FANTNet: Uma Arquitetura para Rede de Trânsito NDN Baseada em Redes Definidas por Software

Eduardo Castilho Rosa

Universidade Federal de Uberlândia
Faculdade de Computação
Programa de Pós-Graduação em Ciência da Computação

Uberlândia
2024

Eduardo Castilho Rosa

FANTNet: Uma Arquitetura para Rede de Trânsito NDN Baseada em Redes Definidas por Software

Tese de doutorado apresentada ao Programa de Pós-graduação da Faculdade de Computação da Universidade Federal de Uberlândia como parte dos requisitos para a obtenção do título de Doutor em Ciência da Computação.

Área de concentração: Ciência da Computação

Orientador: Flávio de Oliveira Silva

Uberlândia
2024

Ficha Catalográfica Online do Sistema de Bibliotecas da UFU
com dados informados pelo(a) próprio(a) autor(a).

R788
2024

Rosa, Eduardo Castilho, 1986-
FANTNet: Uma Arquitetura para Rede de Trânsito NDN Baseada em Redes Definidas por Software [recurso eletrônico] / Eduardo Castilho Rosa. - 2024.

Orientador: Flávio de Oliveira Silva.
Tese (Doutorado) - Universidade Federal de Uberlândia, Pós-graduação em Ciência da Computação.
Modo de acesso: Internet.
Disponível em: http://doi.org/10.14393/ufu.te.2024.629
Inclui bibliografia.
Inclui ilustrações.

1. Computação. I. Silva, Flávio de Oliveira,1970-, (Orient.). II. Universidade Federal de Uberlândia. Pós-graduação em Ciência da Computação. III. Título.

CDU: 681.3

Bibliotecários responsáveis pela estrutura de acordo com o AACR2:

Gizele Cristine Nunes do Couto - CRB6/2091
Nelson Marcos Ferreira - CRB6/3074

UNIVERSIDADE FEDERAL DE UBERLÂNDIA
Coordenação do Programa de Pós-Graduação em Ciência da Computação
Av. João Naves de Ávila, 2121, Bloco 1A, Sala 243 - Bairro Santa Mônica, Uberlândia-MG, CEP 38400-902
Telefone: (34) 3239-4470 - www.ppgco.facom.ufu.br - cpgfacom@ufu.br

ATA DE DEFESA - PÓS-GRADUAÇÃO

Programa de Pós-Graduação em:	Ciência da Computação				
Defesa de:	Tese, 33/2024, PPGCO				
Data:	27 de agosto de 2024	Hora de início:	09:02	Hora de encerramento:	13:32
Matrícula do Discente:	11813CCP001				
Nome do Discente:	Eduardo Castilho Rosa				
Título do Trabalho:	FANTNet: Uma Arquitetura para Rede de Trânsito NDN Baseada em Redes Definidas por Software				
Área de concentração:	Ciência da Computação				
Linha de pesquisa:	Sistemas de Computação				
Projeto de Pesquisa de vinculação:	----------------				

Reuniu-se por videoconferência, a Banca Examinadora, designada pelo Colegiado do Programa de Pós-graduação em Ciência da Computação, assim composta: Professores Doutores: João Henrique de Souza Pereira - FACOM/UFU, Pedro Frosi Rosa - FACOM/UFU, Daniel Nunes Corujo - Universidade de Aveiro, António Luis Duarte Costa - Universidade do Minho e Flávio de Oliveira Silva - Universidade do Minho, orientador do candidato.

Os examinadores participaram desde as seguintes localidades: Flávio de Oliveira Silva - Braga/Portugal, Daniel Nunes Corujo -Aveiro/Portugal e António Luis Duarte Costa -Braga/Portugal. Os outros membros da banca e o aluno participaram da cidade de Uberlândia.

Iniciando os trabalhos o presidente da mesa, Prof. Dr. Flávio de Oliveira Silva, apresentou a Comissão Examinadora e o candidato, agradeceu a presença do público, e concedeu ao Discente a palavra para a exposição do seu trabalho. A duração da apresentação do Discente e o tempo de arguição e resposta foram conforme as normas do Programa.

A seguir o senhor presidente concedeu a palavra, pela ordem sucessivamente, aos examinadores, que passaram a arguir ao candidato. Ultimada a arguição, que se desenvolveu dentro dos termos regimentais, a Banca, em sessão secreta, atribuiu o resultado final, considerando o candidato:

Aprovado

Esta defesa faz parte dos requisitos necessários à obtenção do título de Doutor.

Ressalta-se que o examinador Daniel Nunes Corujo e António Luis Duarte Costa , por

ser estrangeiro, residente em outro país e não possuir CPF registrado no Brasil não assinará a ata de defesa.

O competente diploma será expedido após cumprimento dos demais requisitos, conforme as normas do Programa, a legislação pertinente e a regulamentação interna da UFU.

Nada mais havendo a tratar foram encerrados os trabalhos. Foi lavrada a presente ata que após lida e achada conforme foi assinada pela Banca Examinadora.

Documento assinado eletronicamente por **João Henrique de Souza Pereira**, **Professor(a) do Magistério Superior**, em 02/09/2024, às 10:12, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

Documento assinado eletronicamente por **Pedro Frosi Rosa**, **Usuário Externo**, em 02/09/2024, às 10:18, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

Documento assinado eletronicamente por **Flávio de Oliveira Silva**, **Professor(a) do Magistério Superior**, em 02/09/2024, às 11:43, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

A autenticidade deste documento pode ser conferida no site https://www.sei.ufu.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0, informando o código verificador **5625309** e o código CRC **8A259B1E**.

Referência: Processo nº 23117.053749/2024-23 SEI nº 5625309

Este trabalho é dedicado à minha amada esposa Keicyane Castilho, que sempre esteve ao meu lado e não mediu esforços para me apoiar nos momentos críticos durante essa jornada de aprendizado.

Agradecimentos

Agradeço primeiramente à Deus por ter me sustentado com vida e saúde durante essa tragetória e por ter se feito presente na minha vida durante cada etapa desse trabalho.

À minha querida e amada esposa, Keicyane Castilho, que me apoiou incondicionalmente e foi meu pilar de sustentação durante toda essa jornada.

Aos meus pais, Alice e José, aos meus irmãos Bruno e Natália, à minha cunhada Marisa, e aos meus sogros, Regiani e Moisés. A nossa união foi fundamental para que eu obtivesse o equilíbrio emocional necessário durante os desafios que enfrentei nesse doutorado.

Ao meu orientador, Prof. Flávio Silva, pelos valiosos ensinamentos e sobretudo por me motivar e me fazer ver sempre o lado positivo das coisas.

Aos demais professores da FACOM, em especial Pedro Frosi, Camila Barioni, Lásaro Camargos, Rodrigo Miani e Marcelo Albertini, que contribuíram para que eu obtivesse uma base forte de conhecimento na área.

Aos colegas de laboratório, em especial ao Rodrigo Elias e Italo Cunha, pelos diálogos enriquecedores envolvendo assuntos técnicos que contribuíram para um aprofundamento teórico e aprendizado mútuo.

Às agências de fomento CAPES e CNPQ, pelo incentivo financeiro na forma de auxílios diversos.

Ao Instituto Federal Goiano, Campus Catalão, pelo apoio e concessão de lincença capacitação, a qual me deu condições de dedicar tempo integral à essa pesquisa.

Enfim, agradeço a todos aqueles que direta ou indiretamente esteve envolvido nessa jornada. Sem dúvida, a conclusão desse doutorado de maneira exitosa não seria possível sem a contribuição de cada um de vocês.

"Mas graças a Deus que nos dá a vitória por nosso Senhor Jesus Cristo"
(1º Coríntios 15:57)

Resumo

A arquitetura de Redes de Dados Nomeados (NDN) foi proposta para resolver algumas das limitações existentes na Internet atual. Dado o potencial da NDN como alternativa ao TCP/IP clássico, encaminhar o tráfego entre múltiplos domínios NDN por meio de redes de trânsito de alta velocidade é fundamental. Todavia, o desenvolvimento de encaminhadores de pacotes NDN trás inúmeros desafios, especialmente quando se trata do desenvolvimento de estruturas de dados para a NDN FIB em *hardware*. Embora existam propostas de NDN FIB para comutadores programáveis, dimensioná-las para armazenar milhões de prefixos no ASIC ainda é um problema não resolvido satisfatoriamente. Diante desse contexto, propõe-se nessa tese a FANTNet, uma arquitetura de rede de trânsito NDN baseada em SDN. A arquitetura FANTNet tem como premissa a aceleração do tráfego entre múltiplos domínios NDN através de uma abstração da rede núcleo que utiliza comutadores de borda programáveis. Para otimizar o encaminhamento no núcleo, propõe-se a *Compressed Forwarding Information Base* (CoFIB), uma estrutura de dados para a FIB que é empregada exclusivamente nos comutadores de borda. A CoFIB é implementada como um conjunto de tabelas P4 dispostas nos *pipelines* de ingresso e egresso, onde o algoritmo proposto para o *Longest Name Prefix Matching* (LNPM) utiliza múltiplas recirculações de pacotes. Para reduzir o número dessas recirculações, propõe-se uma heurística de posicionamento das tabelas nos blocos de ingresso e egresso. Como critério para comprimir os prefixos na CoFIB, essa tese introduz o conceito de prefixo nomeado canônico e um algoritmo para a extração desses prefixos da RIB. Resultados experimentais mostram uma redução de até 16,58× no consumo de memória *on-chip* da CoFIB em comparação com o estado da arte, além de uma menor probabilidade de falhas de *lookups* devido ao baixo número de colisões de *hash*. Além disso, os resultados mostram um aumento de 23,17% de pacotes de interesse processados a taxa de linha em comparação com o *baseline* devido a estratégia de otimização de posicionamento de tabelas.

Palavras-chave: NDN, SDN, P4, FIB, comutadores programáveis, prefixos canônicos.

Abstract

The Named Data Networks (NDN) architecture was proposed to solve some of the limitations existing in the current Internet. Given the potential of NDN as an alternative to classic TCP/IP, routing traffic between multiple NDN domains over high-speed transit networks is critical. However, developing NDN packet forwarders for the core network brings numerous challenges, especially regarding the data structures for the NDN FIB on *hardware*. Although there are NDN FIB proposals for programmable switches, scaling them to store millions of prefixes in the ASIC is still a problem that needs to be satisfactorily resolved. Given this context, this thesis proposes FANTNet, a logically centralized NDN transit network architecture based on SDN. The FANTNet's architecture aims to accelerate traffic between multiple NDN domains through an abstraction of the core network that uses programmable edge switches. To optimize forwarding in the core, we propose the *Compressed Forwarding Information Base* (CoFIB), a data structure for the FIB that is used exclusively in edge switches. CoFIB is implemented as a set of P4 tables arranged in the ingress and egress pipelines, where the algorithm proposed for LNPM uses multiple packet recirculations. We propose a heuristic for positioning the tables in the ingress and egress blocks to reduce the number of these recirculations. As a criterion for compressing prefixes in CoFIB, this thesis introduces the concept of canonical named prefixes and an algorithm for extracting these prefixes from the RIB. Experimental results show up to a 16.58× reduction in CoFIB *on-chip* memory consumption compared to the state-of-the-art, as well as a lower probability of *lookup* failures due to the low number of *hash* collisions. Furthermore, the results show a 23.17% increase in interest packets processed at line rate due to the table positioning optimization strategy.

Keywords: NDN, SDN, P4, FIB, programmable switches, canonical prefixes.

Lista de ilustrações

Figura 1.1 – Estrutura da Tese associada a cada questão de pesquisa. 38
Figura 2.1 – Comparação entre as pilhas de protocolos *Internet Protocol* (IP) e *Named Data Networking* (NDN). 42
Figura 2.2 – Pacote de interesse e pacote de dados na arquitetura NDN. 43
Figura 2.3 – Pacote de dados codificado em NDN. 45
Figura 2.4 – Fluxo operacional no plano de dados de um roteador NDN. 49
Figura 2.5 – Separação dos planos de dados e controle na *Software Defined Networking* (SDN). 50
Figura 2.6 – *Pipeline* reconfigurável de combinação-ação para um plano de dados programável. 52
Figura 2.7 – Compilando um programa P4. 53
Figura 4.1 – Visão geral da arquitetura *Fast NDN Transit Networking* (FANTNet). 70
Figura 4.2 – Fluxo operacional de pacotes NDN que não podem ser convertidos. . . 71
Figura 4.3 – Elementos do plano de controle da FANTNet associados aos comutadores de borda para a geração das tabelas do plano de dados. 76
Figura 4.4 – *Workflow* para pacotes de interesse e de dados. 81
Figura 5.1 – Representação de pacotes de interesse e de dados no formato *P4 Name Friendly Format* (P4NF). 84
Figura 5.2 – Fluxograma para extração de cabeçalhos. 86
Figura 5.3 – Formato de entrada em uma tabela da CoFIB. 92
Figura 5.4 – Fluxograma de processamento de pacotes de interesse no *pipeline* de entrada do comutador de borda. 97
Figura 5.5 – Fluxograma de processamento de pacotes de interesse no *pipeline* de saída do comutador de borda. 98
Figura 6.1 – Porcentagem de prefixos canônicos extraídos da *Routing Information Base* (RIB) usando o *Canonical Prefix Extractor* (CPE). 104
Figura 6.2 – Colisão de *hash* de componentes nomeados considerando quatro *datasets* de nomes distintos. 106

Figura 6.3 – Consumo de memória com diferentes valores de P_r. 109
Figura 6.4 – Taxa de ocupação de memória *on-chip* da *Compressed Forwarding Information Base* (CoFIB). 110
Figura 6.5 – Testbed experimental da FANTNet. 112
Figura 6.6 – Vazão em *bits* por segundo (a) e em pacotes por segundo (b) ao enviar Ipkts ordenados pelo número de componentes. 116
Figura 6.7 – Vazão em *bits* por segundo (a) e em pacotes por segundo (b) ao enviar Ipkts em ordem arbritária. 117
Figura 6.8 – CDF do número de passagens no *pipeline* usando o *dataset* 0.5K. 119
Figura 6.9 – CDF do número de passagens no *pipeline* usando o *dataset* 4M. 121

