OSPFN takes the form of an additional program running alongside the regular OSPF daemon. As shown in Fig. 2.5, new contents are advertised by OSPFN by creating a new OLSA that is forwarded by the OSPF daemon to all the routers in its OSPF area. When an OLSA is received by OSPF, it is sent to the OSPFN program. It will in return query OSPF for the next hop in the shortest path to the router from where the OLSA originated. This information, alongside the content name stored in the OLSA is used to fill the FIB of the router.

While OSPFN is very efficient due to its similarities with OSPF, these similarities also constitute its biggest drawback. Indeed, OSPFN is mainly an extension of OSPF which is an IP protocol: IP addresses are used in LSAs to discover the network topology, name routers, etc. As a result, using OSPFN requires that NDN can only be implemented as an overlay over an existing IP network.

To overcome the limitations of OSPFN, a native link-state routing protocol was developed for NDN, namely, the Named-data Link State Routing protocol (NLSR) [31].

NLSR is very similar to OSPFN, except that it uses the native Ipkts and Dpkts of NDN. NLSR first defines a naming scheme for routers and networks, as well as a method to securely distribute the cryptographic keys used to sign and verify the authenticity of the Interest and Data packets.

In NLSR, every router will advertise two types of LSAs: Adjacency and Prefix LSAs. The former contains the list of the router's neighbors while the latter contains the contents stored or directly accessible by the router. Adjacency LSAs have thus the same function as the regular OSPF LSAs while Prefix LSAs have the same function as the OLSAs used in OSPFN.

The set of all the LSAs in the network form the LSDB that is synchronized between all the routers using the ChronoSync protocol [32]. The LSDB is then used to recreate on each router the graph of the network topology on which heuristics are applied to fill the FIB with a set of next hops for each name prefix. Schneider and Zhang examined in [33] different heuristics that can be run on the topology graph created by NLSR in order to have loop free multipath forwarding in NDN.

To decrease the memory requirements and complexity of the FIB in the context of Internet of Things (IoT) deployments of NDN, Muñoz et al. suggest in [34] to advertise hashes of the name prefixes (computed using iterated bloom filters) instead of the full strings.

2.3.3 Inter-Domain Routing Protocols

This section discusses some of the most prominent inter-domain routing protocols that are developed for both the current Internet and for ICN networks.

2.3.3.1 Inter-Domain Routing Protocols in the Current Internet

In the current Internet, the most widely used inter-domain routing protocol is the Border Gateway Protocol (BGP) [35].

BGP routers are located at the edge of the autonomous systems and are explicitly connected to each other. Each BGP router sends, to its neighbors, an advertisement containing the IP prefixes that are reachable from its AS. It will also forward to its neighbors the routes it receives from other BGP routers. An important attribute of BGP routes is the AS_PATH that lists the autonomous systems that will need to be traversed by a packet in order to reach its destination. The AS_PATH is used in order to detect and handle routing loops, and to select the route that is added to the FIB. Indeed, a specific prefix may be reachable through different autonomous systems, and thus it is expected that several BGP routes towards it will be received by a router. The route that will be used and added to the FIB is selected using a predefined policy (e.g., selecting a route with the smallest AS_PATH or a route that does not go through a specific AS).

2.3.3.2 Name-based Inter-Domain Routing Protocols

Unlike name-based intra-domain routing protocols that share a lot of similarities with Internet's intra-domain routing protocols, especially with OSPF, name-based inter-domain routing protocols are quite different from BGP. This is mainly due to a significant difference in the scalability requirements between BGP and the name-based inter-domain routing protocols. Indeed, there are far more different content names than hosts in the current Internet (the AS65000 BGP table contains 6.9×10^5 active routes corresponding to 2.8×10^9 IPv4 addresses [36] while the estimated number of content names in the current Internet is on the order of 10^{12} [37]). Also, the usual techniques used in BGP to manage the scalability like prefix aggregation have a limited efficiency with content names. This is due to the fact that two contents sharing the same prefix may not be located in neighboring parts of the Internet, and thus may not share the same route. As a result, the proposed approaches for inter-domain routing in ICN do not use BGP to advertise name prefixes instead of IP prefixes. A recent research trend examines the possibility to increase the efficiency of name route aggregation by advertising bloom filters instead of names, but this approach was criticized by Katsaros et al. in [38].

To solve these issues, different approaches are developed for name-based inter-domain routing protocols that can broadly be classified as name resolution and route by name approaches.

2.3.3.3 Name Resolution Approaches

In the name resolution approaches, the content retrieval process has two steps:

1. The location (AS) of the content is first retrieved, usually using a Distributed Hash Table (DHT).

2. The request is then forwarded to the AS storing the content using an underlying location-based network that uses the traditional routing approaches (e.g. IP network with BGP).

αRoute, introduced by Ahmed et al. in [39], is one example in this category. It consists of an overlay network implementing a DHT that is used to store the location of every content, and a mapping scheme that maps the nodes of the DHT to the autonomous systems. This mapping is performed in an efficient manner that enables near shortest path routing in the AS network during queries to the DHT. When a content is requested, the request will first get routed by αRoute to the AS storing the content location in the DHT. After retrieving the location, the request is forwarded to the AS that is storing the content.

One of the main advantages of name resolution approaches is their lookup guarantees. Indeed, by querying the DHT, a user can know for sure if a content is available or not in the network, and if it is, its exact location. However, they also suffer from several drawbacks, the main drawbacks are the requirement of a two-step resolution process during content retrieval and the fact that using a DHT makes the routing suboptimal. Indeed, the requests will first have to travel through different autonomous systems that may not be in the shortest path as part of the DHT query process.

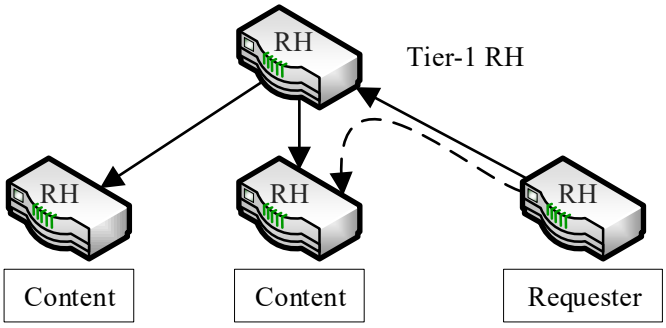


Figure 2.6: The hierarchical organization of the RHs in DONA. The solid arrows represent the different next hops stored for a specific content. The dashed arrow represents the path followed by a query for this content (adapted from [10]).

2.3.3.4 Route by Name Approaches

In the route by name approaches, the content name is directly used to route requests towards their destination. A prominent example in this category is DONA which was introduced by Koponen et al. in [10].

In DONA, every AS has a Resolution Handler (RH) that consists of a table mapping content names to the next hop AS to which a request for these contents needs to be forwarded. Resolution handlers are organized hierarchically following the AS topology of the Internet: every RH has a parent in an upper-tier AS until the root-RH, that is located in a Tier-1 AS, is reached (Fig. 2.6). The contents are added to an RH by register messages sent by the content providers in the AS. These register messages are then forwarded by each RH to its parent which then adds the content to its table. The next hop AS that is set corresponds to the AS of the child RH from which the register message came. When a content is queried, the request is forwarded upward by the resource handlers to their parents until a correspondence is found for the requested content in the table of an RH. The request will then be forwarded to the indicated next hop AS following as a result the reverse path of the register messages.

DONA has several advantages; the main one being its routing performance. Indeed, as the resource handlers are organized hierarchically following the Internet's AS topology,

the content requests are forwarded through a near shortest path route in the AS network. However, it suffers from scalability issues. Indeed, the root-RH would need to store all the content names as every register message will eventually reach it. The requirement of a unique root-RH also induces a single point of failure which can cause reliability and availability issues.

As presented in this section, routing solutions for ICN networks still face both scalability and performance related challenges. These two problems are even more pressing for NDN's inter-domain routing protocols. Additionally, the variable length content names and their expected huge number in an Internet-scale deployment of an ICN network induce several scalability and performance requirements not only for the routing protocols, but also for the NDN forwarding plane. The design of performant scalable forwarding devices that could operate at line speed with high throughput links, as well as the definition of efficient data structures having acceptable space and time complexities to implement the different tables of the forwarding process of NDN, are all challenging tasks that are still unsolved to a large extent.

However, a significant shortcoming of NDN that limits the possible solutions to its scalability challenges comes from the fact that NDN is a distributed network architecture. On the other hand, the IP architecture benefited from several Software-Defined Networking (SDN) developments aiming at separating the control plane from the data plane. The control plane is then implemented by a logically centralized component having a global view of the network. Thus, this global view allows the controller to execute efficient network management algorithms that can solve many of the aforementioned routing scalability challenges. For instance, the FIB size can be reduced by storing in the switches only a partial FIB containing the most frequently used routes while the controller is queried when other routes are needed [40]. Similarly, alternative routing schemes that do not rely on large FIBs in every switch (e.g., source routing) can be used with the support of the controller [41]. In the next section, we provide a background on SDN and present some research proposals combining NDN with SDN.

2.4 Software-Defined Networking

The current Internet was designed as a distributed system where every switch and router runs several distributed algorithms that collaborate with each other to support the forwarding of packets (e.g., the routing algorithms presented in Section 2.3). However, while this distributed design has several advantages in terms of reliability, it introduces an increased complexity when managing the network. For instance, a network operator needs to configure every switch individually, usually using vendor-specific interfaces, whenever a new network behavior is needed. This increased complexity reduced the ability of the network to adapt to new requirements and overall caused a slowdown of innovation in the network layer. A prime example can be seen in the transition from IPv4 to IPv6: while IPv6 was designed several decades ago, its implementation in the current Internet is still incomplete [42]. SDN concepts were introduced in this context to allow for increased flexibility and innovation at the network layer [11].

The main idea of SDN approaches consists of centralizing the network control plane. Distributed network switches and routers consist of two main layers: the data plane and the control plane. The data plane contains the fast packet forwarding logic that processes every packet using simple rules (e.g., the destination-based IP forwarding in traditional IP routers). On the other hand, the control plane consists of the different algorithms running in the router that configure the data plane (e.g., the different routing algorithms that fill the switch forwarding table). The first step performed by SDN approaches is thus to separate the control and data planes and centralize the control plane in a single component: the controller. Then, the different switches become simple forwarding devices configured by the controller using a standardized API: the southbound interface. Network flexibility is thus improved by allowing the programmability of the control plane. Hence, the controller acts as a network operating system providing a global abstract view of the network to several SDN applications running network management algorithms. Thus, implementing a new network behavior becomes a simple update of an SDN application in the controller. Additionally, several algorithms can benefit from

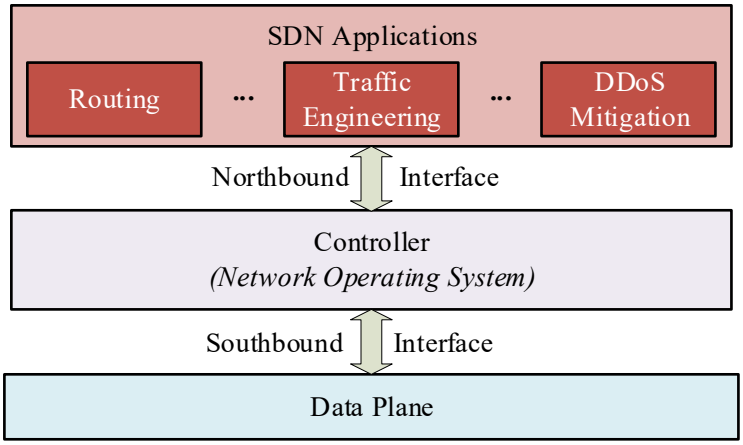


Figure 2.7: Main components in an SDN architecture (adapted from [11]).

the global network view provided by the controller to run more efficiently. For example, contrary to distributed OSPF routers, SDN routers do not need to store the graph of the whole network topology and then compute the shortest paths to each destination. Instead, the controller discovers the network topology and uses an SDN routing application to compute the optimal paths and then update the routing tables of all the managed routers in the domain. Fig. 2.7 shows an overview of a generic SDN architecture.

One of the most prominent components of SDN architectures is the southbound interface. Indeed, it determines the capabilities of the data plane, the structure of the packet processing pipeline, and how it can be configured. The leading southbound interface is OpenFlow [43, 44]. It specifies both the logical architecture of the packet processing pipeline and the network protocol used to configure this pipeline. Every OpenFlow switch is centered around a configurable flow table. The latter is a match-action table that contains flow rules specifying a forwarding action for some packets. More precisely, a flow rule matches on a set of packets, referred to as a flow, using packet header fields values. The set of possible forwarding actions and the possible header fields used during the matching are specified by the OpenFlow specification. However, while the initial OpenFlow specification supported only limited header fields and forwarding actions, the latest version allows for complex behaviors on multiple protocols (e.g., metering and flow table chaining) [44].

While the main southbound interface used by SDN architectures is OpenFlow, there is a large variety of controller network operating systems [45–47]. These controller architectures differ in the different components they contain, the abstract network view they provide to the SDN applications, their support for distributed deployments, and the northbound interface that can be used to program them. However, a significant category of network operating systems consists of network hypervisors [48, 49]. Their main goal is to slice the physical network into different isolated virtual network views that can each be managed by a different controller. Finally, the introduction of SDN led to the development of a large variety of SDN applications for various network management needs like traffic engineering [50, 51], monitoring [52, 53], and security [51].

2.4.1 SDN-based Approaches for NDN

Several research proposals aim at applying SDN concepts to NDN, especially for routing. Like in the current Internet, instead of having every node in the autonomous system recover the whole AS topology and content availabilities, an SDN controller can be used. In SDN-based approaches, routing and content-related information are logically centralized in a single controller that calculates the optimal routes and fills the routers' FIBs. SDAR [54] and SRSC [55] are two examples with SDN-based routing.

These SDN approaches rely mainly on designing an OpenFlow-like protocol with NDN packets and a naming scheme for the controller and the routers' communication. First, every node will advertise to the SDN controller the list of its neighbors and the contents it stores. The SDN controller will then calculate the optimal routes by executing any algorithm used in name-based link-state routing protocols (e.g., Dijkstra's algorithm or some of the heuristics presented in [33]). It will then send these routes to the routers in order to fill their FIBs.

Other approaches use SDN to optimize caching of Dpkts in the CSs. [22, 56, 57]. However, these proposals focus mainly on the cache management algorithm rather than

on the SDN architecture itself.

2.5 Programmable Data Planes

While SDN allowed more programmability and flexibility at the control plane level, the data plane was simplified into a fixed switch architecture like OpenFlow. As a result, the controller has to adapt to the fixed capabilities of the data plane architecture. Additionally, the rigid specification of the data plane prevents it from supporting new protocols and forwarding pipelines. For example, OpenFlow explicitly specifies the header fields that can be used for matching in the flow tables. Supporting a new field thus results in a new OpenFlow specification as well as a forwarding device upgrade. Hence, programmable data planes were introduced to allow more flexibility in the data plane by allowing the control plane to completely specify the packet processing pipeline using a data plane program. One of the most prominent data plane programming languages is P4 [17].

2.5.1 P4 Targets and Architectures

P4 was designed to solve some of the aforementioned limitations of OpenFlow, especially its explicit dependency on packet header fields. Indeed, the P4 language is protocol-independent: it contains a programmable parser component that can recognize and extract any header field needed in the packet processing pipeline. The parsed packets are then processed by match-action pipelines consisting of a sequence of programmable Match-Action Tables (MATs) before being sent to the deparser that converts the packet back to wire format. The MAT structure consists of several matching fields along with a set of programmable actions. Hence, each MAT entry associates values of the matching fields with a programmable action. Every action can then perform several operations like:

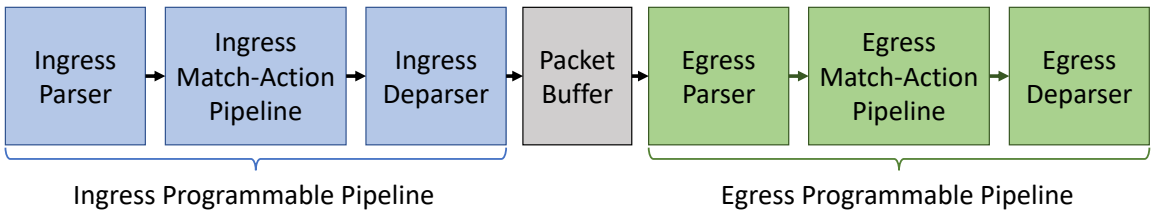


Figure 2.8: The PSA architecture model (adapted from [12]).

- Update values in the standard metadata structure (e.g., set the port through which the packet needs to be forwarded);
- Modify the parsed header fields;
- Modify the value of local variables;
- Call extern functions.

The P4 program contains the parsing/deparsing logic as well as the structure and order of the MATs. On the other hand, the MAT entries can be inserted and modified at runtime by the control plane using the P4Runtime interface [58]. Finally, it is worth noting that P4 does not support loops: all the MATs in the program are executed in order.

P4 is a compiled programming language that is executed by a P4 target [59]. Every P4 target has its P4 compiler and provides a set of standard libraries defining a P4 architecture model. The architecture model defines the P4 program interfaces, i.e., the inputs and outputs of the P4 code. These inputs and outputs are encompassed in the various fields of the standard metadata structure. For instance, an input field for an IP switch could be the packet source port, and an output field may be the face (port) ID of the next-hop. The standard metadata structure is part of a Packet Header Vector (PHV) structure alongside the parsed header fields. Hence, the PHV fields are directly accessible from the P4 MAT actions and can thus be directly modified from the P4 code.

The architecture model also provides an abstraction of the switch and defines the sequence of P4 components (i.e., parsers, MATs, and deparsers) that may constitute a valid P4 program. For example, the Portable Switch Architecture (PSA) [12] provides two packet processing pipelines each with its own parser, deparser, and match-action pipeline: the ingress and the egress pipelines (cf. Fig. 2.8). Finally, the P4 model defines extern functions and objects accessible from the P4 code. Extern objects (e.g., meters, registers, and counters) are used to save the P4 program states, while extern functions can be used to implement complex operations (e.g., hash functions or encryption). As a result, P4 programs are architecture-dependent but target-independent. In other words, a single P4 program can run on multiple targets implementing the same architecture model, like highly specialized hardware-based switches [60] as well as software-based ones [61].

2.6 Intent-Driven Networking

Both SDN and programmable data planes introduce network programmability that can serve as the basis for an application-centric network architecture. However, these solutions rely on a detailed technical knowledge of the network in order to configure it (e.g., to write SDN applications, control plane logics, or data-plane programs). On the other hand, the main premise of IDN is to have networks that are easier and simpler to manage and customize to individual applications and/or industries [19]. IDN allows operators to describe, at a high level of abstraction, the desired business goals as well as how customized network services should behave to serve different applications. IDN can also be employed by application developers to interact directly with the hosted network to specify their required service customization. This section presents a classification of the different types of intents and describes the main intent lifecycle management functionalities based on the model defined by the IRTF Network Management Research Group of the IETF [19].

2.6.1 Intent classification

Within the context of IDN, an intent describes a goal, a constraint, or a desired outcome to be met by the network. The IRTF model classifies network intents along several dimensions [62]. The most relevant of these categories to our context are intent user types, intent type, and intent lifecycle. The first classification category, user types, describes the role of the user issuing the intent, namely, a network operator, an application developer, or an end-user. The second classification category constitutes the most important category used to classify an intent: its type. There are three main intent types:

1. Customer- or application- service intents that describe the desired service quality for a given customer or application (e.g., customers should receive application A videos with high quality and a staleness not exceeding 1 min).

2. Network-service intents describe services that are offered by the network (e.g., best-effort content delivery services should have a maximum latency of 30 ms).

3. Strategy intents describe a desired goal from the perspective of the overall network operation (e.g., reduce overall energy consumption or maintain bandwidth utilization levels and cache occupancy below a given threshold).

The IRTF model includes additional classes in this category, such as cloud management intents (e.g., allocating cloud resources to users), underlay network and service intents (e.g., establishing a connection between two data centers), and operational task intents (e.g., request service migration). We argue that the first three classes are sufficient to describe all network intents, while the remaining classes are geared towards describing network operational policies (e.g., migrating a service) which should be inferred by the management system as a result of higher-level intents. Intents can also be classified according to their lifecycle as either persistent (e.g., all users of given application receive highest video quality) or transient (e.g., remove all cached contents of a given application from the network).

We extend this classification to include two new categories, namely, intent side-effects and intent scope. The first category classifies intents as query, event-reporting, or actionable intents. Query intents answer questions (e.g., how many requests were generated for a given content). Event reporting intents notify the intent user when a certain event takes place (e.g., reporting heavy hitter flows to the network operator). Finally, actionable intents are those that require modifying the data plane. On the other hand, the intent scope specifies the targets of the final realization of the intent. This can include the direct (re)programming of one or more of the data-plane components that process served traffic (e.g., the FIB), or the calibration of switch parameters independently from the served traffic (e.g., configuring values for cache or port queue size). Intents can also result in the execution of one or more of the control-plane functionalities (e.g., calculating optimal placements for a firewall or running a routing algorithm to determine optimal paths) followed by switch configurations.

2.6.2 Intent lifecycle functionalities

Fig. 2.9 depicts the main processing functionalities during the lifecycle of an intent. The figure builds on the IETF standard model [19] and includes two main phases: preproduction and production. During the first phase, the network operator defines the set of intents that the users can employ. Then, depending on the level of automation in the IDN [63], the operator may optionally associate with each intent an intent handler to define the abstract actions that are taken by the network to fulfill the given intent. These handlers can range in complexity from predefined rules to self-reasoning agents that learn and refine the intent handling using feedback from the network. These handlers/rules will aid the intent translation process during the production phase.

The first functionality in the production phase involves ingesting the intents from the users. These users can be network administrators, application developers, or end-users. This step takes place using different text- or voice-based interfaces to type or utter the intents, respectively. Advances in speech recognition and natural language processing

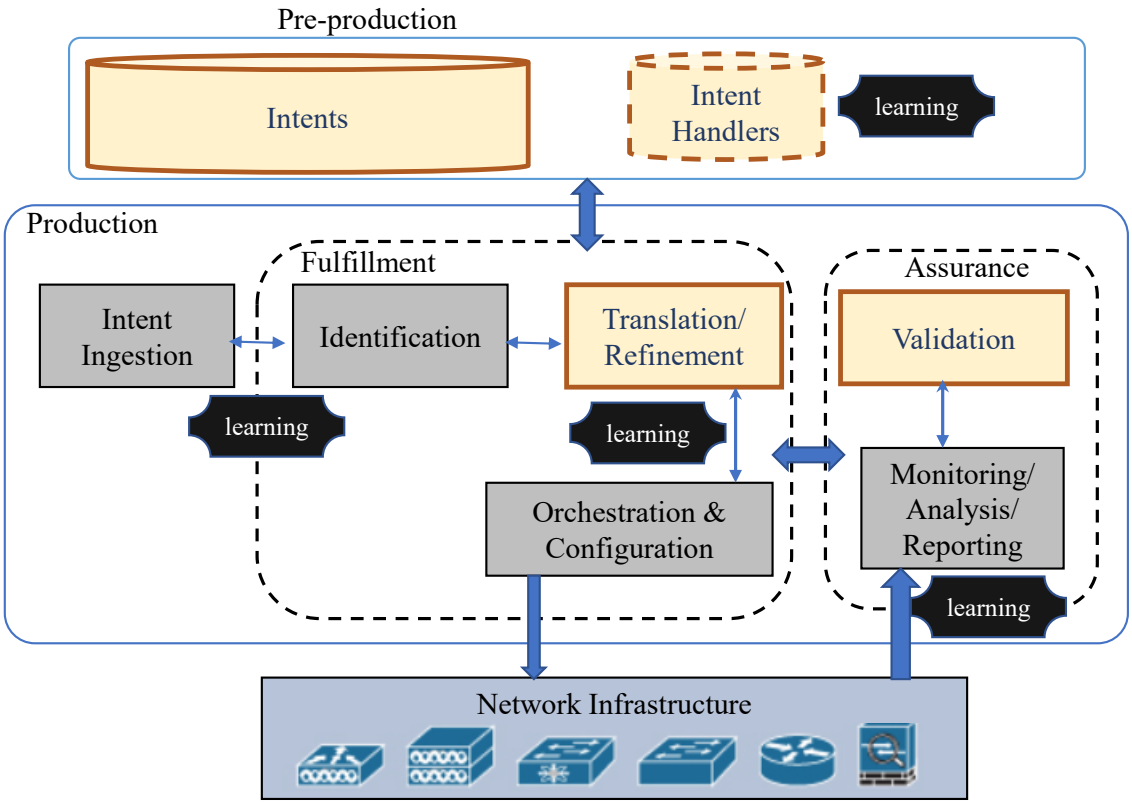


Figure 2.9: Intent life cycle.

allow for the realization of this step [64]. Moreover, the authors in [19] envision this process to eventually include an open dialog between the user and the IDN system in order to aid the user to articulate and clarify the intent gradually.

Once ingested, the intent lifecycle management involves the realization of functionalities that belong to one of two categories, namely, intent fulfillment and assurance [19]. Functions in the first category ensure the realization of the required network configurations to satisfy the intent. Meanwhile, assurance functionalities validate the intents, identify any potential conflicts with already existing ones, and ensure that the corresponding switch configurations realize the goals of the intents and do not drift away from these goals over time.

The first step in intent fulfillment involves identifying the ingested intent. In this step, the intent is rendered in a format that the IDN system can process. This step includes identifying the type of the intent, its application scope, its goals, and/or desired outcomes. It also parses the intent to identify any semantics that the user has provided within ingested intent (e.g., a specific content, time, or service name). The outcome of the identification process is fed to the translation module which maps the intent into actions, management operations or services, as well as network configurations. Any predefined intent rules or handlers that were defined in the pre-production phase can be used as aids to this step. The final stage in the fulfillment of an intent is to translate that intermediate representation into device-specific configurations. The orchestration of the configurations of different devices in the network to respond to different intents also represents an important component of this final stage.

Intent assurance functionalities ensure that the applied network configurations comply with the user intents. These functionalities include intents conflict detection and resolution as well as the assurance that the implemented configurations satisfy the intents. The first step of the intent conflict detection process takes place before the network configurations are deployed. Then during network operation, the traffic is monitored and analyzed to ensure the intents goals are satisfied.

IDN systems are anticipated to be augmented with machine leaning (ML) capabilities [65] that can enhance the performance of various IDN functionalities using learnt experience. For example, intent identification tools may employ ML algorithms to enhance the process of understanding the user input. Similarly, a ML-based translation module can refine its mapping decisions based on the network feedback concerning previous configurations. Finally, ML can be used to monitor and analyze the network feedback and take appropriate actions to correct the data-plane configuration when the network performance shifts away from the intent goals [66].

Using the above framework, we can identify three main areas of research. First, the development of formal models for representing intents and intent handlers is a key step

towards the automation of IDN systems. Second, the development of efficient mechanisms for intent translation into network configurations as well as intent conflict detection and configuration validation before deployment is another challenge. Finally, the last challenge concerns the addition of the necessary intelligence for each IDN system functionalities to ensure its full automation. In our proposed work, we focus on the first two of these challenges.

2.7 Conclusion

In this chapter, we provided a detailed background study on NDN that highlighted its advantages along with several of the challenges it faces. The NDN routing scalability and performance issues constitute one of the most significant of these challenges. While SDN approaches have been successfully used in the context of IP networks to address several similar routing scalability problems, very few research proposals have combined NDN with SDN. Additionally, these research proposals were limited to introducing an OpenFlow-like southbound interface specific to routing or caching without designing a complete integrated architecture that can take advantage of the specificities of NDN. Thus, this thesis aims to improve the current NDN architecture using SDN approaches to address several of the aforementioned limitations. A complementary approach used in this thesis to solve the performance issues faced by NDN lies in the design of novel scalable forwarding tables data structures. Additionally, as recent developments of SDN lead to the concept of programmable data planes, we also aim to design a programmable NDN data plane and discuss how it can be used to evolve NDN from a content-centric architecture to an application-centric one. Finally, we define an IDN model and system for NDN to allow application operators to describe, at a high level of abstraction, how customized network services should process different application packet flows. In the next chapter, we introduce our first thesis contribution: FCTree, our novel scalable forwarding tables data structure.

Chapter 3

FCTree

NDN switches' forwarding information bases (FIBs) store content name prefixes (CNPs) and a corresponding list of next-hops that may have the contents or are on the route to the contents [39]. An NDN switch first receives an interest packet (Ipkt) requesting the delivery of content with a given content name (CN). The router then searches its FIB to find the longest prefix match (LPM) of that CN and forwards the Ipkt through the corresponding port [67]. Once the Ipkt reaches the location of the requested content, a data packet (Dpkt) containing the content is created and forwarded through the reverse path followed by the Ipkt.

LPM is a complex operation in NDN due to the variable length representation of CNs and the size of the FIB in an Internet-scale NDN. Furthermore, stored CNPs are generated from a usually large alphabet space and storing them in an efficient yet searchable manner is very challenging [68]. NDN switches are also designed to store some served contents in their content stores (CSs) to speed up serving newly incoming Ipkts. In turn, caching-aware routing protocols can update the FIBs of neighboring switches to the new locations of the contents [16, 69]. Hence, the data structure of the tables used in the NDN's forwarding plane must support frequent insertions, deletions, and updates as part of its continuous operation.

Research efforts relating to the design of NDN FIBs in the forwarding plane rely broadly on one or more of three well-known solutions, namely, hash tables [39,70], Bloom filters [71] and tries [72, 73]. Structures using the former need sufficient space to store the full CNPs in a table but achieve a constant time search for exact CN matches. They work well for small FIBs, but lookup collisions increase as the FIB size increases. Bloom filter based FIBs are space efficient, but they are probabilistic models and may produce false-positive exact match answers. Both hash tables and Bloom filters do not natively support LPMs and, in general, cannot scale well to the expected large sizes of NDN FIBs, especially in core switches. Trie-based solutions store CNPs in a compressed form and natively support LPMs. However, the space efficiency of tries is limited by the significant number of needed pointers.

In this chapter, we envision that an explicit FIB compression capable of achieving fast LPM and update operations is a challenging necessary step for the scalable operation of NDN's forwarding plane. Based on this premise, we propose a family of new compressed FIB data structures, namely, the front-coded trees (FCTrees) family of FIB structures. Our main contributions are:

1. We propose FCTree, a self-balancing red-black tree (RBT) of buckets of front-coded segments of CNPs.

2. We design novel route LPM, insertion and deletion algorithms that achieve sub-linear time in terms of the FIB size.

3. We demonstrate how to support additional functionalities such as FIB range and wildcard searches in order to serve emerging applications (e.g., in-network search engines).

4. We introduce two additional data structures, namely, statistically compressed FC-Tree (StFCTree) and dictionary compressed FCTree (DiFCTree). StFCTree and DiFCTree achieve a higher compression ratio while providing the control plane with

additional knobs to fine tune the desired trade-off goals between lookup speed and space efficiency.

5. Through theoretical and experimental analysis, we show that FCTrees achieve significant memory compression compared to traditional hash table and trie approaches at the expense of a slight increase in the lookup time.

The rest of this chapter is organized as follows; we identify the main challenges faced by Internet-scale NDN FIBs and define a set of requirements for efficient FIB designs in Section 3.1. Section 3.2 surveys the related work and analyzes the strengths and limitations of existing mechanisms with respect to the proposed requirements. Section 3.3 introduces our first compressed FIB data structure, FCTree. Two additional data structures that further enhance the space efficiency of FCTree are described in Section 3.4. Section 3.5 compares experimentally our proposed solutions to existing approaches using real and large datasets. Finally, Section 3.6 concludes this chapter.

3.1 NDN FIB Design Objectives

Fig. 3.1 shows an overview of the different interactions between a FIB datastructure and other components of the NDN architecture. From these interactions, we identify the main design and functional requirements for developing efficient storage for the FIB tables in the forwarding plane of NDN. We will then show in Section 3.3, how they are satisfied by the proposed FCTree FIB design.

3.1.1 Storage Scalability

NDN routers forward packets by performing LPM on CNs rather than on destination IP addresses [20]. The latter are composed using a two-symbol alphabet, $\{0, 1\}$, and have a comparably bounded namespace size since the length of an IP address is fixed. CNs,

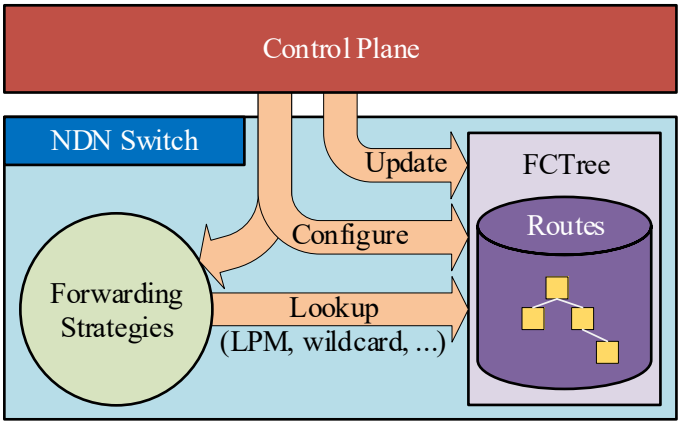


Figure 3.1: Overview of the FIB data structure in the NDN architecture.

however, are generated from a much larger alphabet set and can vary significantly in their length [16]. Intuitively, the number of contents is expected to be orders of magnitude larger than current IP addresses. Additionally, the growth rate of created CNs (e.g., as a result of new news articles, media contents or scientific data) is even exponentially higher than that projected for newly used IP addresses. Hence, the number of routes in a typical NDN FIB is expected to become a significant obstacle to an Internet-scale NDN, specifically with the current trends of designing simple bare-metal software-based switches [74].

3.1.2 Fast CNP Lookup

IP forwarding is performed by extracting the destination IP address and finding its LPM in a single FIB. The fixed length of the IP addresses simplifies the bit-oriented fast searches that can be performed using hardware-based lookup schemes (e.g., ternary content addressable memory-based tries) or by employing parallel lookups using standard SRAMs or DRAMs [75, 76]. On the other hand, serving an Ipkt in an NDN switch involves performing an exact match or an LPM of a variable size name using up to three lookup tables; namely, a pending interest table (PIT) storing already received but unsatisfied interests, a content store (CS) that caches contents, in addition to the

traditional FIB [8]. Hence, name lookup may result in a significant lookup latency in every switch. Furthermore, the sizes of these tables can vary significantly, for example, between edge and core switches. In turn, the variance in the lookup time across these switches may result in a significant delivery-time jitter that can affect the end-users experience. While regular content retrieval applications may not be susceptible to the network delay or jitter, maintaining a negligible and fixed delay is critical for the majority of real-time applications (e.g., streaming [77] and video conference [78]). Hence, the NDN FIB structure must support high-speed LPM with a bounded worst-case delay.

3.1.3 Ease of Maintenance

Route updates in the forwarding plane of IP networks are mainly the result of executing some network applications (e.g., inter- or intra-domain routing protocols) at the control plane [79]. The execution of these applications controls the frequency of these updates. However, in-network caching in NDN contributes significantly to the increase in the frequency of FIB updates as various caching strategies are employed to allow switches to store served contents in their CSs [20]. As a result, the shortest path to a copy of requested content is not static but evolves as the content is cached in several locations. Hence, efficient forwarding operations using the FIBs become very sensitive to the efficiency of FIB insertions and deletions.

3.1.4 Support for Forwarding Programmability

A prominent component in the NDN architecture is its forwarding strategy module [8]. It is responsible for deciding the action that should be performed on any received Ipkt. This module is programmable and can be customized to perform a specific behavior (e.g., load balancing, multipath forwarding, content aware customized delivery, and network level search engine). The design of effective forwarding strategies and their applications is an active research area that aims at adding new NDN functionalities [24, 25, 80]. In

order to offer the most flexibility to the forwarding strategies, the FIB data structure must efficiently support additional types of lookup operations such as range or wildcard search queries that return more than one prefix match.

3.1.5 Adaptability to Switch Heterogeneity

In both IP and NDN networks, different switches are faced with different traffic workloads and have different capabilities. For instance, core switches may be equipped with significantly large resources (e.g., multiple processors and memory hierarchies), but are also required to process a large volume of traffic. On the other hand, routers in access networks may have limited resources and serve smaller traffic volumes. Some switches may have stringent space constraints and require efficient storage management at the expense of a slower lookup speed while a smaller time delay may be preferred over space savings at other switches. Consequently, the FIB data structure should be configurable to adapt to the different specifications of individual routers.