Lista de tabelas

Tabela 3.1 – Arquiteturas baseadas em SDN para o encaminhamento de tráfego NDN. 59
Tabela 3.2 – Principais soluções NDN *Forwarding Information Base* (FIB) para comutadores tradicionais. 66
Tabela 3.3 – Principais soluções NDN FIB baseadas em *Programming Protocol-Independent Packet Processors* (P4). 67
Tabela 3.4 – Comparação entre as principais soluções para a NDN FIB existentes na literatura implementadas em *software* e em *hardware*. 68
Tabela 6.1 – Características de cada *dataset* de nomes utilizado nessa tese. 103
Tabela 6.2 – Comparação entre a FANTNet e as principais arquiteturas baseadas em SDN para o encaminhamento de tráfego NDN. 124
Tabela 6.3 – Comparação entre as principais soluções para a NDN FIB existentes na literatura implementadas em *software* e em *hardware*. 125

Lista de siglas

ALU *Arithmetic Logic Unit*

ANPM *All Name Prefix Matching*

API *Application Programming Interface*

AS *Autonomous System*

ASCII *American Standard Code for Information Interchange*

ASIC *Application-Specific Integrated Circuit*

ASNM *All Sub-Name Matching*

BGP *Border Gateway Protocol*

BM *Bypass Memory*

BMv2 *Behavioral Model Version 2*

CAD *Component Action Data*

CAM *Content Addressable Memory*

CFIB *Control Plane Forwarding Information Base*

CNAME *Canonical Name*

CoFIB *Compressed Forwarding Information Base*

CPE *Canonical Prefix Extractor*

CPU *Central Processing Unit*

CS *Content Store*

CSw *Core Switch*

CPST *Control Plane Shape Table*

DCH *Dual Component Hashmap*

DFIB *Dataplane FIB*

DNS *Domain Name System*

DPST *Data Plane Shape Table*

DRAM *Dynamic Random Acess Memory*

dRMT *disaggregated Reconfigurable Match-Action Table*

ESw *Edge Switch*

ENM *Exact Name Matching*

FANTNet *Fast NDN Transit Networking*

FIB *Forwarding Information Base*

FIFO *First-In First Out*

FPGA *Field-Programmable Gate Array*

FST *Fast Switching Table*

HBM *Hashtray-Based Method*

HCT *Hash Conflicting Table*

ICN *Information-Centric Networking*

IP *Internet Protocol*

IoT *Internet of Things*

JVM *Java Virtual Machine*

LFU *Least Frequently Used*

LIB *Label Information Base*

LNPM *Longest Name Prefix Matching*

LPM *Longest Prefix Matching*

LRU *Least Recently Used*

NPC *NDN Packet Converter*

MPLS *Multi Protocol Label Switching*

NCH *Name Component Hash Table*

NDN *Named Data Networking*

NF *Network Function*

NIC *Network Interface Card*

NLSR *Named-data Link State Routing*

OVS *Open vSwitch*

P4 *Programming Protocol-Independent Packet Processors*

P4RT *P4Runtime*

P4NF *P4 Name Friendly Format*

PDU *Packet Data Unit*

PISA *Protocol-Idependent Switch Architecture*

PIT *Pending Interest Table*

PNA *Portable NIC Architecture*

PSA *Portable Switch Architecture*

QoS *Quality-of-Service*

QUIC *Quick UDP Internet Connection*

RIB *Routing Information Base*

RMT *Reconfigurable Match Tables*

RTT *Round Trip Time*

SDN *Software Defined Networking*

SRAM *Static Random Access Memory*

TCAM *Ternary Content Addressable Memory*

TLS *Transport Layer Security*

TLV *Type-Length Value*

TNA *Tofino Native Architecture*

URL *Uniform Resource Locator*

Sumário

1 INTRODUÇÃO . . . 29
1.1 Contextualização . . . 29
1.2 Caracterização do Problema . . . 31
1.3 Motivação . . . 32
1.4 Objetivos . . . 33
1.5 Hipóteses de Pesquisa . . . 34
1.6 Contribuições . . . 35
1.7 Organização da Tese . . . 38

2 FUNDAMENTAÇÃO TEÓRICA . . . 41
2.1 Introdução . . . 41
2.2 Arquitetura NDN . . . 41
2.2.1 Tipos de Pacotes . . . 42
2.2.2 Plano de Dados . . . 44
2.2.3 Plano de Controle . . . 49
2.3 Paradigma SDN . . . 50
2.3.1 Plano de Dados Programável . . . 51
2.3.2 Linguagem P4 . . . 53
2.4 Considerações . . . 55

3 TRABALHOS RELACIONADOS . . . 57
3.1 Introdução . . . 57
3.2 Arquiteturas para Encaminhamento NDN . . . 57
3.3 Estruturas de Dados para a NDN FIB . . . 60
3.3.1 FIB baseada em Tabelas de *Hash* . . . 60
3.3.2 FIB baseada em *Trie* . . . 62
3.3.3 FIB baseada em Filtros de *Bloom* . . . 64
3.4 Considerações . . . 68

4 FANTNET: PLANO DE CONTROLE . . . 69
4.1 Introdução . . . 69
4.2 Visão Geral . . . 70
4.3 Routing Information Base . . . 72
4.4 Fast Switching Table . . . 73
4.5 Segregação de Prefixos . . . 74
4.6 Dual Component Hashmap . . . 76
4.7 Fases de Operação . . . 79
4.8 Considerações . . . 81

5 FANTNET: PLANO DE DADOS . . . 83
5.1 Introdução . . . 83
5.2 Representação de Nomes . . . 84
5.2.1 Formato P4NF . . . 84
5.2.2 Extração de Cabeçalhos . . . 85
5.3 Estruturas de Dados . . . 87
5.3.1 Tabela CS . . . 87
5.3.2 Tabela PIT . . . 87
5.3.3 Tabelas de moldes . . . 88
5.3.4 Tabelas para armazenamento de prefixos . . . 89
5.3.5 Estrutura CAD . . . 89
5.4 Otimização de Recursos . . . 93
5.4.1 Otimização de Memória *On-Chip* . . . 93
5.4.2 Otimização de Vazão . . . 94
5.5 Lógica do LNPM . . . 96
5.6 Considerações . . . 100

6 AVALIAÇÃO EXPERIMENTAL E DISCUSSÃO . . . 101
6.1 Introdução . . . 101
6.2 Método para a Avaliação . . . 101
6.3 Datasets de Nomes . . . 102
6.4 Análise Quantitativa . . . 104
6.4.1 Taxa de *Offloading* . . . 104
6.4.2 Probabilidade de Encaminhamento Incorreto . . . 105
6.4.3 Consumo de Memória . . . 107
6.5 TestBed Experimental da FANTNet . . . 112
6.6 Experimentos de Simulação . . . 114
6.6.1 Experimento 1: Pacotes de Interesse Ordenados . . . 115
6.6.2 Experimento 2: Pacotes de Interesse Não Ordenados . . . 117
6.6.3 Experimento 3: Passagens no *pipeline* (*Dataset* 0.5K) . . . 118

6.6.4 Experimento 4: Passagens no *Pipeline* (*Dataset* 4M) 120
6.7 Discussão 122
6.7.1 Análise Qualitativa 123
6.7.2 Questões de Pesquisa 126
6.8 Considerações 130

7 CONSIDERAÇÕES FINAIS 131
7.1 Principais Contribuições 132
7.2 Trabalhos Futuros 132
7.3 Produção Bibliográfica 134

REFERÊNCIAS 137

Capítulo 1

Introdução

1.1 Contextualização

A evolução da Internet ao longo das últimas décadas tem evidenciado a capacidade de adaptação da arquitetura TCP/IP em suportar aplicações cada vez mais complexas. No entanto, as limitações no modelo de comunicação centrado em sistemas finais tem motivado a comunidade científica a propor arquiteturas de rede mais aderentes aos requisitos de aplicações emergentes. Essas iniciativas são conhecidas na literatura como abordagens *clean-slate* (REXFORD; DOVROLIS, 2010) e vem sendo propostas como alternativas à arquitetura TCP/IP.

Dentre as propostas de arquiteturas *clean-slate* para a Internet do Futuro, as mais prevalentes são aquelas conhecidas como Redes Centradas na Informação (*Information-Centric Networking* (ICN)) (ALDAOUD et al., 2023). De maneira geral, o princípio básico de uma rede ICN é tratar os dados de forma independente de sua localização física. Através de pacotes contendo nomes estruturados de maneira hierárquica, as redes ICN são capazes de identificar de forma unívoca todas as porções de dados transferidas entre nós da rede sem a necessidade de identificar os hospedeiros que armazenam esses dados. Para garantir a interdependência de localização, os roteadores ICN armazenam pacotes em *cache* ao longo do caminho entre a origem e o destino. Desse modo, é possível diminuir o *Round Trip Time* (RTT) médio consideravelmente (MURALIDHARAN; ROY; SAXENA, 2018) tendo em vista que não somente o destinarário mas qualquer roteador intermediário que armazene o pacote em *cache* pode responder ao requisitante mesmo este não sendo o produtor do conteúdo.

Atualmente, a *Named Data Networking* (NDN) é a arquitetura ICN mais avançada em termos de maturidade e estágio de desenvolvimento (TAKI; MASTORAKIS, 2023). Em NDN, a comunicação entre nós consumidores e produtores é baseada na troca síncrona de pacotes de interesse (Ipkts) e pacotes de dados (Dpkts). O nó consumidor sempre inicia a comunicação enviando um Ipkt para a rede contendo o nome do conteúdo requisitado. O nó produtor, ou algum roteador intermediário com *cache* habilitado, envia um Dptk

com o conteúdo requisitado através do caminho reverso. Para garantir funcionalidades como transmissão *multicast*, encaminhamento baseado em nomes e interdependência de localização, o plano de dados de um roteador NDN utiliza tabelas específicas para armazenamento de Dpkts e de prefixos nomeados. O plano de controle da arquitetura NDN é distribuído e responsável por definir rotas iniciais de longo prazo para Ipkts, diferindo da arquitetura TCP/IP por não ter a função de responder à alterações dinâmicas da topologia da rede.

Embora a arquitetura NDN traga várias novas funcionalidades que resolvem parte das limitações do modelo TCP/IP, o processamento de pacotes no plano de dados de um roteador NDN é muito mais complexo quando comparado ao processamento de um datagrama IP em um roteador tradicional (TARIQ; REHMAN; KIM, 2020). Pacotes NDN carregam nomes de comprimento variável, codificados em *Type-Length Value* (TLV), onde o encaminhamento através da *Forwarding Information Base* (FIB) é realizado com base no algoritmo *Longest Name Prefix Matching* (LNPM). Além disso, o roteador NDN normalmente mantém estado sobre o *status* de cada interface para viabilizar a estratégia de encaminhamento e tanto Ipkts como Dpkts podem ser enviados para múltiplas interfaces de saída, o que requer *engines* de replicação de pacotes mais complexas. Todas essas características tornam o projeto e desenvolvimento de um roteador NDN físico muito desafiador. Isso fica evidente ao observar o excasso número de comutadores NDN de alto desempenho projetados na última década (ROSA; SILVA, 2022b).

O conceito de Redes Definidas por *Software* (*Software Defined Networking* (SDN)), introduzido por volta de 2009 (FEAMSTER; REXFORD; ZEGURA, 2014), foi o primeiro passo concreto no sentido de trazer maior flexibilidade ao processamento de pacotes em redes sob um mesmo domínio administrativo. Através do protocolo OpenFlow (MCKEOWN et al., 2008), o plano de encaminhamento de pacotes passa a ser aberto para o operador de rede, que o controla por meio de *Application Programming Interface* (API)s. Com isso, surgiu a oportunidade de implementar redes NDN por meio da tecnologia SDN, através do protocolo OpenFlow, como em Salsano et al. (2013) e Cabral, Rothenberg e aes (2013). Todavia, embora o protocolo OpenFlow seja capaz de comutar tráfego NDN, ele opera apenas sobre um conjunto fixo de protocolos de rede. Em razão da arquitetura NDN ainda não estar padronizada (TEHRANI et al., 2022), o formato de Ipkts e de Dpkts não pode ser reconhecido nativamente pelo protocolo OpenFlow.

As redes de sobreposição (*overlay*) podem ser utilizadas para resolver o problema da limitação do OpenFlow em reconhecer o formato TLV. Nessa linha, Adrichem e Kuipers (2015) propõem o NDNFlow, uma solução para encaminhar o tráfego NDN através de comutadores OpenFlow que encapsulam pacotes NDN em quadros *Ethernet*. De maneira geral, a técnica de sobreposição permite que pacotes NDN possam ser carregados como carga útil em *Packet Data Unit* (PDU)s da camada de enlace, rede e transporte do modelo TCP/IP. Dessa forma, é possível utilizar os próprios nós da Internet para transportar

os dados sem se preocupar com a implantação de uma nova infraestrutura física. No entanto, utilizar redes de sobreposição faz com que certos recursos nativos da arquitetura NDN, como por exemplo, o *caching* na rede e o encaminhamento baseado em nomes, não funcionem adequadamente em função do plano de dados em encaminhadores de pacotes TCP/IP não serem programáveis.

O surgimento da arquitetura de *pipeline* reconfigurável, conhecida como *Reconfigurable Match Tables* (RMT) (BOSSHART et al., 2013), trouxe um avanço significativo para a indústria de redes ao propor uma lógica de processamento de pacotes totalmente customizável. A ideia é que subconjuntos de *pipelines* físicos em um comutador possam ser alocados em estágios lógicos de maneira flexível. Posteriormente, a arquitetura *disaggregated Reconfigurable Match-Action Table* (dRMT) (CHOLE et al., 2017) surge como uma melhoria à RMT ao possibilitar que os recursos de um comutador programável (memória e processamento) possam ser alocados sob demanda de forma centralizada pelo operador de rede para cada protocolo. Tanto a arquitetura RMT como a dRMT tornaram possível o desenvolvimento de roteadores NDN compatíveis com o paradigma SDN e capazes de processar Ipkts e Dpkts em *hardware* à altas taxas sem a necessidade de redes de sobreposição e de *Application-Specific Integrated Circuit* (ASIC)s NDN específicos.

1.2 Caracterização do Problema

A natureza descentralizada e distribuída do modelo de comunicação orientado em dados torna a NDN uma arquitetura de rede robusta e altamente escalável. No entanto, em uma rede de trânsito que encaminha tráfego NDN, a falta de informações centralizadas sobre a topologia e recursos da rede dificulta a tomada rápida de decisão de encaminhamento de pacotes. Nesse sentido, a utilização do paradigma SDN para viabilizar o encaminhamento de pacotes em alta velocidade entre múltiplos domínios NDN é importante para acelerar a implantação da arquitetura em nível global. Desse modo, a proposta de uma arquitetura centralizada para viabilizar o trânsito de pacotes NDN em alta velocidade vem de encontro com a necessidade de garantir uma engenharia de tráfego mais eficiente além de um gerenciamento e controle mais inteligente no núcleo da rede.

É importante salientar que a utilização do paradigma SDN para viabilizar o encaminhamento de tráfego NDN permite uma lógica de controle centralizada que elimina os problemas de convergência de algoritmos distribuídos. Todavia, os algoritmos centralizados executados no plano de controle precisam gerar e descarregar entradas na FIB para viabilizar o encaminhamento sem a interferência do controlador SDN. No plano de dados da arquitetura NDN, a tabela FIB desempenha um papel essencial para garantir que Ipkts chegem até os nós produtores. Em razão de possuírem comprimento variável e necessitarem de mais espaço de armazenamento na FIB, quando comparados à endereços IP tradicionais, os prefixos nomeados exigem algoritmos mais complexos de compatibilização

de prefixos mais longo (*Longest Prefix Matching* (LPM)) e técnicas de compressão da FIB. Nesse sentido, o principal problema de pesquisa tratado nessa tese é como garantir que tabelas de encaminhamento por nomes sejam processadas a altas velocidades em *harware* ao mesmo tempo que sejam capazes de armazenar milhões de prefixos nomeados dadas as limitações de memória *on-chip* e as restrições de *hardware* que tornam a operação sobre estruturas de dados menos flexíveis quando comparadas ao desenvolvimento em *software*.

1.3 Motivação

Embora existam várias propostas para acelerar o encaminhamento NDN, como em Shi, Pesavento e Benmohamed (2020) e Newberry, Ma e Zhang (2021), o projeto e desenvolvimento de estruturas de dados para a FIB que sejam rápidas e ao mesmo tempo eficientes no consumo de memória é um problema ainda não resolvido satisfatoriamente (TARIQ; REHMAN; KIM, 2020). Soluções em *software* dispõem de grande quantidade de memória para armazenamento de prefixos nomeados. No entanto, a latência de comutação em *software* é da ordem de milisegundos, o que inviabiliza processamento de pacotes em alta velocidade, por exemplo na ordem de *Tbps*. Por outro lado, soluções em *hardware* possibilitam o processamento de pacotes na escala de nanosegundos (FRANCO et al., 2024), mas a quantidade de memória *on-chip* disponível é limitada à algumas dezenas de MB (MIAO et al., 2017), o que restringe a quantidade de prefixos que podem ser armazenados na FIB. O desenvolvimento de uma arquitetura que possibilite o encaminhamento de pacotes NDN em alta velocidade e de maneira flexível, por meio de comutadores programáveis, contribui para a evolução da NDN no longo prazo. Nesse sentido, o projeto de uma estrutura de dados para a FIB que equilibra o consumo de memória com velocidade de comutação é fundamental.

A ideia de implementar o plano de dados NDN em comutadores programáveis não é nova. A arquitetura ENDN (KARRAKCHOU; SAMAAN; KARMOUCH, 2020a) implementa a FIB em comutadores com suporte à linguagem *Programming Protocol-Independent Packet Processors* (P4) através da estrutura de dados FCTree (KARRAKCHOU; SAMAAN; KARMOUCH, 2020b). A FCTree provê mecanismos de compressão de memória, mas é projetada para rodar em *software*, fazendo uso extenso de recursos como ponteiros, recursividade e alocação dinâmica de memória. Esses recursos são difíceis de implementar em comutadores programáveis físicos sem que haja um impacto considerável na latência de processamento. Por outro lado, Takemasa, Koizumi e Hasegawa (2021) desenvolvem o que eles consideram como o primeiro protótipo de um roteador NDN desenvolvido para um comutador programável físico, baseado no ASIC Intel Tofino. Todavia, a tabela FIB nessa solução é implementada em *software* no plano de controle, o que aumenta a latência média de processamento de Ipkts. Signorello et al. (2016) propõem o NDN.p4, uma implementação do plano de dados NDN onde a FIB é projetada utilizando funções de *hash*.

Em NDN.p4, a FIB é implementada como uma única tabela P4 que contém sequências de *hashes* de subprefixos, onde esses subprefixos são utilizados como chaves de compatibilização na *Ternary Content Addressable Memory* (TCAM). Em Miguel, Signorello e Ramos (2018), os autores propõem um método baseado em trilhas de *hash*, chamado aqui de *Hashtray-Based Method* (HBM), que segue os mesmos princípios usados em NDN.p4, mas que reduz o consumo de memória eliminando a necessidade de inserir subprefixos adicionais para cada entrada na FIB. No entanto, embora tanto a NDN.p4 como a HBM serem projetadas para executar a LNPM em uma única passagem no *pipeline*, a utilização de *hashes* de 16 *bits* para gerar as entradas na tabela P4 resulta em um alto número de colisões, o que acaba aumentando o consumo de memória consideravelmente quando grandes *datasets* são utilizados (ROSA; SILVA, 2022b).

As soluções propostas para a NDN FIB em comutadores programáveis apresentam três importantes limitações. A primeira delas está relacionada com o alto número de colisões de *hash* que não é tratado adequadamente em nenhuma proposta. Reduzir o número de colisões em *hardware* não é trivial dadas as restrições e limitações de *targets* P4. Em segundo, as soluções propostas que fazem o descarregamento da FIB na memória do ASIC usam somente os recursos existentes no *pipeline* de ingresso, desperdiçando a metade dos recursos de memória e processamento disponíveis em comutadores programáveis. Por último, as propostas para a NDN FIB não provêem mecanismos para equilibrar o consumo de memória com a latência de processamento. Portanto, a solução apresentada nessa tese é projetada para resolver essas questões em aberto, tornando possível o armazenamento de milhões de prefixos nomeados na memória *Static Random Access Memory* (SRAM) de comutadores programáveis enquanto a latência de processamento é limitada na escala de nanosegundos.

1.4 Objetivos

O objetivo geral dessa tese é projetar uma rede de trânsito baseada em SDN para encaminhar tráfego entre domínios NDN geograficamente dispersos. A arquitetura dessa rede de trânsito utiliza comutadores programáveis na borda e comutadores híbridos (comutadores convencionais ou programáveis) no núcleo. O racional da arquitetura consiste em armazenar os prefixos anunciados que chegam a esses comutadores na *Routing Information Base* (RIB) no plano de controle e depois descarregá-los para a FIB na memória *on-chip* dos comutadores de borda a fim de possibilitar o encaminhamento rápido no plano de dados. Como forma de atingir esse objetivo geral, os seguintes objetivos específicos são definidos:

- ❑ Projetar uma estrutura de dados para a NDN FIB no plano de dados que suporte as operações de inserção, atualização e remoção compatível com a arquitetura de comutadores programáveis.

- Criar métricas para comprimir prefixos nomeados de forma a aumentar a quantidade de prefixos que podem ser armazenados na memória *on-chip* sem que isso impacte de maneira significativa a latência de processamento por pacote.
- Propor um algoritmo para a seleção de prefixos nomeados que serão descarregados da RIB no plano de controle para a FIB no plano de dados.
- Propor um novo algoritmo para a LNPM baseado em múltiplas operações de combinação-ação nos *pipelines* de entrada e saída de comutadores programáveis.
- Propor uma heurística para o posicionamento de tabelas nos *pipelines* de entrada e saída a fim de reduzir o número de recirculações de pacotes necessário para completar a LNPM.

1.5 Hipóteses de Pesquisa

Essa tese propõe uma arquitetura baseada em SDN para garantir que o tráfego entre múltiplos domínios NDN seja processado de maneira flexível e eficiente. Assim, a utilização de comutadores programáveis de borda é parte fundamental da arquitetura proposta. Desse modo, as hipóteses e questões de pesquisas levantadas nesse trabalho estão relacionadas em grande parte à estrutura de dados para a FIB em comutadores programáveis. A seguir são apresentadas as hipóteses definidas nessa tese.

- **Hipótese 1**: *É possível projetar uma arquitetura baseada em SDN para viabilizar o encaminhamento de tráfego NDN entre múltiplos domínios por meio de comutadores de borda programáveis capazes de armazenar prefixos na FIB na ordem de milhões.*

As seguintes questões de pesquisa estão associadas à Hipótese 1.

- **Q1**: Como processar pacotes NDN eficientemente em comutadores programáveis dadas as restrições de *hardware* e a complexidade da codificação TLV?
- **Q2**: Como eliminar a necessidade de armazenar a FIB em comutadores de núcleo e ao mesmo tempo garantir o encaminhamento baseado em nomes entre comutadores de borda?
- **Q3**: Em razão da impossibilidade de armazenar todos os prefixos da RIB na FIB, devido a limitação de memória *on-chip*, qual critério utilizar para selecionar um subconjunto de prefixos para descarregar na FIB?

- **Hipótese 2**: *O uso de funções de hash de 32 bits para armazenar componentes nomeados é capaz de reduzir a probabilidade de colisão e ao mesmo tempo reduzir o consumo de memória on-chip em comutadores programáveis em comparação com o uso de funções de hash de 16 bits.*

As questões de pesquisa associadas à Hipótese 2 são:

- **Q4**: Devido a possíveis problemas de fragmentação, como armazenar componentes com menos de 5 caracteres de forma eficiente na SRAM?

- **Q5**: Quais técnicas de compressão podem ser utilizadas para reduzir a quantidade de memória *on-chip* necessária para armazenar prefixos nomeados na NDN FIB?

❑ **Hipótese 3**: *A utilização de múltiplas tabelas P4 para armazenar componentes nomeados ao invés de uma única tabela diminui a probabilidade de falso encaminhamento devido à colisão de hash que ocorre em um determinado prefixo nomeado.*

A questão de pesquisa associada à Hipótese 3 é:

- **Q6**: Qual critério pode ser utilizado para agrupar componentes nomeados a fim de determinar o número de tabelas P4 a serem utilizadas?

❑ **Hipótese 4**: *É possível limitar a quantidade de recirculações de pacotes de interesse no pipeline de um comutador programável para que a latência da operação LNPM permaneça na ordem de grandeza de nanosegundos.*

A questão de pesquisa associada à Hipótese 4 é:

- **Q7**: *Quais mecanismos podem ser utilizados para diminuir a quantidade de recirculações de pacotes no pipeline e mitigar o efeito dessas recirculações em relação à latência de processamento?*

1.6 Contribuições

Essa tese propõe a *Fast NDN Transit Networking* (FANTNet), uma arquitetura baseada no modelo *Fabric* (CASADO et al., 2012) que possibilita o encaminhamento de tráfego NDN entre múltiplos domínios. O modelo *Fabric* proposto por Casado et al. (2012) refere-se a uma coleção de elementos de encaminhamento que tem como principal função o transporte de pacotes. Esse modelo integra a tecnologia SDN na qual a borda implementa as políticas de rede e gerencia os endereços dos sistemas finais enquanto a *Fabric* interconecta a borda da maneira mais eficiente e rápida quanto possível. Existem duas diferenças principais entre o modelo *Fabric* em Casado et al. (2012) e a arquitetura FANTNet. A primeira é que a *Fabric* utiliza dois controladores independentes (um para a borda e outro para o núcleo) enquanto que a FANTNet utiliza um controlador logicamente centralizado para controlar tanto os comutadores de borda quanto os de núcleo. A segunda é que a *Fabric* é projetada para o tráfego *Internet Protocol* (IP) enquanto a

FANTNet foca no encaminhamento NDN, muito embora o tráfego IP também possa ser encaminhado pela FANTNet.

A arquitetura FANTNet atua como uma rede de trânsito semelhante à definida em Haddadi et al. (2008), onde Sistemas Autônomos (*Autonomous System* (AS)) geograficamente separados são interligados para prover interconexão global entre múltiplos domínios. O termo arquitetura utilizado nessa tese é definido como um cojunto de elementos de *hardware* e *software* que interagem por meio de interfaces bem definidas a fim de garantir a comunicação de dados de maneira flexível.

Para viabilizar o encaminhamento de Ipkts entre múltiplos domínios NDN nativos, propõe-se nessa tese a *Compressed Forwarding Information Base* (CoFIB), uma estrutura de dados para a FIB que roda nos comutadores programáveis de borda. A CoFIB é projetada para lidar com as limitações de memória *on-chip* disponível em comutadores programáveis e é capaz de utilizar a SRAM para armazenar os prefixos nomeados. Como métrica para otimizar espaço no plano de dados, essa tese introduz o conceito de prefixos de nomes canônicos para compactar a FIB de forma que seja possível armazenar milhões de prefixos nomeados na memória do ASIC. Operar nessa escala de magnitude é um aspecto inovador da CoFIB em relação ao estado da arte. No plano de controle, propõe-se um algoritmo para o descarregamento de prefixos canônicos da RIB para a memória *on-chip* dos comutadores programáveis. Propõe-se também um algoritmo para a LNPM que utiliza recirculações de pacotes no *pipeline*, mantendo a latência téorica de processamento na ordem de nanosegundos.