Our objective is to develop a new family of FIB structures, FCTrees, that satisfy the aforementioned requirements.

3.2 Related Work

The Literature includes a large number of existing approaches addressing the scalability challenges faced by NDN content-based routing. The primary objective of these approaches is to reduce the overall state stored in NDN routers as well as the route lookup time. Some of these solutions focus mostly on the reduction of the FIB size while others explore different techniques to decrease the total forwarding state managed in the network and, in turn, reduce the impact of the storage scalability challenges of NDN routers.

3.2.1 Approaches for FIB Size Reduction

In general, existing FIB structures can broadly be classified as those that employ hash tables, Bloom filters or tries.

3.2.1.1 Hash Tables Based FIBs

Hash Tables (HTs) have been used extensively in the Literature as the basis for the NDN FIB data structure [39, 70]. An HT consists of an array containing a fixed number m of cells. A route is inserted in the HT by first applying a hash function to the CNP to generate a hash value in the range $[0, m-1]$. This value is used to select the cell that will be used to store the CNP alongside its list of next hops. When two CNPs are hashed to the same cell, resulting in a collision, a linked list is used to store the CNPs (referred to as chaining). The main advantage of HTs is the constant-time of exact match lookups, insertions, and deletions when no collisions happen. However, as the number of routes stored in the HT increases, with the table size m staying constant, the probability of collisions grows and chaining is used more often. In this case, the time complexity of these operations increases linearly with the HT loading factor (i.e., the ratio between the number of routes n and slots m) [68].

The Name Tree implementation used in NDN Forwarding Daemon (NFD) [70] addresses this problem by continuously resizing the HT whenever the number of routes stored in the FIB reaches a certain predefined threshold (by default 50% of m). Hence, Name Tree incurs the additional cost of resizing the FIB, recalculating the hashes of all the CNPs and moving them to their new positions. On the other hand, αRoute [39] overcomes this limitation by partitioning the routes namespace into fixed partitions, where each partition is hashed and used for LPM operations separately. On the contrary, BFAST [81] reduces collisions probabilities by using several different hash functions and a counting Bloom filter consisting of a counter at each cell of the HT. A route is inserted by applying k hash functions to the CNP and then incrementing the k counters indexed

by these hashes. The route is stored in the cell corresponding to the smallest counter among the k selected counters. The routes stored in other positions are moved to enforce the property that a CNP is always stored in the bucket with the smallest counter among the k counters indicated by the hashes. While performing better than single hashing, BFAST performance still degrades as the FIB grows.

With respect to the requirements introduced in Section 3.1, the constant time complexity of the operations on low collisions makes HTs perform very well during FIB lookup and maintenance operations. However, the performance quickly degrades to linear time as the loading factor increases. As a result, the loading factor needs to be kept low either by oversizing the table or through expensive periodic resize operations. The table size thus constitutes a parameter that can be used to control the space-time trade-off of HTs. Moreover, CNPs are stored in an uncompressed format resulting in a large memory footprint. Azgin et al. improved the space efficiency of HTs by storing the hash value of the CNP instead of the uncompressed string [82]. Finally, as the names are generally spread randomly over the table by the hash function, only exact match lookups are natively supported.

3.2.1.2 Bloom Filters Based FIBs

NameFilter [71] aims at decreasing the space utilization of the HTs by using Bloom Filters (BFs) [83]. A BF is a field of m bits associated to k hash functions returning hashes in the range $[0, m-1]$. A string is stored in a BF by applying the k hash functions to it, and setting to one the k bits indicated by the k hashes. The set membership operation is executed by checking if the k bits pointed out by the k hashes of the queried string are set to one. In NameFilter, every port of the router has an associated BF that stores the CNPs to forward through the port. These BFs are merged in a single structure to decrease the number of memory accesses during lookup. NameFilter significantly reduces the memory size of the FIB while providing constant time lookup and insertion operations. However, BF-based solutions suffer from false-positive errors during lookups

that increase as the number of stored CNPs grows [84]. Additionally, these solutions do not support deletions nor do they offer any additional lookup operations. For a detailed analysis of the limitations of BFs applied to NDN, the reader is referred to Katsaros et al. [38].

3.2.1.3 Trie-based FIBs

Several designs of NDN FIBs that are widely used in the Literature are based on tries. The Character Trie (CT) is a tree where each node corresponds to a CNP and stores the associated next hop list. The root of the CT corresponds to the empty string. Each edge is labeled with a character that extends its source node to form a new prefix. CTs are very efficient at performing LPMs (its complexity is $O(l)$ with l the size of the longest prefix). However, they suffer from a high memory utilization due to a large number of pointers needed (every character added to a prefix increases the number of pointers).

Name Prefix Trie (NPT) [72] overcomes this limitation by reducing the number of nodes in the CT at the expense of increasing the lookup time complexity. NPT aggregates all the characters that form a component to a single node in the tree (e.g., */com/youtube* will be stored in two nodes: one containing *com* and the other containing *youtube*). Wang et al. improved NPT by encoding the name components to reduce the memory size of the tree and implementing the trie using State Transition Arrays (STAs) [72]. STAs allow using binary search during LPM which makes the time complexity of the lookup operation logarithmic on the average number of child edges per node. However, the insertion operation remains linear on the average number of child edges per node. To further reduce the memory footprint, Ghasemi et al. [85] map CN characters to new smaller size ASCII-based characters which are then employed to build a CT. On the other hand, Ctries [73] decrease the memory footprint by using component-based edge labels at the higher levels of the trie and byte-based labels towards the leaves.

While improved, the memory utilization issue of tries remains unsolved due to the significant number of used pointers, chiefly when the CNPs are comprised of a large

Table 3.1: A summary of the advantages and disadvantages of existing FIB data structures.

Requirement	**Hash Table [70]**	**Bloom Filter [71]**	**Character Trie [72]**	**Name Prefix Trie [72]**
Storage scalability	Data stored uncompressed	Lossy compression as data is not explicitly stored	Data stored in compressed form, but huge data structure overhead (pointers)	Data stored in compressed form
CNP lookup	Constant time on average with degradation to linear time as table utilization increases. No explicit LPM support	Constant time in the worst case with false positive error rate that increases as new routes are inserted. No explicit LPM support	Constant time in the worst case	Linear time in the average number of children per node
FIB maintenance operations	Constant time on average with degradation to linear time as table utilization increases	Constant time in the worst case, deletion not supported	Constant time in the worst case	Linear time in the average number of children per node
Additional Lookup Types	Only exact match lookup	Only exact match lookup with a false positive error rate	Exact match, LPM, and prefix search	Exact match, component LPM, and component prefix search
Adaptability to switch heterogeneity	Table size	None	Aggregation level at nodes	Aggregation level at nodes

number of components. In general, as the trie depth increases, trie-based FIBs require a significantly large number of I/Os to traverse downward paths [86]. To reduce frequent memory accesses, Quan et al. [87] proposed a hybrid data structure in which CNPs are split into two parts: a B-Prefix and a T-Suffix. A counting Bloom-filter manages the former part with an associated HT which stores a pointer to the root of a trie that will handle the corresponding T-Suffixes. This data structure shares the advantages of the Bloom filter and trie data structures. However, it still suffers from the problem of high false-positive rates.

CTs meet relatively well the requirements introduced in Section 3.1. CTs store CNPs in a compressed form, as common prefixes between routes are stored once, while providing constant time operations. Moreover, several lookup operations (exact match, LPM, prefix search) are natively supported. However, the space efficiency of CT is limited by the large number of pointers usually needed. Finally, while the granularity level of aggregation at each node can be used as a space-time trade-off parameter (e.g., NPT aggregates by component), the time complexities may become linear.

Little work has been done in the literature to investigate the efficiency of compressed data structures, other than tries, in order to meet the size scalability requirements of NDN FIBs. Furthermore, except for the work of Ascigil et al. [88] where tags are associated with CNs, none of the existing approaches provide additional router functionalities such as wildcard and range-based lookups. These two limitations are the main focus of FCTrees.

A summary of the key advantages and limitations of existing approaches using the FIB requirements identified in the previous section is also presented in Table 3.1. For a detailed review of approaches in each of these classes, the reader is also referred to the work of Li et al. [68].

3.2.2 Approaches for Forwarding State Reduction

Several approaches have also been proposed to reduce the overall memory utilization in NDN routers. Tsilopoulos et al. [89] achieve significant state reduction in routers by having Ipkts tracked every d hops rather than at each consecutive hop in their path. Similarly, Wang et al. [90] classify Ipkts based on their popularity and track only those with high popularity in order to reduce the state stored in NDN routers. Yuan et Crowley [91] develop efficient schemes to reduce the PIT size of each router using novel implementation techniques. On the other hand, Wu et al. [92] reduce the routing table size by introducing a service-oriented network that addresses service names instead of contents. The authors also add a network layer that can forward directly using host addresses when a service is resolved to a specific location. These approaches are complementary to the proposed efficient designs of FIBs.

3.3 Proposed Data Structure

We consider an NDN switch's FIB that contains a set $\mathcal{R}$=$\{r_1, ..., r_n\}$ of n routes for various content name prefixes (CNPs) at a given time instance. Each entry r_i is associated with a list of one or more corresponding next-hop faces. The CNP $r_i \in \mathcal{A}^\infty$ is a variable length string with $\|r_i\|$ symbols generated from an ordered alphabet $\mathcal{A} = \{a_1, a_2, \cdots, a_{|\mathcal{A}|}\}$ of size $|\mathcal{A}|$ with symbol probabilities p_i. N is the total length of the n CNPs in $\mathcal{R}$, where $N = \sum_{i=1}^{n} \|r_i\|$, $N \gg n$. It is worth noting that, in typical implementations of current NDN, r_i is composed of name components separated by "/" to represent a hierarchical namespace (e.g., */ca/uottawa*). However, the proposed data structure is generic enough to include both hierarchical and flat presentations of namespaces. A common prefix between two CNPs or CNs is the substring that is common between them starting from their first symbols. A common component prefix between two CNPs or CNs is the substring of their common prefix that does not contain the last incomplete component. The LPM of a given CN r in an incoming Ipkt, is the route r_i that is a

prefix of r and has the longest common component prefix with r.

Intuitively, without compression, the FIB storing $\mathcal{R}$ requires at least $N \log_2 |\mathcal{A}|$ bits of memory to store the CNPs, where each symbol in $\mathcal{A}$ needs $\log_2 |\mathcal{A}|$ bits to be stored. Additional space will then be needed to index the n routes for non-sequential access.

Our primary objective is to develop a data structure that can efficiently reduce the storage space for $\mathcal{R}$ while not sacrificing the LPM lookup time for r_i in $\mathcal{R}$. Furthermore, the structure should facilitate additional search functionalities where an Ipkt may request k CNs prefixed by a specific name with a wildcard search such as $r*$, (i.e., a wildcard search $r*$ should return a list of up to k routes having r as their prefix).

In the following sections, we first provide a brief description of the front-coding (FC) compression technique [93]. We then show how FC can be employed to store route entries before we finally describe the proposed family of FCTrees FIBs.

3.3.1 Front-coded FIB (FC)

To minimize the memory footprint of $\mathcal{R}$, front-coding (FC), an efficient lossless compression technique [93] can be applied. FC takes advantage of the shared prefixes between consecutive CNPs as follows. The first CNP is stored completely and then each subsequent one is stored as a pair of values $(\|\pi_{i-1}\|, r_i - \pi_{i-1})$ where π_{i-1} is the common prefix between r_i and r_{i-1}. Hence, $r_i - \pi_{i-1}$ is the suffix added to π_{i-1} to construct r_i. As a result, the complete $\mathcal{R}$ set can be stored in a compressed form, $FC(\mathcal{R})$ as follows:

$$FC(\mathcal{R}) := [(0, r_1), (\|\pi_1\|, r_2 - \pi_1), \cdots, (\|\pi_{n-1}\|, r_n - \pi_{n-1})]$$

The memory efficiency of the routers' FIB using FC can be further improved by rearranging the name prefixes based on their lexicographical order as they are inserted. In this manner, we increase the probability of shared prefixes among consecutive CNPs and hence increase their compression ratios.

The space needed to store the CNP suffixes in $FC(\mathcal{R})$ is $\sum_{i=1}^{n} \|r_i - \pi_{i-1}\|$ with the empty prefix π_0. If the route names contain no common prefixes then the size of the stored suffixes in $FC(\mathcal{R})$ reduces to that of $\mathcal{R}$ with no compression. A variable length non-negative integer encoding scheme such as VByte [94] is employed to compress the integer representing the length of the common prefixes π_i, $i = 1, \cdots, n$. The following proposition demonstrates the space efficiency of $FC(\mathcal{R})$.

Proposition 3.1. *The space needed by the $FC(\mathcal{R})$ FIB, that stores $\mathcal{R}$=$\{r_1, ..., r_n\}$, where each CNP $r_i \in \mathcal{A}^\infty$ and $N = \sum_{i=1}^{n} \|r_i\|$, is $\mathcal{C}(FC(\mathcal{R}))$ bits such that:*

$$\mathcal{C}(FC(\mathcal{R})) = (N - n\phi_{\mathcal{R}}) \log_2 |\mathcal{A}| + n \log_2 \left(\frac{N}{n}\right) \tag{3.1}$$

where

$\phi_{\mathcal{R}} \approx \frac{\log_2(n)+\gamma-1}{H(\mathcal{R})} + \frac{H_2(\mathcal{R})}{2H(\mathcal{R})^2}$*;*

$H(\mathcal{R}) = \sum_{a_i \in \mathcal{A}} p_i \log_2 \left(\frac{1}{p_i}\right)$ *is the Shannon Entropy of $\mathcal{R}$;*

$H_2(\mathcal{R}) = \sum_{a_i \in \mathcal{A}} p_i \log_2^2 \left(\frac{1}{p_i}\right)$*;*

γ *is the Euler-Mascheroni constant.*

From the above proposition, it can be seen that $\phi_{\mathcal{R}}$ represents the average length of a shared prefix between two CNPs. In other words, $\phi_{\mathcal{R}}$ represents the average space saved when storing a single route.

To provide a comparison, a hash table, the most commonly used FIB structure, can store $\mathcal{R}$ using $\mathcal{C}(HT(\mathcal{R}))$ bits such that:

$$\mathcal{C}(HT(\mathcal{R})) = N \log_2 |\mathcal{A}| + n \log_2(N) \tag{3.2}$$

The first term represents the space needed to store the n CNPs as indicated before. The latter term is needed to store the optimal table structure (i.e., n pointers to mark the beginnings of n names within the N symbols).

Similarly, a character trie (CT) can store $\mathcal{R}$ using $\mathcal{C}(CT(\mathcal{R}))$ bits such that [95]:

$$\begin{aligned}\mathcal{C}(CT(\mathcal{R})) &\approx N_{CT_{\mathcal{R}}}(|\mathcal{A}|\log_2|\mathcal{A}| - (|\mathcal{A}|-1)\log_2(|\mathcal{A}|-1)) \\ &\approx N_{CT_{\mathcal{R}}}\left(\log_2|\mathcal{A}| + (|\mathcal{A}|-1)\log_2\left(\frac{|\mathcal{A}|}{|\mathcal{A}|-1}\right)\right) \end{aligned} \tag{3.3}$$

where $N_{CT_{\mathcal{R}}} = N - n\phi_{\mathcal{R}}$ is the number of nodes in the CT.

$N_{CT_{\mathcal{R}}}$ represents the space needed to store the compressed version of the CNPs in the CT and is identical to the space saving achieved by $FC(\mathcal{R})$ as discussed in the proof of Proposition 3.1 in the Appendix. As $N_{CT_{\mathcal{R}}} >> n$ in general, $FC(\mathcal{R})$ achieves significant space savings compared to CTs.

While the memory cost $\mathcal{C}(FC(\mathcal{R}))$ is improved relative to hash tables and character tries, to lookup $r = r_i$, the first $i-1$ FIB entries must be reconstructed sequentially by concatenating each name prefix to its predecessor. On average $\frac{n}{2}$ CNP decompressions must be performed, and this can deteriorate to n decoded CNPs in the worst case for each lookup. An additional main limitation with the above representation is that insertion and deletion time complexities in $FC(\mathcal{R})$ also take $\mathcal{O}(n)$.

3.3.2 Bucket-based Front-coded FIB (BFC)

In order to reduce the lookup and update times when using FC, one can partition the list of lexicographically ordered routes into buckets. The first CNP in each bucket is always stored in full and is referred to as the bucket header, the subsequent CNPs in a bucket are differentially encoded using FC. The buckets are then ordered in a list according to the lexicographic order of their headers as shown in Fig. 3.2. We refer to this FIB as a bucket-based front-coded (BFC) FIB.

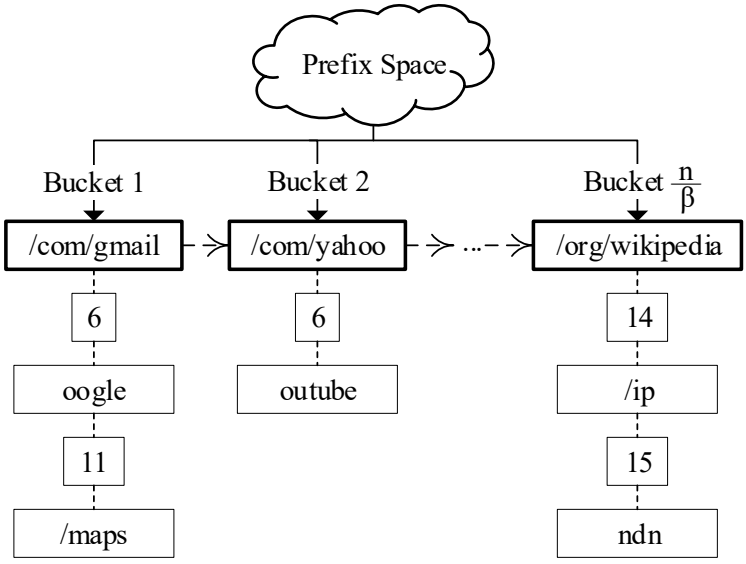


Figure 3.2: BFC FIB with $\frac{n}{\beta}$ buckets. The header of the first bucket stores the content name r_1="/com/gmail" while the second entry in the bucket stores r_2 = "/com/google". Since r_2 shares the prefix "/com/g" of length 6 with r_1, it is stored as (6,"oogle"). Similarly, r_3= "/com/google/maps" is stored as (11, "/maps").

The BFC structure can be constructed incrementally by limiting the number of routes in each bucket to a maximum threshold value $1 \leq \beta \leq n$. The first route in the FIB is inserted in a new bucket. The CNP of the second route to be inserted is compared to the header of this first bucket (i.e., the name prefix of the first route). If the new CNP is lexicographically larger than the first, it is added to the first bucket using simple FC. Otherwise, it becomes the header of a new bucket that is created in the list just before the first one. Consequently, each new route is added to the bucket that precedes another one whose header is lexicographically larger than the route's CNP. We proceed similarly for the next routes where routes are inserted in their appropriate bucket unless this bucket contains β entries. When a route is inserted in a full bucket, we move the last route of this bucket to a newly created bucket. The resulting data structure with buckets $b_1, \cdots, b_{\lceil \frac{n}{\beta} \rceil}$ can be represented as follows:

$$
\begin{aligned}
BFC(\mathcal{R}) := & \\
& \big[\langle b_1 : \quad [(0, r_1), \cdots, (\|\pi_{\beta-1}\|, r_\beta - \pi_{\beta-1})\rangle, \\
& \langle b_2 : \quad (0, r_{\beta+1}), \cdots, (\|\pi_{2\beta-1}\|, r_{2\beta} - \pi_{2\beta-1})\rangle, \\
& \qquad\qquad \cdots \\
& \langle b_{\lceil \frac{n}{\beta} \rceil} : \quad (0, r_{n-\beta}), \cdots, (\pi_{n-1}\|, r_n - \pi_{n-1})\rangle]
\end{aligned}
$$

The space needed by BFC is equal to that of $FC(\mathcal{R})$ in addition to the space needed to store the additional prefixes of the routes in each header and the pointers to each new bucket. The following proposition describes this fact.

Proposition 3.2. *The space needed by the $BFC(\mathcal{R})$ FIB that stores $\mathcal{R}$=$\{r_1, ..., r_n\}$, where each CNP $r_i \in \mathcal{A}^\infty$ and $N = \sum_{i=1}^{n} \|r_i\|$, is $\mathcal{C}(BFC(\mathcal{R}))$ bits such that:*

$$
\begin{aligned}
\mathcal{C}(BFC(\mathcal{R})) = \mathcal{C}(FC(\mathcal{R})) &+ \left(\left\lceil \frac{n}{\beta} \right\rceil - 1\right)\phi_{\mathcal{R}} \log_2 |\mathcal{A}| \\
&+ \left(\left\lceil \frac{n}{\beta} \right\rceil - 1\right) \log_2 \left(N - \left(n + 1 - \left\lceil \frac{n}{\beta} \right\rceil\right)\phi_{\mathcal{R}}\right) \qquad (3.4)
\end{aligned}
$$

In the above proposition, we can verify that when $\beta = n$, $\mathcal{C}(BFC(\mathcal{R}))$ reduces to $\mathcal{C}(FC(\mathcal{R}))$.

In order to update or look up a route, the bucket headers are searched linearly to determine the bucket where the CNP may be located. Using the bucket header, the front-coded CNPs are sequentially decoded inside the bucket and matched against the queried CN. When a route entry is to be deleted, it is removed from its bucket. If a bucket becomes empty, it is deleted, and two consecutive buckets can be merged into one. For a FIB of size n, finding a route takes an average of $\frac{n}{2\beta}$ bucket header comparisons and $\frac{\beta}{2}$ bucket decoding operations. BFC might provide acceptable performance for routers

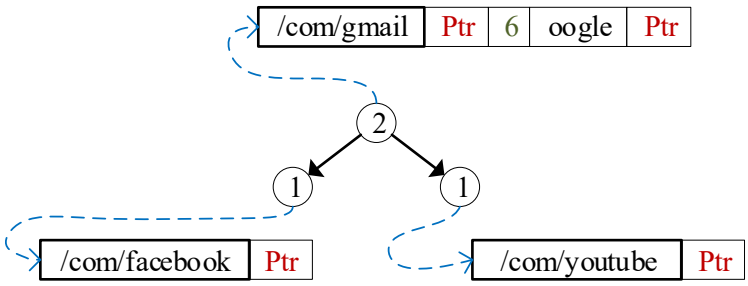


Figure 3.3: FCTree FIB with a maximum bucket size of $\beta = 2$ containing routes for /com/facebook, /com/gmail, /com/google, /com/youtube. Ptr stores the next-hop list.

with commodity multi-core CPUs that can perform simultaneous lookups, one at each bucket [96].

One way to reduce the linear time complexity of BFC is to employ an order preserving perfect hash function [97] to the bucket headers. This function provides an $\mathcal{O}(1)$ access to the appropriate search bucket and uses $\mathcal{O}(\frac{n}{\beta})$ additional space. This implementation reduces the overall access-time complexity of BFC to $\mathcal{O}(1+\beta)$. However, we did not adopt this structure, since it does not efficiently support wildcard and range lookups, and because it requires an apriori knowledge of the content namespace.

3.3.3 Front-coded Tree-based FIB (FCTree)

$BFC(\mathcal{R})$ improves the lookup time compared to $FC(\mathcal{R})$ while providing means to control the space compression using the β parameter. This time gain comes at the expense of a slight increase in the storage space. The proposed $FCTree(\mathcal{R})$ data structure provides a further improvement to the lookup time compared to $BFC(\mathcal{R})$. The efficiency of $FCTree(\mathcal{R})$ stems from its ability to maintain the reduced space of $BFC(\mathcal{R})$ while further reducing the linear-time search needed on the bucket headers to logarithmic time.

In $FCTree(\mathcal{R})$, we structure the buckets using a self-balancing binary search tree. Fig. 3.3 shows an example of the proposed red-black tree (RBT) [98, 99]. Each bucket

is represented by a node in the RBT whose key is the bucket header. Let T be the FCTree representing the FIB routes, then for any node v in T, the left (respectively, right) subtree of v contains only nodes whose keys are lexicographically smaller than v (respectively, larger than v).

To guarantee that the tree is always balanced, the RBT enforces three constraints by performing tree rotations during the insertion and deletion of nodes. Each node in T is marked as either red or black, such that the root of T is black and the parent of any red node in T is also black. Finally, for any node v, every direct path from v to any of its leaf children must contain the same number of black nodes. These properties guarantee that the longest path from the root to a leaf is at most twice the size of the shortest path from the root to a leaf. This ensures that the height of the RBT is $\mathcal{O}\big(\log_2\big(\frac{n}{\beta}\big)\big)$ [99].

Proposition 3.3. *The space needed by the $FCTree(\mathcal{R})$ FIB that stores $\mathcal{R}=\{r_1, ..., r_n\}$, where each CNP $r_i \in \mathcal{A}^\infty$ and $N = \sum_{i=1}^{n} \|r_i\|$, is $\mathcal{C}(FCTree(\mathcal{R}))$ bits such that:*

$$\mathcal{C}(FCTree(\mathcal{R})) \approx \mathcal{C}(BFC(\mathcal{R})) + 2\frac{n}{\beta} \tag{3.5}$$

Moreover, the lookup, insertion and deletion operations need in the worst case $2\log_2(\frac{n}{\beta} + 1) + \beta$ string comparisons.

In the above proposition, we can again verify that when $\beta = n$, $\mathcal{C}(FCTree(\mathcal{R}))$ reduces to $\mathcal{C}(FC(\mathcal{R}))$ (c.f. proof in the Appendix as the first Catalan number $C_1 = 1$).

The LPM operation on a CN is executed as follows. First, a binary search is performed to retrieve the bucket that could contain the full CN. An exact match operation is then performed on the retrieved bucket to check if the full CN has a corresponding route stored in the bucket. If not, the last component of the CN is removed to form the first CNP, and we note that this CNP is always lexicographically smaller than the CN. As a result, the route that corresponds to this CNP can either be stored in the same bucket as the CN, or in another bucket that is necessarily in the left side of the tree with respect

to the first bucket. Hence, we go up the tree until we find an ancestor node whose bucket header is lexicographically smaller than the CNP. We perform a binary search of the CNP on the subtree rooted on the retrieved ancestor node. We then perform an exact match of the CNP on the retrieved bucket. On the other hand, if the root node is reached and its bucket header is lexicographically greater than the CNP, a binary search on the full tree is still performed. The process is then repeated by removing the last component of the CNP until a route is found or there are no more components left in the CNP. In the worst case, the LPM operation would have to return to the root of the tree l times, where l is the number of components in the CN. However, on average, the number of additional string comparisons needed with respect to an exact match on the CN is significantly limited by our LPM algorithm.

The proposed RBT can also be generalized to a B-arity self-balancing tree, where B is the maximum number of children per node in T and $\frac{n}{\beta}$ is the number of buckets in the tree. Hence, the lookup, LPM, insertion and deletion operations all have a time complexity of $\mathcal{O}(\beta + \log_B\left(\frac{n}{\beta}\right))$.

3.3.4 Wildcard Searches in FCTree

In this section, we are interested in adding an additional functionality to the router where the FIB can be queried to return k routes to a content name with a wildcard search $r*$ (e.g., */a/b/mymovie**), where the k returned routes, if any, must have r as their prefix (e.g., */a/b/mymoviepart1* and */a/b/mymoviepart2*). Besides, a range matching request, given two name prefixes r_i and r_j where r_i is lexicographically smaller than r_j, should be allowed. We envision that these functionalities will be essential in future NDN routers. They can serve different types of applications (e.g., real-time streaming [100], network-level search engines [101] and Internet of things applications [88]). They can also resolve problems related to content partitions and visioning [102] in addition to increasing the efficiency of existing routing protocols [103]. These functionalities cannot be easily implemented directly in HT and BF based FIBs since their mapping functions

do not necessarily preserve the routes lexicographical ordering. On the other hand, these two functionalities can be implemented in FCTree as follows.

For wildcard searches, the appropriate bucket for r is first located, and a sequence of decoding operations are performed. The process stops when the k routes are found or whenever a new route is found that is not prefixed by r. Consecutive buckets may be visited to find the needed k routes. The range operation proceeds in the same manner, however, the search starts with the position of the exact match for r_1 and all the following routes are returned as long as they are lexicographically smaller than r_j.

Since our primary focus is on the FIB structure, we leave the details of how the routing protocols and forwarding strategies should use these retrieved routes. One solution to consider is as follows. Wildcard search can be used as a basis for a network level search engine. An auto-completion feature can natively use wildcard searches to suggest the contents available in the network. Wildcard searches can also be performed in the forwarding of special Ipkts that need to target every content starting with a specific prefix. One use of such Interest packet is as follows. A content producer may want to update a meta-data (e.g., copyright notice) associated with all the contents it has published in the network, regardless of their location. The producer could perform this operation by sending a special Interest packet targeting all the contents starting with the producer's prefix.

3.4 Compressed FCTree

In this section, we investigate the ability to achieve further compression for the proposed FCTree structure. We aim at addressing the different requirements of various ingress and egress routers concerning FIB compressibility and LPM speed. While already permitting significant gains regarding space utilization compared to traditional FIB implementations, the memory footprint of FCTree can be further decreased using additional compression techniques applied to the strings stored in the buckets. Hence, by

proposing some alternative additional compression structures and allowing the control plane to specify the appropriate FIB structure for each router, we can achieve a better overall network routing performance. In order to further decrease the memory footprint of FCTree, we apply additional compression to the name prefixes in the buckets. Compression can be applied to both the headers and front-coded content suffixes within each bucket.

A chosen compressed representation for the bucket headers should be order-preserving to allow fast content-name lookups. Since the tree traversal during LPM is performed on the bucket headers, compressing these headers should maintain the same ordinal structure of the tree in the compressed representation. If the bucket header compression is not order-preserving, the compressed name cannot be used directly in the lookup process and must be decompressed first. The time penalty, as the bucket headers are decompressed before every string comparison, becomes very expensive. A more efficient approach would be to compress the CN that needs to be searched and then to perform LPM by comparing this compressed CN with the compressed bucket headers.

Two families of compression techniques that are applied to the proposed FCTree FIB are described next.

3.4.1 Statistically Compressed FCTree (StFCTree)

Our goal is to compress FCTree further using knowledge of the statistical distribution of the number of each symbol in the FIB. By using a variable-length code that assigns shorter codewords to the most frequently used symbols in the FCTree structure, the size of the encoded namespace is reduced. In order to be space and time efficient, these codewords must satisfy the following three properties. First, for these codewords to be instantaneously decodable without the need for additional delimitators, these variable length codes are selected to be prefix-free codes(i.e., no codeword is a prefix of another). Secondly, a symbol with a smaller frequency in the FCTree must be represented with

a smaller codeword than any other symbol that appears with a higher frequency. Finally, the codewords must maintain the same ordinal relationship of their corresponding alphabet symbols.

Let $p'_1, \cdots, p'_{|\mathcal{A}|}$, be the occurrence probabilities of the alphabet symbols $a_1, \cdots, a_{\mathcal{A}}$ in FCTree, and $c_1, \cdots, c_{|\mathcal{A}|}$ be the newly derived codewords satisfying the above properties. Our objective is to create a new structure StFCTree($\mathcal{R}$) that results from replacing all the symbols a_i with their codewords c_i, such that the space of StFCTree($\mathcal{R}$) is minimized. To realize this objective, we employ Hu-Tucker's algorithm [104]. The scheme relies on using an estimate of the symbol occurrence probabilities and constructs the codewords as a binary search tree, hence preserving the lexicographical order of the symbols. Once the codeword tree is constructed, the tree is traversed from the root to each leaf once to find the corresponding binary codeword by assigning zero and one respectively to every left and right child. Hu-Tucker's algorithm produces the optimal variable-length order-preserving prefix-free code for a given statistical distribution of the alphabet symbols [104]. The tree is constructed offline in $\mathcal{O}(|\mathcal{A}| \log |\mathcal{A}|)$ steps and uses $\mathcal{O}(|\mathcal{A}|)$ space to store the codewords. These codewords are used to replace all the FCTree FIB entries as it is constructed hence creating the StFCTree.

The main advantage of StFCTree is that LPM can still be performed on the compressed structure as follows. When a new route query arrives, the new content name for that queried route is compressed using the obtained codewords. This step introduces a small but negligible increase in the LPM time compared to the increase that would be induced if the bucket headers were decompressed before every comparison during the binary search. The compressed content name is then compared with the compressed bucket headers and then within the correct bucket in the StFCTree FIB as was the case for the FCTree structure. As is the case for all statistical compressors, the main limitation of StFCTree is the need for a good estimate of the symbol occurrence probabilities in FCTree.

3.4.2 Dictionary Compressed FCTree (DiFCTree)

To increase the compression ratio of FCTree, we construct an additional lookup dictionary. The dictionary stores CNP components that are repeated in the FIB. For each CNP containing a repeated component, this component is replaced with the position in the dictionary where it is stored. This approach avoids the limitation of statistical compression schemes that require the overall statistical properties of the FIB.

We note that only content suffixes in the front-coded representation of the buckets should be analyzed for component repetitions and not the complete CNPs. As a result, we avoid storing components in the dictionary that are already compressed by front-coding. Furthermore, we only store patterns that represent a full name component in any CNP front-coded suffix (i.e., not the first partially stored component in the front-coded suffix). Consequently, the '/' component separator can be replaced by a marker character indicating whether the following component is a reference to a location in the dictionary or an uncompressed component.

To construct the dictionary for DiFCTree, we modify the existing Lempel-Ziv techniques, LZ77 [105], and LZ87 [106]. In LZ77, a sliding window is used to detect repetitions in a fixed set of strings, and any repeated sequence of characters is replaced by a pointer to the first location where this sequence appeared. On the other hand, LZ78 constructs an explicit dictionary where repeated words are stored and replaced in the data stream by their position in the dictionary. However, LZ78 constructs its dictionary in a deterministic fashion that can also be followed during the decompression in order to rebuild the dictionary. Thus, LZ78 does not require its dictionary to be transmitted with the compressed data stream.

In our solution, we build an explicit dictionary and employ a circular buffer of a fixed size C to store patterns that are candidates to be saved in the dictionary. For each insertion of a CNP in the FIB, the compression is performed component by component. Once the correct bucket in the DiFCTree is retrieved, the remaining suffix is compressed

as follows. Each component in the suffix is looked up in the dictionary. If found, it is replaced by its position. Otherwise, another lookup is performed to search for the component in the circular buffer. The circular buffer stores sequentially the name components that are the next candidates to be stored in the dictionary. Hence, it acts as a sliding window, similar to that of LZ77, to help identify name component repetitions. If a match to the name component is found in the circular buffer, it is removed from the circular buffer, added to the explicit dictionary, and replaced in the inserted route by the position where it was inserted in the dictionary. The '/' character that acts as the name component separator is replaced by a special character that indicates that the following component is compressed. Two hash tables, one for the dictionary and one for the circular buffer are used to speed up the lookup process in both the dictionary and the circular buffer. These hash tables store only the index positions of the strings in the dictionary or the circular buffer to reduce their memory consumption. As a result, compared to LZ77 and LZ78, we no more require the data to take the form of a fixed data stream, at the expense of the need to store the dictionary explicitly.

It is worth noting that the compression techniques proposed above are applied to the buckets after each update operation to the FCTree structure (e.g., insertion of a new route or deletion of an expired one). The bucket is first decompressed fully, then the desired insertion or deletion operation is performed, and if it resulted in a modification of the bucket content, the bucket is compressed and then stored. This results in a bounded increase to the operations time that keeps their time complexities logarithmic on the number of buckets in the tree.

The control plane can adjust the size of the circular buffer of the switch as the FIB is built. At the bootstrap time, a large buffer can help to speed up the process of building the dictionary. Once the dictionary is populated with the most commonly used components in the CNPs, a small buffer is sufficient to capture newly repetitive components.

3.5 Performance Evaluation

In this section, we evaluate the performance of FCTrees against that of hash tables (HTs), character tries (CTs), name prefix tries (NPTs) and BFAST-aCBF [81], a data structure combining Bloom filters with a hash table.