Em contraste com arquiteturas existentes para o encaminhamento de tráfego entre domínios NDN, como a NDNFab (MADUREIRA et al., 2021), a FANTNet descarrega a FIB para o plano de dados (*fast path*), tornando o encaminhamento de Ipkts mais veloz. Já em relação a soluções para a NDN FIB baseadas em P4, a CoFIB foi projetada para ser executada em comutadores de borda programáveis em uma rede baseada em SDN. A CoFIB utiliza tabelas de *hash* como estrutura de dados subjacentes, semelhante à maioria dos modelos de NDN FIB na literatura. No entanto, em vez de usar uma grande tabela P4 para armazenar os prefixos nomeados, a CoFIB reduz a probabilidade de colisão de *hash* ao utilizar múltiplas tabelas nos *pipelines* de ingresso e egresso e uma função com comprimento de *hash* mais longo. Além disso, a CoFIB explora toda a memória SRAM disponível nas unidades de controle de entrada e saída. Consequentemente, o algoritmo de LNPM proposto pode realizar duas operações de combinação-ação por passagem no *pipeline*, sendo essa uma outra característica inovadora da CoFIB. Isso exige múltiplas recirculações de pacotes para apenas alguns grupos de Ipkts.

Para minimizar o número de recirculações necessárias para completar o LNPM, essa tese propõe uma heurística para posicionamento de tabelas nos *pipelines* de entrada e saída de acordo com a distribuição de nomes. Além disso, a CoFIB utiliza recursos da linguagem P4 como *action profiles* e *action selectors* como um método adicional para

reduzir o consumo de memória de tabelas que contêm apenas uma ação. Assim, a CoFIB pode escalar para milhões de prefixos nomeados armazenados na memória SRAM, mantendo a latência média por pacote em uma escala de nanossegundos. Pelo levantamento bibliográfico realizado, a CoFIB é a primeira estrutura de dados para a NDN FIB a explorar completamente a disponibilidade de memória SRAM em ASICs programáveis para armazenar prefixos de nomes em vez de depender apenas de memória TCAM que, além de disponível em menor capacidade, é um tipo de memória cara que consome muita energia.

O plano de controle logicamente centralizado da FANTNet permite que cada rede de trânsito possa operar como uma abstração de um comutador NDN em grande escala, onde as interfaces de saída correspondem aos comutadores de borda conectados a cada domínio NDN nativo. Desse modo, propõe-se uma alteração na operação da FIB onde, ao invés da LNPM encontrar um conjunto de interfaces de saída para encaminhar o Ipkt, as entradas na CoFIB contém prefixos nomeados associados aos identificadores dos comutadores de borda que respondem por tais prefixos. Desse modo, a operação de LNPM na arquitetura proposta retorna um identificador que representa o conjunto de comutadores de borda pelos quais o Ipkt será encaminhado. Como base nesse identificador, o pacote é comutado à velocidade de linha na *Fabric* (malha de comutadores de núcleo).

Os comutadores programáveis empregados na borda da arquitetura FANTNet desempenham um papel fundamental no encaminhamento de pacotes em alta velocidade. Desse modo, as principais contribuições desse trabalho estão relacionadas ao projeto dos componentes do plano de dados dos comutadores programáveis e aos elementos do plano de controle. De maneira resumida, as principais contribuições desse trabalho são:

- Uma arquitetura baseada no paradigma SDN para encaminhamento em alta velocidade de tráfego entre múltiplos domínios NDN. A arquitetura proposta provê uma abstração da rede de trânsito como um único roteador NDN em larga escala.

- Uma nova estrutura de dados baseada em *hash* para a NDN FIB, chamada CoFIB, totalmente compatível com comutadores programáveis e que minimiza o consumo de memória em comparação com soluções na literatura.

- A introdução do conceito de prefixo nomeado canônico como métrica de compressão da tabela FIB bem como um algoritmo para a extração de nomes canônicos da RIB.

- Um algoritmo para a LNPM que recircula cuidadosamente Ipkts no *pipeline*, realizando operações de combinação-ação tanto no bloco de controle de entrada como no de saída.

- Uma heurística para otimizar o posicionamento de tabelas P4 nos *pipelines* de entrada e saída a fim de reduzir a quantidade de recirculações de pacotes e melhorar a vazão.

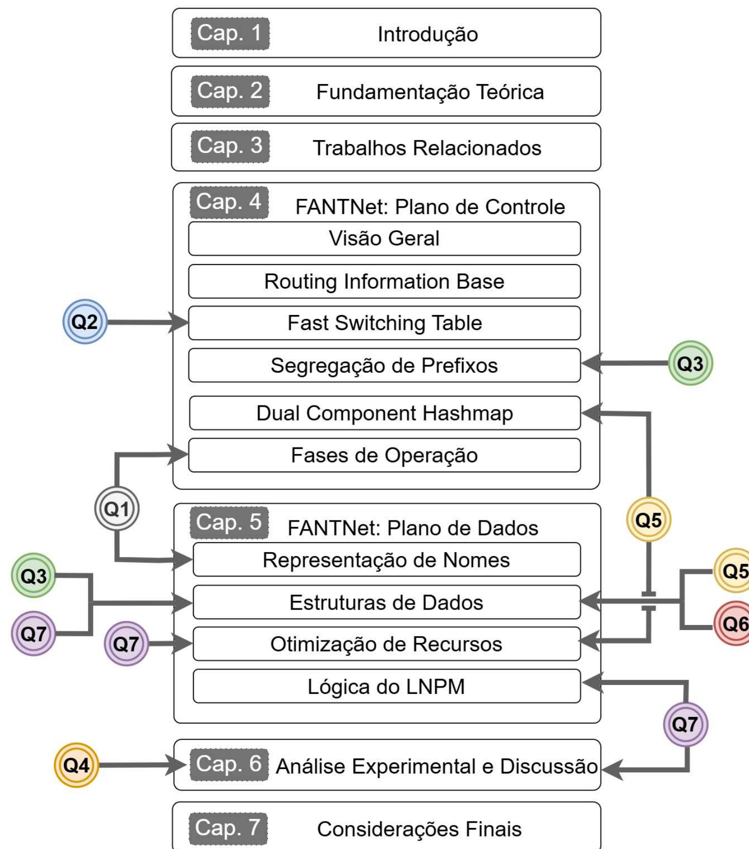

Figura 1.1 – Estrutura da Tese associada a cada questão de pesquisa.

1.7 Organização da Tese

O restante dessa tese está organizado da seguinte maneira. O Capítulo 2 apresenta uma visão geral da arquitetura NDN e os fundamentos do paradigma SDN. A arquitetura NDN é apresentada de acordo com as funções do plano de dados e do plano de controle. No plano de dados, são abordados conceitos como a codificação de Ipkts e Dpkts e as tabelas *Content Store* (CS), *Pending Interest Table* (PIT) e FIB. Já no plano de controle, são abordados detalhes sobre o roteamento em NDN e o papel da tabela RIB na formação das entradas da FIB. Os fundamentos das redes SDN são apresentados com o foco no plano de dados, que é abordado em função das principais arquiteturas de elementos de rede e *hardware* programáveis.

O Capítulo 3 apresenta a revisão de literatura relacionada ao tema de pesquisa dessa tese. Os trabalhos relacionados à arquiteturas para encaminhamento NDN são comparados qualitativamente entre si enquanto os trabalhos relacionados à FIB são classificados de acordo com as estruturas de dados utilizadas. Ao final do capítulo as propostas para a NDN FIB que rodam em comutadores convencionais e comutadores programáveis são apresentadas separadamente, destacando suas características e limitações.

O Capítulo 4 fornece uma descrição detalhada da arquitetura FANTNet com foco nos elementos do plano de controle. A arquitetura proposta contém comutadores de borda

programáveis e comutadores convencionais que são utilizados para encaminhar o tráfego entre múltiplos domínios NDN. Além de descrever as funções desses comutadores, esse capítulo introduz também um elemento de rede utilizado para converter pacotes NDN nativos para um formato específico. As estruturas de dados do plano de controle necessárias para viabilizar o encaminhamento de pacotes NDN são apresentadas em detalhes bem como os algoritmos para a inserção, atualização e remoção de prefixos nomeados tanto nas tabelas do plano de controle quanto nas tabelas do plano de dados.

O Capítulo 5 descreve a CoFIB e demais componentes do plano de dados da FANTNet. Os detalhes sobre o formato de codificação de nomes proposto e o fluxograma para a extração de cabeçalhos são apresentados. No *pipeline* de processamento de entrada e saída, o conceito de moldes de prefixos, prefixos canônicos e prefixos conflitantes são introduzidos bem como a lógica de processamento de pacotes para garantir que a operação de LNPM seja realizada com sucesso. Nesse capítulo, apresenta-se também a heurística de posicionamento de tabelas proposta para aumentar a vazão e as técnicas de compressão de memória utilizando recursos nativos da linguagem P4.

O Capítulo 6 apresenta os procedimentos utilizados para avaliar a FANTNet, especificamente a estrutura de dados CoFIB. O método para a avaliação inclui análises quantitativas e qualitativas feitas por meio de *datasets* de nomes reais e sintéticos. A análise quantitativa provê resultados relacionados à taxa de descarregamento de prefixos canônicos, a probabilidade de encaminhamento incorreto devido à colisões de *hash*, o consumo de memória, e a vazão. Já a avaliação qualitativa consiste em comparar a arquitetura FANTNet e estrutura de dados CoFIB com os trabalhos correlatos em relação às suas características de desempenho.

O Capítulo 7 traz as considerações finais sobre a tese apresentada, bem como as limitações e os trabalhos futuros. Esse capítulo consolida as respostas para as questões apresentadas na Seção 1.5, onde as principais contribuições são destacadas novamente em conjunto com a listagem das principais publicações resultantes desse trabalho.

A Figura 1.1 mostra a relação entre as questões de pesquisa levantadas nesse capítulo e às seções da tese em que elas são endereçadas. Como parte das questões de pesquisa estão centradas no eixo concepção-estrutura-experimentação, algumas delas são tratadas em diferentes partes do texto. De fato, a FANTNet e a CoFIB são apresentadas seguindo uma abordagem *top-down*, onde a concepção, a estruturação e a experimentação dos elementos do plano de controle e do plano de dados ocorrem em diferentes locais ao longo do texto.

Capítulo 2

Fundamentação Teórica

2.1 Introdução

Desde a sua concepção, no início da última década, os principais componentes da arquitetura NDN vem sendo desenvolvidos em *software* (LI et al., 2019). Em função do baixo custo e da possibilidade de extensiva experimentação, as soluções em *software* permitem uma ampla variedade de testes antes da arquitetura atingir um nível de maturidade suficientemente bom para ser utilizada em larga escala. No entanto, para escalar a arquitetura NDN ao tamanho da Internet atual, além do desenvolvimento de módulos NDN em *software*, é necessário também focar em desempenho e viabilizar o processamento das estruturas de dados NDN em *hardware*. Com os recentes avanços em comutadores programáveis e linguagens de domínio específico como P4, é possível desenvolver protótipos de roteadores NDN em *hardware* sem que sejam necessários anos para desenvolver ASICs específicos para a arquitetura (BOSSHART et al., 2013).

A solução proposta nessa tese envolve a implementação de uma estrutura de dados para a FIB baseada em SDN. Dessa forma, esse capítulo visa descrever os fundamentos das redes NDN bem como apresentar uma visão geral de SDN e dos principais componentes da arquitetura de comutadores programáveis. Nesse sentido, a Seção 2.2 apresenta uma visão geral da NDN incluindo detalhes sobre os tipos de pacotes e seus formatos além das principais estruturas de dados de um elemento de rede NDN. A Seção 2.3 apresenta os fundamentos das redes SDN e de planos de dados programáveis, incluindo uma descrição em alto nível da linguagem P4 e as principais arquiteturas de *pipeline* de processamento de pacotes. Por fim, a Seção 2.4 traz as considerações parciais sobre esse capítulo.

2.2 Arquitetura NDN

A arquitetura NDN (ZHANG et al., 2014) é o resultado de anos de pesquisa empírica em redes de computadores para resolver as limitações da arquitetura TCP/IP. Inspirada no conceito de Redes Centradas na Informação (ICN) (AHLGREN et al., 2012), os ele-

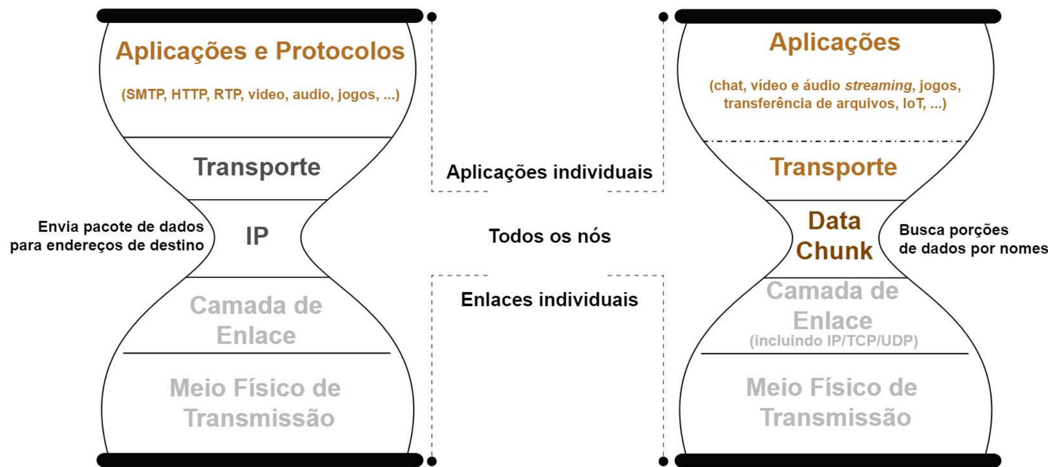

Figura 2.1 – Comparação entre as pilhas de protocolos IP e NDN (adaptado de Jacobson et al. (2009) e Afanasyev et al. (2018)).

mentos de uma rede NDN trocam informações através de Ipkts e Dpkts nomeados que dispensa a necessidade de identificar os nós comunicantes como ocorre na Internet atual. A Figura 2.1 mostra o modelo em camadas da arquitetura NDN em comparação com a pilha TCP/IP. A principal diferença entre as duas arquiteturas está no protocolo da camada de rede, onde ocorre uma mudança no modelo de comunicação tradicional centrado em sistemas finais (IP) para um modelo inovador centrado em dados que desacopla identificação de localização. Dessa forma, as redes NDN são capazes de garantir funcionalidades inexistentes na arquitetura TCP/IP como, por exemplo, *caching* em roteadores, segurança ao nível de dados e encaminhamento baseado em nomes.

2.2.1 Tipos de Pacotes

Na arquitetura NDN, os nós comunicantes são classificados em nós consumidores e nós produtores de maneira similar aos conceitos de sistemas finais clientes e sistemas finais servidores no modelo TCP/IP. Desse modo, a comunicação em NDN é sempre conduzida pelo consumidor através do envio de Ipkts requisitando determinados conteúdos, onde o produtor se encarrega de gerar e enviar ao consumidor os Dpkts correspondentes. Diferentemente da arquitetura TCP/IP, na qual os pacotes de um mesmo fluxo contém os mesmos identificadores (IPs e portas), em NDN os Ipkts e Dpkts contém um campo específico que identifica de forma unívoca as porções de dados transmitidas entre os nós NDN. Os pacotes NDN também contém campos específicos para garantir funcionalidades como segurança, *caching* na rede e transmissão *multicast*. A Figura 2.2 mostra os campos para cada tipo de pacote existente na arquitetura NDN.

A principal informação contida em um pacote NDN, seja ele de dados ou de interesse, é o campo de nome de conteúdo. De maneira similar ao conceito de *Uniform Resource Locator* (URL) existente nas redes TCP/IP, um nome NDN é formado por um prefixo

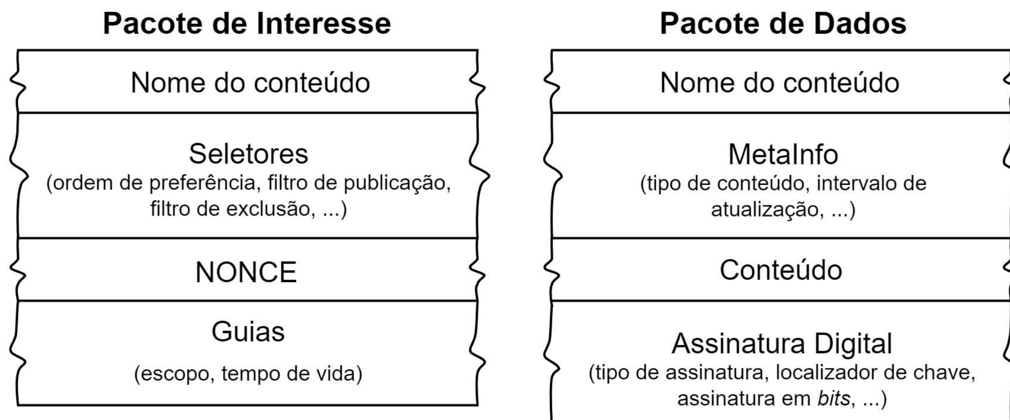

Figura 2.2 – Pacote de interesse e pacote de dados na arquitetura NDN (adaptado de Zhang et al. (2014)).

de tamanho variável, que faz referência ao produtor do conteúdo, e por um sufixo que identifica os segmentos de dados gerados por aquele produtor. A especificação NDN assume que os nomes são hierarquicamente estruturados. Assim, um vídeo produzido pela FACOM/UFU pode ter o nome */ufu/facom/videos/demo.mpg*, onde o caracter '/' separa os componentes nomeados. Essa estrutura hierárquica permite que as aplicações possam ser contextualizadas e que os elementos de dados possam ser correlacionados (ZHANG et al., 2014). Além disso, essa estrutura de hierarquia possibilita agregação de nomes, onde o prefixo */ufu* poderia corresponder ao sistema autônomo que produz o conteúdo. Desse modo, um nome em um pacote NDN assume diferentes papéis, podendo não apenas identificar um sistema final como também representar um *frame* em uma transmissão de vídeo de fluxo contínuo, identificar uma porção de dados correspondente à um arquivo ou codificar um comando para acender uma lâmpada em um sistema de Internet das Coisas (*Internet of Things* (IoT)) (ZHANG et al., 2014).

2.2.1.1 Formato de Pacotes

De acordo com a especificação da arquitetura NDN, os Ipkts e Dpkts são codificados por meio do formato TLV (NDN PROJECT TEAM, 2016). O formato TLV é um esquema de codificação genérico usado para representar elementos de informação em protocolos de comunicação que são opcionais e que não necessariamente precisam estar em uma ordem pré definida. Assim, a utilização do formato TLV dispensa a necessidade de cabeçalhos fixos nos pacotes. Devido a sua flexibilidade, além de ser empregada na codificação dos pacotes NDN, o formato TLV é adotado também em uma ampla variedade de protocolos de rede como nas extensões do *Transport Layer Security* (TLS) (IETF, 2018), nos parâmetros do *Border Gateway Protocol* (BGP) (IETF, 2006), em quadros *Quick UDP Internet Connection* (QUIC) (IETF, 2021), e na ASN.1 DER (ITU-T, 2021).

A codificação TLV corresponde a um conjunto de blocos que podem estar sequenciais

ou aninhados dentro de um pacote de rede. De maneira simplificada, um bloco TLV contém três campos que codificam uma porção de dados específica de um pacote. O primeiro campo é o *Type* e representa um código que identifica o tipo de informação contida no bloco. Esse campo contém valores pré-determinados que indicam a semântica dos dados armazenados no bloco (cadeia de caracteres, valor temporal, *hash*, número em ponto flutuante, etc). O segundo campo é o *Length* e indica o comprimento em *bytes* do campo de dados (*Value*) subsequente. Em protocolos que usam o TLV como TLS (IETF, 2018), BGP (IETF, 2006) e QUIC (IETF, 2021), os campos *Type* e *Length* são de comprimento fixo para facilitar a operação de extração de cabeçalhos (*parsing*). O último campo de um bloco TLV é o *Value* que corresponde à sequência de *bytes* dos dados do bloco, cujo comprimento é dado pelo campo *Length* e a semântica pelo campo *Type*.

Um pacote NDN é formado por uma coleção de blocos TLV_1, TLV_2, ..., TLV_n encapsulados em um bloco principal TLV_0 (ZHANG et al., 2014). Diferentemente de protocolos que usam o TLV em sua forma original, com os campos *Type* e *Length* de comprimento fixo, em NDN esses campos são definidos com um número variável de *bytes*. O objetivo dessa mudança é garantir maior flexibilidade no suporte a futuros protocolos, muito embora isso torna o processamento de pacotes NDN mais complexo. A regra para determinar o tamanho desses campos segue uma lógica simples. Caso o primeiro octeto para representar o campo seja menor que 252, o valor do campo é codificado em um único *byte*. Caso contrário, o primeiro octeto do campo assume os valores 253, 254 ou 255 seguidos de 2, 4 ou 8 *bytes*, respectivamente. Além disso, cada bloco TLV_i pode conter subblocos aninhados e também blocos em ordem arbitrária. No entanto, um princípio de projeto é manter a ordem de TLV_i sempre determinística e manter o nível de aninhamento o menor possível para minimizar a sobrecarga de processamento e as chances de erros (ZHANG et al., 2014). A Figura 2.3 mostra um Dpkt codificado em TLV que contém o nome "*/file/a*". Os valores sublinhados (em azul) representam os campos *Type*, os valores em vermelho indicam o campo *Length* e os demais o campo *Value*.

É importante observar que o formato de um pacote NDN não possui um cabeçalho de comprimento fixo nem possui um campo contendo a versão de protocolo. A utilização da codificação TLV proporciona a flexibilidade de adicionar novos tipos e eliminar tipos antigos conforme o protocolo evolui com o tempo. A ausência de um cabeçalho fixo também torna possível a arquitetura NDN suportar pacotes pequenos eficientemente com baixo *overhead*. Apesar de não haver suporte para fragmentação ao nível de rede, caso seja necessário é possível realizar a fragmentação e remontagem salto a salto, conforme em Afanasyev et al. (2015).

2.2.2 Plano de Dados

Em uma rede NDN, a comunicação ocorre quando um nó consumidor deseja obter dados de um nó produtor. Assim, o nó consumidor coloca o nome da porção de dados

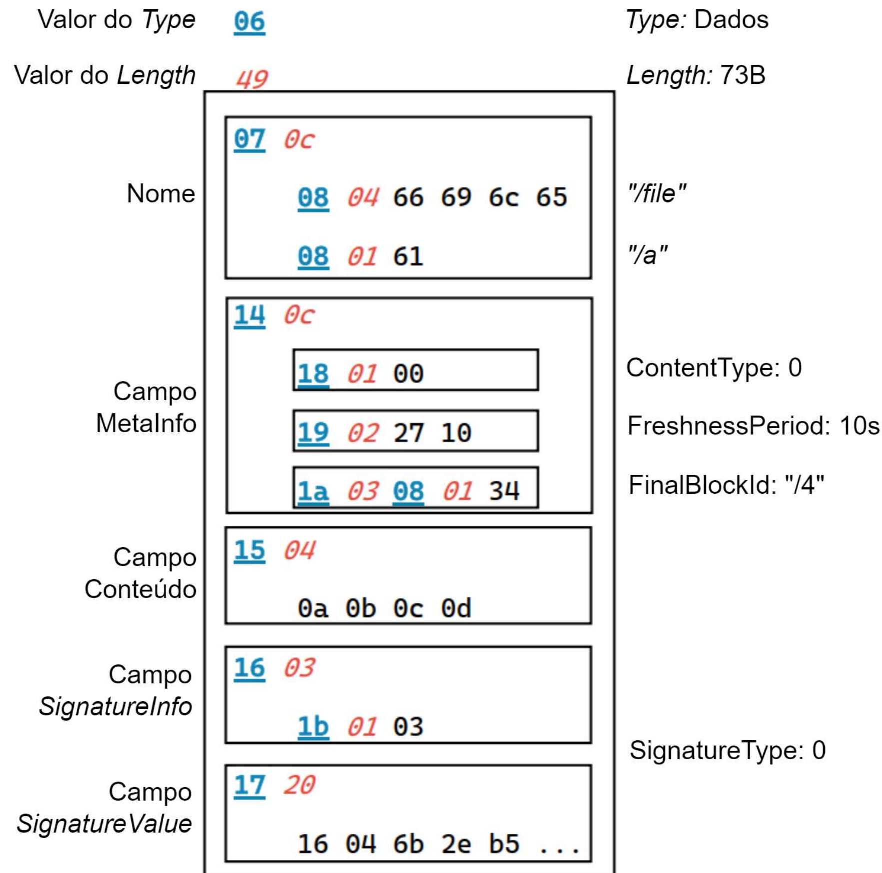

Figura 2.3 – Pacote de dados codificado em NDN (adaptado de Ma, Afanasyev e Zhang (2022).

desejada em um Ipkt e encaminha esse pacote para a rede. Os roteadores NDN ao longo do caminho usam o nome para encaminhar esse Ipkt até o nó produtor. Quando o Ipkt chega até um nó que contém o dado requisitado, esse nó encapsula o conteúdo em um Dpkt e encaminha-o de volta para o nó consumidor usando o caminho reverso.

Para obter dados gerados dinamicamente por produtores, os nós consumidores devem ser capazes de deterministicamente construir o nome para a porção de dados desejada. Essa tarefa pode ser realizada de duas formas. A primeira é através de um algoritmo determinístico que permite ao produtor e consumidor chegarem a um mesmo nome com base nas informações disponíveis a ambos. A segunda forma é através do campo de seletores disponível no Ipkt que, em conjunto com a compatibilização de prefixo nomeado mais longo (LNPM), permite que o nome seja construído iterativamente (ZHANG et al., 2014).

Para garantir o encaminhamento correto de Ipkts e Dpkts, os roteadores NDN ao longo do caminho implementam três tabelas no plano de dados. A primeira delas é a CS, responsável por fazer *cache* na rede de Dpkts. A segunda é a PIT, responsável por agregar Ipkts pendentes de resposta. A última é a FIB, que armazena prefixos nomeados juntamente com suas interfaces de saída. As subseções seguintes descrevem melhor o

papel de cada uma dessas tabelas.

2.2.2.1 Content Store

A CS é responsável por armazenar Dpkts em *buffers* nos roteadores NDN. Essa funcionalidade permite que Dpkts recém chegados ao roteador possam ser posteriormente disponibilizados para outros nós consumidores. Dessa forma, a CS garante a entrega de dados independente de localização, tendo em vista que não apenas o nó produtor pode responder a consultas mas qualquer roteador intermediário que mantém cópia dos Dpkts. Além disso, a CS desempenha também um papel importante na arquitetura NDN, no que diz respeito à provisão de Qualidade de Serviço (*Quality-of-Service* (QoS)) às aplicações, ao diminuir o tráfego no núcleo da rede e o RTT para a obtenção de Dpkts por nós consumidores.

Em geral, a CS consiste em uma tabela chaveada por um índice que aponta para um determinado Dpkt. Ao receber um Ipkt, o roteador NDN faz uma busca na CS utilizando o nome contido no pacote como índice. Caso a busca tenha sucesso, o pacote é retirado do *buffer* e é encaminhado para o consumidor através da interface de entrada do Ipkt. Caso a busca falhe, o pacote segue o fluxo normal no *pipeline*. Por outro lado, ao receber um Dpkt, um algoritmo de compatibilização exata de nome, conhecido na literatura como *Exact Name Matching* (ENM) (LI et al., 2019), é usado para verificar a existência ou não desse Dpkt na CS. Caso o pacote não exista, ele é armazenado dependendo dos parâmetros contidos no cabeçalho do pacote.

Em relação à operação de compatibilização de nomes na CS (*name lookup*), ao receber um Ipkt duas possibilidades podem ocorrer no roteador. Primeiro, o nome de algum Dpkt na CS pode ser idêntico ao nome contido no Ipkt. Uma segunda possibilidade é apenas uma parte do nome no Ipkt (prefixo) ser compatível com algum Dpkt na CS. Em ambos os casos, a arquitetura NDN prevê que a CS possa responder com o pacote que compatibilizou melhor segundo algum critério. Assim, o algoritmo de *lookup* por nome para essa situação é conhecido como *All Sub-Name Matching* (ASNM) (LI et al., 2019), o qual é definido como uma consulta a Dpkt cujo o nome começa com o prefixo contido no Ipkt. Utilizando o algoritmo ASNM, o roteador pode retornar qualquer Dpkt relacionado ao Ipkt assim que possível. Enquanto isso, esse algoritmo ajuda o consumidor a aprender mais sobre os nomes dos dados disponíveis e a contruir os nomes completos em futuros Ipktss (LI et al., 2019).