The HT implementation we use is Name Tree, the FIB data structure used in NFD [70]. Name Tree employs an HT that stores nodes which are also part of an associated tree structure. Each node contains a CNP and its associated FIB and PIT entries. The root of the tree structure is the node corresponding to the *'/'* name. The tree structure links together the CNPs in a hierarchical manner (e.g., *'/a/b/c'* is stored in the tree as *'/'* $\rightarrow$ *'/a'* $\rightarrow$ *'/a/b'* $\rightarrow$ *'/a/b/c'*). During the forwarding of an Ipkt, the node that contains the PIT entry is first located by performing an exact match lookup operation in the HT and created if it does not exist, then the FIB entry of the longest content prefix is found by going up the tree structure. To provide a fair comparison with other implementations, we eliminated all additional operations (e.g., PIT insertions and lookups) and separated the FIB structure in the Name Tree from its PIT. We use two HT configurations; the first HT, referred to as *HT (resized)*, is the default configuration in NFD that periodically doubles the size of its table when the loading factor reaches 0.5. The second configuration is a traditional fixed size HT with a table size of 500,000 which induces an average of 20 collisions per cell for 10 million routes. BFAST was configured with a hash table size of 500,000 similarly to the fixed size HT. The Bloom filters of BFAST were configured with 5 hash functions and a size of 50 million resulting in a false positive rate of 10^{-1} for 10 million routes [81].

We implemented CT, NPT, BFAST and our family of FCTrees as alternative FIB implementations for NFD. For CT, we used a pointer implementation where every node contains a fixed array of 96 child pointers corresponding to the size of our alphabet (i.e., the number of printable ASCII characters). For NPT, we also used a pointer implementation where every node contains a chained list of pointers to its children.

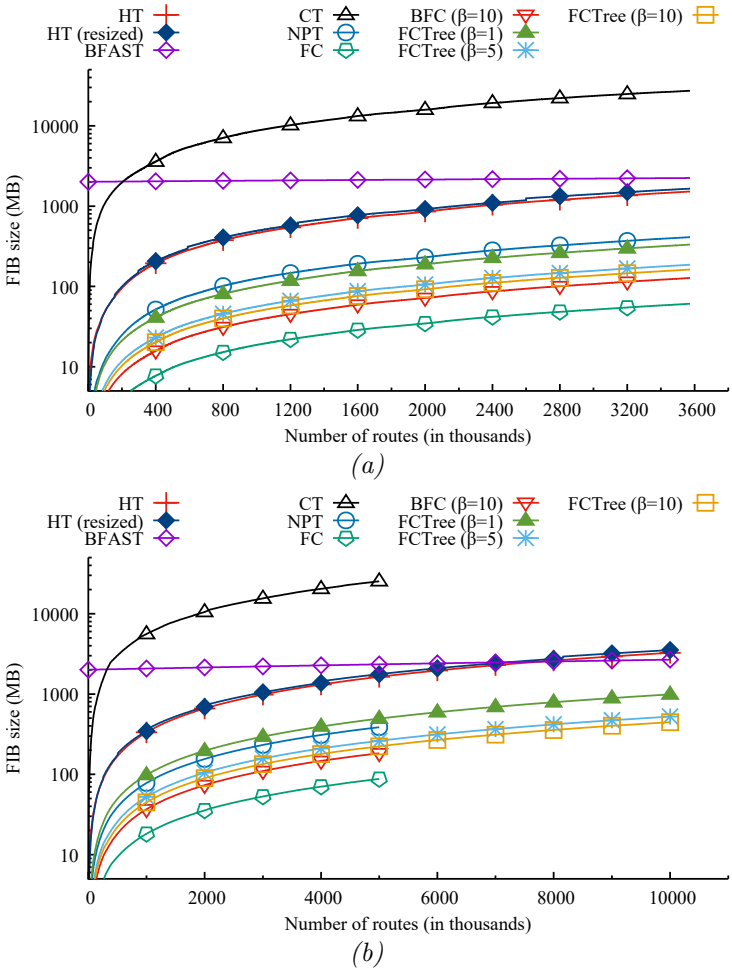


Figure 3.4: Memory footprint for different FIB sizes. (a) DMOZ dataset. (b) Domains dataset.

FCTrees were implemented with an RBT (i.e., we have $B = 2$).

We used ndnSIM [107] to execute our test scenarios. We utilized two sources to construct realistic content prefix datasets that we used in all our experiments: the URLs from DMOZ [108] and the top ten million domains taken from the Open PageRank initiative [109]. The DMOZ dataset contains a smaller number (3.6 million) of longer CNPs compared to the domains dataset. Hence, we can measure the performance of our proposed FCTrees under different CNP dataset characteristics.

All the experiments have been conducted on a server running Ubuntu 19.04 64 bits with an Intel Xeon E31220 (3.1 GHz, 8MB cache) and 32 GB of RAM.

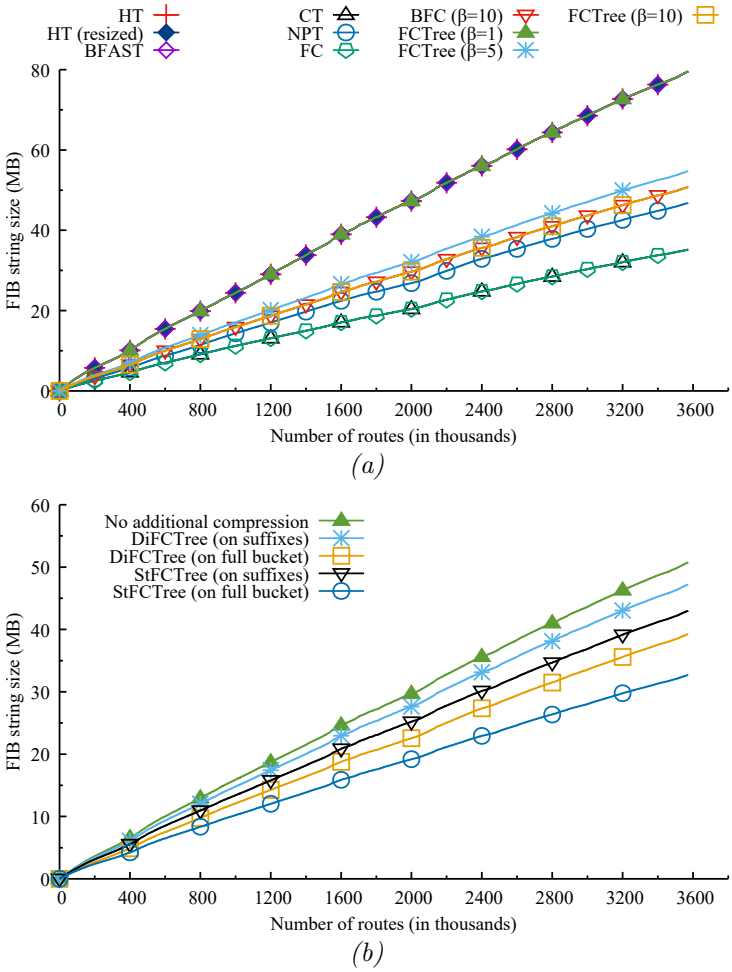


Figure 3.5: CNPs memory utilization for different FIB sizes on the DMOZ dataset. (a) Comparison of FCTree with traditional approaches. (b) Impact of additional compression on FCTree ($\beta = 10$).

3.5.1 Storage Scalability

In the first experiment, we measure the storage size of the FIB structure as we increase the number of routes. We start with an empty FIB and insert additional routes until the entire dataset is loaded. We employ two metrics to measure the storage size: the full memory utilization of the FIB and the compressed string size of the stored CNPs. We hence can differentiate between space savings achieved by the compression of the CNPs and the overhead of the data structure. For NPT, FC, and BFC, the experiment had to be terminated before all the routes in the dataset were inserted due to the considerable

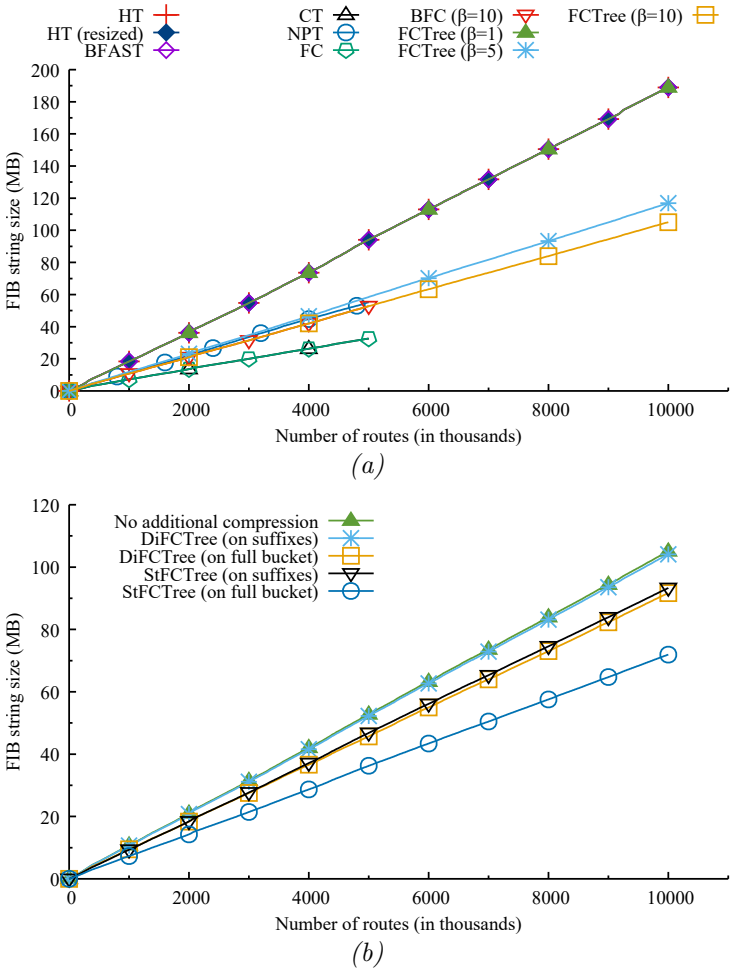


Figure 3.6: CNPs memory utilization for different FIB sizes on the domains dataset. (a) Comparison of FCTree with traditional approaches. (b) Impact of additional compression on FCTree ($\beta = 10$).

per route insertion time needed as shown in Fig. 3.7a.

Figs. 3.4a and 3.4b plot the total memory footprint of the different FIB implementations, respectively, for the DMOZ and domains datasets. On the other hand, Figs. 3.5a and 3.6a plot the size of the compressed CNPs stored in the different FIB implementations for both datasets. We first note that there is no significant difference between the datasets. Hence, the compression techniques used by the different implementations behave equally well with different CNP average sizes.

However, the figures show that for both datasets, FCTrees achieve the most space

savings compared to HT, CT, NPT, and BFAST. This significant difference can be attributed to two factors. First, FCTree (with $\beta > 1$) compresses the CNPs using front-coding while other techniques like HT do not perform any compression. Additionally, FCTree has a smaller data structure overhead compared to the other implementations. The space efficiency of FCTree's data structure is especially visible when comparing it to HT and CT. CT achieves the highest compression level for the CNPs but uses the most memory because of its expensive tree structure. On the other hand, both HT and FCTree with $\beta = 1$ use no CNP compression while FCTree ($\beta = 1$) still achieves a significant memory reduction compared to HT. We verify that the string compression level of FC and CT are exactly the same as expected from our theoretical analysis. Moreover, the increase of the memory footprint from FC to BFC to FCTree is visible.

Figs. 3.5b and 3.6b show the additional space savings achieved by StFCTree and DiFCTree on the compressed CNP size when $\beta = 10$. We note that the statistical compression achieves higher compression ratios than the dictionary compression. Moreover, the dictionary compression behaves better with longer CNPs (i.e., the DMOZ dataset). This can be attributed to the fact that small CNPs are already compressed significantly by front-coding leaving suffixes that may not contain a full component. However, the statistical compression requires an apriori estimate of the dataset symbol statistics while the dictionary compression does not.

3.5.2 Update Operations

In this experiment, we measure route insertion speeds achieved by the different FIB implementations. As the number of routes has a more significant impact on the asymptotic behavior of the insertion performance than the average size of CNPs, we chose to use the domains dataset. Similar to the first experiment, we start with an empty FIB and gradually insert additional routes while measuring the time taken to perform each route insertion operation. The experiment is repeated, and average values for several runs are obtained.

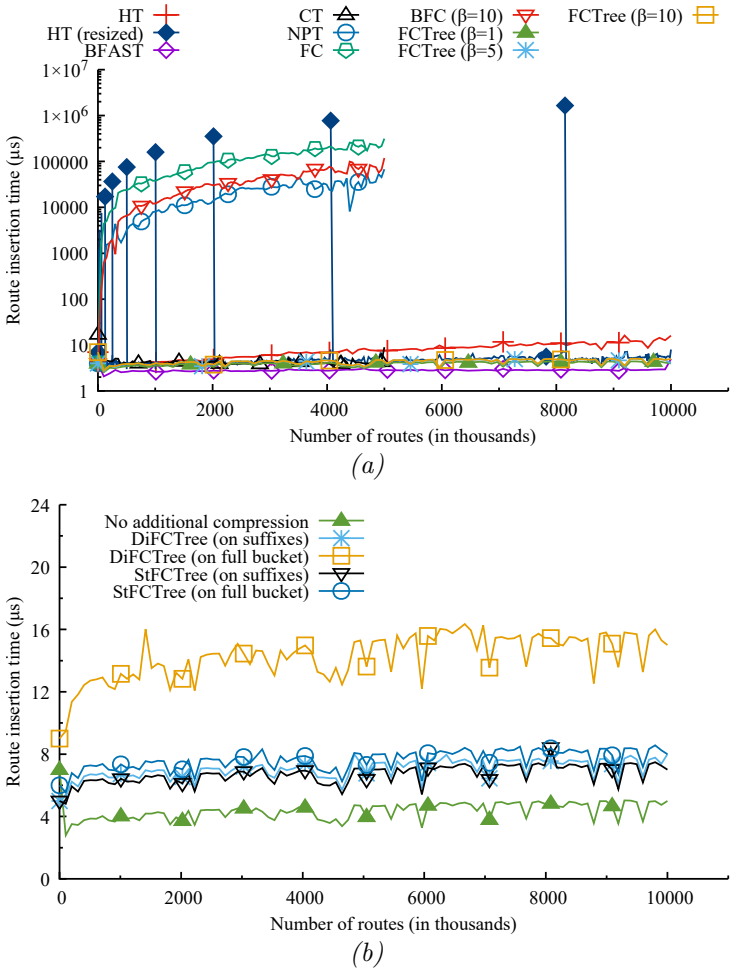


Figure 3.7: Insertion time for different FIB sizes on the domains dataset. (a) Comparison of FCTree with traditional approaches. (b) Impact of additional compression on FCTree ($\beta = 10$).

Fig. 3.7a shows that the majority of the FIB implementations have an insertion speed of the same order of magnitude (less than 10 µs) at all time points except when the HT performs an expensive table resize operation. FCTree's β parameter has a negligible impact on the insertion time. The figure also shows the linear increase in insertion time of FC, BFC and NPT. Finally, we can verify that BFAST's constant time performance does not degrade as fast as the fixed size HT.

Fig. 3.7b shows that the dictionary and statistical compression techniques increase the insertion time of FCTree by a small bounded amount. The increase, in the case of DiFCTree on the full bucket, is higher because of the additional bucket decompression

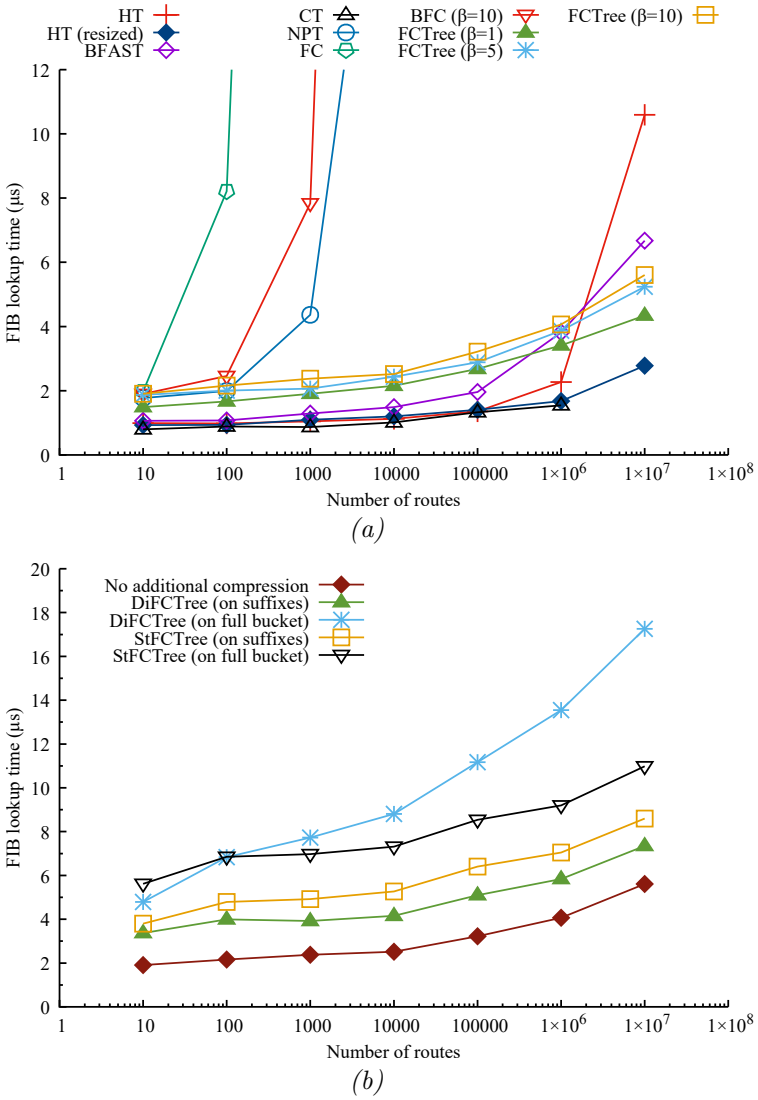


Figure 3.8: LPM time for different FIB sizes on the domains dataset. (a) Comparison of FCTree with traditional approaches. (b) Impact of additional compression on FCTree ($\beta = 10$).

during every bucket header comparison. However, all the techniques result in an insertion performance of the same order of magnitude (10 µs).

3.5.3 CNP Lookup Speed

In this experiment, we create a small network consisting of a consumer, a producer, and an NDN router connecting them using ndnSIM [107]. The router's FIB is filled with a

variable number of routes from the domains dataset. The consumer then sends Ipkts for a random selection of contents, expanded with 0 to 5 components following a uniform distribution, distributed according to a Zipf-Mandelbrot distribution (parameters $q = 0.7$ and $s = 0.7$ as set by the simulator) [110]. Additionally, 25% of the Ipkts are matched by the default route. We then measure the time taken by the LPM operation during the router's Ipkt forwarding process. The experiment is repeated for $10,000$ Ipkts to obtain average values.

Fig. 3.8a shows the measured LPM time for the different FIB implementations. We first note the linear time behavior of NPT, FC, and BFC that quickly makes them unscalable. On the other hand, FCTrees display a clear logarithmic behavior from 10 to 10,000 routes and from 10,000 routes to 10 million routes. The discontinuity at 10,000 routes is explained by an increasing number of cache misses as the FIB memory footprint grows above the 8MB CPU cache size (Fig. 3.4b). The impact of FCTree's β parameter is small (less than $1\,\mu s$). CT achieves the best LPM time, and its $\mathcal{O}(1)$ complexity is again clearly shown in Fig. 3.8a. On the other hand, HT's constant time complexity is relatively well preserved by the periodic table resize operation until large numbers of routes are inserted and the collisions linear time effect becomes more visible, especially for the fixed size HT. Finally, we can verify that BFAST's performance degrades from constant time to linear time at a slower rate than the fixed size HT. While FCTrees are slightly slower than HT and CT, their performance is still of the same order of magnitude.

Fig. 3.8b measures the impact on the LPM time of the additional compression techniques used in DiFCTree and StFCTree. We note that these techniques introduce a bounded small, yet noticeable, increase to the LPM time. However, these techniques do not change the time complexity of the LPM operation that stays logarithmic. Consequently, these techniques can be used to allow the operation of memory-constrained edge routers where the memory scalability is more important than the latency of the operations.

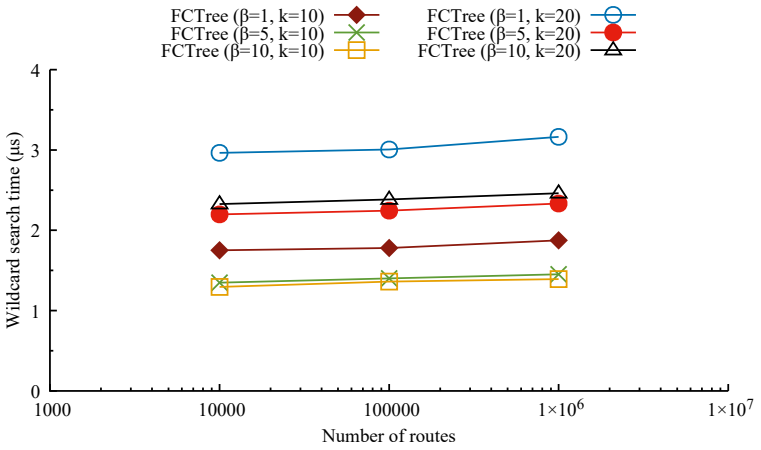


Figure 3.9: Wildcard search time for different FIB sizes on the DMOZ dataset.

3.5.4 Wildcard Search Performance

In this experiment, we populate FCTree with the DMOZ routes and then perform wildcard searches that return up to $k = 10$ and $k = 20$ routes. We then measure the time taken to perform these searches. The experiment is repeated and averaged over 1000 runs.

As shown in Fig. 3.9, in general, all searches remain between 1 to 3 µs. The worst performance is when $\beta = 1$ and the router is asked to return up to 20 routes since the binary tree traversal dominates the search time. On the other hand, when $\beta = 10$ and the wildcard request was set to return 10 routes, the lookup operation was very efficient. This efficiency stems from the hardware cache that can accelerate consecutive memory accesses in the same bucket. Finally, for all the plots in Fig. 3.9, we can see that the wildcard search operation remains logarithmic.

3.6 Conclusion

In this chapter, we designed and analyzed a family of novel compressed forwarding information base (FIB) data structures for Named Data Networking (NDN) routers. The first proposed structure, FCTree, is a space-efficient data structure that relies on local

compressions of lexicographically ordered name-based routes. These routes are stored as partitions, or buckets, in a self-balancing searchable tree structure. The significant space savings, when compared to traditional solutions, come at the expense of a small increase in the lookup time. More precisely, FCTree has a longest prefix matching (LPM) time complexity that is logarithmic in the number of buckets used in the tree but efficiently meets the scalability requirement of an Internet-scale deployment of NDN. We have also shown that new functionalities such as wildcard searches for content names can be efficiently supported. Finally, we introduced additional compression techniques that can be applied to FCTree in order to further decrease its memory footprint at the expense of a slight increase in the lookup time. We thus provide the control plane with different knobs to adjust the space-time trade-off of the FIB data structure in order to adapt to the specific requirements of each switch optimally. In the next chapter, we introduce our proposed programmable ENDN data plane.

Chapter 4

Programmable Data Plane

The simple design of NDN allows it to serve remarkably well the needs of static content delivery applications at the expense of less adaptability to other types of applications. For instance, NDN lacks a native push mechanism critical for several applications (e.g., IoT) [111]. As a result, similar to IP, several network requirements must still be implemented at the application level in NDN (e.g., Ipkt pipelining to support real-time traffic [77, 78, 112] and virtual Ipkt polling to support push traffic [111]).

Both IP and NDN provide a simple but limited delivery service to the applications. Hence, to meet the evolving application needs, multiple network-related services were implemented at the application level, sometimes resulting in reduced application performance. A prominent example is the TCP congestion avoidance mechanism, which is implemented at the end-points, that suffers from delayed detection of congestions at the network core.

This problem is becoming more prominent as network operators are increasingly transformed into content and service providers. In this case, a single administrative entity manages the network (e.g., ISP networks, content provider distributed data centers) and the applications hosted on it. Thus, it is critical for these network managers to optimize their network resources to best serve their application needs. However, while

a simple best-effort delivery mechanism, as provided by IP and NDN, is efficient for a global networking architecture as proven by the current Internet, it lacks sufficient manageability for these network operators. Following this trend, recent research efforts also aimed to adapt existing network architectures to specific applications (e.g., IoT and smart grid). One example is the work of Ravikumar et al. who modify NDN to serve the needs of Wide Area Measurement Systems [113].

In our proposed work, we believe that hosted applications should provide the network with requirements for processing their flows. The network control plane uses these requirements to generate the corresponding data plane configuration. More precisely, we envision that the application and network layers must be subject to a strict separation of concerns. The application layer's primary concern is content generation and identification of content sharing and delivery patterns. The network layer's primary concern is implementing the applications' requirements into data plane configurations. The network layer should provide a store of services to the application layer to achieve this separation of concerns. Examples of these services include various content delivery patterns (e.g., publish/subscribe or notification), content name rewriting, and adaptive forwarding (e.g., choose the next-hop port based on measured statistics). The application layer defines its content delivery requirements by declaring a content namespace (i.e., a set of content names usually defined by a content name prefix using the same hierarchical format as NDN [8]) and a set of chosen content delivery services chained together. Additionally, the catalog of content delivery services may be extended at any time to provide new functionalities required by novel applications.

In this chapter, we introduce the ENDN data plane that allows the implementation of this rich catalog of content delivery services. ENDN enhances the traditional NDN data plane by providing a fully programmable forwarding pipeline consisting of two main modules. The first, the Enhanced packet Processing (EProcessing) module, is a table-based scalable ingress pipeline. The EProcessing module enables the forwarding of packets using different content delivery patterns. For each packet, it generates a set of

metadata fields to be used by the second programmable egress module, the Forwarding Logic. In turn, this module consists of a set of custom forwarding functions programmable using an extended version of P4 (a commonly used data plane programming language [17]). It is worth noting that ENDN operates using the standard NDN packets. ENDN switches can thus coexist/communicate with NDN switches.

Our main contributions can be summarized as follows:

1. We enhance the design of the traditional NDN forwarding pipeline and the existing FIB and PIT tables in order to provide support for several content delivery patterns (e.g., publish-subscribe, push, and stateless pull).

2. We extend the NDN data plane with a new P4 target architecture. This novel P4 target architecture allows the definition and execution of multiple isolated P4 forwarding functions, thus, enabling P4 code run-time pluggability.

3. We extend the P4 language with several extern functions allowing network services to access or modify the string-based NDN packet content names.

The rest of this chapter is organized as follows; Section 4.1 surveys the related work. In Section 4.2, we present the architecture and operation of the ENDN data plane. Section 4.3 illustrates the capabilities of the ENDN data plane with a prototype implementation. Finally, Section 4.4 concludes this chapter.

4.1 Related Work

Our proposed architecture relates to two main streams of research efforts: network programmability and NDN forwarding functionalities.

4.1.1 Network Programmability

As the basis of the current Internet, many solutions have been developed to increase the efficiency of the TCP/IP network layer. While early approaches focused on specific hardware solutions (e.g., firewall and load-balancer) or specific application-level solutions (e.g., Content Delivery Networks and DNS), recent solutions adopt software-based network designs.

Software-defined networks (SDN) [11] improve the manageability and evolvability of the network layer by separating the control and data planes. The control plane is logically centralized in a programmable entity, the controller, having a global view of the network. By providing programmability to the network layer, SDN allowed the implementation of several algorithms to support application network requirements [114–116]. However, the lack of data plane programmability in SDN, especially in the implementations based on the OpenFlow protocol [44], results in several problems. For instance, the controller's need for an accurate global view of the state of the data plane may induce a significant monitoring traffic overhead from the switches [117, 118]. Moreover, the simple design of the OpenFlow switches relies on strong assumptions coupling higher layer applications to the header fields of lower-layer network protocols. As a result, enforcing application requirements through an OpenFlow SDN network is particularly tedious and error-prone due to possible low-level rules conflicts [119].

P4 [17] is a data plane programming language that was introduced in large part to mitigate some of the aforementioned limitations of OpenFlow. P4's design stems from the idea that the controller should be able to inform the data plane on how to process packets instead of adapting to a fixed switch architecture. As a result, the data plane becomes protocol independent and programmable by the controller. P4 is used to specify how a switch parses packets and then forwards them to the next hops. A single P4 program defines the complete behavior of a switch and is used to configure it. Several proposals used P4 to implement complex forwarding behaviors in the data plane [120–122]. However, even with P4 programmability, the current TCP/IP architecture

lacks adaptability to the needs of applications without increasing their design complexity.

4.1.2 NDN Forwarding Functionalities

As a clean-slate network architecture, NDN mitigates several limitations of TCP/IP, especially for simple content delivery at the network layer. Several solutions are proposed in the Literature to support additional application requirements in NDN efficiently. Some of the most studied applications in NDN are real-time applications with strict low latency delivery. Indeed, NDN's strict pull mechanism may not be adapted to real-time applications as the delay is at least equal to a round-trip-time. Persistent interests (PI) [123] constitute one of the early solutions proposed to support real-time traffic. Ipkt pipelining has also been used as a solution to support real-time applications in NDN [77, 78, 112]. Another type of applications, Wide Area Measurement Systems, have been investigated by Ravikumar et al. who propose several solutions to adapt NDN to their specific requirements [113].

Some efforts have been proposed in the Literature to support additional content delivery patterns in NDN. Amadeo et al. presented three solutions to enable push traffic in NDN [111]: interest notifications, unsolicited data, and virtual interest polling. On the other hand, Ghali et al. challenge the benefits of strict stateful forwarding for all traffic in NDN [124]. They introduce a stateless delivery service where Ipkts and Dpkts have an additional source and destination address, respectively, that are used to route Dpkts.

NDN's design allows programmability in the data plane through forwarding strategies (FS) [20]. In the NDN Forwarding Daemon (NFD) [70] implementation, FSs are pieces of code bound to a specific content namespace. FSs main goal is to determine on which next-hops Ipkts have to be forwarded, enabling adaptative forwarding [24, 125, 126]. However, NDN forwarding strategy capabilities are not standardized and are thus implementation-specific. Moreover, FSs are selected by network operators, and NDN offers no guarantee

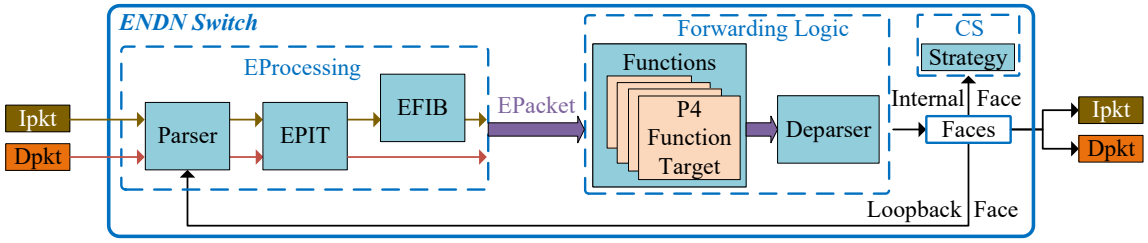


Figure 4.1: Overview of the ENDN data-plane architecture.

for application developers that a specific FS is configured in the network [127]. As a result, FSs are in practice application-agnostic and focus mainly on network management issues.

Several approaches have been proposed to build an NDN network using SDN solutions [128]. Some of these methods focus on providing controller-based routing [54, 129, 130]. Sun et al. introduce a controller-aided traffic management application that classifies NDN flows in different priority classes and assigns them to queues in network devices [131]. Other approaches address cache management challenges using controller-based solutions [22, 56, 57]. Finally, Signorello et al. present a proof-of-concept of a P4 implementation of an NDN switch and highlight the shortcomings of P4 in supporting a stateful variable length protocol, as in the case of NDN [132].

In summary, while most approaches focus on improving NDN through application-level solutions, few approaches address the problem of making the NDN data plane sufficiently programmable to adapt to the applications needs. Moreover, while FSs constitute an early attempt to introduce more programmability in the data plane, the lack of direct control of the applications on the FSs deployed in the network limits the practical use cases of FSs. These two limitations constitute the main motivations for the ENDN data plane.

4.2 Proposed Data Plane Design

The design of ENDN's data plane aims at providing the control plane with run-time programmability of the data plane using P4. As shown in Fig. 4.1, packets are first processed by the EProcessing module. It parses the packets and queries the EPIT and EFIB forwarding tables to construct an Enhanced Packet (EPacket) containing additional metadata stored in the corresponding EPIT/EFIB entry (e.g., P4 function name, and next-hop list). The EPacket is then passed to the Forwarding Logic module, which contains a set of independent P4 functions. The desired P4 function is selected to process the supplied EPacket. After the P4 function is executed, the EPacket is deparsed and forwarded through the selected next-hop faces.

In contrast to existing approaches (e.g., Signorello et al. [132]), ENDN does not implement the full data plane functionality using a single P4 target. We limit the usage of P4 to only realizing complex forwarding functionalities. Furthermore, instead of using a single P4 program per switch to contain services, in ENDN, multiple P4 functions run in isolation with separate targets and, thus, maintain separate states. These design choices stem from the following two main reasons.

The first is to simplify the (re)configuration process when a namespace or its services are added/modified. Without isolated P4 function targets, the complete P4 code of the switch must be replaced, which is equivalent to an expensive firmware upgrade operation. The new P4 code contains the new logic inferred from the set of services linked to the namespace. With isolated P4 function targets, we allow the update of a single P4 function without impacting other parts of the switch, thus enabling run-time P4 code pluggability. Having multiple isolated P4 programs run in parallel on the same hardware was achieved using FPGAs in the context of network slicing [133]. However, our design differs in the nature of the P4 programs that are run in parallel: the P4 functions are not full P4 programs defining the switch's whole behavior. Since the parsing and deparsing operations are carried out outside of the target, our P4 functions are lightweight

P4 match-action pipelines acting on a parsed EPacket constructed by the EProcessing module.

The second reason is to overcome the limitations of P4 with respect to string processing [59]. P4 was introduced in the context of the TCP/IP architecture with a strict line-speed performance objective. P4 does not allow loops and allows only compile-time string constant values. As a result, a full implementation of the NDN data plane in P4 requires strict assumptions and constraints on the format of content names (e.g., number of content name components) [132]. In ENDN, the EProcessing module pre-processes the Ipkts and Dpkts headers to generate corresponding EPackets through simple EFIB and EPIT table lookup operations. The lookup mechanism used in the EFIB and EPIT tables is the longest component prefix match used in NDN. We employ our previously developed data structure, FCTree, to implement the EPIT and EFIB tables, but other optimized time- and space-efficient data structures can be used [68]. The EPackets can be processed by P4 code efficiently without being subject to the string processing limitations of P4. Finally, in order to allow complex forwarding functions that require access and modification to content names in the packet headers, we extend P4 with a set of extern functions that are not implemented in the P4 language.

The following sections describe the main components of the ENDN data plane in more detail.

4.2.1 EProcessing Module

The EProcessing module contains a parser and two forwarding tables that are based on the NDN forwarding pipeline. However, the structure of the existing FIB and PIT (Fig. 4.2), as well as the Ipkt and Dpkt forwarding pipeline (Fig. 4.3), are modified. The main motivation of these modifications is to support some core content delivery pattern services (e.g., push, publish/subscribe) natively as described next.

The first modification we propose is to promote the PIT as a routing table for Dpkts.

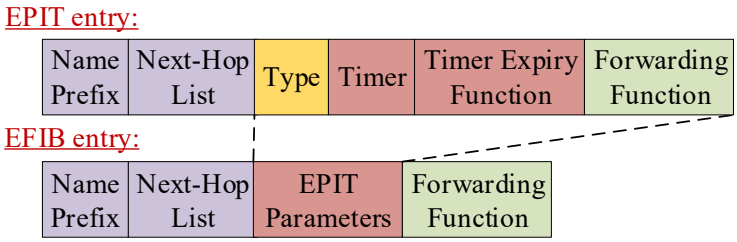


Figure 4.2: Structure of EPIT and EFIB tables.

A traditional NDN PIT enforces the one-to-one relationship between Ipkts and Dpkts, leading to the strict pulling mechanism of NDN. In order to support additional content delivery patterns, ENDN's data plane has an Enhanced PIT (EPIT) where entries do not correspond only to previously forwarded Ipkts. The controller can insert some EPIT entries to configure the forwarding of Dpkts proactively. EPIT entries contain a content name prefix and the associated list of next hops towards which Dpkts are forwarded. Rather than an exact match, the lookup operation in the EPIT uses a longest prefix match (LPM) operation on the Dpkt content name (same lookup operation as the one used in the NDN FIB for Ipkts).

EPIT entries have a new field, type, that can be set as either persistent or non-persistent. Persistent EPIT entries are not removed when a matching Dpkt is forwarded while non-persistent EPIT entries are. All EPIT entries contain an expiration timer that is used as a form of garbage collection to delete unused EPIT rules. The controller can configure the data plane to execute a special P4 function when an EPIT entry expires, or an LPM operation is unsuccessful, for example, to notify the control plane or the application if needed.

As shown in Fig. 4.2, the Enhanced FIB (EFIB) contains the same fields as the EPIT. Those fields marked as the EPIT Parameters in Fig. 4.2 are used when inserting a new EPIT entry. The EPIT Parameters field also specifies whether an EPIT entry needs to be created when forwarding an Ipkt, thus allowing stateless pull forwarding similarly to the work of Ghali et al. [124]. Fig. 4.3 describes the overall processing of the packets using the new EPIT and EFIB fields. Finally, both EPIT and EFIB entries

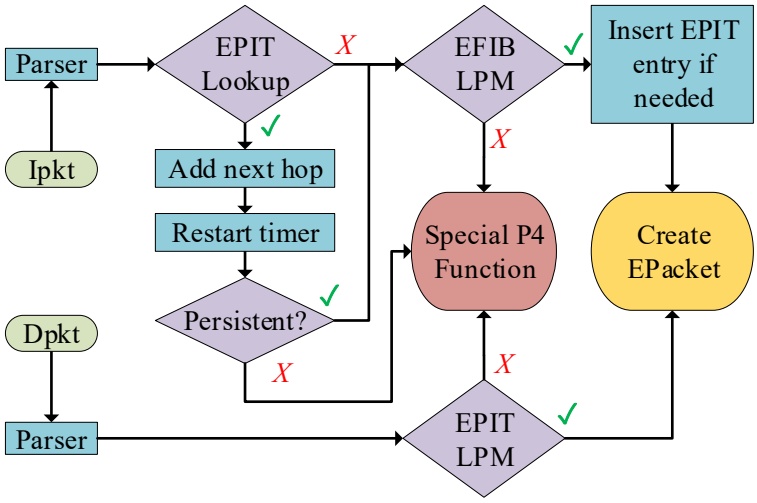


Figure 4.3: Ipkt and Dpkt EProcessing pipeline.

have a forwarding function field that contains the name of the P4 function to execute alongside a set of parameters to pass to the P4 function.

These modifications allow ENDN's data plane to support three types of content delivery patterns:

1. *Push traffic*: a push mechanism can be configured by the control plane by proactively inserting non-expiring persistent EPIT entries across the desired Dpkt push path. For instance, a home automation application can send a push fire alarm Dpkt that is forwarded to the firemen station without the need of an Ipkt.

2. *Publish/Subscribe traffic*: a publish/sbscribe mechanism similar to PIs [123] can be achieved by creating a persistent EPIT entry during the forwarding of Ipkts. The set of newly created persistent EPIT entries results in a Dpkt path that is reserved in the network. The EPIT timer controls the lifetime of this reserved path. Nevertheless, the lifetime of this reserved path can be extended by sending an Ipkt that resets the timers of the corresponding EPIT entries (cf., Fig. 4.3). The publish/subscribe content delivery pattern can be used by live video streaming applications (e.g., TV channels) to forward content to viewers in an efficient multicast manner.

3. *Pull traffic with receiver flow control*: EFIB entries configured to create a non-persistent EPIT entry after an Ipkt is forwarded result in the traditional NDN pull mechanism. The one-to-one relationship between Ipkts and Dpkts allows receiver flow control by accurately controlling the rate of Dpkt received. This content delivery pattern can be used by a broad range of traditional content delivery applications (e.g., content download and video-on-demand).

As shown in Fig. 4.4, once a packet is processed by the EProcessing pipeline, the set containing the parsed packet headers (the most important one being the content name), and the retrieved EPIT/EFIB entry (containing metadata such as the list of potential next-hops and the P4 function parameters) are used to form the EPacket. The target ID corresponding to the P4 function to run is retrieved using the P4 function name extracted from the EPIT/EFIB entry. Finally, the EPacket is passed to the Forwarding Logic module described next.

4.2.2 Forwarding Logic Module

As shown in Fig. 4.1, the Forwarding Logic module contains two components: Functions and Deparser. The Functions component represents a set of P4 code organized in independent P4 functions, each running in isolation. The Deparser is a component that can update the packet header fields (e.g., content name) based on values provided by the P4 function.

In ENDN, we design a novel P4 target aiming at providing support for several categories of services described in Chapter 1. The next sections explain the design choices outlining our P4 target.

4.2.2.1 Forwarding Logic Design

As shown in Fig. 4.4, ENDN defines a new P4 architecture model corresponding to the P4 function target. Each target contains a P4 program that can optionally call one

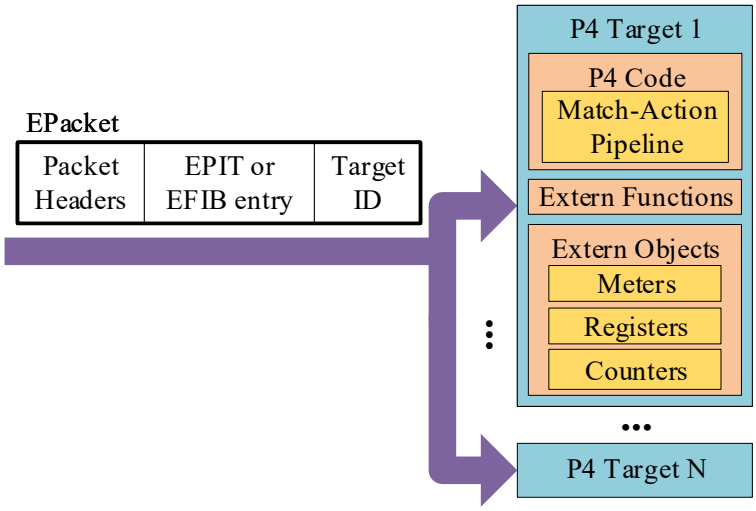


Figure 4.4: Architecture of the Functions component.

or more extern functions and objects. Each target can be dedicated to serving a single namespace or can be shared by multiple ones from different applications. In the first case, the P4 program can be highly customized and maintain application-related values (e.g., meter thresholds, constant content names for NAT like functionalities, or the face ID for static backup forwarding). Multiple applications can also share P4 programs by allowing application-specific values to be stored in the EProcessing module tables and passed to a generic P4 code as parameters. The first scenario allows for highly customizable services at the expense of switch scalability due to larger numbers of functions used. On the other hand, the second scenario achieves higher scalability at the expense of reduced per-application customizability. This choice of P4 program sharing is made by network operators and embedded in the control plane logic.

A P4 program can be updated efficiently using a three-step process. First, the controller compiles the P4 code and installs it in a new target. Then, it modifies the mapping of the updated function name to that of the new target ID. Finally, the target executing the previous version of the P4 program is deleted. As a result, the installation or update of P4 programs requires no downtime to the switch, thus, enabling run-time programmability.

To keep our P4 targets as lightweight as possible, we limit the P4 programs to a single match-action pipeline without a parser or deparser. In contrast to IP, the parser logic is performed by the EProcessing module and not by the P4 target. In an IP P4 switch, the parser determines which protocol is supported in the switch. For example, for an IP P4 switch to implement one of the following protocols: IPv6, IPv4, ICMP, ARP, TCP, or UDP headers, its parser must be programmed to extract the needed header fields. On the other hand, ENDN supports only two packet types, NDN's Ipkts and Dpkts.

Finally, ENDN has a single deparser component common to all the P4 functions. It can modify the header fields (e.g., the content name) with the new field values updated in the match-action pipeline. In case of modified packets and as Ipkts and Dpkts are signed, the modification of the header fields will result in the invalidation of the original signatures. Several approaches can be used to preserve the validity of the packet signatures. One approach is to use encapsulation where the original packet is embedded in the payload of a new packet signed by the switch. A second approach is for switches to re-sign the packets, and for applications using content rewrite services to trust the keys used by the network switches.

4.2.2.2 Match-Action Pipeline

Each P4 match-action pipeline contains a sequence of P4 actions connected using match tables. The match tables select the next P4 action to execute based on the value of some P4 fields (e.g., header fields and P4 variables). Match tables and P4 actions can use inputs that can be retrieved from the metadata available in the EPacket. There are three possible outcomes of the match-action pipeline. The first outcome is to select which ports the packet should be forwarded to among the list of possible next-hops retrieved from the EPIT/EFIB entry. This includes forwarding the packet to a single port, multicasting to multiple ports, or dropping the packet. It can also decide additional port forwarding parameters such as queue priorities and traffic shaping. The second outcome is to modify the header fields of the packet before it is forwarded. Finally, the third outcome is to

update the state of the P4 program represented by extern objects. For example, the average rate of forwarded packets can be stored in a register.

It is worth noting that P4 accepts only numerical value fields (i.e., binary fields of a certain bit-size). As a result, strings cannot be passed directly to the P4 function. While most of the match-action pipeline's inputs and outcomes can be expressed as a numerical value (e.g., the type of packet, Ipkt or Dpkt, can be stored in a boolean), the content name header field is impacted by this limitation.

To allow for the manipulation of data from the content name header field of packets, we added several extern functions to our P4 function target. Some of these functions allow the P4 code to access information from the content name. For instance, we added a function that takes a name component position in parameter and returns the hash of the component. The component hash can then be used to query match tables and thus perform a specific P4 action based on the value of a content name component. Other functions control the deparser: they allow the P4 code to insert or delete a name component at a specific position.

Finally, we add an extern function that allows the P4 code to create a new Dpkt with a specific name and payload (both string constants). This Dpkt is processed as a new incoming packet by forwarding it for reprocessing through the loopback face (cf. Fig. 4.1).

4.2.2.3 P4 Function Examples

The design of our P4 target allows us to implement a broad range of complex networking services.

For instance, a round-robin load-balancer P4 function would cycle through the next-hop list by selecting a specific next-hop and storing it in a stateful register. After each packet is forwarded, the selected next-hop value stored in the stateful register is updated.

Similarly, a congestion avoidance mechanism would measure the throughput of pack-

ets forwarded through a specific face using a meter. When the measured throughput reaches a predefined threshold, the packets are forwarded through a second face.

Another example is a RTT-face ranking mechanism. The timestamp on which an Ipkt is forwarded is stored in an array of stateful registers indexed by the chosen next-hop. When the corresponding Dpkt is received, the source face ID is mapped to the chosen Ipkt next-hop number using a table. The Ipkt next-hop is then used to retrieve the previously stored Ipkt forwarding timestamp from the stateful register array. The RTT is thus calculated and used to rank the next-hops.

Custom monitoring services can be achieved by implementing matching rules that decide what to monitor and when to send monitoring Dpkts. For example, when Ipkt requests exceed a certain threshold, a monitoring Dpkt is created using an extern function and forwarded to the application or the controller following non-expiring persistent EPIT entries.

Finally, the application can request a switch to flag Ipkts with a region ID. The region ID is represented by a name component that is inserted at a specific position in the content name through a call to a deparser extern function.

4.2.3 ENDN Content Store

The CS is a crucial component of the forwarding pipeline of NDN: every Ipkt and Dpkt is processed by the CS [8]. While some applications may benefit from in-network caching, other ones serve contents that must not be cached (e.g., user authentication data). However, applications do not have control over the CS caching decisions in NDN switches. In ENDN, caching control is explicit and linked to the published namespaces. The CS is not part of the standard Ipkt and Dpkt forwarding pipelines but appears instead as a data plane application accessible through an internal face. Dpkts are explicitly cached by adding a next-hop to the switch's CS in the EPIT entry. Similarly, the CS is not always queried during the forwarding of an Ipkt: the CS is explicitly checked by adding

a next-hop to it in the EFIB entry.

In ENDN, the CS strategy implements the cache replacement strategy that determines which previously cached Dpkt is replaced by a new one. Examples of cache replacement strategies include the least-recently or least-frequently used strategies.

4.2.4 Hardware Feasibility

The design of the ENDN data plane relies on a novel P4 target architecture. While traditional ASIC P4 hardware uses a fixed P4 target architecture, newer FPGA-based hardware allows for the definition of a custom P4 target architecture. One of the most prominent examples of such hardware is the NetFPGA SUME card, a PCI-Express network board with a fully programmable FPGA [134]. Ibanez et al. show how the NetFPGA SUME card can be configured to realize a custom P4 target with P4 extern functions implemented in the FPGA using a Hardware Description Language (HDL) [60]. Also, Krude et al. show how to implement several isolated P4 targets on an FPGA [133]. The EProcessing module, the P4 function targets, and the extern functions of an ENDN switch can be directly programmed on an FPGA using a HDL. The P4 functions are then compiled to run on the custom P4 function targets in a manner similar to [60].

4.3 Evaluation

We implemented ENDN's data plane by modifying the NDN Forwarding Daemon (NFD) [70] and using libraries from the Behavioral Model (BMv2) software switch to implement our P4 targets [61]. The version of the P4 language accepted by the ENDN's P4 function target is the standard $P4_{16}$ version. ENDN switches are used to create two test topologies within the ndnSIM simulator [107].

The two test scenarios show how different services are represented and enforced by our architecture.

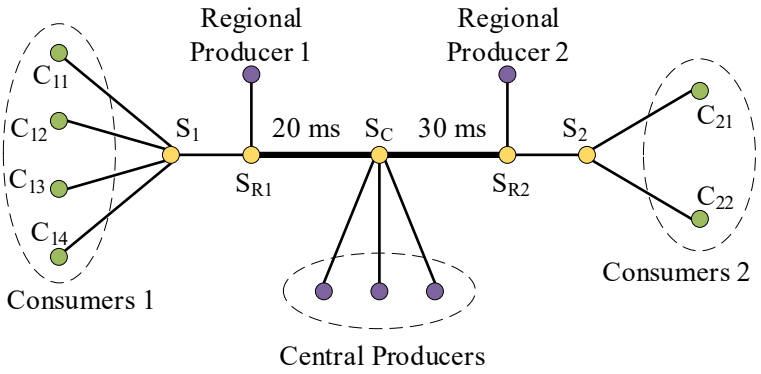


Figure 4.5: Simulation topology of the first scenario.

4.3.1 First Test Scenario

The first test scenario compares ENDN to NDN using the topology shown in Fig. 4.5. This topology contains two network regions, each containing a group of consumers (Consumers 1 and Consumers 2). Regional Producer 1 (RP_1) is connected to the switch S_{R1} and serves four consumers connected to an edge switch S_1. Similarly, RP_2 is connected to the switch S_{R2} and serves two consumers connected to an edge switch S_2. The core of the network contains a global data center formed by a group of central producers. The link between S_{R1} and S_c has a delay of 20 ms while the link between S_{R2} and S_c has a delay of 30 ms. All the other links introduce a delay of 10ms. All the links have a rate of 10 Mbps and cannot cause any packet loss.

This scenario simulates a content provider, managing a distributed data-center, that requires geo-gating to restrict content access in each region. For instance, the first region's consumers are not allowed to access the content specific to the second region. Hence, the provider defines three namespaces: */content*, */content/region1*, and */content/region2*. The last two contain content names that are accessible from their respective regions. The consumers of both regions are only aware of and must use the common */content* namespace (geo-gating). Hence, the */content* namespace becomes an alias of */content/region1* in the first region, and of */content/region2* in the second. RP_1 and RP_2 serve contents specific to their region: they can only produce Dpkts in the namespace */content/region1*

or */content/region2*. The central producers contain all the content of the application: they can answer all the namespaces. Furthermore, each regional producer can serve up to 250 Ipkts/s. Additional requests are automatically offloaded to the central producers. Finally, switch S_c serves the three central producers and implements a load-balancer to divide the requests among them.

The following ENDN services are linked to the namespaces:

1. *Content delivery pattern service*: a pull-based content delivery service for all the namespaces. The controller interprets this request by configuring all the switches with the correct EFIB routes. These routes only allow Ipkts of the */content* namespace in S_1 and S_2. These EFIB routes create non-persistent EPIT entries during the forwarding of Ipkts.

2. *Content name rewrite service*: is implemented as P4 programs installed in S_1, S_2, S_{R1}, and S_{R2}. For example, the P4 programs add the name component *region1* to */content* Ipkts going through S_1 and remove the name component *region1* from */content/region1* Dpkts going through S_{R1} (similarly for *region2*).

3. *Load balancer service*: balance the Ipkt requests in S_c between the central producers using a round-robin algorithm. It results in the generation of a P4 function installed in S_c.

4. *Flow limit service*: limit the flow of Ipkts forwarded to the regional producer in S_{R1} for */content/region1* to 250 Ipkts/s (similarly for *region2*). The excess requests are forwarded to S_c. It results in the generation of P4 functions with meters installed in S_{R1}, and S_{R2}.

To compare the performance of ENDN against traditional NDN [70], we first note that NDN does not support namespace aliasing and cannot provide applications with the means to control the forwarding strategies (FSs) deployed in the network. Hence, additional logic must be built in the application to realize the same scenario. Nevertheless, we

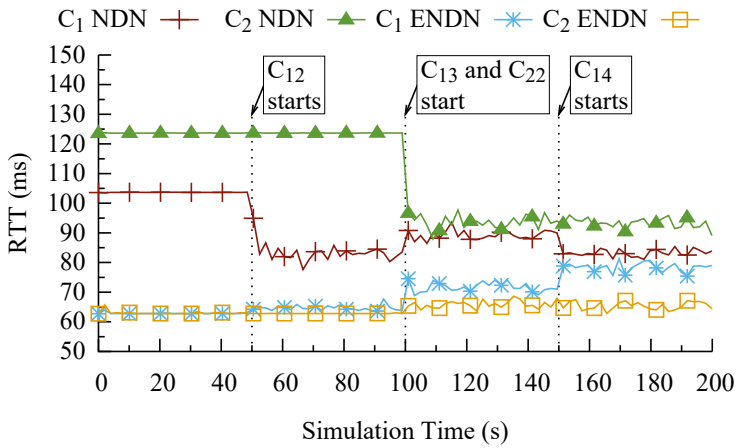


Figure 4.6: RTT experienced by the different consumer groups for NDN and ENDN.

assume that the application provider has access to the switch S_c and can install a round-robin load-balancer FS for the */content* namespace. The NDN data plane configuration mainly consists of FIB routes: the requests to */content/region1* and */content/region2* are restricted to the corresponding region and forwarded to the regional producer. On the other hand, the requests to */content* are forwarded to the central producers to perform further analysis to determine the region of the consumer using application level solutions.

Fig. 4.6 shows the average RTT experienced by Consumers 1 (C_1) and Consumers 2 (C_2) for both NDN and ENDN as individual consumers start requesting content at different time points. In the case of NDN, consumers alternate between requesting the */content* namespace and their regional namespace (e.g., C_{11} requests */content* while C_{12} requests */content/region1*). For example, C_2 experiences the highest delay during the first 100 seconds because all the Ipkts request the */content* namespace and are thus forwarded to the central producers. When C_{22} starts, it requests the */content/region2* namespace thus reducing the average RTT experienced by C_2. It explains the alternating increases and decreases of RTT as consumers start their requests. On the other hand, ENDN forwards all the requests to the regional producers until the threshold is exceeded, which explains the staircase-shaped RTT curves as more and more requests are forwarded

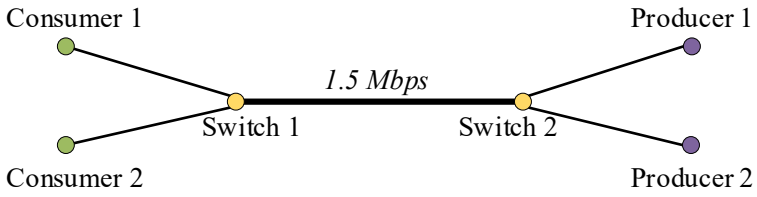


Figure 4.7: Simulation topology of the second scenario.

to the central producers when more consumers are active. The adaptability of ENDN to the application requirements thus results in a lower latency compared to NDN.

4.3.2 Second Test Scenario

The second test scenario aims to show how ENDN can adapt to the requirements of several applications using the topology shown in Fig. 4.7. This topology contains two consumers (C_1 and C_2) served by switch S_1 and two producers (P_1 and P_2) served by switch S_2. S_1 and S_2 are separated by a single link that may get congested. All the links have a rate of 1.5 Mbps, introduce a delay of 10 ms, with no packet loss. The topology simulates a part of an ISP network hosting two applications: P_1 and P_2 host an IoT and a video on demand (VOD) applications respectively to serve their respective consumers C_1 and C_2. The IoT application uses two namespaces: */sensor* is a stream of sensor data periodically published (e.g., a temperature sensor values generated every minute) to which consumers can subscribe, while */sensor/alert* corresponds to high priority push notifications (e.g., fire alarm). */sensor* is thus a publish/subscribe namespace that is configured with EFIB entries generating persistent EPIT entries. On the other hand, the */sensor/alert* namespace is configured with persistent non-expiring EPIT entries forwarding Dpkts towards C_1. Moreover, the switches contain a P4 function that sets a high queue priority for the */sensor/alert* Dpkts. The VOD application uses one main namespace, */VOD*, that is used to interact with the application (e.g., search for a specific video or access reviews and video related metadata). The */VOD/video* sub-namespace corresponds to the video streaming traffic. Contents in this namespace are retrieved by

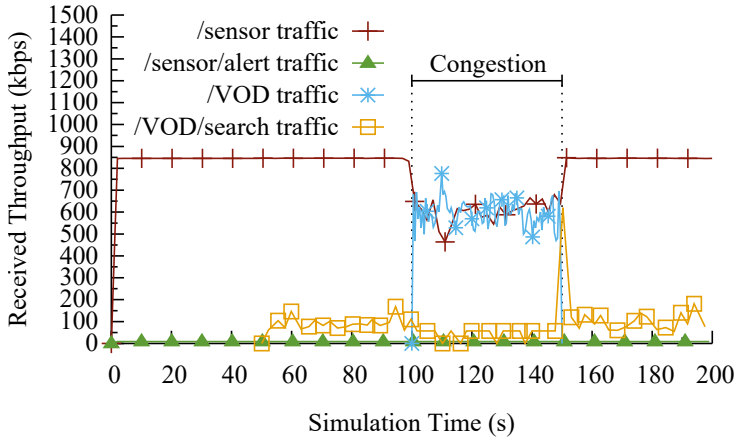


Figure 4.8: Dpkt received throughput for the different flows without congestion detection.

the traditional NDN pull mechanism that is configured with EFIB entries generating non-persistent EPIT entries.

At the beginning, C_1 subscribes to the */sensor* traffic. The */sensor/alert* traffic consists of a single push notification sent every second to C_1. After 50 seconds, C_2 starts browsing the VOD application, thus generating the */VOD* traffic. Finally, between 100 seconds and 150 seconds, C_2 watches a video, which adds the */VOD/video* traffic in the network and results in congestion. Without congestion detection, the congestion results in packet losses that are shown in Fig. 4.8 as a reduction in the received throughput.

To mitigate the effects of the congestion, the IoT application requests that the network sends it a Dpkt having the content name */sensor/rate/reduce* when the congestion starts, and another Dpkt having the content name */sensor/rate/restore* when the congestion is finished. When the IoT application receives the */sensor/rate/reduce*, it reduces the rate at which it generates Dpkts by half, and inversely when it receives the */sensor/rate/restore* Dpkt. The IoT application then declares a new */sensor/rate* push namespace and link it with a customized monitoring service that results in the generation of a P4 function installed in S_2. This P4 function measures all the Dpkt traffic forwarded to S_1 using a meter to detect the congestion, and then calls an extern function to send the */sensor/rate* Dpkts. The effect of this congestion detection and mitigation

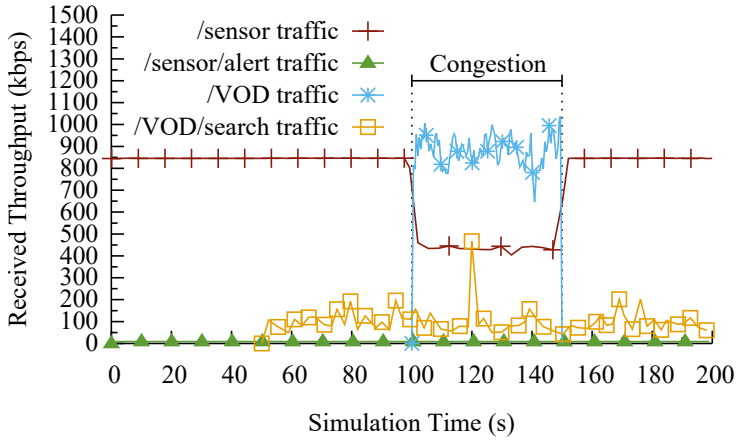


Figure 4.9: Dpkt received throughput for the different flows with congestion detection.

mechanism is clearly visible in Fig. 4.9: only the */sensor* traffic received throughput is reduced during the congestion. The congestion did not cause any packet losses.

4.4 Conclusion

In this chapter, we designed ENDN's data plane. We extended existing P4 target architectures to allow the execution of multiple P4 programs in isolation. ENDN's data plane permits the installation and update of individual P4 programs without incurring any downtime on the switch, thus, enabling P4 code run-time pluggability. Moreover, we overcame the limitations of P4 in string-based content name processing by adding several extern functions. Finally, we illustrated with a network simulation how several services are translated to a data-plane configuration. By adapting to the application needs using a broad set of customizable services, ENDN achieves network efficiency with low latency. In the next chapter, we explore in more details how services are represented in our proposed architecture, introduce the ENDN control plane, and explain how the data plane can be generalized to support other packet types.

Chapter 5

Control Plane and Protocol Independence

The vision of future applications includes some extensive use of holograms and virtual and augmented reality in various social and industrial tasks. This vision also includes full reliance on smart programs running in clouds or at the network's edge to automate various end-user devices or industrial machinery. The realization of this vision must be accompanied by a major shift in the role of the network infrastructure; while current networks are regarded as source-to-destination traffic delivery channels, next-generation networks will play the role of a dispersed fabric that seamlessly augments the capabilities of running devices, machines, and things [135, 136].

To realize this role, network services must be advanced in several directions; first, novel network architectures must be capable of offering a plethora of finely customized and tuned flavors of traditional services (e.g., load balancers, firewalls, and various scheduling algorithms). Second, as newer protocols and applications emerge, corresponding supporting services must be easily implemented. For example, streamed holograms may require the support of multiple flow synchronization, on-time flow delivery, and dynamic ranking for packet priorities [135]. Third, while some services (e.g., firewalls) may

be easily configured to be shared among several applications, network providers must allow specific applications access to individually customized processing. One example is to allow applications to configure selective payload drop or compression to further reduce the experienced latency. Finally, some latency-critical application processing may also be offloaded to the network (e.g., filtering video frames in the network to guide a car in real-time [136]).

Recent advances in software-defined networking (SDN) [11], with full separation of the control and data planes, are paving the way to realize these services. As a result, applications can communicate their requirements to the control plane, which, in turn, translates and enforces them at the programmable data planes (PDPs). The latter further allows the forwarding devices to execute custom forwarding logic eliminating the need for continuous communication with the controller.

In the previous chapter, we introduced the ENDN PDP that allows NDN namespaces to be associated with a stateful isolated P4 function implementing complex custom network behaviors. This design allows us to support several network services at the data-plane level. In this chapter, we present the operation of the ENDN control plane that provides hosted applications with a catalog of continuously evolving network services that can be chained to process application packet flows. In the NDN context, application packet flows can be identified with a content name prefix (i.e., a namespace): they correspond to a sequence of Ipkts and Dpkts that are sent by the application following a content delivery pattern (e.g., pull or publish/subscribe). However, in general, an application packet flow can be identified by a set of packet header field values (e.g., a tuple of IP addresses and TCP ports). We can thus generalize the packet format accepted by our data plane to non-NDN packets. Hence, while our architecture is based on the NDN vision, we believe this vision is not necessarily tied to the current specific NDN packet format or the NDN protocol. Supporting additional protocols not only allows for interoperability with current IP networks, it also makes our data-plane future-proof.

As seen in the previous chapter, our ENDN data plane is based on P4, a prominent

protocol-independent programming language [17] that can be used to instruct switches to execute programs at line rate. Each P4 program defines packet header parsing rules and a pipeline of match action tables (MATs) to control the packets' forwarding behavior. However, due to the limitations of P4 parsers in string processing, our proposed ENDN PDP did not use P4 parsers: the P4 functions only contain a sequence of MATs. Additionally, ENDN P4 functions cannot access the packet payload to perform application-data-specific computations (e.g., computer vision algorithms [136]). In this chapter, we thus also extend the ENDN data plane to make it protocol-independent through programmable parsing.

Hence, we introduce a novel PDP design, referred to as Extended-P4 (EP4). First, a fast parse and process path for fast-tracked packets with low latency constraints is performed by the PDP's enhanced parser (EParser). After parsing, the EParser either assigns a forwarding decision (i.e., output port and scheduling policy), for fast-tracked and best-effort packets, or an enhanced processing decision, using a unique processing behavior ID (PBID) that is passed to the next module, the extended processor (EProcessor). The latter can perform just-in-time parsing of the packet to only extract the packet payload data required for the specific behavior corresponding to the PBID. The EProcessor is also designed to execute runtime-pluggable P4 behaviors on packet flows. The controller can achieve the desired balance between scalability and switch resources utilization on one side and per-application customization and PDP management complexity on the other by varying the level of P4-program sharing among several flows. Hence, EP4 generalizes both the EProcessing and Forwarding Logic modules introduced in the last chapter into the novel protocol-independent EParser and EProcessor modules.

Our main contributions can be summarized as follows:

1. We provide a definition of ENDN network services in a protocol-independent way, and thus introduce an extensible catalog of network services that can be used by hosted applications.

2. We introduce the ENDN control plane operations needed to support our extensible catalog of network services and translate the application service requirements into data-plane configurations.

3. We design EP4, a novel protocol-independent PDP that extends the ENDN PDP to allow low-latency packet fast-tracking and just-in-time parsing of packet payloads.

The rest of this chapter is organized as follows; Section 5.1 surveys related work. Section 5.2 presents an overview of our application-centric ENDN architecture. Section 5.3 defines the network services provided by our architecture, and introduces the ENDN control plane. We design EP4, our protocol-independent PDP in Section 5.4. Section 5.5 illustrates the functionalities of our proposed architecture with a prototype implementation. Finally, Section 5.6 concludes this chapter.

5.1 Related Work

5.1.1 Network Services with PDPs

Related to our work are several approaches implementing particular services at the PDPs. These approaches can be classified into two categories: traditional network services and in-network processing. The former act on the packet-forwarding behavior of the network. We can cite in this category: security [137, 138], monitoring [122, 139], and forwarding patterns [140, 141]. On the other hand, in-network processing allows the applications to offload some of their computations such as database queries [142], media processing [121, 143], or computer vision algorithms [136] to the PDP. Finally, Nour et al. introduce the "compute-less" paradigm that aims at reducing the usage of network computational resources by optimizing the offloading of tasks to the network (e.g., identify and reuse the results of similar tasks from different applications) [144]. Our architecture complements these approaches and aims to provide a framework for their integration as

independent network services chosen by applications. We thus make applications the main stakeholders that can define their desired underlying network behavior.

5.1.2 Protocol-Independent PDPs

PDPs rely on languages such as P4 [17] to specify the packet processing logic of switches. P4 programs are target-independent. In other words, they can run on highly specialized hardware-based switches [60] as well as software-based ones [61]. However, P4 is architecture-dependent; an architecture model provides an abstraction of the switch and defines the sequence of P4 components (i.e., parsers, MATs, and deparsers) that may constitute a valid P4 program along with definitions of additional permissible functions, defined as externs and metadata structures. The portable switch architecture (PSA) [12] is the most commonly adopted architecture and provides a single parser and ingress and egress pipelines.

P4-enabled switches still fall short of realizing the aforementioned vision of services. The majority of P4-based PDPs employ single monolithic programs to control the full behavior of the switch. These programs usually have large memory footprints and necessitate some downtime for any required modifications by the controller [145]. Another limitation is the rigid sharing of a single parser and a pipeline of MATs for all the packets. This design falls short of taking advantage of current switches' multicore processors and results in increased latency for some packets that might not need to be fully parsed or processed [146]. Furthermore, existing P4 parsers are stateless and only depend on header field values. These parsers cannot be customized to perform application-specific dynamic parsing based on stateful computations [147].

To extend PSA with the ability to support multiple P4 programs, rather than a single monolithic one, P4Visor [148] facilitates the merging of different P4 programs at compile time. However, modifications to these programs necessitate the recompilation of the end-program. On the other hand, HyperVDP [149] overcomes this problem by virtualizing

the PDP into multiple switch instances, each with a separate pipeline, while DejaVu [150] uses the recirculation of packets back to the pipeline to emulate multiple ones. μP4 [151] develops a logical architecture with sequential or parallel sets of parsers and MATs at the expense of lacking detailed specifications of architecture-specific features such as registers and externs. Complete isolation of flows is achieved in MTPSA [145] and P4vbox [152] which extend PSA with individual parsers and pipelines but lack the flexibility of adjusting to the processing needs of low latency packets and incur a high overhead due to unnecessary replication of pipelines.

Similar to these approaches, our work extends PSA in order to implement generalized and customizable network services. However, the proposed PDP design aims at realizing several advantages:

1. Balance between the switch parsing and processing speed and flow processing isolation.
2. Late parsing for some headers that can be dependent on already collected data and metadata in the packet header vector (PHV) or on switch state.
3. Customizable runtime reconfiguration of the processing logic.
4. A clear focus on the desired set of extensible services to be offered by the controller based on the capabilities of the PDP.

5.2 Proposed Architecture

Our proposed ENDN architecture design stems from the need to expose an extensible and evolving catalog of network services to the applications. These services include traditional (e.g., source-destination delivery, firewalls, and load balancers) as well as emerging (e.g., in-network (de)compression, encryption/decryption, caching, and re-transmission) services.

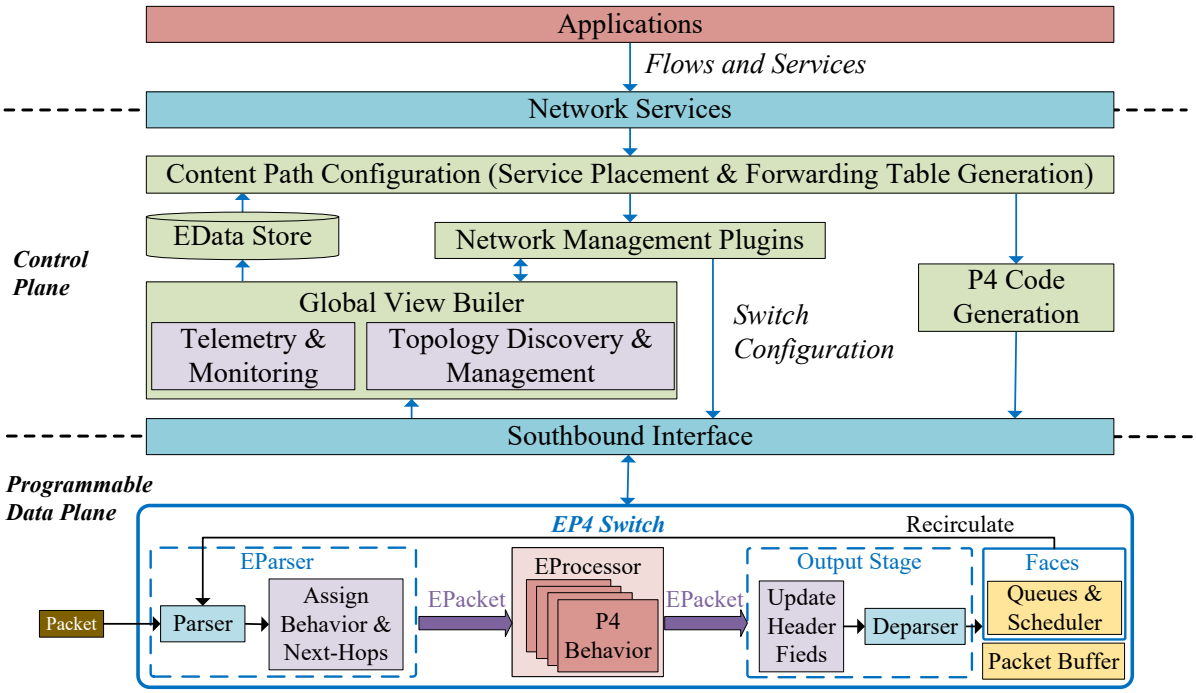


Figure 5.1: Overview of the ENDN architecture.