Embora seja importante a CS armazenar Dpkts por muito tempo, ela é uma estrutura de *cache* temporária (armazenamento não persistente) e o uso da CS deve levar em conta a quantidade de Dpkts e a capacidade de memória. Dessa forma, uma política de substituição de *cache* deve ser implementada para determinar quais conteúdos devem ser substituídos quando um Dpkt chega e a CS está cheia. Na literatura, existem três políticas de substituição principais. A primeira delas é a *First-In First Out* (FIFO), na qual

a substituição ocorre para o primeiro pacote que chegou na CS. O segundo critério é o baseado em popularidade, conhecido como *Least Recently Used* (LRU) e *Least Frequently Used* (LFU), onde o Dpkt menos popular ou acessado com menos frequência é substituído, respectivamente. Existe também o critério de substituição baseado em prioridades, na qual pacotes com baixa prioridade são substituídos primeiro (LI et al., 2019). Em função da necessidade de suportar a política de substituição de *cache* e também a alta frequência de acesso, a estrutura de dados mais utilizada para indexar a CS são as listas de salto (LI et al., 2019).

2.2.2.2 Pending Interest Table

Da perspectiva de um Ipkt recém chegado ao roteador NDN, quando a CS não contém o Dpkt correspondente, a PIT é a primeira a ser consultada para checar se existe algum Ipkt pendente que contenha o mesmo nome de conteúdo. De maneira geral, a PIT armazena todos os Ipkts pendentes que foram encaminhados mas os correspondentes Dpkts não chegaram ao roteador. Essa tabela tem a finalidade de evitar o envio de informações desnecessárias na rede, tendo em vista que apenas o primeiro Ipkt é enviado e os demais, que contém o mesmo nome, ficam pendentes na tabela. Dessa forma, assim como a CS, a PIT contribui para a diminuição do tráfego na rede.

Conforme visto na Figura 2.2, cada Ipkt carrega um *nonce* aleatório gerado pelo consumidor. Ao receber o Ipkt, tanto o nome de conteúdo quanto o *nonce* são registrados na PIT. Assim, caso o roteador NDN receba um Ipkt contendo um nome previamente armazenado, através do *nonce* ele é capaz de identificar se esse Ipkt é exatamente o mesmo enviado anteriormente ou se é um novo Ipkt proveniente do mesmo consumidor. Como os Dpkts seguem o caminho reverso, esse recurso na PIT evita laços infinitos de qualquer tipo de pacote NDN (YI et al., 2013).

Para exemplificar, quando um Ipkt com o nome */br/ufu/videos* chega ao roteador e não é satisfeito pela CS, a PIT usa o algoritmo ENM para criar ou atualizar uma entrada PIT com o nome */br/ufu/videos*, por meio da tupla ⟨*nome, lista de interfaces, lista de nonces, tempo de expiração*⟩. Por outro lado, quando um Dpkt chega contendo o nome */br/ufu*, por exemplo, o algoritmo *All Name Prefix Matching* (ANPM) (LI et al., 2019) é utilizado para encontrar todos os Ipkts pendentes cujos componentes fazem parte do prefixo contido no Dpkt recém chegado.

Considerando que o *lookup* é realizado na PIT independentemente se o pacote é de interesse ou dados, a PIT é altamente dinâmica (LI et al., 2019). Assim, para evitar que a PIT esgote toda a memória disponível em um curto período de tempo, as estradas na PIT precisam ser removidas após alguns instantes. Esse valor de *timeout*, tipicamente da ordem de alguns segundos, também fornece proteção contra ataques simples de *buffer overflow* (XIE; WIDJAJA; WANG, 2012). Dada a natureza dinâmica da PIT, conforme reportado por Xie, Widjaja e Wang (2012), em um cenário onde os Dpkts são corretamente

transmitidos em resposta a Ipkts, em uma razão de 50%, a frequência de operações na PIT é da ordem de 6 milhões por segundo. Assim, a frequência de operações de leitura e escrita na PIT é alta, o que sugere a utilização de estruturas de dados que atendem a esse requisito, como os filtros de *bloom* (LI et al., 2019).

2.2.2.3 Forwarding Information Base

A tabela FIB é uma estrutura de dados que desempenha um papel fundamental na arquitetura NDN, visto que ela é responsável pela interconexão entre nós consumidores e nós produtores quando a CS e a PIT não contém entradas compatíveis. De maneira simplificada, a FIB é uma tabela que contém prefixos nomeados associados a uma ou mais interfaces de saída. Quando Ipkts compatibilizam seus nomes com as entradas na FIB, eles são encaminhados para a interface de saída determinada pela estratégia de encaminhamento. Em caso de não compatibilização, os Ipkts podem ser encaminhados para alguma interface padrão para fins de gerenciamento ou serem descartados. As entradas na FIB são realizadas por meio de algoritmos de roteamento e da estratégia de encaminhamento.

Em relação à operação de compatibilização de nomes, somente Ipkts tem seus prefixos consultados na FIB. Assim, a FIB não é acessada tão frequentemente quanto a CS e a PIT. Dado que os nomes em NDN são hierarquicamente estruturados, o Ipkt deve ser encaminhado com base no resultado do algoritmo LNPM. Esse algoritmo é similar à tradicional compatibilização de prefixo mais longo (LPM) usado nas redes IP, exceto que os prefixos a serem compatibilizados são nomeados, de comprimento variável e pode haver mais que uma interface de saída por prefixo.

Quanto ao tamanho da NDN FIB, o número esperado de prefixos a serem armazenados é muito maior que a quantidade de prefixos IP em roteadores tradicionais (LI et al., 2019). Embora o número de entradas na FIB seja determinado pelo projeto de *namespace* e da efetividade da agregação de prefixos, pelo menos 10 milhões de nomes são esperados para lidar com uma rede da escala da Internet atual (YUAN, 2015). Song et al. (2015) sugerem a possibilidade de escalar a FIB para armazenar bilhões de prefixos nomeados.

A frequência de acesso à tabela FIB depende da taxa com que os Ipkts vão ser satisfeitos tanto na CS como na PIT. Em geral, a FIB requer mais operações de leitura do que de escrita, considerando que a FIB é responsável por ler a informação do próximo salto para um dado Ipkt. Em função da natureza hierárquica dos prefixos NDN, as estruturas de dados em *software* mais comuns usadas na implementação da FIB são as baseadas em *Trie* (LI et al., 2019) e em *hardware* são as tabelas de *hash*. A Figura 2.4 mostra o fluxo operacional de Ipkts e Dpkts em um roteador NDN típico.

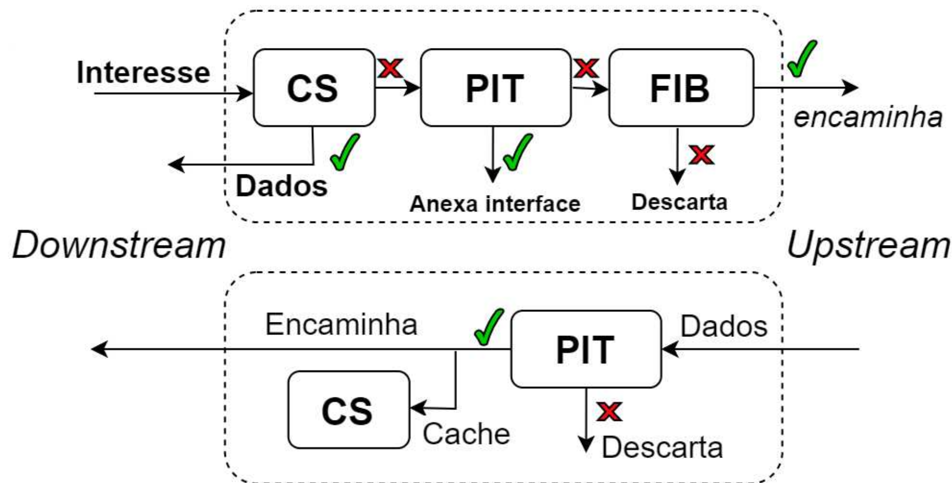

Figura 2.4 – Fluxo operacional no plano de dados de um roteador NDN (adaptado de Meddeb et al. (2018)).

2.2.3 Plano de Controle

O plano de controle na arquitetura NDN é responsável pela execução de funções relacionadas ao roteamento. Em NDN, algoritmos de roteamento convencionais de estado de enlace e de vetor de distâncias podem ser empregados de forma adaptada através do anúncio de prefixos nomeados ao invés de prefixos IP tradicionais. Contudo, existem algumas outras particularidades específicas associadas ao roteamento NDN como a possibilidade de cada prefixo estar associado a múltiplas interfaces de saída e o papel da estratégia de encaminhamento.

Quando o prefixo em um Ipkt compatibiliza na FIB, a operação de LNPM retorna um conjunto de interfaces pelas quais esse Ipkt pode ser enviado. A estratégia de encaminhamento é um conceito introduzido em Yi et al. (2013) que possibilita ao roteador NDN selecionar, com base em algum critério, um subconjunto de interfaces, dentre as possíveis, pelas quais o Ipkt será encaminhado. Em Ahdan, Situmorang e Syambas (2017), por exemplo, a estratégia de encaminhamento prevê enviar o Ipkt para todas as interfaces retornadas pela operação de LNPM. Por outro lado, na estratégia de encaminhamento proposta por Yi et al. (2014), o conjunto de cada prefixo compatibilizado na FIB é ranqueado e o pacote é enviado para a interface melhor posicionada no *ranking*. Enviar o Ipkt para múltiplas interfaces diminui o atraso para a obtenção do Dpkt mas aumenta a sobrecarga na rede. Por outro lado, enviar o pacote apenas para uma interface diminui a sobrecarga mas aumenta o atraso.

A função de roteamento em uma rede NDN é mais simples quando comparada ao roteamento IP em razão da estratégia de encaminhamento adotada pelos roteadores NDN. Enquanto que o roteamento em uma rede IP é responsável por determinar todas as rotas entre nós e também lidar com recuperação de falhas em enlaces, o roteamento em NDN é responsável apenas por criar rotas de longo prazo.

Em relação ao roteamento em NDN, uma das principais estruturas de dados do plano de controle é a tabela RIB. Essa tabela armazena e agrega rotas para conteúdos que foram aprendidos de várias fontes, incluindo protocolos de roteamento, rotas configuradas manualmente e anúncio de aplicações. Em razão da heterogeneidade das informações contidas na RIB, ela pode se tornar instável. Desse modo, apenas quando a RIB se torna consistente é que os prefixos são descarregados na FIB para permitir o rápido *lookup* no plano de dados durante o processo de encaminhamento (NEWBERRY; MA; ZHANG, 2021).

2.3 Paradigma SDN

Assim como a arquitetura TCP/IP, as redes NDN foram projetadas para serem descentralizadas e distribuídas. No entanto, esse modelo de comunicação torna o gerenciamento de rede e a engenharia de tráfego desafiadores. Em razão do tempo necessário para convergir algoritmos distribuídos, determinadas aplicações nessas arquiteturas podem ser impactadas negativamente como aquelas que desempenham funções relacionadas à roteamento, balanceamento de carga, detecção de falhas e replicação de dados. Assim, as redes SDN surgem como alternativas para lidar com essas limitações, trazendo maior flexibilidade no processamento de pacotes para redes localmente administradas.

As redes SDN surgiram por volta de 2009 (FEAMSTER; REXFORD; ZEGURA, 2014) trazendo o conceito de separação física entre os planos de dados e de controle, que eram originalmente integrados com o *hardware* de rede. Conforme mostra a Figura 2.5, o princípio de operação em uma rede SDN envolve um programa logicamente centralizado, normalmente escrito em uma linguagem de alto nível (p. ex., Python, Java, C), que toma decisões lógicas sobre como encaminhar pacotes em cada comutador da rede. O canal de comunicação entre o *software* de alto nível e o *hardware* subjacente pode ser qualquer coisa que o dispositivo de rede seja capaz de entender. Como exemplo, os pri-

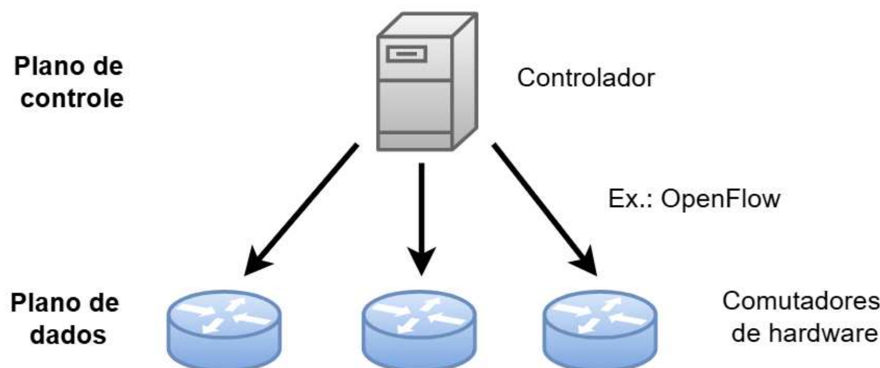

Figura 2.5 – Separação dos planos de dados e controle na SDN (adaptado de (TANENBAUM; FEAMSTER, 2019)).

meiros controladores SDN usavam o próprio BGP como plano de controle (FEAMSTER; BORKENHAGEN; REXFORD, 2003). Após um período de experimentação e testes, tecnologias como OpenFlow (MCKEOWN et al., 2008), YANG (IETF, 2010) e NETCONF (IETF, 2011) surgiram como formas mais flexíveis de comunicar informações do plano de controle com dispositivos de rede (TANENBAUM; FEAMSTER, 2019).