As shown in Fig. 5.1, the applications specify their requirements to the control plane by associating their flows, represented by possible protocol header values, with a list of network services. The control plane then enforces these requirements by translating these services into P4 programs optimally placed in the data plane. In turn, our proposed EP4 data plane contains three main components: an extended parser (EParser), an enhanced processor (EProcessor), and an output stage. The former performs fast common packet parsing and processing. This stage's outcome is either a forwarding decision to specific ports and/or a processing decision identified by a PBID. These are stored in an enhanced packet (EPacket) alongside the packet header vector (PHV) consisting of the parsed header fields as well as any additional metadata (e.g., incoming port number and queue statistics). In turn, the EProcessor executes the selected P4 program corresponding to the PBID.

Finally, the output stage module prepares the packets to be forwarded by the faces. It contains two main components: *update header fields* and a *deparser*. The former performs

any operation needed to form valid packets (e.g., update length fields and recalculate checksums). The deparser then converts the EPacket back to wire format. The EParser and output stage constitute the switch's relatively static components configured using table entries. On the other hand, the EProcessor contains the custom dynamic programs of our data plane pipeline that are updated frequently. The following sections provide more details about these components.

5.3 Control Plane

5.3.1 Network Services

A network service represents an abstract forwarding or processing logic that acts on a flow of packets given certain inputs (e.g., packet headers and flow states). A simple example is a session-aware load-balancer, which divides a flow of packets between multiple servers while ensuring that the same session's packets are sent to the same server. As can be seen from this example, a service can output the packets of a flow towards a specific destination and store some local state (e.g., the server towards which packets of a specific session have been sent). Another example of a service is to selectively drop specific marked segments of the packet payload when the network is congested. In this example, the application can mark non-critical packets or parts of their payloads with tags that can be identified and dropped at the PDP level.

For completeness, we identify the following set of primitive actions that can be employed by the controller to build the offered catalog of services to hosted applications:

1. *Simple forwarding:* output a packet towards one (unicast) or many (multicast) ports, drop, cache in buffer or forward the packet to the controller.

2. *Packet scheduling:* specify the packets' queueing and scheduling policies (e.g., priority and maximum burst size).

3. *Packet headers update:* change the value of some header fields, add/delete fields (e.g., flag packets with a VLAN tag).

4. *State update:* store and update values representing a desired state in persistent memory. For example, this state can reflect statistics about a flow, a set of flows, a port, or a list of source IP addresses (e.g., state of a tracked TCP connection in a stateful firewall).

5. *Packet generation/replication:* create a new packet with custom headers and payload (e.g., fragment a packet or send a notification packet to an application service or the controller).

6. *Vendor-defined:* a set of additional custom logic functions, such as a checksum computation or encryption/decryption, that can be provided by the switch vendor or operator using P4 extern functions.

The execution of one or more actions depends on a given set of input primitives:

1. *Packet content:* representing the set of data contained in the packet (e.g., header values). Source routing is an example of how header fields can control the forwarding behavior of a packet.

2. *Packet metadata:* defines the parameters that can be collected during the packet's journey in the switch (e.g., incoming port and queue state at the time of the packet arrival).

3. *Saved state:* which represents the results of previous computations on a packet flow (e.g., metered throughput).

4. *Controller configurations:* representing values configured by the controller that define the behavior of the service (e.g., bandwidth threshold for a rate limiter).

ENDN defines a service as an algorithm associated with a flow that applies one or more actions based on a set of inputs. While relatively simple, the set of actions and inputs allow for the specification of rich stateful network services. These services can be chained together to form the desired behavior for a specific application flow. The chain of services associated with a flow is converted to a data plane configuration by generating P4 code and placing it in the correct switches. In other words, some services are singleton services that correspond to P4 code executed at a single switch (e.g., firewall or load-balancer), while other services correspond to end-to-end algorithms executed in every node along the path followed by the flow (e.g., congestion detection). It is worth noting here that services are defined in a protocol-independent way. However, in a single network domain, we can assume that all the switches process a certain layer-2 or layer-3 packet format (e.g., IP, Ethernet, or NDN packets). As a result, we can assume that the P4 code of the services assumes the existence of a common parser that processes the common layer-2 or layer-3 packet: the P4 code of the services does not contain this common parser. Finally, services also have an attribute that indicate if an instance of the service can be shared by multiple flows. For example, stateless services can always be shared, while services that manage a state can only be shared if they contain a logic that can handle multiple flows. To summarize, an ENDN service contains the following attributes:

1. *Identifier:* identifies the service in the catalog.
2. *Parameters:* used to parametrize the behavior of the service (e.g., specify a meter threshold for the detection of heavy hitters).
3. *P4 code:* P4 code templates containing the service implementation following the format introduced by Riftadi et al. [153].
4. *Type:* whether the service is end-to-end or singleton.
5. *Shareability:* whether the service instances can be shared by multiple flows.

The application operators associate, in the control plane, their application flows with a list of parametrized services selected from the extensible service catalog. The control plane will then proceed to generate the corresponding data-plane configuration as follows.

5.3.2 Generation of the Data-Plane Configuration

As shown in Fig. 5.1, the control plane must first build a global view of the network by discovering the network's topology and performing telemetry/monitoring measurements. The result of this process is stored in the EDataStore, and is available to the different control plane components. Based on the information in the EDataStore, and the different application flows that were declared to the control plane, the Content Path Configuration module will compute the different routes corresponding to the application flows. This routing process has been covered in detail in Chapter 2. Instances of each network service are then placed in switches along the computed flow paths based on switch resource constraints and service attributes (e.g., shareability and type). The details of this service placement logic consist in solving constrained optimization problems and are outside the scope of this thesis. Finally, for every flow and switch, the control plane will generate the P4 code corresponding to the chain of service instances placed on that switch by combining the P4 templates as shown by Riftadi et al. [153]. The control plane operation can also be extended with several network management plugins that can modify the generated data-plane configuration. For instance, a cache management plugin can add next-hops to the CS based on a cache optimization algorithm that uses the global network view.

The data-plane configuration generated by the control plane consists of table values corresponding to the flows' route configuration and the associated P4 code for every flow and switch. It is then installed in the data plane using the P4Runtime API [58]. The next sections show how this configuration is executed by the data plane.

5.4 Protocol-Independent PDP: EP4

5.4.1 Data Plane: EParser

As shown in Fig. 5.1, a received packet is first parsed by the EParser that converts it into an EPacket. The EParser also performs the first stage of the packet processing using its own match action tables (MATs). The EParser design achieves several objectives:

1. Save switch resources with minimal processing for best-effort packets.
2. Reduce the latency experienced by fast-tracked ultra-low latency packets.
3. Identify the required processing logic for other packets to realize new services.
4. Serve packets with different protocol stacks.

The first objective is generic and straightforward to implement. For instance, the EParser can parse IP packets to extract the IP and TCP/UDP header and then use the destination IP address to query a routing table to get the next-hops. Achieving the second goal is challenging; however, there are already several solutions in place for fast-tracked packet processing. One example is to allow the first ingress switch to fully parse and process the packet (using the EProcessor module, as will be explained later) and then add additional tags in the header with full forwarding behavior instructions that can be parsed and processed by the EParser in the following switches using minimum clock cycles [146]. To realize advanced PDP services, the EParser matches the parsed fields against a processing behavior table (PBT) to fetch a unique PBID. Alternatively, the PBID can be directly extracted from an additional header like in the segment routing approach [154]. Finally, the EParser can be programmed to serve multiple protocol stacks, such as NDN packets, using additional parsing and lookup tables such as EFIB and EPIT tables. Hence, the EParser is a generalization of the EProcessing module of the ENDN data plane presented in the previous chapter. It is worth noting that the exact implementation of the parser module is thus protocol-specific.

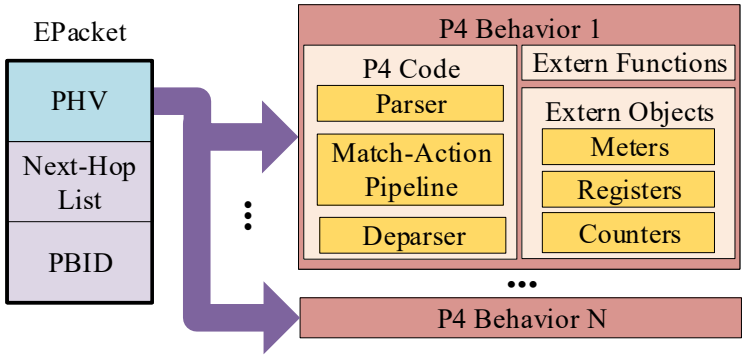


Figure 5.2: Architecture of the P4 behaviors.

We can hence define two logical components in the EParser: the *parser* and the *assign behavior and next-hops* components. The *assign behavior and next-hops* component is table-based and configured by the control plane to define the different application flows and associated routes and behaviors. The parser extracts the headers whose fields are used in the *assign behavior and next-hops* component. For fixed-length header-field protocols like IP, the EParser can be implemented in P4 using a standard P4 parser and match-action unit. For variable-length header-field protocols like NDN, the EParser can be implemented in hardware using FPGAs [152] or in software using optimized data structures like FCTree.

5.4.2 Data Plane: EProcessor

The EProcessor implements the custom logic corresponding to the list of network services of each flow. It consists of several isolated P4 execution environments (EE), referred to as P4 behaviors, each running a specific P4 code. As shown in Fig. 5.2, the received EPacket contains the PHV, optionally a list of next-hops, and/or the PBID. The EPacket's PBID is used to select the P4 behavior to execute. The P4 code is a full P4 program consisting of a parser, a match-action unit, and a deparser. In contrast to the EParser, the P4 behavior parser can use state and data computed by the EParser processing stage and communicated through metadata fields in the EPacket's PHV. This parser can thus perform complex stateful just-in-time parsing (e.g., parse flow control headers if the

throughput of the flow is above a threshold). This parser also allows extracting and modifying data in the payload, thus providing support for application-specific headers. The match-action unit can call one or more extern functions and objects. Extern objects manage the stateful objects in P4 that store data specific to flows (e.g., throughput meter internal state). Extern functions allow extending the P4 language with additional features. As P4 does not allow loops, we added extern functions that manage the next-hop list in the EPacket. We also added extern functions that create a new packet with custom headers and send it through the deparser.

Each P4 behavior can be used by multiple flows or assigned to a single one. A shared P4 behavior improves the scalability by reducing the number of P4 behaviors needed. However, in that case, additional parameters need to be passed to the generic P4 code to specify the application-specific logic. These parameters are stored in the EParser tables and then as metadata fields in the EPacket's PHV. On the other hand, a flow-specific P4 behavior is beneficial for flows having a frequently updated list of services (e.g., application offloading computation to the network), as in that case, other flows can be impacted if they share the same behavior. Sharing a P4 behavior among multiple flows is a trade-off between scalability and frequency of updates, which can be decided by a control plane algorithm. The EProcessor can also store packets in a packet buffer for later use (e.g., retransmision).

5.5 Evaluation

We implemented EP4 switches by modifying the Behavioral Model (BMv2) [61] P4 software switch and integrating it with the network simulator ns-3 [155]. Our EParser, EProcessor, and output stage are realized as separate P4 targets that accept standard $P4_{16}$ code with extern function extensions to simulate a test scenario described in the next section.

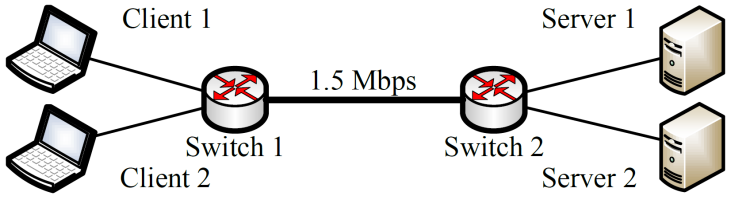


Figure 5.3: Simulation topology.

5.5.1 Test Scenario

The test scenario simulates a distributed video streaming application hosted in two servers: server 1 (S_1) and server 2 (S_2). This application communicates in the network using IP. At the beginning of the simulation, the first client (C_1) starts watching a video hosted in S_1. At t=10s, the second client (C_2) starts watching a video from S_2 and stops watching it at t=30s. Fig. 5.3 shows the simulation topology. C_1 and C_2 are served by switch 1 (Sw_1) and S_1 and S_2 are connected to switch 2 (Sw_2). All the links used have a 1.5Mbps capacity and introduce a propagation delay of 10ms. Finally, Sw_1 and Sw_2 are connected through a link having a 1.5Mbps capacity and a propagation delay of 20ms.

Both videos watched by C_1 and C_2 result in traffic of 1Mbps consisting of application packets of 512 bytes generated by S_1 and S_2. These packets are then buffered at the client side before playback. As both S_1 and S_2 traffic go through the link between Sw_1 and Sw_2, they can cause congestion. Thus, the application requests congestion avoidance and fair bandwidth sharing services from the network. We compare two approaches implementing these services: the standard TCP approach with our proposed network service approach using UDP. Both approaches are implemented using EP4 switches running the same configuration. TCP is executed at the endpoints and relies on a complex algorithm [156] using timers and acknowledgments to detect the maximum throughput that can be transmitted to the network. In this case, the EParser performs the standard IP parsing and routing and bypasses the EProcessor. In our network service approach, we define a custom bandwidth sharing service that executes on every switch along the paths

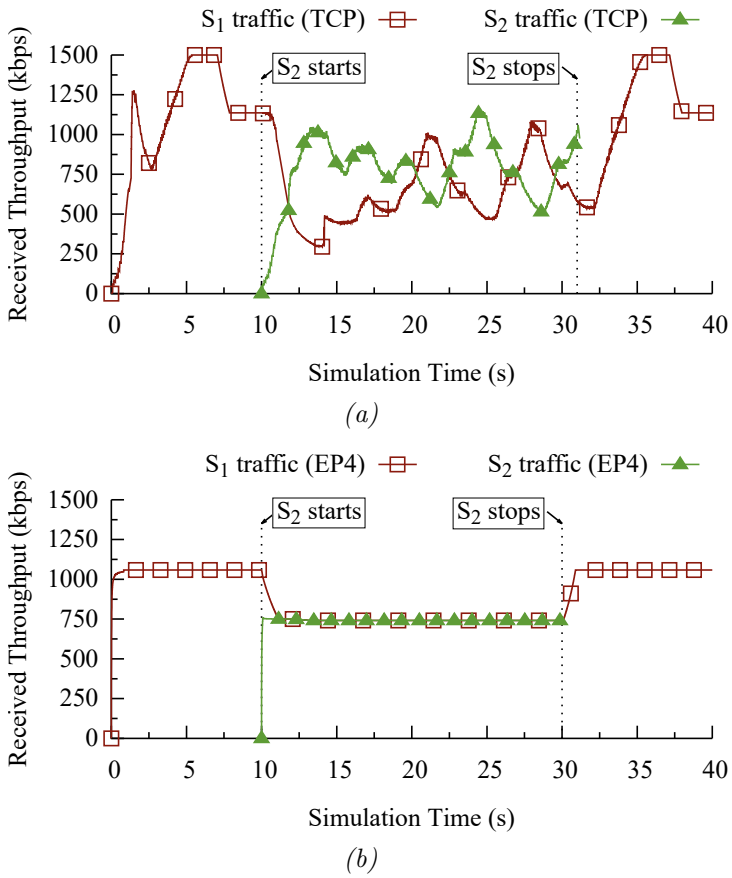


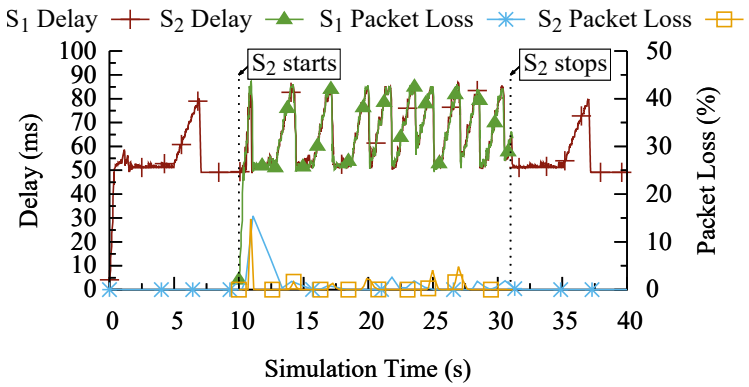
Figure 5.4: Received throughput for the different flows. (a) When TCP is used. (b) When an application bandwidth sharing network service is used.

followed by our application packets. In this case, the EParser also performs IP parsing and routing, but it also assigns to the UDP flow a PBID corresponding to our service. The latter performs just-in-time parsing of application headers to share the bandwidth as follows. It first detects the start and end of video playback by recognizing the server responses to the client's play and pause request packets. The number of active video flows is stored in each switch for every output port. Periodically, the client will send an *allowed throughput query* packet forwarded to the server through the reverse path used by the video packets. The service enforces this reverse-path forwarding by storing the source port from which video packets were received for every server. The service recognizes this *allowed throughout query* packet, parses it, and fills the allowed throughput field based on the number of active flows.

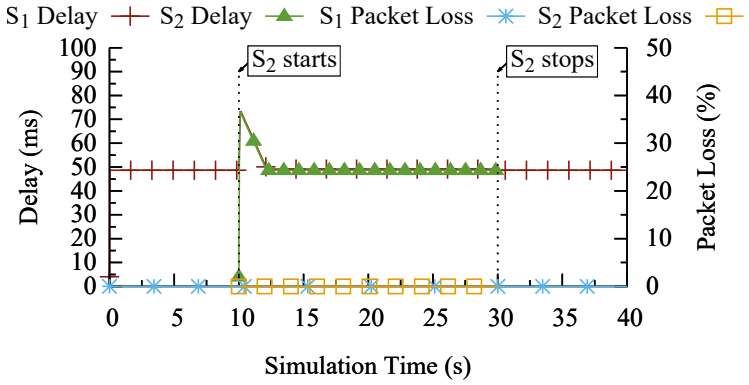
5.5.2 Results

Fig. 5.4a shows the received throughput achieved by TCP for each server. The slow-start and additive-increase/multiplicative decrease (AIMD) phases of TCP appear clearly for S_1 traffic between t=0s and t=10s. As the application generates data at a fixed 1Mbps rate, the TCP received throughput reaches the maximum link capacity of 1.5Mbps until it catches up to the application. The received throughput then stabilizes at around 1.1Mbps, corresponding to the application's 1Mbps data generation rate. When S_2 is transmitting data, TCP ensures on average a fair sharing of bandwidth between the two traffic through oscillations of the AIMD algorithm, as can be seen in Fig. 5.4a. When the S_2 traffic is finished, S_1 traffic received throughput stabilizes again around 1.1Mbps. As expected, TCP performed a fair sharing of bandwidth between S_1 and S_2. However, TCP also resulted in an increased and unstable delay as well as some packet losses as can be seen in Fig. 5.5a. TCP is indeed executed in the endpoints and estimates the available network capacity without any network support. Consequently, TCP may sometimes transmit at an increased rate that would start to congest the network, thus increasing the delay.

In comparison, Fig. 5.4b shows the received throughput achieved by our EP4 service. S_1 traffic throughput is here stable at 1.1Mbps until S_2 traffic starts. When both S_1 and S_2 traffic are transmitted in the network, our network service informs the servers about the allowed throughput. The servers reduce their transmission rate to 750kbps, thus preventing any congestion. The lack of congestion is visible in Fig. 5.5b: the delay is stable at 48ms (the minimal transmission and propagation delay in the network), and there is no packet loss. Our architecture, by allowing the network to collaborate with the application, efficiently shared the bandwidth between the different video traffic with a low, stable delay. As a result, this test scenario shows that application-aware network services can improve network performance compared to traditional endpoint-based solutions. Additionally, we also show that our architecture is now protocol-independent, and can also apply to applications that do not use NDN packets.



(a)



(b)

Figure 5.5: End-to-end delay and packet loss for the different flows. (a) When TCP is used. (b) When an application bandwidth sharing network service is used.

5.6 Conclusion

In this chapter, we extended the ENDN data plane presented in the previous chapter to introduce an application-aware network architecture with a P4-programmable data plane. ENDN's control plane allows applications to link their content namespaces to a set of programmable services from an extensible catalog of network services. The controller then translates these services to data-plane configurations comprised of table values and P4 programs. To realize this goal, we design a protocol-independent data plane with optional minimal parsing and processing, and customized forwarding. The latter feature is realized with run-time pluggable P4 programs containing customizable parsers serving flows encapsulated in arbitrary protocols. Finally, we show with a net-

work simulation that application-aware network services result in an increased network performance compared to traditional solutions where the applications have to adapt to the network's simple best-effort service. In the next chapter, we introduce the ENDN intent-service layer that allows operators to express application network requirements using simple high-level intents. The intent-service layer then converts these intents into the network services presented in this chapter.

Chapter 6

Mapping Application Intents to Data-Plane Configurations

Named data networking (NDN) [8, 68] and intent-driven networking (IDN) [65, 157] are two orthogonal research paradigms that aim at revolutionizing the current use of networks from conventional communication services into integral components of next-generation applications. Examples of these applications include time-sensitive, content-centric, and dispersed applications that allow humans to interact seamlessly with virtual objects within the context of virtual and augmented reality. Industrial automation functionalities built on top of sensors and connected machines represent another example of these applications. On the one hand, NDN facilitates building advanced applications by shifting the application developers' focus away from address- and location-centric communication and towards a simplified content-centric one. On the other hand, IDN allows network operators and hosted application developers to describe what is required from the network at a high level of abstraction without being concerned about how these requirements should be implemented at the network data-plane [65].

In the previous chapters, we gradually enhanced NDN to allow network operators to take advantage of switch functionalities to offer novel per-application, per-content,

or per-consumer highly customized network services (e.g., time-sensitive delivery using prefetching and caching, semantics-based forwarding, and content encryption and decryption). Our vision was motivated by emerging technologies, such as software-defined networks [158], that succeeded in separating the network control functionality from that of the data-plane packet forwarding process. More recently, the emergence of switch programming languages, such as P4 [17], has enabled the notion of programmable data-planes (PDPs). Using these languages, the control plane can continuously configure and fine-tune the switch behavior with respect to packet parsing and processing [5].

Despite these advances, operators are still limited by the current network management tools to direct the installation of per-flow or per-path switch configurations. These tools may require error-prone manual configuration and policy validation [65]. In addition, control-plane functionalities (e.g., routing, traffic engineering, and congestion control) still require manual parameter setting and have a network-wide service focus that lacks the needed per-application customization. Finally, these tools provide no direct means for application developers or users to define their network service requirements in a simple declarative manner without the need for technical knowledge of the network infrastructure.

The emerging concept of IDN attempts to bridge the gap between network management complexity and the emerging network service demands on one side and advances in data-plane programmability on the other [157]. The main premise of IDN is to allow operators and application developers to describe *what* is expected from the network serving the applications but not *how* that behavior is implemented using intents [159]. IDN tools can then automatically *"convert, verify, deploy, configure and optimize"* [65] the network to satisfy these intents. The realization of IDN necessitates addressing three main challenges: first, the development of expressive intent and network state models. Second, the realization of new mechanisms to automate intent validation and mapping to data-plane configurations. Third, novel intelligent machine learning-based techniques must be developed to allow the network to continuously self-adapt and self-heal to maintain the

satisfaction of these intents [19].

This chapter addresses the first two challenges described above. We consider a single ENDN domain with EP4 programmable switches that each can process packets using a chain of stateful match-action tables (MATs). We propose a novel Intent-Service Layer (ISL) built on top of the ENDN control plane that models and captures high-level intents and transforms them into a list of ENDN services to be deployed in the data plane. In the ISL, intents are first captured as uttered or written sentences. These are tokenized and classified using preexisting intent templates. A slot filling procedure is then employed to extract a set of intent parameters from the uttered words. The output from this phase is then translated using Event-B modeling into abstract Event-B machines (EBM) [160] which provide abstract descriptions of the desired network behavior to satisfy the given intents. Each EBM describes a desired behavior as a set of events acting on an abstract state representing the network. Abstract EBMs are then refined using existing tools, such as Rodin [161], to gradually introduce network-specific configurations implementing the desired behavior until a concrete EBM is developed. The concrete EBMs closely resemble the structure of the programmable MATs in the data plane. Hence, we use concrete EBMs as our new ENDN service representation. Our ENDN service catalog is thus enriched and managed by the ISL, and is thus generalized to become an Intent library. These services are then transformed or compiled into equivalent data-plane behaviors satisfying the intent.

The adoption of EBM modeling serves two main purposes. First, the highly abstract model of the EBMs describing the intents represents ideal means to capture the intent goals. Meanwhile, refinement, a key feature of EBM modeling, allows for the gradual mapping of these hardware and software-independent abstract EBMs towards the concrete EBMs representing the corresponding data-plane configurations. Second, Event-B is also a formal method to design EBMs that are correct by construction. Our ISL benefits from this feature by formally representing an intent's requirements and constraints on the network states by defining strict rules referred to in the EBM as invariants. For a

machine to be correct, i.e., performing as intended, these invariants must always be preserved after every event and refinement operation. These verification steps are referred to as proof obligations and are carried out using automated tools such as Rodin [161]. To this end, the main contributions of this chapter can be summarized as follows:

1. We develop a general framework for the lifecycle management of intents within the context of ENDN and analyze the main challenges for its realization. We then propose ISL, a novel component of our ENDN architecture that focuses on modeling and mapping ENDN intents into ENDN services that are translated into data-plane configurations.

2. Within our ISL, we define a novel networking intent model that is inspired by existing virtual assistants.

3. We propose a novel intent to network service mapping process using Event-B modeling. The proposed work demonstrates how EBM modeling language and refinement tools can be used efficiently to automate the steps of intent processing, validation, and translation to *correct* network and domain-dependent configurations.

4. We describe how ENDN services represented by concrete EBMs are translated into data-plane configurations.

The remainder of this chapter is organized as follows; Section 6.1 surveys the related Literature. In Section 6.2, we provide an overview of our proposed ISL. Section 6.3 is then dedicated to describing the adopted models and their mapping steps. Simulation results are presented in Section 6.4. Finally, Section 6.5 concludes this chapter.

6.1 Related Work

In this section, we review the Literature with respect to intent modeling, translation, and validation.

Existing network data models such as the management information bases (MIBs) and YANG (yet another next-generation) were developed specifically for low-level device configuration [162]. They are accompanied by a suite of client-server protocols such as the simple network management protocol (SNMP) and the network configuration protocol (NETCONF) to interact with and configure devices. While they provide a good abstraction for device configurations, they are not suitable for representing the high-level abstraction of network intents.

While recent research efforts have proposed several novel intent models, they have been mostly focused on defining intents that directly capture desired network or service configurations rather than abstract or declarative user or operator goals. For example, one of the earliest approaches for intent modeling is the model built within the SDN-based Open Network Operating System (ONOS) [163]. The model defines a set of predefined connection-oriented intents (e.g., topology, end-points connection, or service chain intents) and then provides a one-to-one mapping of these intents to network policies. Similarly, the IETF NEMO project and its extension defined in [164] focus on intents relating to network operations, such as selecting or changing a routing path. Other approaches utilize intent models built as extensions of the Topology and Orchestration Specifications for Cloud Applications (TOSCA) model [165]. However, they are also limited to direct mapping of network-oriented low-level intents into policies. Chopin [166] is another framework for specifying intents for cloud resource usage between end-points. It uses a fixed intent template that defines the desired traffic source and destination as well as the required resources between these end-points. The authors in [153] develop a novel intent definition language for applications hosted on IP networks. In their model, intents must clearly identify the two communicating end-points and the desired data-plane service (e.g., drop heavy hitters), which is then configured statically in the data-plane. In a similar manner, an intent model was developed in [167] to describe flow-rule intents for vehicular networks. The authors in [159] provide a more expressive model of service-oriented intents that allows an application to identify a service (e.g., caching or resource

provisioning). However, the intents are also pre-associated with a set of policies that describe the required behavior of the service in more detail.

In summary, existing network intent models are limited to describing communication-oriented requirements rather than aiming at capturing the operator or application goals from the underlying network. The majority of these existing models assume that the served applications have a detailed knowledge of the network topology and the exact configurations of the resource demands for their traffic flows. In other words, they identify network configurations using low-level vocabulary (e.g., allocated bandwidth between two end-points). A detailed comparison of existing IDN models and their limitations is presented in [159].

In contrast to the aforementioned models, highly expressive and well-developed intent models were developed for software applications such as those used by personal assistants [168, 169]. Moreover, intents capture and interpretation using these models have been addressed extensively in the field of natural language processing [170].

The majority of existing solutions in the Literature for intent to network configuration focus on the direct mapping of intents into policies [19, 164]. However, one of the main limitations of this approach is that the rigid modeling of policies as events-condition-actions fails to capture intent goals except in the context of predefined services such as network slicing [171]. A different approach is used in [166] where intents are translated directly into optimization problems for resource assignment and allocation. Overall, the Literature is limited to approaches that map intents to policies or limited direct network configurations.

Table 6.1 presents a summary of the intent models and domains of applicability of the major existing solutions in the Literature. Most of these solutions are domain-specific, and, hence, provide an intent model that captures requirements specific to a certain use case. Additionally, these solutions all apply only to a topology-centric IP-based network. To the best of the author's knowledge, the proposed work is the first attempt to build an intent model and an intent-to-data-plane mapping mechanism with

Table 6.1: Summary of the analysis of intent-based solutions in the Literature

Intent-Based Solution (authors)	**Intent Model/Scope**	**Applicability Domain**
Han et al. [163]	Virtual network model with middleboxes	Define virtual networks
Tsuzaki et al. [164]	Event-Condition-Action based on network resources (e.g., switch bandwidth)	Reactive network path change based on bandwidth usage
Khan et al. [165]	5G service model and Quality of Service (QoS) requirements	Automate 5G network configuration and slicing
Heorhiadi et al. [166]	Network resources cost and constraints between endpoints	Network resource management and optimization
Riftadi et al. [153]	P4 services between endpoints for a specific traffic flow	Create a combined P4 code containing multiple services
Singh et al. [167]	QoS requirements	Data dissemination in vehicular edge networks
Alamei et al. [159]	Service requirements	Cloud CDN
Proposed work	Name-based service requirements and constraints represented using EBMs	NDNs

a particular focus on NDNs. As NDN names are generic and can identify both contents and network resources, an NDN-based intent model offers a higher-level of abstraction compared to IP-based models. Thus, application developers can define high-level custom network services applied to their contents and flows without any prior knowledge of the underlying network topology or endpoints.

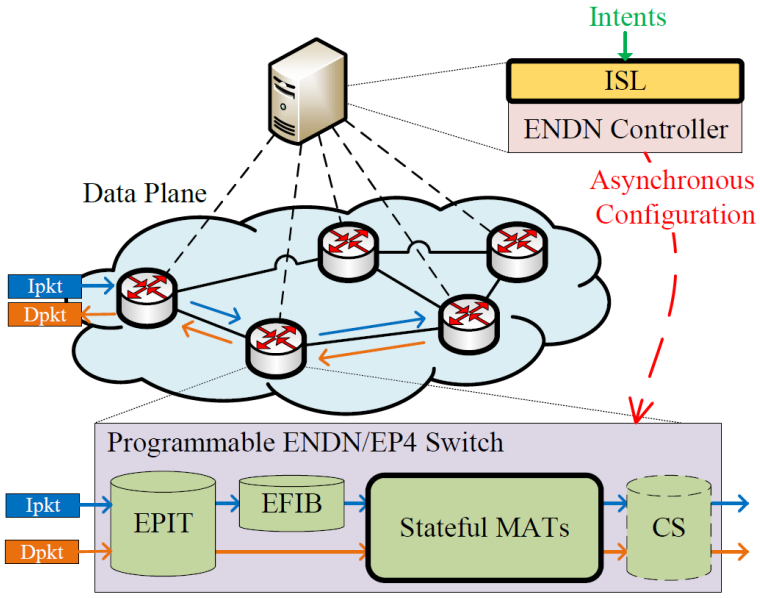


Figure 6.1: Proposed network model.

6.2 Proposed Architecture

6.2.1 Network Model

As shown in Fig. 6.1, the goal of our complete ENDN architecture is to receive intents from network operators or application developers and then translate them into a programmable data-plane configuration. The target network consists of a single domain managed by a single controller containing our proposed ISL. At the data-plane level, the ENDN/EP4 switches contain stateful programmable Match-Action Tables (MATs), implemented by P4 functions/behaviors, that can process packets according to custom rules. These MATs can be semantically represented as a set of rules of the form *if (conditions) then actions*, where the conditions and actions apply to packet fields (e.g., content name), switch metadata (e.g., queue length or output port), or custom saved states. Furthermore, the access to the CS is controlled by the MATs as shown in Fig. 6.1. The stateful programmable data plane allows highly dynamic per-packet forwarding decisions to be executed directly in the data plane with little involvement from the controller. As a result, communication between switches and the controller for data-plane configuration

is carried-out only when a new intent is requested: every intent is translated to stateful MAT entries in the data plane.

6.2.2 Overview of the proposed ISL

Fig. 6.2 provides a schematic description of the main components of our proposed ISL. As per the IETF IDN model presented in Chapter 2, the processes of ISL operate in two phases: production and pre-production. The production phase corresponds to the intent to data-plane configuration mapping process that is executed each time a new intent is uttered. On the other hand, the pre-production phase consists of defining all the different mapping rules used during the production phase. For instance, the different types of intents that the users can request are defined by the operator in a library of intents templates during the pre-production phase. These intent templates are related to a service or a network strategy. Examples of service intents are: to *forward a given list of contents to certain subscribers*, to *cache contents belonging to a particular namespace for a specific duration* or to *distribute requests equally among several producers*. Examples of strategy intents are: to *maintain average utilization of a server to a certain level* or to *create three classes of services for contents*. Intents also have parameters called slots (e.g., a content namespace or a traffic threshold).

The production phase consists of an intent processing workflow containing three main steps: identification, translation, and configuration. These steps are closely related to the stages of a generic IDN intent lifecycle described in Chapter 2. The validation process is done in parallel with the translation and configuration steps using the proof engine of the Event-B formal method.