O protocolo OpenFlow (MCKEOWN et al., 2008) é o padrão atualmente utilizado em SDN para viabilizar a comunicação entre os planos de controle e de dados. A ideia inicial do OpenFlow era permitir a gravação em uma memória endereçavel por conteúdo que age como uma tabela de combinação-ação. Tabelas de combinação-ação possibilitam que um comutador possa identificar pacotes por meio de um ou mais campos em seus cabeçalhos (p. ex., endereço MAC, endereço IP). Ao encontrar correspondência, os comutadores são capazes de executar um conjunto de ações simples baseadas nos valores desses campos como encaminhar o pacote para uma porta específica, enviar o pacote para o controlador de *software* centralizado ou descartá-lo (TANENBAUM; FEAMSTER, 2019).

A primeira versão do protocolo OpenFlow (versão 1.0) tinha uma única tabela de combinação-ação, na qual as entradas podiam se referir a correspondências exatas em campos de cabeçalho de pacotes padronizados ou entradas curinga. Versões posteriores do OpenFlow (p. ex., OpenFlow 1.3) adicionaram operações mais complexas, incluindo cadeias de tabelas e suporte a novos protocolos de rede. Apesar de incorporar várias melhorias importantes, poucos fornecedores implementaram essas novas versões em seus equipamentos, muito em função da complexidade em expressar a semântica necessária para processar corretamente os pacotes. A utilização de combinações genéricas de conjunções AND e OR para definir a lógica de encaminhamento se mostrou uma tarefa difícil, especialmente para programadores, tendo em vista que não havia uma linguagem de programação de alto nível para expressar comportamentos específicos. Assim, a utilização do protocolo OpenFlow acabou sendo limitada (TANENBAUM; FEAMSTER, 2019).

É importante ressaltar que o propósito inicial do protocolo OpenFlow não era possibilitar um controle flexível da rede, mas sim tornar as APIs de equipamentos de redes abertas. Os comutadores de rede da época já possuíam tabelas de pesquisa baseadas em TCAM e o OpenFlow surgiu como uma iniciativa de mercado para permitir que programas externos escrevessem nela. Porém, não demorou muito para que pesquisadores em redes começassem a considerar formas mais flexíveis de controle no plano de dados por meio de um projeto de *hardware* de rede mais elaborado (TANENBAUM; FEAMSTER, 2019).

2.3.1 Plano de Dados Programável

Como forma de contornar as limitações do OpenFlow, os avanços subsequentes em SDN visaram tornar o próprio *hardware* de rede programável. A partir de 2014, uma série de inovações em *hardware* programável, tanto em *Network Interface Card* (NIC)

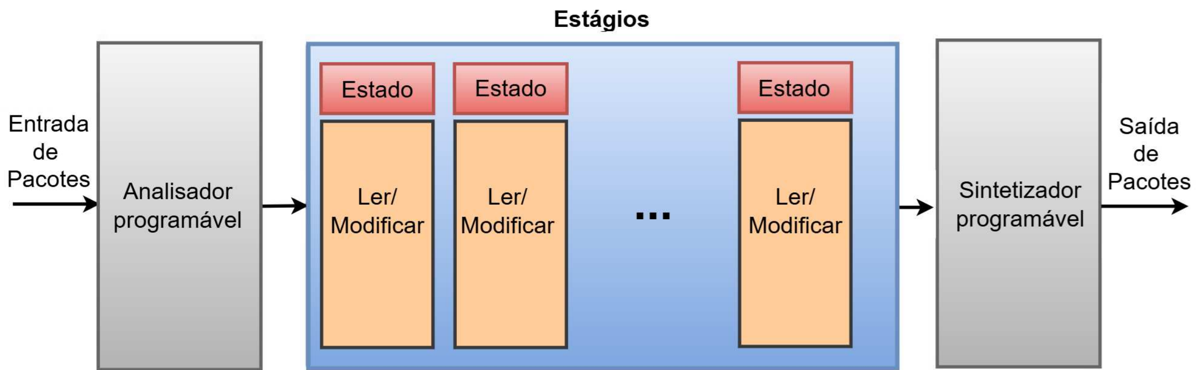

Figura 2.6 – *Pipeline* reconfigurável de combinação-ação para um plano de dados programável (adaptado de (TANENBAUM; FEAMSTER, 2019)).

quanto em comutadores físicos e *software switches*, permitiu que os operadores de rede personalizassem e customizassem não somente o formato de pacotes mas também toda a lógica de encaminhamento de pacotes.

A arquitetura geral de um elemento de rede programável é chamada de arquitetura de *switch* independente de protocolo (*Protocol-Idependent Switch Architecture* (PISA)), conforme mostra a Figura 2.6. Também conhecida como tabelas de combinação reconfiguráveis (RMT) (BOSSHART et al., 2013), a arquitetura PISA inclui estruturas para reconhecer, extrair e construir cabeçalhos de pacotes à taxa de linha de forma totalmente customizável. Essa arquitetura possui também um conjunto fixo de *pipelines* de pocessamento, cada um com bancos de memória SRAM e TCAM para tabelas de combinação-ação personalizadas, alguma quantidade de memória de registrador e unidades de processamento (*Arithmetic Logic Unit* (ALU)) para a execução de operações lógicas e aritméticas convencionais. Essa arquitetura resolve o problema da limitação de protocolos que o OpenFlow suporta e isso torna possível o processamento de pacotes em formatos não padronizados à altas taxas, como é o caso da arquitetura NDN.

Embora a arquitetura PISA contenha a especificação de como um comutador programável pode ser implementado, ela não é instanciada em nenhum equipamento físico ou virtual. Na realidade, a arquitetura PISA é um modelo de comutador programável que os fabricantes de equipamentos de redes podem utilizar para criar suas próprias arquiteturas. As arquiteturas proprietárias, também chamadas de *targets*, podem conter um número variado de *pipelines* de entrada e saída, com diferentes quantidades de memória SRAM e TCAM em cada estágio, e podem conter recursos específicos como a possibilidade de clonar pacotes do bloco de entrada para o bloco de saída, resubmeter pacotes que passaram pelo *pipeline* para o bloco de saída novamente, e recircular pacotes múltiplas vezes no bloco de entrada. Exemplos de arquiteturas específicas incluem o modelo de referência *v1model*, a arquitetura de comutador portável (*Portable Switch Architecture* (PSA)) (ONF, 2022b) e a arquitetura de NIC portável (*Portable NIC Architecture* (PNA)) (ONF,

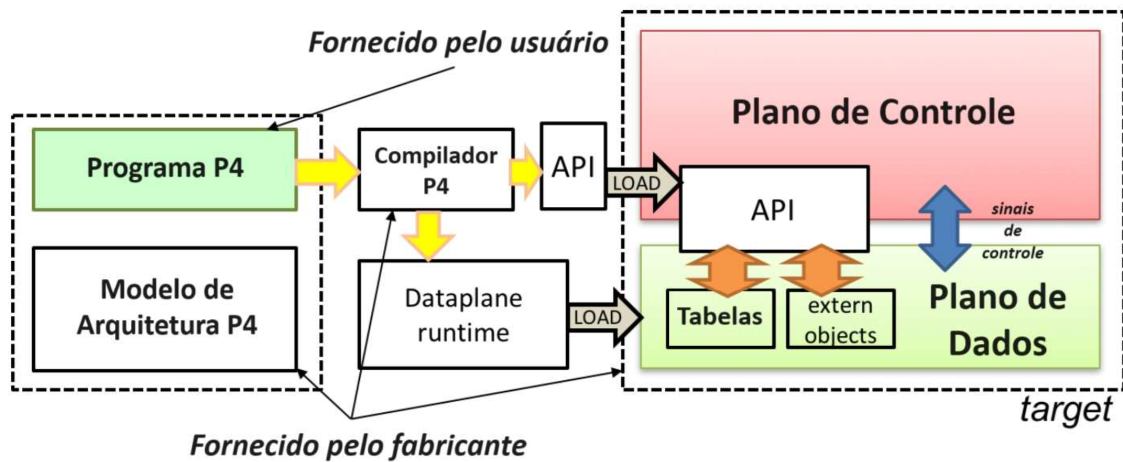

Figura 2.7 – Compilando um programa P4 (adaptado de (ONF, 2023)).

2022a).

Dentre as arquiteturas de planos de dados programáveis para comutadores físicos, a mais prevalente é a *Tofino Native Architecture* (TNA) (KUMAR, 2021). Essa arquitetura é uma iniciativa da *Barefoot Networks*, hoje incorporada pela Intel, em tornar aberta a especificação do *chipset* programável *Barefoot Tofino* (Intel Tofino). Essa arquitetura permite o processamento customizado de pacotes na entrada e saída de um comutador programável, independentemente do protocolo. Essa capacidade de processamento personalizado em ambas as direções possibilita uma série de análises detalhadas em pacotes como o tamanho e quantidade de tempo de espera em filas, bem como o encapsulamento e desencapsulamento customizados. O então ASIC *Barefoot Tofino*, hoje Intel Tofino, também permite o gerenciamento ativo de filas de saída, com base em metadados disponíveis nas filas de entrada (TANENBAUM; FEAMSTER, 2019).

2.3.2 Linguagem P4

A configuração de um *pipeline* de processamento de pacotes em uma determinada arquitetura de elemento de rede programável é realizada por meio de linguagens de programação de alto nível e de propósito específico. Embora existam outras linguagens para a programação do *pipeline* de processamento de pacotes, como a Frenetic (FOSTER et al., 2011) e DOMINO (SIVARAMAN et al., 2016), a linguagem P4 vem se tornando padrão em expressar como pacotes são processados pelo plano de dados de um elemento de encaminhamento programável, seja ele um NIC, *Field-Programmable Gate Array* (FPGA), *software switch*, ou ASIC.

A Figura 2.7 mostra o processo de configuração de um dispositivo de rede programável. O usuário ou operador escreve um programa P4 incorporando elementos externos e específicos de uma determinada arquitetura. Esse programa é compilado gerando tanto a configuração do plano de dados quanto as APIs para o plano de controle. A

P4Runtime (P4RT) (ONF, 2020) é a principal API utilizada na comunicação entre os planos de dados e controle. A P4RT padroniza o gerenciamento de planos de dados de maneira independente de fornecedor. Isso faz com que programas escritos em P4 possam ser compilados para diferentes dispositivos. A P4RT permite operações para inserções de regras, exclusões ou atualizações de elementos do plano de dados de um dispositivo P4. Para viabilizar isso, a P4RT depende do gRPC (GRPC, 2024), uma estrutura de alto desempenho para chamadas de procedimentos remotos desenvolvida pela Google.

A linguagem P4 é agnóstica quanto ao protocolo, mas permite ao programador expressar um conjunto rico de formato de pacotes e comportamentos do plano de dados. Para atingir esse objetivo, a linguagem provê uma série de abstrações, dentre as quais as principais são:

- **Tipos de cabeçalhos**: São estruturas (p. ex., *struct* em C) que permitem representar o formato de cada cabeçalho de pacote e seus campos de comprimento fixo e variável.

- **Parsers**: Também chamados de extrator ou analisador de cabeçalhos, conciste em máquinas de estado finitas que descrevem o fluxograma para extrair cada cabeçalho específico dentro de um pacote, levando em consideração o comprimento de seus campos.

- **Tabelas**: As tabelas P4 são estruturas de armazenamento abstratas que associam chaves definidas pelo usuário com determinadas ações. O compilador configura a profundidade de cada tabela de acordo com valores definidos pelo usuário e a largura da tabela é determinada pelo comprimento da chave.

- **Ações**: São fragmentos de código que descrevem como campos de cabeçalhos de pacotes ou metadados são manipulados. As ações podem incluir valores de parâmetros que são recebidos do plano de controle em tempo de execução.

- **Unidades de Combinação-Ação**: Desempenha três operações básicas. Primeiramente, essas unidades constroem as chaves de compatibilização a partir dos campos do pacote ou de valores nos metadados computados. Após o cálculo da chave, essas unidades realizam a pesquisa na tabela correspondente. Finalmente, caso ocorra compatibilização, a ação correspondente é executada juntamente com seus parâmetros.

- **Fluxo de controle**: Essa estrutura expressa um programa imperativo que descreve o processamento de pacotes em um *target*, incluindo a sequência de chamadas à unidades de combinação-ação. A remontagem de pacotes (*deparsing*) também pode ser realizada usando o fluxo de controle.

- **Externs**: Objetos *externs* são elementos específicos da arquitetura que podem ser manipulados por programas P4 através de APIs bem definidas, mas cujo comportamento interno é *hard-wired* e não programável (por exemplo, unidades de *checksum*). Os *externs* são fundamentais para estender a linguagem P4 de forma a possibilitar que operações não definidas pela especificação sejam invocadas dentro do código.

- **Metadados**: São estruturas de dados associadas a cada pacote que possibilita a passagem de valores entre os vários estágios do *pipeline* de processamento. Os metadados podem ser definidos pelo usuário ou definidos pela arquitetura. Metadados definidos pelo usuário podem incluir variáveis que devem ser persistidas para cada pacote ao longo do processamento e os metadados intrínsecos da arquitetura são aqueles que não podem ser modificados como, por exemplo, a porta de entrada de um pacote, o comprimento da fila, etc.

Em razão dos requisitos de latência de processamento de pacotes serem bem estritos no plano de dados, a linguagem P4 foi projetada para não conter elementos tipicamente encontrados em outras linguagens de programação de alto nível como *loops*, ponteiros e recursividade. Assim, códigos escritos em P4 fazem com que os pacotes sejam processados de maneira determinística a fim de garantir taxas da ordem de *Tbps*.

Por outro lado, em comparação com sistemas de processamento de pacotes de última geração, como aqueles baseados na gravação de microcódigo em *hardware* personalizado, a linguagem P4 oferece várias vantagens significativas. A primeira delas é a flexibilidade. Através da linguagem P4 várias políticas de encaminhamento de pacotes podem ser expressas por meio programas simples em alto nível. Isso contrasta com tecnologias de comutadores tradicionais, que expõem mecanismos de encaminhamento de função fixa para seus usuários. Além disso, a expressividade da linguagem P4 torna possível a escrita de algoritmos de processamento de pacotes sofisticados e independentes de *hardware* usando apenas operações lógicas e aritméticas de propósito geral e consultas a tabelas. Esses programas podem ser portáveis para diferentes *targets* de *hardware* desde que eles implementem as mesmas arquiteturas. Portanto, o processamento de pacotes na arquitetura NDN tem muito a se beneficiar ao utilizar a linguagem P4 e isso contribui para a aceleração da implantação da arquitetura em nível global.