During the identification step, intents are captured using a chat interface [172] or with the help of a smart assistant similar to Amazon's Alexa [168]. The intent detection and slot filling [173] operations are then performed. In this step, an intent is identified by contrasting it against the built-in intents from the intent library, and a list of label-

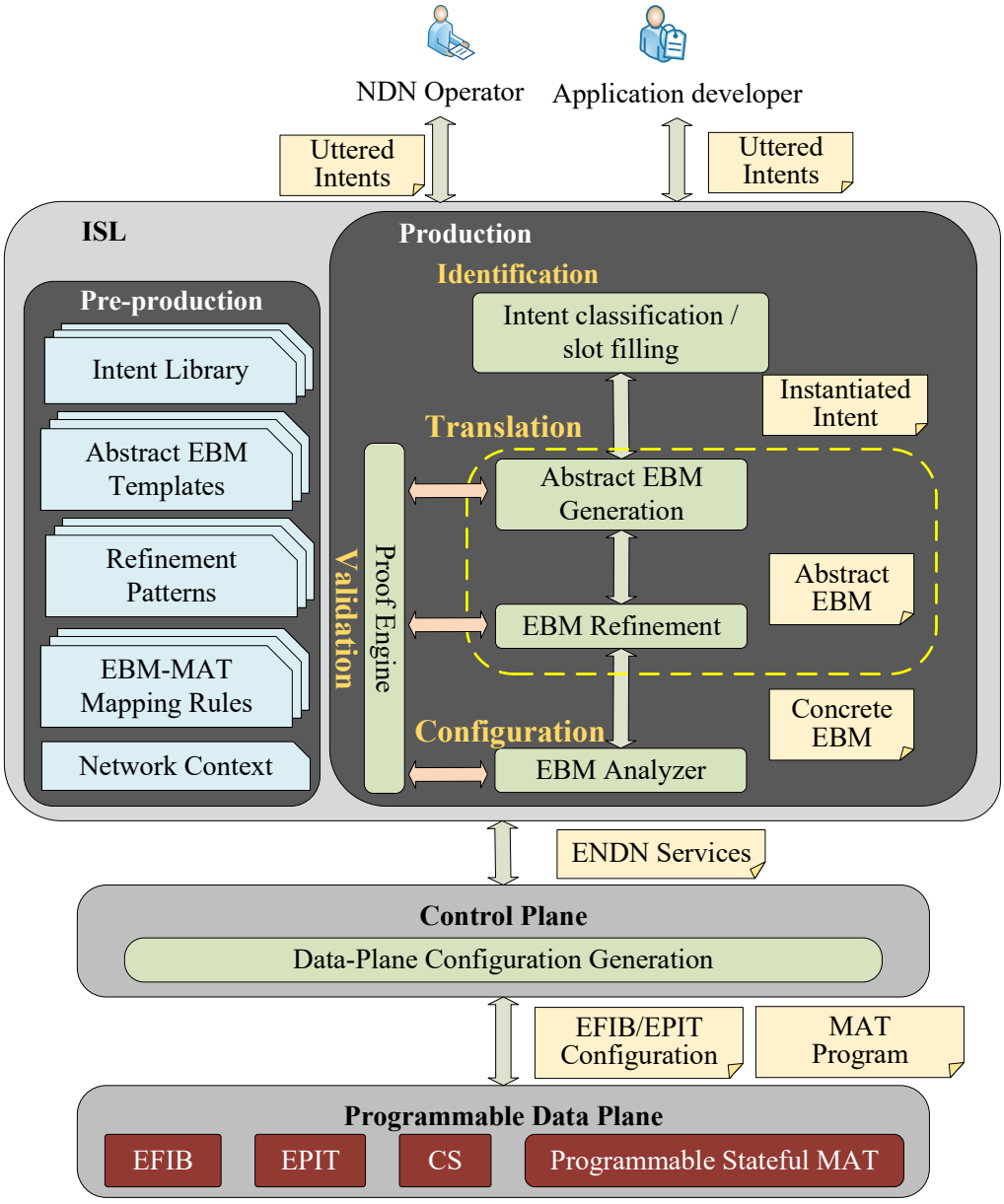


Figure 6.2: Proposed intent mapping architecture.

value pairs representing intent slot parameters is generated (e.g., time intervals, content names, or producers IDs).

Once the slot labels and values are obtained, they are fed into the first module of intent translation: the abstract Event-B machine (EBM) generation. Every intent

template is associated with an abstract EBM during the pre-production phase. This EBM contains an abstract implementation of the desired network behavior to fulfill the intent. Event-B [160] is a formal method that allows developers to model a discrete system control problem using a set of state variables in an EBM. Constraints, called invariants, are then added to the possible values of the state variables to represent the expected system behavior when the problem is solved (e.g., a counter can never reach a certain threshold). Events acting on the state variables are then created and proven to be compliant with the constraints, thus resulting in an event-based algorithm that solves the control problem. Event-B can thus create programs that are proven to be correct by construction using its proof engine [161]. Our architecture uses Event-B to model the programmable network behavior in response to each desired intent. EBM events follow the *if (condition) then action* semantic. This representation facilitates the refinement of the abstract machines into corresponding Match-Action Table (MAT) rules in the data-plane. In this case, EBM state variables correspond to packet headers, traffic statistics, switch values (e.g., queue size), packet metadata (e.g., packet source and destination), and in-network custom saved states (e.g., the last measured RTT), and thus correspond to the different inputs and outputs of the network.

In the abstract EBM, intent slot values are mapped to EBM parameters, and the semantics of the intent result in several invariants that ensure that the EBM implements the required intent behavior. Once the abstract EBM is instantiated with the slot values, it is refined using several refinement patterns [174] defined in the pre-production phase until a final EBM, called concrete EBM, is reached. EBM refinement is an essential part of the Event-B method: it gradually adds more details to the EBM while ensuring the invariants are always met until the problem is completely solved. The main goal of the refinement step is to transition between two different EBM representations. The abstract EBM representation is high-level and allows the intent requirements to be defined conveniently using abstract variables. On the other hand, the concrete EBM representation is switch-dependent and thus close to the data-plane MAT structures. As a result, the re-

finement patterns map abstract EBM variables and events into concrete EBM constructs to adapt to the network capabilities. For instance, a load balancer intent can balance the load between two producers using a specific load distribution algorithm (e.g., round-robin, congestion-aware, or based on the source region of the packets). The abstract EBM would then contain the generic load balancing algorithm and an abstract variable specifying the load distribution algorithm to use. On the other hand, the concrete EBM would contain the full implementation of the load balancer with the load distribution algorithm in the case of a P4 implementation, or an action to forward the packets to a load balancer middlebox if it is available in the network. The proof engine is executed during every refinement to ensure that the refined EBMs do not violate the invariants of the abstract EBM. Hence, the concrete EBM is proved to be compliant with the intent requirements set at the abstract EBM level.

Once the concrete EBM corresponding to the intent has been generated, it is processed by the EBM analyzer module. The main outcome of this module is to add this new concrete EBM to a list of ENDN services attached to a namespace. However, as multiple intents can be configured in the network, we first need to check that these intents do not result in conflicting data-plane configurations. Therefore, the EBM analyzer first performs consistency checks among multiple intents. More precisely, through the composition of different EBMs representing different intents [175], we can ensure that the invariants of an EBM are not violated by the processing done in another EBM. Hence, we can verify that a new intent does not conflict with existing ones. Once the concrete EBM passes the consistency checks, it is further analyzed to extract the namespaces it applies to. It can then be added as a new ENDN service to the list of services that are attached to these namespaces. These services are then processed by the control plane to generate the corresponding data-plane configuration. Finally, it is worth noting that some EBM variables are mapped into the execution of generic control-plane functionalities (e.g., a routing scheme to find the shortest path, or an optimal network function placement algorithm).

Table 6.2: Summary of the different variables

Variable	**Definition**
n	Intent name
$l \in \mathcal{L}$	Slot labels
$\mathbf{s_i} = (l_{i1}, l_{i2}, \cdots)$	Intent slot labels sequence
$\mathbf{w} = (w_1, w_2, \cdots)$	Tokenized intent words
$\mathbf{v_i} = (w_1, w_2, \cdots)$	Slot values vector
$\mathcal{C}(\mathbf{S}, \mathbf{C}, \mathbf{A})$	Event-B context
$\mathbf{S}$	Event-B context sets
$\mathbf{C}$	Event-B context constants
$\mathbf{A}$	Event-B context axioms
$\mathcal{M}(\mathbf{V}, \mathbf{I}, \mathbf{E})$	EBM
$\mathbf{V}$	EBM variables
$\mathbf{I}$	EBM invariants
$\mathbf{E} = (e_1, e_2, \cdots)$	EBM events
$G_e(V)$	Guard condition of event e
$A_e(V)$	Action of event e

The following sections provide a brief description of our proposed models and intent lifecycle functionalities.

6.3 Proposed Intent Lifecycle

In this section, we describe in detail the different steps of the intent lifecycle of our ISL. Table 6.2 contains a summary of the different mathematical variables used in this section.

6.3.1 Intent Creation and Identification

In our model, at an abstract level, a single ENDN domain can be regarded as a black-box that provides end-points (e.g., users, devices, and applications) with customizable contents. Customization includes various delivery patterns (request/receive, pub-

lish/subscribe, notifications, etc.), content processing services (e.g., encryption, filtering, and synchronization of multiple streams) as well as quality guarantees (e.g., reliability, delivery speed, and latency). Furthermore, it provides additional delivery services (e.g., access control, caching, request filtering, load balancing, geo-gating, and delivery quality assurance). The network blackbox also provides monitoring (e.g., reporting the number of requests from a certain user) and event-reporting (e.g., reporting an alarm when the number of content requests in a geographical area exceeds a given threshold) services. From the perspective of the network operator, the network blackbox is composed of a number of abstract services (e.g., content request/response handlers, content filtering, firewalls, and access control) that act on resources (e.g., consumers and producers lists, content namespaces, abstract communication channels to consumers, producers, and contents or caches) that must be configured in order to satisfy the requirements of the offered services.

These requirements are defined as intents that are instances of intent templates. The latter are created by the network operator and are stored in an intents library during the pre-production phase. They are defined using semantic frames [170, 176]. Each frame, or intent template, contains a unique intent name n and a set of entities, referred to as slots that are placeholders for the values of attributes needed to describe the intent. The intent template also provides a set of different example utterances that the intent owner can use. These samples can be communicated to the application developer as hints.

Formally, an intent template is identified by its name n and defines different sequences $\mathbf{s_1}, \mathbf{s_2}, \cdots$ of *slot labels* from a set $\mathcal{L}$ such that $\mathbf{s_i} = (l_{i1}, l_{i2}, \cdots)$. Each slot label $l \in \mathcal{L}$ describes an object that the users may mention in the intent. Fig. 6.3 depicts three examples of different intent templates. The first is an intent to describe a load balancing mechanism that an application developer can request. The template indicates the set of slot labels with their types that can be used in that intent (e.g., *cn*, *c1*, and *p1*). The possible sequences of slots are defined by the uttered samples. For example, in the first uttered sample: *"distribute the received requests for cn using mechanism between p1*

```
{"intentsLibrary": [
 {"intent": { "name": "LoadBalance",
   "slots": [ {"name": "cn",            "type":  "CONTENT_NAMESPACE},
              {"name": "c1",            "type":  "CONSUMER"} ,
              {"name":  "p1",           "type":  "PRODUCER"}, {"name": "p2", …},
              {"name":  "mechanism",    "type":  "DISTRIBUTION_MECHANISM"},
              {"name":  "ss",           "type":  "PRODUCER_STATE"},
              {"name": "t",             "type":  "TIME_RANGE"}, …]
  "utteredSamples": [
      "Distribute the received requests for  {cn} using  {mechanism} between
            {p1} and {p2}",
      "{p1} and {p2} should serve {cn} when {p3} is {ss}",
      "{p1} and {p2} should serve  {cn}  between {t}" }"]
 },
 {"intent": { "name": "Cache",
    "slots": […,{"name": "cp", "type": "CONTENT_PROPERTY"}, …]}
    "utteredSamples": [
       "Cache  contents that are {cp} when produced by {p1}",
       "After delivering {cn} cache it for {t}"]
 },
 {"intent": {"name": "HeavyHitter",
    "slots": […,{"name": "cr", "type": "CONSUMER_REGION"}, …],
    "utteredSamples": [
         "Block  heavy hitters for {cn}", "Report heavy hitters in {cr}"]
 }
]}
```

Figure 6.3: Examples of built-in intent templates

and p2", indicates that the expected slot labels are $\mathbf{s} = \{cn, mechanism, p1, p2\}$. The second intent template in the figure describes an intent to cache contents in the network when they satisfy certain properties (e.g., cache contents generated by producer *p1* in the last hour). Finally, the third template describes an intent requesting to block or report heavy hitters (i.e., consumers who send many Ipkts to a given type of content) in a certain region.

At production time, users utter an intent to describe the desired outcome guided by the samples of uttered intents. The identification module tokenizes the intent into words $\mathbf{w} = (w_1, w_2, \cdots)$ that are processed in two steps: intent classification, i.e., mapping the uttered words to the correct intent n, and a second phase of slot filling that identifies a corresponding sequence $\mathbf{s_i} = (l_{i1}, l_{i2}, \cdots)$ and a corresponding subset of the tokenized

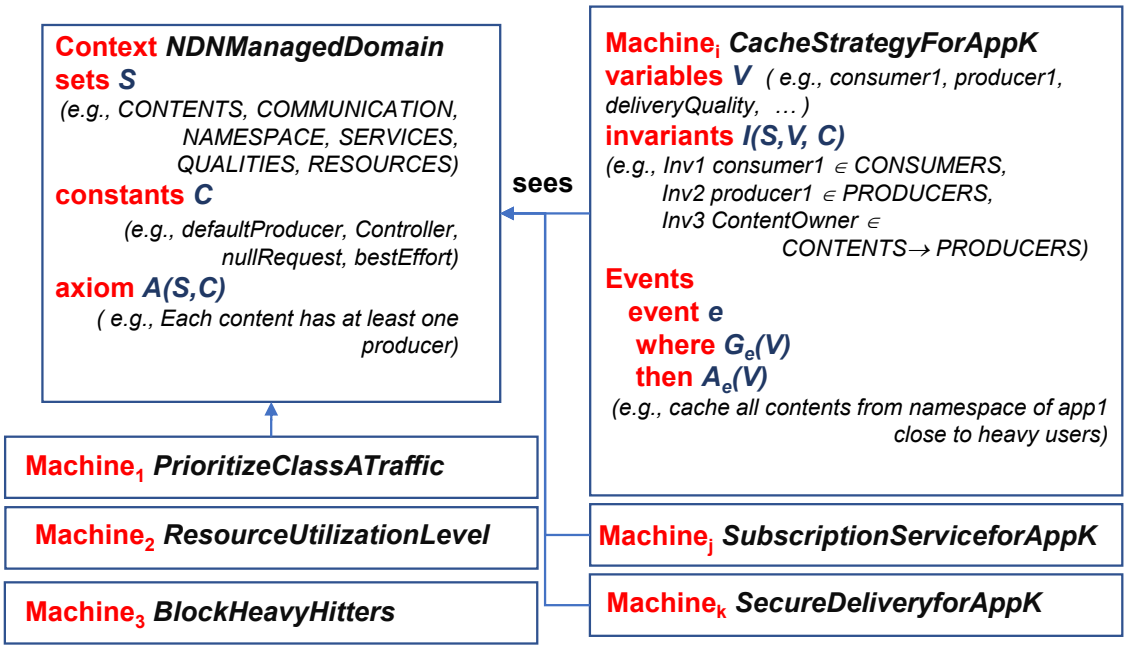


Figure 6.4: Examples of Abstract EBMs

words stored in the vector $\mathbf{v_i} = (w_1, w_2, \cdots)$ storing the corresponding values of the slots. For example, using the first intent template in Fig. 6.3, when the user utters "***Producer1*** *and* ***Producer2*** *should serve* ***Video*** *between* ***3:00pm to 5:00pm***", the identification module's output is the intent template name *LoadBalanceAction*, the slot labels sequence $\mathbf{s} = \{$p1, p2, cn, t$\}$, and the slot values $\mathbf{v} = \{$"Producer1", "Producer2", "Video", "3:00pm to 5:00pm"$\}$. It is worth noting that slot values correspond to abstract values that can later be mapped to concrete network-specific values. For instance, the **Video** slot value corresponds to a content name prefix identifying all the video contents of a specific application in the previous example.

We adopt open-source machine learning-based tools, such as DeepPavlov [177] in this phase. Models of the intents are first defined and stored as JSON objects data sets. The tool is then trained using a graphical user interface until it can correctly identify the intents. When a new intent template is added, the system is retrained to recognize the intent. The outcome of the intent identification phase is a selected intent and a list of slot labels and their values that are then passed to the intent translation phase.

```
CONTEXT NetworkContext extends GenericContext
SETS Producers LoadBalanceAlgorithms Namespaces
CONSTANTS P1 P2 P3 AnyProducer NoProducer RoundRobin WRR
AnyNamespace Video Music
AXIOMS
@axm1 partition(Producers,{P1},{P2},{P3},{AnyProducer},
{NoProducer})
@axm2 partition(LoadBalanceAlgorithms, {RoundRobin}, {WRR})
@axm3 partition(Namespaces, {AnyNamespace},{Video}, {Music})
END
```

Figure 6.5: An Event-B context

6.3.2 EBM Templates and Intent Translation

We will first describe the abstract EBM templates that the operator creates for each intent and slot sequence. As shown in Fig. 6.4, we implement an intent behavior in Event-B using two components: a context $\mathcal{C}$ and an abstract machine $\mathcal{M}$. The context $\mathcal{C}$ defines the relatively static state of the network and is shared by all the machines. On the other hand, every machine implements the behavior of a specific intent.

As shown in Fig. 6.2, the network context is created during the pre-production time but can be updated during production. In Event-B, the context is used to define new data types that are associated with the variables representing the state of EBMs [160]. In our architecture, we thus use the context to represent the types of different resources and objects that are available or can be manipulated in the network. Examples are producers, consumer regions, content namespaces, or scheduling algorithms. Fig. 6.5 shows the Event-B code of a network context which contains three sections: Sets, Constants, and Axioms. Hence, the context can be modelized by the set of sets $\mathcal{C}$(**S**, **C**, **A**). Here, **S** lists all the types (i.e., the categories of objects or resources that comprise or interact with the network). The constants set **C** stores possible elements of the sets in **S** (e.g., the possible content producers). Here constants can also refer to names of algorithms or control-plane mechanisms that can be resolved during refinement. For example, the *LoadBalanceAlgorithms* set stores the constants *RoundRobin* and *WRR* that correspond to different scheduling algorithms for a load balancer. Finally, the axioms set **A** is used

Table 6.3: Intent to EBM mapping

Intent Templates	**Abstract EBM templates**
Template name n and slot sequence $\mathbf{s_i}$	Machine $\mathcal{M}_i$
Predefined slot types (e.g., Producers and Namespaces)	Sets in $\mathbf{S}$ in the network context $\mathcal{C}$
Slot labels $l_{ij} \in \mathbf{s}_i$	Template parameter variables in $\mathbf{V}$ in $\mathcal{M}_i$
Possible slot values	Constants in $\mathbf{C}$ of the context $\mathcal{C}$
Semantic of the intent behavior	Constraints represented by invariants $\mathbf{I}$ of $\mathcal{M}_i$
Intent behavior implementation	Concrete EBM $\mathcal{M}_i\prime$ logic (events $\mathbf{E}$ acting on variables $\mathbf{V}$)

mainly to link constants to their set (e.g., *axm1* in Fig. 6.5). But it is worth noting that axioms can also be used to specify properties of sets and constants (e.g., every content namespace must have at least one producer).

A machine template $\mathcal{M}$ contains the implementation of an intent behavior. Table 6.3 summarizes how intents are mapped to EBMs. At the level of the intent, the network is seen as a blackbox whose expected outcomes are specified. However, in the EBM, we go inside this blackbox and model how the network processes packets to satisfy the intent. In NDN, the network processes two types of packets: Ipkts and Dpkts. Hence, our EBMs specify the stateful treatment of Ipkts and Dpkts inside the network. More precisely, an EBM models an ENDN network and its possible packet processing actions using a set of variables $\mathbf{V}$. The variables have a type that can either be a native type (e.g., boolean or integer) or one of the new types defined in the context $\mathcal{C}$.

The EBM variables can be classified into four categories: packet variables, flow variables, abstract variables, and slot parameters. Packet variables correspond to any data specific to a single packet. Hence, they are used to represent header fields (e.g., content name), individual packet forwarding actions (e.g., drop or forward to a specific destination), or metadata (e.g., queue priority, received timestamp, or source region). Packet variables are thus reinitialized each time the network receives a new packet. On the other

hand, flow variables represent stateful information that is kept in the network. Examples are data managed by stateful algorithms (e.g., number of packets sent to a specific destination) or contents cached in the network. Abstract variables are only allowed at the level of abstract EBMs and correspond to parts of the packet processing treatment that have not yet been specified in detail. For instance, an abstract EBM may have an abstract variable representing the result of a congestion detection mechanism without detailing how this mechanism works. This abstract EBM would then specify how to process packets in case of congestion based on the value of this abstract variable. The refinement process eliminates the abstract variables by replacing them with the corresponding algorithms. The operator has the complete freedom to decide on the abstraction level that is represented by these abstract variables. A higher level of abstraction will provide more flexibility to adapt to different network domains and capabilities at the expense of more refinement steps. Finally, slot parameters are used to make an EBM generic by allowing its behavior to be parametrized.

Packet processing actions are represented in EBMs by a set of events **E** that act on the variables **V**. The events have an *if (condition) then action* semantic, where both the condition and actions are relative to the variables **V**. Hence, an event $e \in \mathbf{E}$ can be formally modeled as a conditional statement: $e := if(G_e(V))\ then$ V:= $A_e(V)$. The event guard G_e contains a list of logical conditions on the values of the EBM variables V that can trigger the event. On the other hand, the event action $A_e(V)$ specifies how variables are modified when e is executed. Hence, each event that is triggered brings the network from one state to another state. The possible states of the machine are restricted by several conditions on the variables represented by the set of invariants **I**. Finally, it is worth noting that each machine contains an initialization event that is executed as the first event in the machine. It assigns different values to the machine variables in order to define the desired initial state (e.g., the number of received requests for a specific content is initialized to 0, or the cached contents set is initialized with the empty set).

To better explain how EBMs work, we will present in detail a simple load balancer

```
MACHINE LoadBalancer SEES NetworkContext
VARIABLES processingStepIpkt processingStepDpkt
allowDpkt deliveryPattern previousIpktDestination
ipktDestination ipktContentName slotLoadBalancedNamespace
slotProducer1 slotProducer2 slotLoadBalancerAlgorithm
INVARIANTS
...
@inv3 ipktContentName ∈ Namespaces
@inv4 slotLoadBalancedNamespace ∈ Namespaces
@inv5 slotProducer1 ∈ Producers
@inv6 slotProducer2 ∈ Producers
@inv7 slotLoadBalancerAlgorithm ∈ LoadBalanceAlgorithms
@inv8 ipktContentName = slotLoadBalancedNamespace ⇒
      ipktDestination ∈ {slotProducer1, slotProducer2}
EVENTS
EVENT INITIALISATION THEN
...
@act5 previousIpktDestination ≔ AnyProducer
...
@act8 slotLoadBalancedNamespace ≔ Video
@act9 slotProducer1 ≔ P1
@act10 slotProducer2 ≔ P2
@act11 slotLoadBalancerAlgorithm ≔ RoundRobin
END
EVENT receiveIpkt ANY previousDestination contentName
WHERE
@grd1 processingStepIpkt = 0
@grd2 deliveryPattern ≠ Push
@grd3 previousDestination ∈ Producers
@grd4 contentName ∈ Namespaces
THEN
@act1 processingStepIpkt ≔ 1
@act2 previousIpktDestination ≔ previousDestination
@act3 ipktContentName ≔ contentName
@act4 ipktDestination ≔ slotProducer1
END
EVENT processIpktToP1 WHERE
@grd1 ipktContentName = slotLoadBalancedNamespace
THEN
@act1 processingStepIpkt ≔ processingStepIpkt + 1
@act2 ipktDestination ≔ slotProducer1
END
EVENT processIpktToP2 WHERE
@grd1 ipktContentName = slotLoadBalancedNamespace
THEN
@act1 processingStepIpkt ≔ processingStepIpkt + 1
@act2 ipktDestination ≔ slotProducer2
END
EVENT processIpktOtherNamespace WHERE
@grd1 ipktContentName ≠ slotLoadBalancedNamespace
THEN
@act1 processingStepIpkt ≔ processingStepIpkt + 1
@act2 ipktDestination ≔ previousIpktDestination
END
EVENT deliverIpkt THEN
@act1 allowDpkt ≔ TRUE
@act2 processingStepIpkt ≔ 0
END
EVENT receiveDpkt ...
EVENT processDpkt ...
EVENT deliverDpkt ...
```

Figure 6.6: The Load Balancer abstract EBM

intent example. The application developer wants to distribute the load of requests for a video namespace between two producers using the round-robin algorithm. The following intent is then uttered: *"Distribute received requests for **Video** between **P1** and **P2** using the **RoundRobin** algorithm"* and the following slot values are extracted: **Video**, **P1**, **P2**, and **RoundRobin**. These slot values are then passed to the abstract EBM template shown in Fig. 6.6: they serve to initialize the *slotLoadBalancedNamespace*, *slotProducer1*, *slotProducer2*, and *slotLoadBalancerAlgorithm* template parameter variables in the *INITIALISATION* event. Fig. 6.6 shows that EBMs have three main sections: *VARIABLES*, *INVARIANTS*, and *EVENTS*. The *VARIABLES* section lists all the variables used by the EBM while the *INVARIANTS* section is initially used to specify the type of these variables. *ipktContentName* and *ipktDestination* are examples of packet variables, while this EBM contains no flow or abstract variables. Every event contains

a guards section introduced by the *WHERE* keyword that contains several conditions on the variables, as well as an actions section introduced by the *THEN* keyword where variables are modified. There are two main events for the Ipkts and Dpkts: a receive event that initializes the packet variables and a deliver event that allows the receive event to be triggered again. The receive event has an event parameters section introduced by the *ANY* keyword to represent the possible initialization values of packet variables constrained by a guard condition. For instance, the event parameter *contentName* of the *receiveIpkt* event, alongside the guard *"contentName ∈ Namespaces"*, specify that the Ipkt content name header field can be any namespace from the *Namespaces* set defined in the context (cf. Fig 6.5). Finally, there are three events that process Ipkts with the following behavior: if the Ipkt content name is the same as the load-balanced namespace specified in the intent, then the packet is either forwarded to the first or second producers; otherwise, no action is done. As a result, the abstract EBM only describes the details of the namespace check and the packet forwarding, while the exact implementation of the load balancing algorithm is left for the refinement process.

It is worth noting here an essential capability of Event-B that comes from the expressiveness of invariants. While several invariants specify the type of variables, other invariants are used to put constraints on the values of variables. For example, *inv8* in Fig. 6.6 imposes that Ipkts requesting content in the *slotLoadBalancedNamespace* can only be forwarded either to *slotProducer1* or *slotProducer2*. This constraint corresponds to one part of the semantic of the load balancer intent. Hence, invariants can also be used to represent the expected outcomes of an intent behavior using constraints on variables. Examples of constraints that can be represented as invariants are: *the currently served request must belong to the set of authorized contents*, *the requesting user location must be within a certain geographical area*, or *the number of responses should not exceed the number of requests in a pull delivery pattern*. All the events of the EBM are then checked using the Event-B proof engine (cf. Fig. 6.2) to make sure they do not violate the constraints set by invariants. As a result, both the invariants and the proof engine

```
MACHINE LoadBalancerRR REFINES LoadBalancer SEES
NetworkContext
VARIABLES processingStepIpkt … slotLoadBalancerAlgorithm
currentPosition numIpktsP1 numIpktsP2
INVARIANTS
@inv1 currentPosition ∈ ℕ
@inv2 numIpktsP1 ∈ ℕ
@inv3 numIpktsP2 ∈ ℕ
@inv4 currentPosition < 2
@inv5 currentPosition = 0 ⇒ numIpktsP1 = numIpktsP2
@inv6 currentPosition = 1 ⇒ numIpktsP1 = numIpktsP2 + 1
EVENTS
EVENT INITIALISATION EXTENDS INITIALISATION THEN
@act12 currentPosition ≔ 0
@act13 numIpktsP1 ≔ 0
@act14 numIpktsP2 ≔ 0
END
EVENT receiveIpkt EXTENDS receiveIpkt END
EVENT processIpktToP1 EXTENDS processIpktToP1 WHERE
@grd2 currentPosition = 0
@grd3 processingStepIpkt = 1
THEN
@act3 numIpktsP1 ≔ numIpktsP1 + 1
@act4 currentPosition ≔ 1
END
EVENT processIpktToP2 EXTENDS processIpktToP2 WHERE
@grd2 currentPosition = 1
@grd3 processingStepIpkt = 1
THEN
@act3 numIpktsP2 ≔ numIpktsP2 + 1
@act4 currentPosition ≔ 0
END
EVENT processIpktOtherNamespace EXTENDS
processIpktOtherNamespace WHERE
@grd2 processingStepIpkt = 1
END
EVENT deliverIpkt EXTENDS deliverIpkt WHERE
@grd1 processingStepIpkt = 2
END
EVENT receiveDpkt EXTENDS receiveDpkt END
EVENT processDpkt EXTENDS processDpkt END
EVENT deliverDpkt EXTENDS deliverDpkt END
```

Figure 6.7: The resulting concrete EBM of the Load Balancer with Round Robin

result in the *"correct by construction"* feature of Event-B.

Abstract EBMs are refined to gradually have additional implementation details until the intent behavior is completely specified. In Event-B, a refinement extends an initial EBM by adding new variables, invariants, and events [160]. Events of the abstract EBM can also be refined by adding new guards and actions, with the restriction that the refined

event results in exactly the same outcome on the variables of the abstract EBM. This restriction ensures that refined versions of an event may not violate the invariants of the abstract EBM. In other words, refinements are syntactical extensions of an EBM that preserve the invariants. Fig. 6.7 shows the concrete machine resulting from the refinement of the abstract load balancer machine of Fig. 6.6 when the round-robin algorithm is used. The *currentPosition*, *numIpktsP1* and *numIpktsP2* flow variables are added alongside three invariants that impose the round-robin scheduling constraint (*inv4*, *inv5*, and *inv6*). The *processIpktToP1* and *processIpktToP2* events are then refined accordingly by adding new guards and actions. In our architecture, we use the refinement patterns concept introduced by Iliasov et al. [174]. Refinement patterns allow us to automate the implementation of refinements by formally specifying every EBM syntactical modification that is part of a refinement. Refinement patterns also have applicability conditions that allow them to be triggered when needed. For instance, the refinement that led to the concrete machine of Fig. 6.7 was triggered by the presence of the value *RoundRobin* in the *slotLoadBalancerAlgorithm* variable.

When the concrete machine is created, it is processed by the EBM analyzer in order to associate it to the namespaces it needs to process as part of a list of ENDN services. The next section describes the different processes performed by the EBM analyzer.

6.3.3 EBM Analyzer

The EBM analyzer first performs several consistency checks on the concrete EBM to make sure it does not conflict with other intents already configured in the network. These consistency checks are based on the fact that every EBM has invariants that specify the expected outcome of the corresponding intent behavior. Consequently, we can check that two EBMs do not conflict with each other by validating the events of the first EBM against the invariants of the second EBM and vice-versa. In order to perform these consistency checks, the two EBMs have to be composed to create a combined EBM containing the invariants and events of both machines. The details of EBMs

composition are outside the scope of this thesis. However, several efficient schemes exist in the Literature [175]. The creation of the combined EBM results in the generation of several invariant preservation proof obligations. The Event-B proof engine then examines these proof obligations that require that all events preserve the invariants. Automated tools like Rodin [161] can automatically process most if not all proof obligations; any remaining ones may be proved manually. If a proof obligation can not be proved, it means that the two intents, or their implementations, are conflicting. The new intent is rejected, and the user who submitted the intent is notified. Once the concrete EBM is validated, it is further analyzed to infer the different namespaces it processes.

Concrete EBMs represent a single ENDN service that potentially processes several namespaces. The last step performed by the EBM analyzer is thus to infer these namespaces from the concrete EBM based on the values of the *ipktContentName* and *dpktContentName* variables. For instance, our load balancer example only processes the *Video* namespace. It is worth noting that, in general, a concrete EBM does not necessarily perform checks on the *ipktContentName* and *dpktContentName* variables. In that case, the concrete EBM applies to all the namespaces. However, we limit this case to intents uttered by network operators: application intents must explicitly specify their namespaces in the concrete EBM. Once the namespaces are extracted, the concrete EBM is added to the list of ENDN services attached to each of these namespaces and passed to the control plane. The data-plane configuration is then updated.

6.3.4 Data-Plane Configuration Generation

As presented in the previous chapter, the control plan processes the namespaces and list of associated services in order to generate the data-plane configuration. However, in the previous chapter, we directly used P4 templates as our service representation. Here, services are represented by concrete EBMs. We thus have to translate these EBMs to equivalent P4 code.

Table 6.4: EBM to MAT mapping rules

Concrete EBM Component	**MAT Component**
Constants in the context $\mathcal{C}$ (e.g., content producer)	Switch-specific constants (e.g., output port leading to the producer)
Packet variables in $\mathbf{V}$	Packet header fields, metadata, or special function calls (e.g., execute a meter)
Flow variables in $\mathbf{V}$	MAT stateful fields (e.g., P4 registers)
Processing step value	MAT table
Event guards G_e	MAT matching key and rules
Event actions A_e	MAT actions

Our ENDN/EP4 switches contain programmable P4 MATs as part of both the Ipkt and Dpkt pipelines. A P4 MAT can be used to select custom forwarding actions based on values derived from packet header fields, metadata, or measured statistics. The possible actions include forwarding the packet to one or more network ports, dropping it, sending it to the CS, notifying the controller, modifying header fields, as well as storing a custom state in the switch. The MAT execution structure can be modeled as a collection of conditional rules of the form *if (condition on fields) then do action*. The MAT execution structure thus closely resembles the event execution model of EBMs. Hence, we can map EBMs to MATs by following the rules in Table 6.4.

We can classify the EBM components into four categories: events, variables, context constants, and non-mappable components (e.g., invariants). Events are directly mapped to MAT rules: event guards are mapped to rule conditions, while event actions are mapped to rule actions. Only packet and flow variables can be mapped to an MAT component. Abstract variables are processed by the different refinement patterns, and are thus not allowed at the level of the concrete machine, while slot parameter variables are considered as constants. Packet variables are standard and have special mapping rules to MAT fields: they are mapped to packet header fields (e.g., content name), function calls (e.g., execute a meter), or metadata fields (e.g., source and destination ports). Flow variables are usually custom and are mapped to stateful variables in the MAT (e.g., P4

```
#define contentNS_hash {Video_hash}
#define P1_port {Producer1_port}
#define P1_forwardingHint {Producer1_forwardingHint}
...

control P4_MAT(inout standard_metadata_t std_meta) {
   bit<64> ipktContentName_hash;
   register<bit<64>>(1) numIpktsP1;
   bit<64> numIpktsP1_local;
   ...
   register<bit<64>>(1) currentPosition;

   action processIpktToP1() {
      std_meta.selectedOutputPort = P1_port;
      setForwardingHint(P1_forwardingHint);
      numIpktsP1.read(numIpktsP1_local, 0);
      numIpktsP1_local = numIpktsP1_local + 1;
      numIpktsP1.write(0, numIpktsP1_local);
      currentPosition.write(0, 1);
   }
   action processIpktToP2() {
      ...
   }
   action processIpktOtherNamespace() {}

   table processingStepIpkt1 {
      key={
         ipktContentName _hash: exact;
         currentPosition _local: exact;
      }
      actions={
         processRequestToP1;
         processRequestToP2;
         processRequestToP3;
         processIpktOtherNamespace;
      }
      const entries={
         (contentNS_hash, 0) : processIpktToP1();
         (contentNS_hash, 1) : processIpktToP2();
         (_, _): processIpktOtherNamespace();
      }
   }
   action receiveIpkt() {
      getContentNameHash(ipktContentName_hash);
      currentPosition.read(currentPosition _local, 0);
      processRequest.apply();
   }
   action receiveDpkt() {}

   apply {
      if (std_meta.isInterest) {
            receiveIpkt();
      } else {
            receiveDpkt();
      }
   }
}
```

Figure 6.8: The resulting P4 code of the Round Robin Load Balancer

registers). Finally, the context constants are translated to local values for the switch (e.g., a producer is mapped to an output port number and a forwarding hint value). It is worth noting that we can also have special flow variables in EBMs. These can be used to specify some requirements on the EFIB and EPIT rules (e.g., the EFIB routes need to be computed using the shortest path algorithm).

Fig. 6.8 shows an example of a P4 code corresponding to the round robin load balancer concrete EBM of Fig. 6.7. The different Event-B components are mapped to the corresponding P4 structures: flow variables become registers (in blue in the code), packet variables become metadata fields or function calls (in green in the code), and context constants become *define* statements (in red in the code). In the concrete EBM, a special variable called *processingStepIpkt* allows the events to be organized as possible alternative in a specific processing step of Ipkts. For example, in Fig. 6.7, the *receiveIpkt* event corresponds to the processing step 0, then the *processIpktToP1*, *processIpktToP2*, and

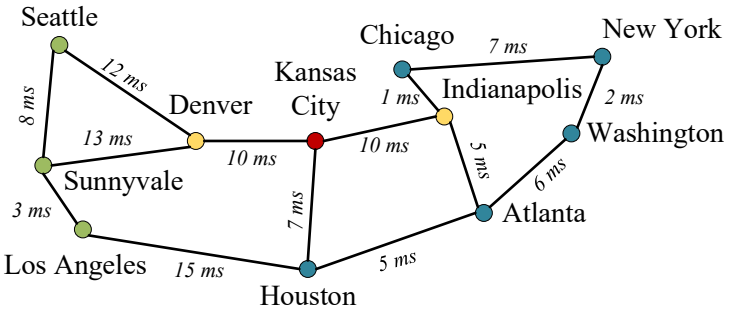


Figure 6.9: The Abilene topology

processIpktOtherNamespace can happen at the processing step 1, finally the *deliverIpkt* event happens during processing step 2. Events that are on the same processing step are mutually exclusive, and thus correspond to different match-action rules in a single MAT. Every processing step thus results in the creation of a new P4 table (e.g., *processingStepIpkt1* table in Fig. 6.8), except for the processing steps of the receive and deliver events. The actions of the events are then mapped to P4 actions accessible from their associated processing step table. Finally, the event guards become entries in the corresponding P4 table. The resulting P4 code can then be installed in the data plane.

6.4 Performance Evaluation

This section demonstrates the advantages of our proposed ISL. More precisely, declarative goals are expressed as intents, and then translated into data-plane configurations. We then measure the performance gains achieved by these intents when compared to the performance of a traditional NDN configured with shortest path routes and best route strategies [70]. Our experiments employ the Abilene topology [178] built using ENDN/EP4 switches within the ndnSIM simulator [107].

6.4.1 Test Scenario

Fig. 6.9 shows the Abilene topology used in our simulation. All links have a rate of 1Mbps and introduce a propagation delay based on the geographical distance between the cities. We consider a content delivery application with content geo-gating requirements where access to contents is restricted based on the geographical region of the users. More precisely, users from cities on the east coast of the United States (blue nodes in Fig. 6.9) can only access content specific to their region, and similarly for users from west coast cities (green nodes in Fig. 6.9). Denver and Indianapolis are regional producers that cache the content of their region, and Kansas City is a national producer that can serve requests from both regions while ensuring the geo-gating restrictions using an application-level logic. To configure the network, the application developer initially defines three intents (words in italic correspond to slot values, and the application namespace is */MyApp*):

- **I1:** *Indianapolis* can only serve requests for */MyApp* content coming from the *east coast*.
- **I2:** *Denver* can only serve requests for */MyApp* content coming from the *west coast*.
- **I3:** *Kansas City* can serve all requests for */MyApp* content.

Additionally, the application developer would like to limit the content requests served by regional producers by automatically offloading any excess requests towards Kansas City. This results in two additional intents:

- **I4:** Limit the */MyApp* content requests served by *Indianapolis* to *100 requests/s* and offload any excess requests to *Kansas City*.
- **I5:** Limit the */MyApp* content requests served by *Denver* to *100 requests/s* and offload any excess requests to *Kansas City*.

We also consider a second application that requires content from the east coast requested by users in the west coast to be delivered with the lowest delay. The content of this application is urgent, so the application developer agreed with the network providers to have the application traffic forwarded with a higher priority. Additionally, the application developer requests proactive caching of the contents in the west coast when the number of requests reaches a certain threshold. In this case, the reception of a new request triggers a secondary request initiated by the P4 code to retrieve other available contents from the east coast to cache them locally. As a result, the application developer selects two intents:

- **I6:** Serve */UrgentContent* traffic with *high* priority.
- **I7:** Proactively cache */UrgentContent* contents in the *east coast* if the number of requests reaches *20 requests per day*.

Finally, the network operator selects a strategy intent that locally avoids congestion in the network by always providing two alternative paths to any destination in every switch. The shortest path is used unless the link utilization reaches 90%. In that case, an alternative path is used. This results in the following intent:

- **I8:** Avoid congestion in the network by keeping the link utilization below *90%*.

Intents **I1** and **I2** correspond to the same intent template with different slot values (and similarly for intents **I4** and **I5**). The different intents are then processed by our architecture and result in several P4 functions that are placed in the switches as follows:

- The P4 functions corresponding to intents **I1** and **I2** are placed in the east coast and west coast nodes respectively (i.e., the green and blue nodes in Fig. 6.9). These P4 functions add a forwarding hint towards Indianapolis or Denver to the */MyApp* Ipkts originating from the east or west coasts respectively.

- **I3** is translated to a P4 function placed in Kansas City that automatically sends Ipkts to the central producer even if a forwarding hint to Denver or Indianapolis is present.

- **I4** and **I5** are mapped to a rate-limiting P4 function placed in Denver and Indianapolis. It measures the rate of */MyApp* requests and offloads any traffic over 100 requests/s to Kansas City.

- **I6** is implemented as a P4 function that requires all */UrgentContent* packets to be processed with a high queue priority. This P4 function is installed in all the switches along the path followed by */UrgentContent* packets.

- **I7** is translated to a P4 function placed in Denver that proactively caches the */UrgentContent* in the local CS.

- The P4 function generated by **I8** is placed in all the switches. It processes all Ipkts containing forwarding hints towards specific destinations (e.g., Denver or Indianapolis in our scenario) by sending them to a secondary path in case of congestion. The algorithm also makes sure to check the source port from which packets are received to avoid creating forwarding loops by sending the packet back through the face from where it was received.

At t=0s, consumers from every city of the east and west coasts (i.e., the green and blue nodes in Fig. 6.9) start requesting */MyApp* content at a rate proportional to the size of their population. From t=100s to t=150s, there is a rush period where additional traffic is added, resulting in congestion on the east coast. Finally, the */UrgentContent* located in Atlanta is requested by a consumer in Seattle at a slow exponentially distributed rate with a mean of 1 request/s during the entire simulation time. The RTT (including transmission, propagation, and queuing delays), packet loss rate, and received Dpkt throughput are measured for the */MyApp* traffic originating from every city as well as for the */UrgentContent* traffic. We then compare the performance of our ENDN network

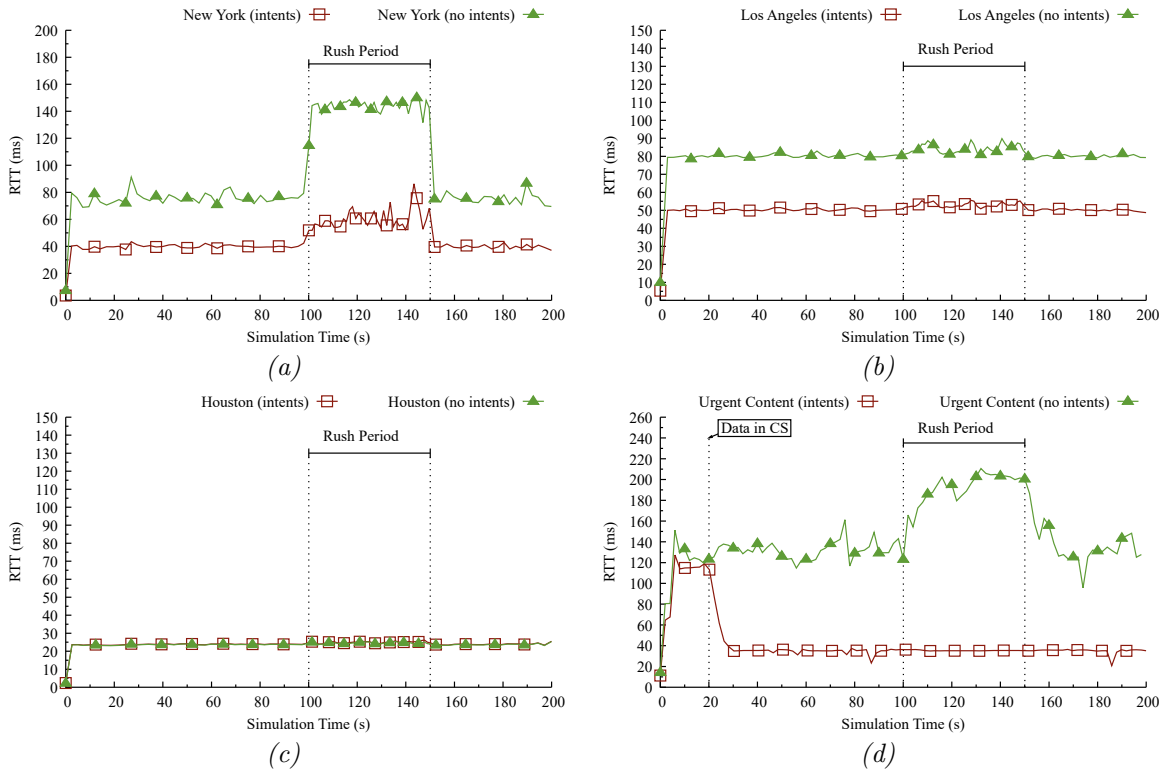


Figure 6.10: Measured RTT for the /MyApp traffic coming from different cities and for the /UrgentContent traffic.

configured using the intents described above against that of a standard NDN network with no intents. The latter forwards all */MyApp* requests to Kansas City using the shortest path as geo-gating can only be guaranteed by the central producer.

6.4.2 Experimental Results

Fig. 6.10 shows the measured RTT for the */MyApp* traffic originating from Los Angeles, Houston, and New York. The effects of satisfying intents **I1** and **I2** are visible in Figs. 6.10a and 6.10b: the RTT increases by around 30ms when no intents are used because the packets are served by the central producer in Kansas City instead of the regional producers. This additional delay is consistent with the propagation delay of 10ms between the regional producers and Kansas City added twice to the transmission delay of a 1KB Dpkt over the 1Mbps links. On the other hand, **I3** allows the requests originating

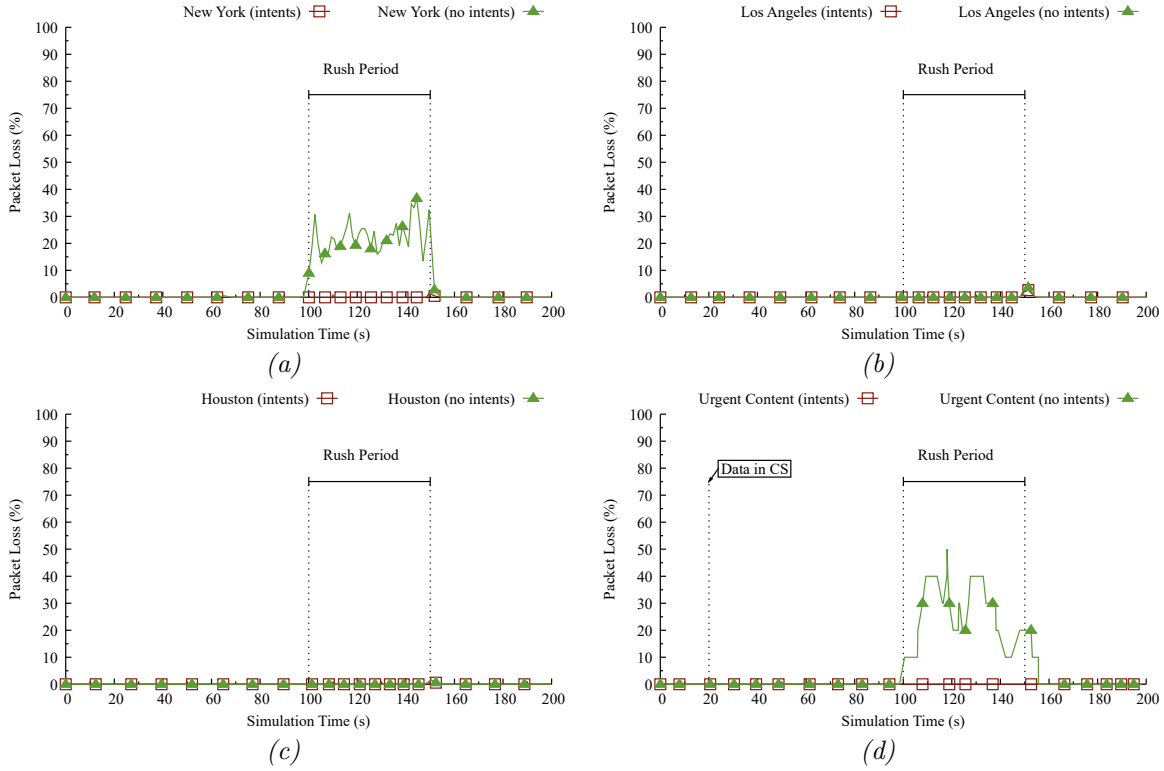


Figure 6.11: Measured packet loss for the /MyApp traffic coming from different cities and for the /UrgentContent traffic.

from Houston to be processed directly by the central producer, which is closer than the regional producers. During the rush period, the traffic is increased, which causes the Indianapolis rate-limiting threshold defined by **I4** to be reached. The excess traffic is thus offloaded to Kansas City which causes the RTT of the New York traffic to increase slightly in Fig. 6.10a.

Fig. 6.10d clearly shows the effects of intents **I6** and **I7**. At around t=20s, the Denver switch proactively caches the */UrgentContent* Dpkts which causes a significant decrease of the RTT. Additionally, the delay remains unchanged during the rush period when intents are used as the */UrgentContent* traffic is treated with a high queue priority.

The effect of the congestion avoidance intent **I8** is mainly visible during the rush period. During this time, the traffic increases, as shown in the throughput plots of Fig. 6.12, which causes congestion in the east coast. This causes an increase in delay for the

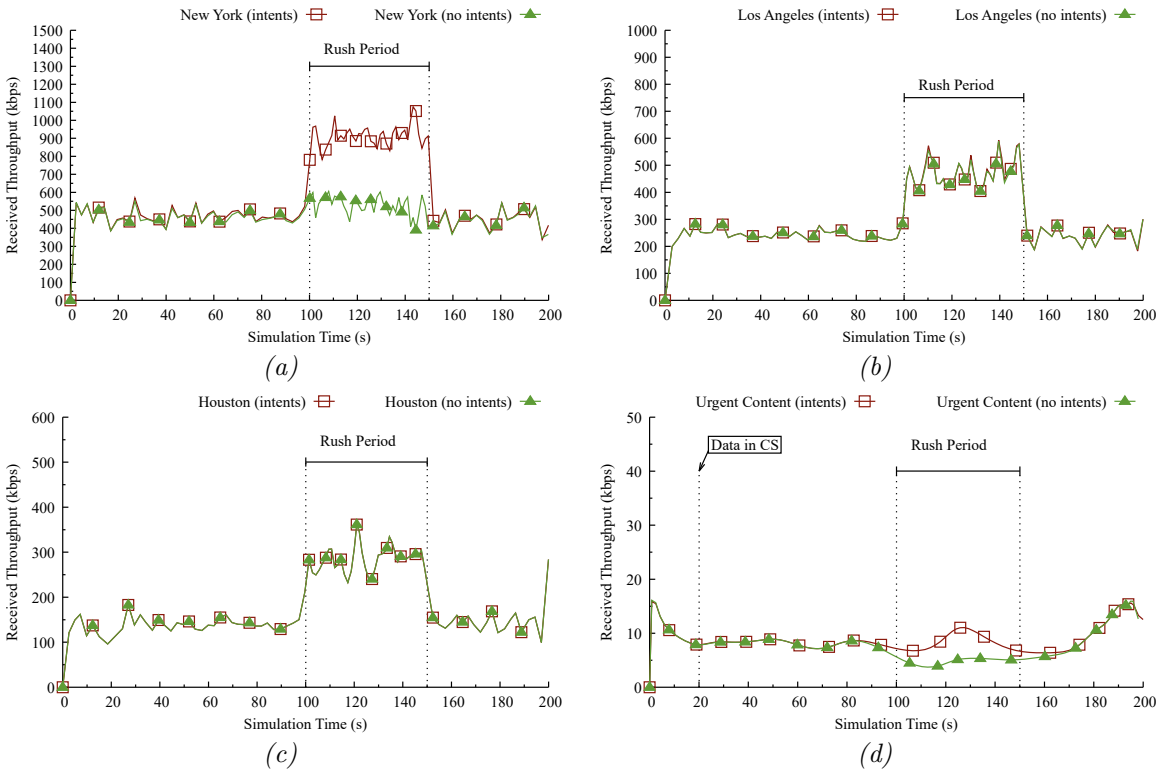


Figure 6.12: Received Dpkt throughput measured for the /MyApp traffic coming from different cities and for the /UrgentContent traffic.

New York and */UrgentContent* traffic (cf. Figs. 6.10a and 6.10d) when no intents are used. The congestion also causes an increase in packet loss and a decrease in received throughput as shown in Figs. 6.11a, 6.11d, 6.12a, and 6.12d. On the other hand, the P4 function that was generated from **I8** has successfully avoided the congestion. Hence, there is no degradation of performance when intents are used.

Our proposed architecture has allowed the network to adapt to the needs of the application and network operators while improving network manageability and configuration through intents. This, in turn, resulted in a better network performance compared to traditional non-intent-based networks.

6.4.3 Computational Cost

Finally, we analyze the computational cost that is introduced by the intents on both the control and data planes.

At the control plane, our architecture processes intents asynchronously from the data plane operations: data-plane configurations are generated once when an intent is processed but are not modified later during packet processing. More precisely, every intent is completely translated into an autonomous data plane configuration/program that does not interact with the control plane. Hence, the communication overhead between the control and data planes takes place once. The operator can limit data-plane updates to batch processes. The main control-plane cost is incurred during the installation of a new configuration in the data plane. However, our programmable data-plane architectures allows fast runtime reconfigurability of P4 programs which makes the impact of data-plane reconfigurations minimal on the switch operation.

At the data-plane level, P4 programs introduce a processing delay dependent on the switch hardware or software implementation [179]. Several high-performance P4 switch implementations were proposed in the Literature to significantly reduce this processing delay, especially using FPGAs [60] or GPUs [180]. It is worth noting, however, that most high-performance P4 switches are limited in the number of P4 programs that can be executed in parallel (e.g., P4VBox can execute up to 13 P4 programs in parallel [152]). Hence, the main data-plane cost overhead can be characterized by the number of P4 functions that are needed in the network for a specific set of intents. In our test scenario, we notice that some intents are mapped to a single P4 function (e.g., **I3** or **I4**), while other intents are implemented as a P4 function placed in every switch (e.g., **I6** or **I8**). It is worth noting though that several intents correspond to the same intent template with different slot values. These intents can thus be shared at the data-plane level by calling the same P4 function using different parameters. The number of P4 functions at the data-plane level can thus be reduced using P4-function sharing. In previous chapters, we have discussed the trade-off between scalability and intent customizability and performance

that depends on the available MAT resources at the data-plane level. This trade-off is decided by the network operator and embedded in a control-plane logic at the level of the Content Path Generation module presented in the previous chapter. The details of this logic consist in solving constrained optimization problems and are outside the scope of this thesis.

6.5 Conclusion

This chapter proposed a novel Intent-Service Layer (ISL) that captures high-level service intents and translates them into data-plane configurations. Our ISL employs the Event-B modeling and refinement concepts to represent high-level intents using abstract Event-B Machines (EBMs) and then refine them to machines that can be used to configure the data plane. We have provided a detailed description of the modeling and mapping steps for translating intents to EBMs and refining these machines. Finally, we showed how these produced EBMs could be translated to instructions on the data-plane match action tables. Experimental evaluation results demonstrate the feasibility and efficiency of the various functionalities of the architecture.

Chapter 7

Conclusion and Future Work

In this thesis, we introduce ENDN, an enhanced NDN architecture aimed at adapting to next-generation Internet applications' requirements. ENDN allows applications to specify their custom forwarding requirements using high-level abstract intents. The latter are processed by a novel Intent-Service Layer (ISL) that links the application flows, identified by namespaces, with a list of content delivery services implementing the requested intents. The intents are chosen from an extensible library of intents representing a catalog of network services available to the application developers. The control plane then transforms the application service requirements into a data-plane configuration. The first part of the data-plane configuration is table-based and specifies the behavior of a scalable ingress pipeline. The latter offers several content delivery patterns (e.g., pull or push) using the EFIB and EPIT tables implemented with our scalable FCTree data structure. The ingress pipeline also associates application flows with a P4 behavior that corresponds to the second part of the data-plane configuration. P4 functions are executed in isolated targets in the programmable egress pipeline, thus allowing for efficient configuration updates. Our proposed data plane has also been generalized to allow interfacing with non-NDN networks using programmable parsers, thus resulting in a protocol-independent data plane.

7.1 Limitations and Future Research Areas

This section discusses several assumptions and limitations of our proposed work and highlights future research opportunities:

1. **Intent model:** The proposed intent model takes a major step forward towards representing intents that can capture the operator and developer goals in a much more declarative way compared to the traditional event-condition-action models [157]. However, the model relies on predefined classes of intents where users must utter one of the predefined intents. This model represents a closed-world model. An open-world intent model can accept and identify unknown, not previously seen, intents from the users. In the Literature, several open-world and multiple intents models have been developed in other contexts, such as chatbots [172], but remain a challenge for IDN.

2. **Single vs. multiple network domains** In the proposed work, we considered a single subnetwork with a single control domain. Extending the proposed work to multiple independent domains that necessitate the collaboration and orchestration between several controllers is left as future work.

3. **Learning and run-time adaptation:** Thus far, our work has focused on the mapping of user intents to PDP configurations while assuring conflict resolution and validation before they are installed. We believe that producing an efficient intent model and intent to data-plane translation methodology represents a first step towards realizing self-configuring and healing IDN. Hence, the challenge of monitoring and analyzing the network behavior and adapting it at run-time remains a future work.

4. **Scalability and resource management:** ENDN's control plane transforms the different requirements of applications into a data plane configuration containing several P4 functions. The maximum number of P4 functions that can be installed

in an ENDN switch impacts the generation of the data plane configuration. The control plane must decide which application behaviors can be implemented in a single P4 function shared by multiple namespaces and which behaviors need to use a new custom P4 function.

5. **Trust and security:** ENDN provides an extensive control of the network to the applications which may result in security vulnerabilities. Moreover, while ENDN shares several security features with NDN (e.g., packet signing and Ipkt Denial of Service (DoS) mitigation through EPIT aggregation), some additional functionalities are susceptible to attacks to which NDN is immune (e.g., push traffic DoS). However, while a study of the security implications of the ENDN design is an area of potential future work, it is outside of the scope of this thesis.

List of Publications

Journal Papers

- O. Karrakchou, N. Samaan, and A. Karmouch, "Mapping applications intents to programmable NDN data-planes via Event-B machines," *IEEE Access*, vol. 10, pp. 29 668–29 686, 2022

- O. Karrakchou, N. Samaan, and A. Karmouch, "FCTrees: A front-coded family of compressed tree-based FIB structures for NDN routers," *IEEE Trans. Netw. Service Manag.*, vol. 17, no. 2, pp. 1167–1180, Jun. 2020

Conference Proceedings (with oral presentation)

- O. Karrakchou, N. Samaan, and A. Karmouch, "EP4: An application-aware network architecture with a customizable data plane," in *Proc. IEEE 22nd Int. Conf. on High Performance Switching and Routing (HPSR)*, 2021, pp. 1–6

- O. Karrakchou, N. Samaan, and A. Karmouch, "ENDN: An enhanced NDN architecture with a P4-programmabIe data plane," in *Proc. 7th ACM Conf. Information-Centric Networking (ICN)*, 2020, pp. 1–11

- O. Karrakchou, N. Samaan, and A. Karmouch, "FCTree: A space efficient FIB data structure for NDN routers," in *Proc. IEEE 43rd Conf. Local Computer Networks*

(LCN), 2018, pp. 589–596

- O. Karrakchou, N. Samaan, and A. Karmouch, "A survey of routing mechanisms in ICNs," in *Proc. 6th Int. Conf. Multimedia Computing and Systems (ICMCS)*, 2018, pp. 1–6

Poster Presentation (invited)

- O. Karrakchou, N. Samaan, and A. Karmouch, "A novel P4 target architecture for runtime-reconfigurable NDN data planes," in *NDN Community Meeting*, NIST, 2020

Bibliography

[1] O. Karrakchou, N. Samaan, and A. Karmouch, "A survey of routing mechanisms in ICNs," in *Proc. 6th Int. Conf. Multimedia Computing and Systems (ICMCS)*, 2018, pp. 1–6.

[2] O. Karrakchou, N. Samaan, and A. Karmouch, "FCTree: A space efficient FIB data structure for NDN routers," in *Proc. IEEE 43rd Conf. Local Computer Networks (LCN)*, 2018, pp. 589–596.

[3] O. Karrakchou, N. Samaan, and A. Karmouch, "FCTrees: A front-coded family of compressed tree-based FIB structures for NDN routers," *IEEE Trans. Netw. Service Manag.*, vol. 17, no. 2, pp. 1167–1180, Jun. 2020.

[4] O. Karrakchou, N. Samaan, and A. Karmouch, "EP4: An application-aware network architecture with a customizable data plane," in *Proc. IEEE 22nd Int. Conf. on High Performance Switching and Routing (HPSR)*, 2021, pp. 1–6.

[5] O. Karrakchou, N. Samaan, and A. Karmouch, "ENDN: An enhanced NDN architecture with a P4-programmabIe data plane," in *Proc. 7th ACM Conf. Information-Centric Networking (ICN)*, 2020, pp. 1–11.

[6] O. Karrakchou, N. Samaan, and A. Karmouch, "Mapping applications intents to programmable NDN data-planes via Event-B machines," *IEEE Access*, vol. 10, pp. 29 668–29 686, 2022.

[7] "NDN Packet Format Specification version 0.3." [Online]. Available: https://named-data.net/doc/NDN-packet-spec/current/ [Accessed: 2021-07-05]

[8] L. Zhang, A. Afanasyev, J. Burke, V. Jacobson, k. claffy, P. Crowley, C. Papadopoulos, L. Wang, and B. Zhang, "Named data networking," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 66–73, Jul. 2014.

[9] L. Wang, A. K. M. M. Hoque, C. Yi, A. Alyyan, and B. Zhang, "OSPFN: An OSPF based routing protocol for named data networking," NDN, Tech. Rep. NDN-0003, 2012.

[10] T. Koponen, M. Chawla, B.-G. Chun, A. Ermolinskiy, K. H. Kim, S. Shenker, and I. Stoica, "A data-oriented (and beyond) network architecture," in *Proc. ACM SIGCOMM Conf.*, 2007, pp. 181–192.

[11] D. Kreutz, F. M. V. Ramos, P. E. Veríssimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-defined networking: A comprehensive survey," *Proc. IEEE*, vol. 103, no. 1, pp. 14–76, Jan. 2015.

[12] "P4˜16˜ Portable Switch Architecture (PSA)." [Online]. Available: https://p4.org/p4-spec/docs/PSA.html [Accessed: 2020-12-10]

[13] L. Lambert, "Lawrence Roberts (1937–)," in *The Internet: A Historical Encyclopedia : Biographies. Vol. 1.* Santa Barbara, CA, USA: ABC-CLIO, 2005, p. 204.

[14] G. Maier, A. Feldmann, V. Paxson, and M. Allman, "On dominant characteristics of residential broadband internet traffic," in *Proc. 9th ACM SIGCOMM Conf. Internet Measurement (IMC)*, 2009, pp. 90–102.

[15] B. Ahlgren, C. Dannewitz, C. Imbrenda, D. Kutscher, and B. Ohlman, "A survey of information-centric networking," *IEEE Commun. Mag.*, vol. 50, no. 7, pp. 26–36, Jul. 2012.

[16] M. F. Bari, S. R. Chowdhury, R. Ahmed, R. Boutaba, and B. Mathieu, "A survey of naming and routing in information-centric networks," *IEEE Commun. Mag.*, vol. 50, no. 12, pp. 44–53, Dec. 2012.

[17] P. Bosshart, G. Varghese, D. Walker, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, and A. Vahdat, "P4: Programming protocol-independent packet processors," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 87–95, Jul. 2014.

[18] O. Karrakchou, N. Samaan, and A. Karmouch, "A novel P4 target architecture for runtime-reconfigurable NDN data planes," in *NDN Community Meeting*, NIST, 2020.

[19] A. Clemm, L. Ciavaglia, L. Z. Granville, and J. Tantsura, "Intent-Based Networking - Concepts and Definitions." [Online]. Available: https://datatracker.ietf.org/doc/html/draft-irtf-nmrg-ibn-concepts-definitions-05 [Accessed: 2020-12-09]

[20] V. Jacobson, D. K. Smetters, J. D. Thornton, M. F. Plass, N. H. Briggs, and R. L. Braynard, "Networking named content," in *Proc. 5th Int. Conf. Emerging Networking Experiments and Technologies (CoNEXT)*, 2009, pp. 1–12.

[21] C. Yi, A. Afanasyev, L. Wang, B. Zhang, and L. Zhang, "Adaptive forwarding in named data networking," *SIGCOMM Comput. Commun. Rev.*, vol. 42, no. 3, pp. 62–67, Jun. 2012.

[22] N. Aloulou, M. Ayari, M. F. Zhani, and L. Saidane, "A popularity-driven controller-based routing and cooperative caching for named data networks," in *Proc. 6th Int. Conf. Network of the Future (NOF)*, 2015, pp. 1–5.

[23] M. Yu, R. Li, Y. Liu, and Y. Li, "A caching strategy based on content popularity and router level for NDN," in *Proc. 7th IEEE Int. Conf. on Electronics Information and Emergency Communication (ICEIEC)*, 2017, pp. 195–198.

[24] J. Garcia-Luna-Aceves, "A fault-tolerant forwarding strategy for interest-based information centric networks," in *Proc. IFIP Networking Conf. (IFIP Networking)*, 2015, pp. 1–9.

[25] H. Qian, R. Ravindran, G.-Q. Wang, and D. Medhi, "Probability-based adaptive forwarding strategy in named data networking," in *Proc. IFIP/IEEE Int. Symp. Integrated Network Management (IM)*, 2013, pp. 1094–1101.

[26] M. Mosko, I. Solis, and E. Uzun, "CCN 1.0 protocol architecture," Palo Alto Research Center, Tech. Rep., 2015.

[27] S. Traverso, M. Ahmed, M. Garetto, P. Giaccone, E. Leonardi, and S. Niccolini, "Temporal locality in today's content caching: Why it matters and how to model it," *SIGCOMM Comput. Commun. Rev.*, vol. 43, no. 5, pp. 5–12, Nov. 2013.

[28] L. Wang, S. Bayhan, J. Ott, J. Kangasharju, A. Sathiaseelan, and J. Crowcroft, "Pro-Diluvian: Understanding scoped-flooding for content discovery in information-centric networking," in *Proc. 2nd ACM Conf. Information-Centric Networking (ICN)*, 2015, pp. 9–18.

[29] "RIP Version 2 (RFC2453)." [Online]. Available: https://datatracker.ietf.org/doc/html/rfc2453 [Accessed: 2021-07-05]

[30] "OSPF Version 2 (RFC2328)." [Online]. Available: https://datatracker.ietf.org/doc/html/rfc2328 [Accessed: 2021-07-05]

[31] A. K. M. M. Hoque, S. O. Amin, A. Alyyan, B. Zhang, L. Zhang, and L. Wang, "NLSR: Named-data link state routing protocol," in *Proc. 3rd ACM SIGCOMM Workshop Information-centric Networking (ICN)*, 2013, pp. 15–20.

[32] Z. Zhu and A. Afanasyev, "Let's ChronoSync: Decentralized dataset state synchronization in named data networking," in *Proc. 21st IEEE Int. Conf. Network Protocols (ICNP)*, 2013, pp. 1–10.

[33] K. Schneider and B. Zhang, "How to establish loop-free multipath routes in named data networking," NDN, Tech. Rep. NDN-0044, 2017.

[34] C. Muñoz, L. Wang, E. Solana, and J. Crowcroft, "I(FIB)F: Iterated Bloom filters for routing in named data networks," in *Proc. Int. Conf. Networked Systems (NetSys)*, 2017, pp. 1–8.

[35] "BGP-4 (RFC4271)." [Online]. Available: https://datatracker.ietf.org/doc/html/rfc4271 [Accessed: 2021-07-05]

[36] "AS65000 - BGP Table Statistics." [Online]. Available: http://bgp.potaroo.net/as1221/ [Accessed: 2018-01-07]

[37] "We knew the web was big..." [Online]. Available: https://googleblog.blogspot.com/2008/07/we-knew-web-was-big.html [Accessed: 2018-01-07]

[38] K. V. Katsaros, W. K. Chai, and G. Pavlou, "Bloom filter based inter-domain name resolution: A feasibility study," in *Proc. 2nd ACM Conf. Information-Centric Networking (ICN)*, 2015, pp. 39–48.

[39] R. Ahmed, F. Bari, S. R. Chowdhury, G. Rabbani, R. Boutaba, B. Mathieu, R. Ahmed, F. Bari, S. R. Chowdhury, G. Rabbani, R. Boutaba, and B. Mathieu, "αRoute: routing on names," *IEEE/ACM Trans. Netw.*, vol. 24, no. 5, pp. 3070–3083, Oct. 2016.

[40] P. K. Dey and M. Yuksel, "An economic analysis of cloud-assisted routing for wider area SDN," *IEEE Trans. Netw. Service Manag.*, vol. 17, no. 1, pp. 445–458, Mar. 2020.

[41] S. Li, K. Han, N. Ansari, Q. Bao, D. Hu, J. Liu, S. Yu, and Z. Zhu, "Improving SDN scalability with protocol-oblivious source routing: A system-level study," *IEEE Trans. Netw. Service Manag.*, vol. 15, no. 1, pp. 275–288, Mar. 2018.

[42] M. Nikkhah and R. Guérin, "Migrating the Internet to IPv6: An exploration of the when and why," *IEEE/ACM Trans. Netw.*, vol. 24, no. 4, pp. 2291–2304, Aug. 2016.

[43] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling innovation in campus networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, Mar. 2008.

[44] "OpenFlow switch specification version 1.3.0," Open Networking Foundation, ONF TS-006, Jun. 2012.

[45] "Open Network Operating System (ONOS) SDN Controller for SDN/NFV Solutions." [Online]. Available: https://opennetworking.org/onos/ [Accessed: 2021-06-23]

[46] "OpenDaylight, A Linux Foundation Collaborative Project." [Online]. Available: https://www.opendaylight.org/ [Accessed: 2021-06-23]

[47] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker, "NOX: Towards an operating system for networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 3, pp. 105–110, Jul. 2008.

[48] R. Sherwood, G. Gibb, K.-K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. Parulkar, "Flowvisor: A network virtualization layer," *OpenFlow Switch Consortium, Tech. Rep*, vol. 1, p. 132, 2009.

[49] X. Jin, J. Gossels, J. Rexford, and D. Walker, "CoVisor: A compositional hypervisor for software-defined networks," in *Proc. 12th USENIX Symp. on Networked Systems Design and Implementation (NSDI 15)*, 2015, pp. 87–101.

[50] M. S. Seddiki, M. Shahbaz, S. Donovan, S. Grover, M. Park, N. Feamster, and Y.-Q. Song, "FlowQoS: QoS for the rest of us," in *Proc. 3rd Workshop on Hot Topics in Software Defined Networking (HotSDN)*, 2014, pp. 207–208.

[51] M. Qilin and S. Weikang, "A load balancing method based on SDN," in *Proc. 7th Int. Conf. on Measuring Technology and Mechatronics Automation (ICMTMA)*, 2015, pp. 18–21.

[52] C. Yu, C. Lumezanu, Y. Zhang, V. Singh, G. Jiang, and H. V. Madhyastha, "FlowSense: Monitoring network utilization with zero measurement cost," in *Proc. Passive and Active Measurement*, ser. Lecture Notes in Computer Science, M. Roughan and R. Chang, Eds., 2013, pp. 31–41.

[53] N. L. M. van Adrichem, C. Doerr, and F. A. Kuipers, "OpenNetMon: Network monitoring in OpenFlow software-defined networks," in *Proc. IEEE Network Operations and Management Symp. (NOMS)*, 2014, pp. 1–8.

[54] Y. Liu and H. Wadekar, "SDAR: Software defined intra-domain routing in named data networks," in *Proc. IEEE 15th Int. Symp. Network Computing and Applications (NCA)*, 2016, pp. 158–161.

[55] E. Aubry, T. Silverston, and I. Chrisment, "SRSC: SDN-based routing scheme for CCN," in *Proc. 1st IEEE Conf. Network Softwarization (NetSoft)*, 2015, pp. 1–5.

[56] R. Jmal and L. C. Fourati, "An OpenFlow architecture for managing content-centric-network (OFAM-CCN) based on popularity caching strategy," *Comput. Stand. Inter.*, vol. 51, pp. 22–29, Mar. 2017.

[57] J. Cao, D. Pei, X. Zhang, B. Zhang, and Y. Zhao, "Fetching popular data from the nearest replica in NDN," in *Proc. 25th Int. Conf. Computer Communication and Networks (ICCCN)*, 2016, pp. 1–9.

[58] "P4Runtime." [Online]. Available: https://p4.org/p4runtime/spec/v1.3.0/P4Runtime-Spec.html [Accessed: 2021-01-28]

[59] "P4˜16˜ Language Specification." [Online]. Available: https://p4.org/p4-spec/docs/P4-16-v1.2.0.html [Accessed: 2020-05-25]

[60] S. Ibanez, G. Brebner, N. McKeown, and N. Zilberman, "The P4->NetFPGA workflow for line-rate packet processing," in *Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays (FPGA)*, 2019, pp. 1–9.

[61] "P4lang/behavioral-model." [Online]. Available: https://github.com/p4lang/behavioral-model [Accessed: 2020-05-25]

[62] L. Chen, O. Havel, A. Olariu, P. Martinez-Julia, J. Nobre, and D. Lopez, "Intent Classification." [Online]. Available: https://datatracker.ietf.org/doc/html/draft-irtf-nmrg-ibn-intent-classification-05 [Accessed: 2021-11-23]

[63] Y. Wang, R. Forbes, U. Elzur, J. Strassner, A. Gamelas, H. Wang, S. Liu, L. Pesando, X. Yuan, and S. Cai, "From design to practice: ETSI ENI reference architecture and instantiation for network management and orchestration using artificial intelligence," *IEEE Commun. Stand. Mag.*, vol. 4, no. 3, pp. 38–45, Sep. 2020.

[64] S. Perez-Soler, S. Juarez-Puerta, E. Guerra, and J. de Lara, "Choosing a chatbot development tool," *IEEE Software*, vol. 38, no. 4, pp. 94–103, Jul. 2021.

[65] L. Pang, C. Yang, D. Chen, Y. Song, and M. Guizani, "A survey on intent-driven networks," *IEEE Access*, vol. 8, pp. 22 862–22 873, 2020.

[66] X. Li, N. Samaan, and A. Karmouch, "An automated VNF manager based on parameterized action MDP and reinforcement learning," in *Proc. IEEE Int. Conf. Communications (ICC)*, 2021, pp. 1–6.

[67] W. So, A. Narayanan, and D. Oran, "Named data networking on a router: Fast and DoS-resistant forwarding with hash tables," in *Proc. ACM/IEEE Symp. Architectures Networking and Communications Systems (ANCS)*, 2013, pp. 215–225.

[68] Z. Li, Y. Xu, B. Zhang, L. Yan, and K. Liu, "Packet forwarding in named data networking requirements and survey of solutions," *IEEE Commun. Surveys Tuts.*, vol. 21, no. 2, pp. 1950–1987, 2nd Quart. 2019.

[69] Sung-Won Lee, Dabin Kim, Young-Bae Ko, Jae-Hoon Kim, and Myeong-Wuk Jang, "Cache capacity-aware CCN: Selective caching and cache-aware routing," in *Proc. IEEE Global Communications Conf. (GLOBECOM)*, 2013, pp. 2114–2119.

[70] A. Afanasyev, J. Shi, B. Zhang, L. Zhang, I. Moiseenko, Y. Yu, W. Shang, Y. Huang, J. P. Abraham, and S. DiBenedetto, "NFD developer's guide," NDN, Tech. Rep. NDN-0021, 2018.

[71] Y. Wang, T. Pan, Z. Mi, H. Dai, X. Guo, T. Zhang, B. Liu, and Q. Dong, "NameFilter: Achieving fast name lookup with low memory cost via applying two-stage Bloom filters," in *Proc. IEEE Conf. Computer Communications (INFOCOM)*, 2013, pp. 95–99.

[72] Y. Wang, K. He, H. Dai, W. Meng, J. Jiang, B. Liu, and Y. Chen, "Scalable name lookup in NDN using effective name component encoding," in *Proc. IEEE 32nd Int. Conf. Distributed Computing Systems (ICDCS)*, 2012, pp. 688–697.

[73] M. Liu, T. Song, Y. Yang, and B. Zhang, "A unified data structure of name lookup for NDN data plane," in *Proc. 4th ACM Conf. Information-Centric Networking (ICN)*, ser. ICN '17, 2017, pp. 188–189.

[74] V. Sourlas, O. Ascigil, I. Psaras, and G. Pavlou, "Enhancing information resilience in disruptive information-centric networks," *IEEE Trans. Netw. Service Manag.*, vol. 15, no. 2, pp. 746–760, Jun. 2018.

[75] H. Asai and Y. Ohara, "Poptrie: A compressed trie with population count for fast and scalable software IP routing table lookup," in *Proc. ACM SIGCOMM Conf.*, 2015, pp. 57–70.

[76] M. Bando, Y.-L. Lin, and H. J. Chao, "FlashTrie: Beyond 100-Gb/s IP route lookup using hash-based prefix-compressed trie," *IEEE/ACM Trans. Netw.*, vol. 20, no. 4, pp. 1262–1275, Aug. 2012.

[77] K. Matsuzono and H. Asaeda, "NRTS: Content name-based real-time streaming," in *Proc. 13th IEEE Annu. Consumer Communications & Networking Conf. (CCNC)*, 2016, pp. 537–543.

[78] P. Gusev and J. Burke, "NDN-RTC: Real-time videoconferencing over named data networking," in *Proc. 2nd ACM Conf. Information-Centric Networking (ICN)*, 2015, pp. 117–126.

[79] M. Caria, A. Jukan, and M. Hoffmann, "SDN partitioning: A centralized control plane for distributed routing protocols," *IEEE Trans. Netw. Service Manag.*, vol. 13, no. 3, pp. 381–393, Sep. 2016.

[80] H. Ben Abraham and P. Crowley, "Controlling strategy retransmissions in named data networking," in *Proc. ACM/IEEE Symp. Architectures Networking and Communications Systems (ANCS)*, 2017, pp. 70–81.

[81] H. Dai, J. Lu, Y. Wang, T. Pan, and B. Liu, "BFAST: High-speed and memory-efficient approach for NDN forwarding engine," *IEEE/ACM Trans. Netw.*, vol. 25, no. 2, pp. 1235–1248, Apr. 2017.

[82] A. Azgin, R. Ravindran, and G. Wang, "H2N4: Packet forwarding on hierarchical hash-based names for content centric networks," in *Proc. IEEE Int. Conf. Communications (ICC)*, 2015, pp. 5639–5645.

[83] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, Jul. 1970.

[84] K. Christensen, A. Roginsky, and M. Jimeno, "A new analysis of the false positive rate of a Bloom filter," *Inf. Process. Lett.*, vol. 110, no. 21, pp. 944–949, Oct. 2010.

[85] C. Ghasemi, H. Yousefi, K. G. Shin, and B. Zhang, "A fast and memory-efficient trie structure for name-based packet forwarding," in *Proc. IEEE 26th Int. Conf. Network Protocols (ICNP)*, 2018, pp. 302–312.

[86] C. Dillabaugh, M. He, and A. Maheshwari, "Succinct and I/O efficient data structures for traversal in trees," *Algorithmica*, vol. 63, no. 1, pp. 201–223, Jun. 2012.

[87] W. Quan, C. Xu, J. Guan, H. Zhang, and L. A. Grieco, "Scalable name lookup with adaptive prefix Bloom filter for named data networking," *IEEE Commun. Lett.*, vol. 18, no. 1, pp. 102–105, Jan. 2014.

[88] O. Ascigil, S. Reñé, G. Xylomenos, I. Psaras, and G. Pavlou, "A keyword-based ICN-IoT platform," in *Proc. 4th ACM Conf. Information-Centric Networking (ICN)*, 2017, pp. 22–28.

[89] C. Tsilopoulos, G. Xylomenos, and Y. Thomas, "Reducing forwarding state in content-centric networks with semi-stateless forwarding," in *Proc. IEEE Conf. Computer Communications (INFOCOM)*, 2014, pp. 2067–2075.

[90] X. Wang, W. Wang, C. Zeng, R. Dai, S. Wang, and S. Xu, "Reducing the size of pending interest table for content-centric networks with hybrid forwarding," in *Proc. IEEE Int. Conf. Communications (ICC)*, 2016, pp. 1–6.

[91] H. Yuan and P. Crowley, "Scalable pending interest table design: From principles to practice," in *Proc. IEEE Conf. Computer Communications (INFOCOM)*, 2014, pp. 2049–2057.

[92] Q. Wu, Z. Li, J. Zhou, H. Jiang, Z. Hu, Y. Liu, and G. Xie, "SOFIA: Toward service-oriented information centric networking," *IEEE Netw.*, vol. 28, no. 3, pp. 12–18, May 2014.

[93] I. H. Witten, A. Moffat, and T. C. Bell, *Managing Gigabytes: Compressing and Indexing Documents and Images*, 2nd ed. San Francisco, CA, USA: Morgan Kaufmann Publishers, 1999.

[94] H. E. Williams and J. Zobel, "Compressing integers for fast file access," *Comput. J.*, vol. 42, no. 3, pp. 193–201, Jan. 1999.

[95] D. Benoit, E. D. Demaine, J. I. Munro, R. Raman, V. Raman, and S. S. Rao, "Representing trees of higher degree," *Algorithmica*, vol. 43, no. 4, pp. 275–292, Dec. 2005.

[96] K. Suksomboon, N. Matsumoto, S. Okamoto, M. Hayashi, and Y. Ji, "Configuring a software router by the Erlang-k-based packet latency prediction," *IEEE J. Select. Areas Commun.*, vol. 36, no. 3, pp. 422–437, Mar. 2018.

[97] B. S. Majewski, N. C. Wormald, G. Havas, and Z. J. Czech, "A family of perfect hashing methods," *Comput. J.*, vol. 39, no. 6, pp. 547–554, Jan. 1996.

[98] B. Pfaff, "Performance analysis of BSTs in system software," in *Proc. Joint Int. Conf. Measurement and Modeling of Computer Systems (SIGMETRICS/Performance)*, 2004, pp. 410–411.

[99] S. Hanke, "The performance of concurrent red-black tree algorithms," in *Proc. Int. Workshop Algorithm Engineering*, 1999, pp. 286–300.

[100] K. Matsuzono, H. Asaeda, and T. Turletti, "Low latency low loss streaming using in-network coding and caching," in *Proc. IEEE Conf. Computer Communications (INFOCOM)*, 2017, pp. 1–9.

[101] X. Qiao, G. Nan, Y. Peng, L. Guo, J. Chen, Y. Sun, and J. Chen, "NDNBrowser: An extended web browser for named data networking," *J. Netw. Comput. Appl.*, vol. 50, pp. 134–147, Apr. 2015.

[102] R. S. Antunes, M. B. Lehmann, R. B. Mansilha, L. P. Gaspary, and M. P. Barcellos, "NDNrel: A mechanism based on relations among objects to improve the performance of NDN," *J. Netw. Comput. Appl.*, vol. 87, pp. 73–86, Jun. 2017.

[103] E. Hemmati and J. Garcia-Luna-Aceves, "A comparison of name-based content routing protocols," in *Proc. IEEE 12th Int. Conf. Mobile Ad Hoc and Sensor Systems (MASS)*, 2015, pp. 537–542.

[104] T. Hu and A. C. Tucker, "Optimal computer search trees and variable-length alphabetical codes," *SIAM J. Appl. Math.*, vol. 21, no. 4, pp. 514–532, Dec. 1971.

[105] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Trans. Inf. Theory*, vol. 23, no. 3, pp. 337–343, May 1977.

[106] J. Ziv and A. Lempel, "Compression of individual sequences via variable-rate coding," *IEEE Trans. Inf. Theory*, vol. 24, no. 5, pp. 530–536, Sep. 1978.

[107] S. Mastorakis, A. Afanasyev, I. Moiseenko, and L. Zhang, "ndnSIM 2: An updated NDN simulator for NS-3," NDN, Tech. Rep. NDN-0028, 2016.

[108] "DMOZ." [Online]. Available: https://dmoztools.net/ [Accessed: 2018-04-16]

[109] "Top 10 million websites based on Open data from Common Crawl & Common Search." [Online]. Available: https://www.domcop.com/top-10-million-websites [Accessed: 2019-01-11]

[110] S. Mastorakis, A. Afanasyev, and L. Zhang, "On the evolution of ndnSIM: An open-source simulator for NDN experimentation," *SIGCOMM Comput. Commun. Rev.*, vol. 47, no. 3, pp. 19–33, Sep. 2017.

[111] M. Amadeo, C. Campolo, and A. Molinaro, "Internet of things via named data networking: The support of push traffic," in *Proc. Int. Conf. and Workshop Network of the Future (NOF)*, 2014, pp. 1–5.

[112] V. Jacobson, D. K. Smetters, N. H. Briggs, M. F. Plass, P. Stewart, J. D. Thornton, and R. L. Braynard, "VoCCN: Voice-over content-centric networks," in *Proc. Workshop Re-Architecting the Internet (ReArch)*, 2009, pp. 1–6.

[113] G. Ravikumar, D. Ameme, S. Misra, S. Brahma, and R. Tourani, "iCASM: An information-centric network architecture for wide area measurement systems," *IEEE Trans. Smart Grid*, vol. 11, no. 4, pp. 3418–3427, Jul. 2020.

[114] V. Varadharajan, K. Karmakar, U. Tupakula, and M. Hitchens, "A policy-based security architecture for software-defined networks," *IEEE Trans. Inf. Forensics Security*, vol. 14, no. 4, pp. 897–912, Apr. 2019.

[115] Y.-J. Chen, L.-C. Wang, M.-C. Chen, P.-M. Huang, and P.-J. Chung, "SDN-enabled traffic-aware load balancing for M2M networks," *IEEE Internet Things J.*, vol. 5, no. 3, pp. 1797–1806, Jun. 2018.

[116] S. Tomovic and I. Radusinovic, "Toward a scalable, robust, and QoS-aware virtual-link provisioning in SDN-based ISP networks," *IEEE Trans. Netw. Service Manag.*, vol. 16, no. 3, pp. 1032–1045, Sep. 2019.

[117] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee, "DevoFlow: Scaling flow management for high-performance networks," in *Proc. ACM SIGCOMM Conf.*, 2011, pp. 254–265.

[118] S. R. Chowdhury, M. F. Bari, R. Ahmed, and R. Boutaba, "PayLess: A low cost network monitoring framework for software defined networks," in *Proc. IEEE Network Operations and Management Symp. (NOMS)*, May 2014, pp. 1–9.

[119] R. Bifulco and F. Schneider, "OpenFlow rules interactions: Definition and detection," in *Proc. IEEE SDN Future Networks and Services (SDN4FNS)*, 2013, pp. 1–6.

[120] F. Paolucci, F. Civerchia, A. Sgambelluri, A. Giorgetti, F. Cugini, and P. Castoldi, "P4 edge node enabling stateful traffic engineering and cyber security," *J. Opt. Commun. Netw.*, vol. 11, no. 1, p. A84, Jan. 2019.

[121] G.-R. Wang, C. Chen, C.-W. Chen, L.-W. Pan, Y.-R. Wang, C.-L. Fan, and C.-H. Hsu, "Streaming scalable video sequences with media-aware network elements implemented in P4 programming language," in *Proc. IEEE/IFIP Network Operations and Management Symp. (NOMS)*, 2018, pp. 1–2.

[122] L. Castanheira, R. Parizotto, and A. E. Schaeffer-Filho, "FlowStalker: Comprehensive traffic flow monitoring on the data plane using P4," in *Proc. IEEE Int. Conf. Communications (ICC)*, 2019, pp. 1–6.

[123] C. Tsilopoulos and G. Xylomenos, "Supporting diverse traffic types in information centric networks," in *Proc. ACM SIGCOMM Workshop Information-Centric Networking (ICN)*, 2011, pp. 13–18.

[124] C. Ghali, G. Tsudik, E. Uzun, and C. A. Wood, "Closing the floodgate with stateless content-centric networking," in *Proc. 26th Int. Conf. Computer Communication and Networks (ICCCN)*, 2017, pp. 1–10.

[125] K. Lei, J. Wang, and J. Yuan, "An entropy-based probabilistic forwarding strategy in named data networking," in *Proc. IEEE Int. Conf. Communications (ICC)*, 2015, pp. 5665–5671.

[126] D. Posch, B. Rainer, and H. Hellwagner, "SAF: Stochastic adaptive forwarding in named data networking," *IEEE/ACM Transactions on Networking*, vol. 25, no. 2, pp. 1089–1102, Apr. 2017.

[127] H. Ben Abraham, J. Parwatikar, J. DeHart, A. Drescher, and P. Crowley, "Decoupling information and connectivity via information-centric transport," in *Proc. 5th ACM Conf. Information-Centric Networking (ICN)*, 2018, pp. 54–66.

[128] Q.-Y. Zhang, X.-W. Wang, M. Huang, K.-Q. Li, and S. K. Das, "Software defined networking meets information centric networking: A survey," *IEEE Access*, vol. 6, pp. 39 547–39 563, 2018.

[129] E. Aubry, T. Silverston, and I. Chrisment, "Implementation and evaluation of a controller-based forwarding scheme for NDN," in *Proc. IEEE 31st Int. Conf. Advanced Information Networking and Applications (AINA)*, 2017, pp. 144–151.

[130] J. V. Torres, I. D. Alvarenga, R. Boutaba, and O. C. M. B. Duarte, "An autonomous and efficient controller-based routing scheme for networking Named-Data mobility," *Comput. Commun.*, vol. 103, pp. 94–103, May 2017.

[131] Q. Sun, W. Wendong, Y. Hu, X. Que, and G. Xiangyang, "SDN-based autonomic CCN traffic management," in *Proc. IEEE Globecom Workshops (GC Wkshps)*, 2014, pp. 183–187.

[132] S. Signorello, R. State, J. François, and O. Festor, "NDN.p4: Programming information-centric data-planes," in *Proc. IEEE NetSoft Conf. and Workshops (NetSoft)*, 2016, pp. 384–389.

[133] J. Krude, J. Hofmann, M. Eichholz, K. Wehrle, A. Koch, and M. Mezini, "Online reprogrammable multi tenant switches," in *Proc. 1st ACM CoNEXT Workshop on Emerging in-Network Computing Paradigms (ENCP)*, 2019, pp. 1–8.

[134] "NetFPGA." [Online]. Available: https://netfpga.org/NetFPGA-SUME.html [Accessed: 2020-08-13]

[135] FG-NET-2030, "Network 2030." [Online]. Available: https://www.itu.int/en/ITU-T/focusgroups/net2030/Documents/White_Paper.pdf [Accessed: 2020-12-13]

[136] R. Glebke, J. Krude, I. Kunze, J. Rüth, F. Senger, and K. Wehrle, "Towards executing computer vision functionality on programmable network devices," in *Proc. 1st ACM CoNEXT Workshop on Emerging in-Network Computing Paradigms (ENCP)*, 2019, pp. 15–20.

[137] E. O. Zaballa, D. Franco, Z. Zhou, and M. S. Berger, "P4Knocking: Offloading host-based firewall functionalities to the network," in *Proc. 23rd Conf. on Innovation in Clouds, Internet and Networks and Workshops (ICIN)*, 2020, pp. 7–12.

[138] Â. C. Lapolli, J. A. Marques, and L. P. Gaspary, "Offloading real-time DDoS attack detection to programmable data planes," in *Proc. IFIP/IEEE Int. Symp. Integrated Network Management (IM)*, 2019, pp. 19–27.

[139] B. Guan and S.-H. Shen, "FlowSpy: An efficient network monitoring framework using P4 in software-defined networks," in *Proc. 90th IEEE Vehicular Technology Conf. (VTC-Fall)*, 2019, pp. 1–5.

[140] R. Kundel, C. Gärtner, M. Luthra, S. Bhowmik, and B. Koldehofe, "Flexible content-based publish/subscribe over programmable data planes," in *Proc. IEEE/IFIP Network Operations and Management Symp. (NOMS)*, 2020, pp. 1–5.

[141] C. Wernecke, H. Parzyjegla, G. Mühl, P. Danielis, and D. Timmermann, "Realizing content-based publish/subscribe with P4," in *Proc. IEEE Conf. Network Function Virtualization and Software Defined Networks (NFV-SDN)*, 2018, pp. 1–7.

[142] M. Tirmazi, R. Ben Basat, J. Gao, and M. Yu, "Cheetah: Accelerating database queries with switch pruning," in *Proc. ACM SIGMOD Conf.*, 2020, pp. 2407–2422.

[143] E. F. Kfoury, J. Crichigno, and E. Bou-Harb, "Offloading media traffic to programmable data plane switches," in *Proc. IEEE Int. Conf. Communications (ICC)*, 2020, pp. 1–7.

[144] B. Nour, S. Mastorakis, and A. Mtibaa, "Compute-less networking: Perspectives, challenges, and opportunities," *IEEE Network*, vol. 34, no. 6, pp. 259–265, Nov. 2020.

[145] R. Stoyanov and N. Zilberman, "MTPSA: Multi-tenant programmable switches," in *Proc. 3rd P4 Workshop in Europe (EuroP4)*, 2020, pp. 43–48.

[146] G. C. Sankaran and K. M. Sivalingam, "Collaborative packet header parsing in NetFPGA-based high speed switches," *IEEE Networking Letters*, vol. 2, no. 3, pp. 124–127, Sep. 2020.

[147] N. Gebara, A. Lerner, M. Yang, M. Yu, P. Costa, and M. Ghobadi, "Challenging the stateless quo of programmable switches," in *Proc. 19th ACM Workshop on Hot Topics in Networks (HotNets)*, 2020, pp. 153–159.

[148] P. Zheng, T. A. Benson, and C. Hu, "Building and testing modular programs for programmable data planes," *IEEE J. Sel. Areas Commun.*, vol. 38, no. 7, pp. 1432–1447, Jul. 2020.

[149] C. Zhang, J. Bi, Y. Zhou, and J. Wu, "HyperVDP: High-performance virtualization of the programmable data plane," *IEEE J. Sel. Areas Commun.*, vol. 37, no. 3, pp. 556–569, Mar. 2019.

[150] D. Wu, A. Chen, T. S. E. Ng, G. Wang, and H. Wang, "Accelerated service chaining on a single switch ASIC," in *Proc. 18th ACM Workshop on Hot Topics in Networks (HotNets)*, 2019, pp. 141–149.

[151] H. Soni, M. Rifai, P. Kumar, R. Doenges, and N. Foster, "Composing dataplane programs with μP4," in *Proc. ACM SIGCOMM Conf.*, 2020, pp. 329–343.

[152] M. Saquetti, G. Bueno, W. Cordeiro, and J. R. Azambuja, "P4VBox: Enabling P4-based switch virtualization," *IEEE Commun. Lett.*, vol. 24, no. 1, pp. 146–149, Jan. 2020.

[153] M. Riftadi and F. Kuipers, "P4I/O: Intent-based networking with P4," in *Proc. IEEE Conf. Network Softwarization (NetSoft)*, 2019, pp. 438–443.

[154] S. Previdi, R. Shakir, C. Filsfils, B. Decraene, L. Ginsberg, and S. Litkowski, "Segment Routing Architecture (RFC8402)." [Online]. Available: https://tools.ietf.org/html/rfc8402 [Accessed: 2020-12-14]

[155] "Ns-3." [Online]. Available: https://www.nsnam.org/ [Accessed: 2020-12-09]

[156] E. Blanton and M. Allman, "TCP Congestion Control (RFC5681)." [Online]. Available: https://tools.ietf.org/html/rfc5681 [Accessed: 2020-12-09]

[157] E. Zeydan and Y. Turk, "Recent advances in intent-based networking: A survey," in *Proc. IEEE 91st Vehicular Technology Conf. (VTC-Spring)*, 2020, pp. 1–5.

[158] N. Anerousis, P. Chemouil, A. A. Lazar, N. Mihai, and S. B. Weinstein, "The origin and evolution of open programmable networks and SDN," *IEEE Commun. Surveys Tuts.*, vol. 23, no. 3, pp. 1956–1971, 2021.

[159] S. Alalmaei, Y. Elkhatib, M. Bezahaf, M. Broadbent, and N. Race, "SDN heading north: Towards a declarative intent-based northbound interface," in *Proc. 16th Int. Conf. on Network and Service Management (CNSM)*, 2020, pp. 1–5.

[160] J.-R. Abrial, *Modeling in Event-B: System and Software Engineering.* Cambridge University Press, 2010.

[161] "Event-B.org." [Online]. Available: http://www.event-b.org/install.html [Accessed: 2021-04-22]

[162] M. Paliwal, D. Shrimankar, and O. Tembhurne, "Controllers in SDN: A review report," *IEEE Access*, vol. 6, pp. 36 256–36 270, 2018.

[163] Y. Han, J. Li, D. Hoang, J.-H. Yoo, and J. W.-K. Hong, "An intent-based network virtualization platform for SDN," in *Proc. 12th Int. Conf. on Network and Service Management (CNSM)*, 2016, pp. 353–358.

[164] Y. Tsuzaki and Y. Okabe, "Reactive configuration updating for intent-based networking," in *Proc. Int. Conf. on Information Networking (ICOIN)*, 2017, pp. 97–102.

[165] T. A. Khan, A. Muhammad, K. Abbas, and W.-C. Song, "Intent-based networking platform: An automated approach for policy and configuration of next-generation networks," in *Proc. 36th Annual ACM Symp. on Applied Computing (SAC)*, Mar. 2021, pp. 1921–1930.

[166] V. Heorhiadi, S. Chandrasekaran, M. K. Reiter, and V. Sekar, "Intent-driven composition of resource-management SDN applications," in *Proc. 14th Int. Conf. on Emerging Networking EXperiments and Technologies (CoNEXT)*, 2018, pp. 86–97.

[167] A. Singh, G. S. Aujla, and R. S. Bali, "Intent-based network for data dissemination in software-defined vehicular edge computing," *IEEE Trans. Intell. Transp. Syst.*, pp. 1–9, 2020.

[168] T. Kollar, D. Berry, L. Stuart, K. Owczarzak, T. Chung, L. Mathias, M. Kayser, B. Snow, and S. Matsoukas, "The Alexa meaning representation language," in *Proc. Conf. of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 3 (Industry Papers) (NAACL-HLT)*, 2018, pp. 177–184.

[169] A. de Barcelos Silva, M. M. Gomes, C. A. da Costa, R. da Rosa Righi, J. L. V. Barbosa, G. Pessin, G. De Doncker, and G. Federizzi, "Intelligent personal assistants: A systematic literature review," *Expert Systems with Applications*, vol. 147, p. 113193, Jun. 2020.

[170] Z. Zhang, Z. Zhang, H. Chen, and Z. Zhang, "A joint learning framework with BERT for spoken language understanding," *IEEE Access*, vol. 7, pp. 168 849–168 858, 2019.

[171] K. Abbas, T. A. Khan, M. Afaq, and W.-C. Song, "Network slice lifecycle management for 5G mobile networks: An intent-based networking approach," *IEEE Access*, vol. 9, pp. 80 128–80 146, 2021.

[172] H. E, Z. Zhan, and M. Song, "Table-to-Dialog: Building dialog assistants to chat with people on behalf of you," *IEEE Access*, vol. 8, pp. 102 313–102 320, 2020.

[173] S. Chen and S. Yu, "WAIS: Word attention for joint intent detection and slot filling," in *Proc. AAAI Conf. on Artificial Intelligence*, vol. 33, 2019, pp. 9927–9928.

[174] A. Iliasov, E. Troubitsyna, L. Laibinis, and A. Romanovsky, "Patterns for refinement automation," in *Formal Methods for Components and Objects*, ser. Lecture

Notes in Computer Science, F. S. de Boer, M. M. Bonsangue, S. Hallerstede, and M. Leuschel, Eds. Berlin, Heidelberg: Springer, 2010, pp. 70–88.

[175] R. Silva and M. Butler, "Shared event composition/decomposition in Event-B," in *Formal Methods for Components and Objects*, ser. Lecture Notes in Computer Science, B. K. Aichernig, F. S. de Boer, and M. M. Bonsangue, Eds. Berlin, Heidelberg: Springer, 2012, pp. 122–141.

[176] C. J. Fillmore and C. Baker, *A Frames Approach to Semantic Analysis*. Oxford University Press, Dec. 2009.

[177] O. Sattarov, "Natural Language Processing with DeepPavlov Library and Additional Semantic Features," in *Artificial Intelligence: 5th RAAI Summer School, Dolgoprudny, Russia, July 4–7, 2019, Tutorial Lectures*, ser. Lecture Notes in Computer Science, G. S. Osipov, A. I. Panov, and K. S. Yakovlev, Eds. Cham: Springer International Publishing, 2019, pp. 146–159.

[178] "Abilene Topology." [Online]. Available: https://uit.stanford.edu/service/network/internet2/abilene [Accessed: 2021-11-10]

[179] H. Harkous, M. Jarschel, M. He, R. Priest, and W. Kellerer, "Towards understanding the performance of P4 programmable hardware," in *Proc. ACM/IEEE Symp. on Architectures for Networking and Communications Systems (ANCS)*, 2019, pp. 1–6.

[180] P. Li and Y. Luo, "P4GPU: Accelerate packet processing of a P4 program with a CPU-GPU heterogeneous architecture," in *Proc. ACM/IEEE Symp. Architectures Networking and Communications Systems (ANCS)*, 2016, pp. 125–126.

[181] J. Bourdon, "Size and path length of Patricia tries: Dynamical sources context," *Random Struct. Algorithms*, vol. 19, no. 3-4, pp. 289–315, Oct. 2001.

[182] R. P. Stanley and F. Sergey, *Enumerative Combinatorics: Volume 2*. Cambridge, UK: Cambridge University Press, 1999.

Appendices

Appendix A

Proofs of FC, BFC, and FCTree Space Utilization

A.1 Proof of Proposition 3.1

To calculate the space $\mathcal{C}(FC(\mathcal{R}))$ needed by the front-coding of $\mathcal{R}$, we note that $FC(\mathcal{R})$ stores the same number of characters as the number of nodes $N_{CT}(\mathcal{R})$ in a CT built using the n routes in $\mathcal{R}$. Bourdon [181] derived the asymptotic behavior of the size S_n and average depth D_n of standard tries (STs) constructed from n strings emitted by a probabilistic dynamical source. However, an ST in Bourdon's work is a compressed version of a CT built from the same set $\mathcal{R}$. In an ST, parent nodes of leaf nodes that contain a single child are all aggregated in a unique leaf node (e.g., an ST storing two strings *"abc"* and *"abde"* would have a single aggregated leaf node storing *"de"* and a normal leaf node storing *"c"*). The number of nodes in a CT can thus be divided between the number of internal nodes S_n and the number of characters aggregated in the leaf nodes nA_n of the corresponding ST:

$$N_{CT_{\mathcal{R}}} = S_n + nA_n \tag{A.1}$$

The number of internal nodes in an ST constructed using n strings from a memoryless source with symbol probabilities p_i from an ordered alphabet $\mathcal{A} = \{a_1, a_2, \cdots, a_{|\mathcal{A}|}\}$ of size $|\mathcal{A}|$ is $S_n{=}\frac{n}{H(\mathcal{R})}$, where $H(\mathcal{R}) = \sum_{a_i \in \mathcal{A}} p_i \log_2 \left(\frac{1}{p_i}\right)$ is the Shannon entropy of $\mathcal{R}$ [181].

On the other hand, to calculate the number of characters aggregated in the leaf nodes of an ST, we note that every string r_i stored in the trie ends with a leaf node. As a result, using the average size of a string $avg(||r_i||) = \frac{N}{n}$, where $N = \sum_{i=1}^{n} \|r_i\|$, and the average depth D_n of the ST, we find that the average number of characters aggregated on a leaf node is [181]:

$$A_n = \frac{N}{n} - D_n \approx \frac{N}{n} - \frac{\log_2(n)}{H(\mathcal{R})} - \frac{H_2(\mathcal{R})}{2H(\mathcal{R})^2} - \frac{\gamma}{H(\mathcal{R})} \tag{A.2}$$

where $\gamma \approx 0.577$ is the Euler-Mascheroni constant and $H_2(\mathcal{R}) = \sum_{a_i \in \mathcal{A}} p_i \log_2^2 \left(\frac{1}{p_i}\right)$.

Hence, the total number of characters stored in $FC(\mathcal{R})$ is:

$$N_{CT_{\mathcal{R}}} = S_n + nA_n \approx N - n\left(\frac{\log_2(n) + \gamma - 1}{H(\mathcal{R})} + \frac{H_2(\mathcal{R})}{2H(\mathcal{R})^2}\right) \tag{A.3}$$

Since each character can be stored in $\log_2 |\mathcal{A}|$ bits, the total space occupied by the front-coded strings in $FC(\mathcal{R})$ is:

$$N_{CT_{\mathcal{R}}} \log_2 |\mathcal{A}| = (N - n\phi_{\mathcal{R}}) \log_2 |\mathcal{A}| \tag{A.4}$$

where $\phi_{\mathcal{R}} \approx \frac{\log_2(n)+\gamma-1}{H(\mathcal{R})} + \frac{H_2(\mathcal{R})}{2H(\mathcal{R})^2}$.

Finally, to store the n integers marking the length of the common prefixes, we need total storage of $n \log_2 \left(\frac{N}{n}\right)$.

Adding these two terms together we obtain the result in the proposition.

A.2 Proof of Proposition 3.2

A bucket-based front-coded FIB $BFC(\mathcal{R})$ employs front-coding only within each bucket. Bucket headers are left uncompressed. Hence, $BFC(\mathcal{R})$ uses $\mathcal{C}(FC(R))$ bits in addition to the prefixes added to the CNPs in each bucket header (except the first since it is already used without compression in $FC(\mathcal{R})$). Since the average length of common prefixes is $\phi_{\mathcal{R}}$, the length of the additional characters in the headers is $(\lceil \frac{n}{\beta} \rceil - 1)\phi_{\mathcal{R}}$.

Finally, we need to store $\lceil \frac{n}{\beta} \rceil - 1$ pointers to the position of the headers of the new buckets. The whole bucket-based front-coded representation of $\mathcal{R}$ now needs $N - n\phi_{\mathcal{R}} + (\lceil \frac{n}{\beta} \rceil - 1)\phi_{\mathcal{R}}$ characters. We thus need $\log_2(N - (n + 1 - \lceil \frac{n}{\beta} \rceil)\phi_{\mathcal{R}})$ bits per pointer. Hence, $\mathcal{C}(BFC(R))$ is the sum of these terms.

A.3 Proof of Proposition 3.3

The buckets in the front-coded tree-based FIB $FCTree(\mathcal{R})$ have the same memory footprint as the bucket-based front-coded FIB $BFC(\mathcal{R})$. The internal structure of the buckets is identical in $BFC(\mathcal{R})$, and the same number of pointers are needed to access those buckets.

The difference between $FCTree(\mathcal{R})$ and $BFC(\mathcal{R})$ lies in the additional space needed by $FCTree(\mathcal{R})$ to store the binary tree structure containing the pointers to the buckets. In general, the number of different binary trees with k nodes is the Catalan number C_k [182]. Hence, the space required to represent a binary tree with k nodes is $\log_2(C_k)$. In the case of FCTree, we have $\lceil \frac{n}{\beta} \rceil$ buckets and thus $\lceil \frac{n}{\beta} \rceil$ nodes in the tree. Hence, the tree structure of FCtree can be stored in $\log_2 \left(C_{\lceil \frac{n}{\beta} \rceil} \right) \approx 2\frac{n}{\beta}$ [95] which is expressed as the second term of $\mathcal{C}(FCTree(R))$.

Since the maximum height of an RBT with k nodes is $2\log_2(k + 1)$ [98], the worst case lookup, insertion or deletion will traverse $2\log_2(\frac{n}{\beta} + 1)$ bucket headers to find the bucket and then perform β additional comparisons within the bucket in the worst case.