2.4 Considerações

Esse capítulo forneceu uma visão abrangente dos principais elementos da arquitetura NDN e do paradigma SDN. Embora concebidos como modelos de redes para resolver problemas distintos, os pontos comuns de ambos foram explorados com foco nos aspectos do plano de dados e plano de controle. Os principais componentes da arquitetura NDN abordados foram as tabelas CS, PIT e FIB enquanto o paradigma SDN foi abordado em

relação às arquiteturas de plano de dados programáveis e a linguagem P4. O próximo capítulo aprofunda esses temas, descrevendo os trabalhos relacionados envolvendo arquiteturas para o encaminhamento NDN integradas à SDN e as estruturas de dados para a NDN FIB.

Capítulo 3

Trabalhos Relacionados

3.1 Introdução

As arquiteturas propostas na literatura para acelerar o tráfego NDN envolvem projetar as tabelas internas de um roteador NDN (CS, PIT e FIB). Essas tabelas são tipicamente implementadas utilizando estrutura de dados de propósito geral como as *Tries*, Filtros de *Bloom* e Tabelas de *Hash* (LI et al., 2019). Em relação à tabela FIB, inúmeras soluções em *hardware* e *software* vem sendo propostas para acelerar o *lookup* baseado em nomes. Algumas soluções focam nos aspectos do plano de dados enquanto outras incorporam uma visão arquitetural mais completa, envolvendo também o plano de controle.

Esse capítulo faz um levantamento dos principais trabalhos correlatos e está está estruturado da seguinte maneira. Inicialmente, a Seção 3.2 apresenta os trabalhos relacionados às arquiteturas para o encaminhamento NDN baseadas em SDN. Posteriormente, a Seção 3.3 taz os trabalhos relacionados às estruturas de dados para a NDN FIB, onde eles são classificados e comparados qualitativamente em função de suas características e limitações. Por fim, a Seção 3.4 traz as considerações parciais desse capítulo.

3.2 Arquiteturas para Encaminhamento NDN

Para resolver as limitações do modelo de comunicação descentralizado da arquitetura NDN, vários trabalhos propõem integrar arquiteturas do tipo ICN/NDN com redes SDN. Os trabalhos apresentados nessa seção envolvem as arquiteturas mais proeminentes para encaminhamento de tráfego NDN utilizando modelos centralizados e que envolvem programabilidade no plano de dados.

Adrichem e Kuipers (2015) propõem o NDNFlow, uma implementação em *software* livre da arquitetura NDN baseada em SDN. Em relação à infraestrutura, eles propõem uma topologia de rede heterogênea composta por dois tipos de encaminhadores de pacotes: 1) Comutadores OpenFlow-ICN e 2) Comutadores OpenFlow tradicionais. Ao receberem Ipkts, os comutadores de borda NDN reencaminham esses pacotes para um módulo de

gerenciamento NDN dentro do controlador centralizado. Esse módulo é responsável por definir uma rota NDN fim-a-fim entre a origem o destino utilizando o protocolo OpenFlow e os dados são carregados posteriormente em quadros Ethernet. Para validar a proposta, os autores realizaram experimentos comparando o desempenho da NDN com relação ao protocolo IP em diferentes configurações. Comparado com outras implementações SDN, os autores argumentam que o NDNFlow é arquiteturalmente menos complexo de desenvolver, fácil de extender novas funcionalidades e aplicável em ambientes com múltiplos encaminhadores de pacotes. No entanto, o NDFlow não permite alguns recursos nativos da NDN como o encaminhamento baseado em nomes na FIB.

A proposta NDNFab (MADUREIRA et al., 2021) é uma solução que combina modelos centrados em conteúdo com modelos baseados em caminhos. Para conseguir isso, a NDNFab utiliza roteadores NDN na borda da rede, com operação baseada em SDN no núcleo. Os roteadores NDN são projetados usando comutadores programáveis na borda e o encaminhamento de tráfego no núcleo é realizado utilizando *Segment Routing* (SR). A principal limitação da NDNFab é que a FIB não é descarregada para o comutador programável de borda. Em vez disso, uma FIB global é implantada no plano de controle de maneira centralizada onde o LNPM é executado em *software*, o que compromete a taxa de transferência de Ipkts na FIB.

Para melhorar o desempenho do encaminhamento de pacotes em NDN, Karrakchou, Samaan e Karmouch (2020a) propõem a *Enhanced NDN* (ENDN), uma arquitetura para encaminhamento NDN que utiliza SDN de maneira integrada com a linguagem P4. O principal objetivo do plano de controle do ENDN é fornecer às aplicações um catálogo de serviços de entrega de conteúdo. Esses serviços são traduzidos em configurações de plano de dados que estão instalados nos comutadores de rede. As aplicações vinculam seus *namespaces* para uma lista de serviços de rede selecionados no catálogo de serviços ENDN. Esses serviços são totalmente programáveis e extensíveis via gerador de código P4 no plano de controle. No entanto, a FIB em ENDN é implementada por meio da estrutuda de dados FCTree (KARRAKCHOU; SAMAAN; KARMOUCH, 2020b), conforme será visto na seção 3.3.2, o que limita a utilização da ENDN em cenários onde milhões de nomes devem ser armazenados na memória *on-chip* dos comutadores programáveis.

No trabalho de Takemasa, Koizumi e Hasegawa (2021) é implementado um protótipo de roteador NDN utilizando o comutador programável Intel Tofino. Chamada aqui de NDN-Tofino, essa solução é a primeira tentativa de implementar a NDN em comutadores programáveis. Embora isso aumente o desempenho, somente os Dpkts são encaminhados no plano de dados (ASIC). Os Ipkts são redirecionados para um encaminhador NDN no plano de controle para serem processados em *software*. Pegasus (LONG et al., 2023) é outra proposta de roteador NDN baseado no Intel Tofino. Semelhante ao NDN-Tofino, o Pegasus combina o comutador programável com servidores comuns para realizar o encaminhamento de pacotes NDN em alta velocidade. No entanto, Pegasus

Tabela 3.1 – Arquiteturas baseadas em SDN para o encaminhamento de tráfego NDN.

Arquitetura	**Características Gerais**	**Características de Desempenho**				
		FIB na SRAM	Utiliza P4	Co-Design HW/SW	Latência por Pacote (*ns*)	Escalável (HW)
NDNFab	• Comutadores programáveis na borda • Comutadores híbridos no núcleo • FIB na DRAM no plano de controle • Encaminhamento via *Segment Routing* (SR)	○	●	●	○	○
ENDN	• Comutadores programáveis em todo domínio • FIB emprega a estrutura FCTree • Utiliza *externs* para processar *strings*	●	●	○	○	○
NDNFlow	• Comutadores OpenFlow em todo domínio • Canal de comunicação separado para ICN e IP • Tunelamento de tráfego entre comutadores não OpenFlow	●	○	○	●	○
Pegasus	• Comutadores programáveis em todo domínio • Descarrega o *parser* no plano de dados • Implementa a FIB no plano de controle	○	●	●	●	○
NDN-Tofino	• Comutadores programáveis em todo domínio • Somente Dpkts são processados no plano de dados • A FIB é armazenada na DRAM	○	●	●	●	○

acelera o encaminhamento descarregando a lógica de *parsing* dos Ipkts para o plano de dados. A abordagem de implementar a FIB usando memória *Dynamic Random Acess Memory* (DRAM), como nas soluções NDN-Tofino e Pegasus, pode facilmente escalar para milhões e até mesmo bilhões de prefixos nomeados. Todavia, essa abordagem trás um impacto severo no desempenho de *lookup*, aumentando a latência da operação LNPM para a ordem de microsegundos ou mesmo milisegundos.

A Tabela 3.1 traz um resumo sobre as principais arquiteturas de encaminhamento NDN via SDN destacando as principais características de cada proposta. As características incluem se a FIB, ou parte dela, é armazenada na SRAM, se a arquitetura é programável via P4, se existe uma abordagem de *Co-Design* para distribuir o processamento entre os planos de dados e controle, se a solução garante processamento por pacote na ordem de nanosegundos e se a solução é escalável em *hardware*. Em função da limitação de

memória *on-chip* disponível em comutadores programáveis modernos, o termo escalável em *hardware* usado nessa tese refere-se a possibilidade de armazenamento de prefixos nomeados na ordem de milhões.

3.3 Estruturas de Dados para a NDN FIB

3.3.1 FIB baseada em Tabelas de *Hash*

As tabelas de *hash* (HT) são estruturas de dados compactas que armazenam pares de chave-valor. Geralmente, a HT é composta de um dicionário ou *array* associativo e uma função de mapeamento chamada função *hash*, onde essa função é responsável por calcular as chaves associando-as a valores. Quando uma chave precisa ser inserida, a HT utiliza essa função de *hash* para mapear a chave em um *slot* correspondente ao valor e a chave é armazenada nesse *slot*. Como a HT oferece a vantagem da pesquisa rápida de nomes, muitas soluções para a NDN FIB baseadas em *hash* vem sendo propostas.

O primeiro roteador baseado em conteúdo que suporta o encaminhamento por nomes em alta velocidade usando HT é o Caesar (PERINO et al., 2014). O roteador Caesar distribui a FIB em diferentes placas de linha e realiza a LNPM separadamente para cada subgrupo de prefixos. A divisão da FIB entre diferentes placas de linha otimiza o uso de memória mas aumenta a complexidade do LNPM e o número de operações de comutação por pacote. Para equilibrar o consumo de memória e a velocidade de comutação os autores do Caesar utilizam tabelas de *hash* baseadas na função crc64 para armazenar os prefixos na FIB. No entanto, o uso de funções de *hash* de 64 *bits*, embora possam ser mais resistentes quanto à colisões, tendem a consumir mais memória.

Li et al. (2014) propõem um algoritmo de LNPM baseado em um esquema de redução do espaço de nomes para otimizar o consumo de memória. O algoritmo proposto por Li et al. (2014) utiliza uma combinação das estruturas de dados *fatTree*, tabelas de *hash* tradicionais e vetores. Através dessa estrutura de dados híbrida e um algoritmo de busca otimizado é possível aumentar o número de prefixos que a FIB pode armazenar além de melhorar a taxa de processamento de Ipkts. No entanto, ao utilizar múltiplas estruturas de dados, as informações redundantes sobre os prefixos armazenados impactam o consumo de memória.

SACS (HUANG; WANG, 2018) é um outro método que foca na operação de LNPM, assim como em Li et al. (2014). Os prefixos em SACS são transformados em moldes antes de serem armazenados. O molde definido em SACS é variável e consiste em uma sequência de valores numéricos que representam a quantidade de caracteres no prefixo. Esses moldes são armazenados em memória TCAM que apontam para tabelas de *hash* armazenadas em memória SRAM. Quando comparada à métodos tradicionais baseados em *hash*, a abordagem usada no SACS melhora a latência considerando que vários acessos

à memória são eliminados devido a filtragem pelos moldes. Contudo, o SACS requer mais memória quando comparado à soluções em *software* como o NFD (NDN PROJECT TEAM, 2019), por exemplo. Isso porque a solução SACS necessita manter tanto o molde quanto a máscara em um *slot* da tabela de *hash*. Além disso, a utilização de moldes de comprimento variável dificulta o processamento em *hardware*.

A NDN-DPDK (SHI; PESAVENTO; BENMOHAMED, 2020) é uma solução que implementa a FIB em *software* usando uma tabela de *hash* chaveada pelos prefixos nomeados. Um algoritmo LNPM de 2 estágios é proposto para acelerar o encaminhamento à até 100 Gbps durante a execução em uma máquina *x86*. A proposta NDN-DPDK incorpora vários aprimoramentos arquiteturais, incluindo algoritmos e estruturas de dados eficientes, além de um *kernel* otimizado para reduzir a sobrecarga envolvida nas chamadas de sistema. Todavia, na proposta NDN-DPDK, a *thread* de entrada responsável por processar os Ipkts se torna o gargalo quando 8 ou mais *threads* de encaminhamento são usadas, o que impacta a escalabilidade da solução. Além disso, o NDN-DPDK não fornece nenhuma análise de consumo de memória para avaliar sua viabilidade quando milhões de prefixos nomeados são armazenados.

NDN.p4 (SIGNORELLO et al., 2016) é a primeira implementação de um roteador NDN usando a linguagem P4. A FIB em NDN.p4 é projetada como uma única tabela de combinação-ação no *pipeline* de ingresso e é configurada para armazenar entradas de largura fixa. Cada entrada na tabela FIB em NDN.p4 é decomposta em um número de subentradas proporcional ao número de componentes do nome. Por exemplo, o prefixo */a/ab/abc* é decomposto em três *hashes* de 16 *bits* de acordo com cada possível subprefixo ($h(/a)$, $h(/a/ab)$ e $h(/a/ab/abc)$), como em Agrawal e Sherwood (2008). A operação de LNPM é realizada em um ciclo usando uma operação de compatibilização ternária na TCAM. Porém, para cenários onde milhões de prefixos nomeados devem ser armazenados, o uso de entradas adicionais na FIB para cada prefixo pode exceder a capacidade de memória TCAM disponível no ASIC do comutador, principalmente em razão da NDN.4 utilizar apenas os recursos de processamento e memória disponíveis no *pipeline* de ingresso. Além disso, para lidar com o alto número de potenciais colisões de *hash* causadas pelo uso da função crc16, tanto o consumo de memória como a latência de processamento aumentam significativamente quando milhões de prefixos são armazenados na FIB.

Como forma de resolver os problemas de escalabilidade da proposta NDN.p4, um método baseado em trilha de *hash* (HBM) é proposto por Miguel, Signorello e Ramos (2018) para reduzir o consumo de memória TCAM. A ideia principal do método HBM é gerar uma entrada FIB para cada prefixo concatenando *hashes* dos componentes nomeados em sequência. Assim, a solução HBM é capaz de reduzir significativamente o consumo de memória. No entanto, como em NDN.p4, a medida em que o número de entradas na FIB aumenta, o uso da função de *hash* crc16 aumenta a probabilidade de colisão e isso impede a adoção da solução HBM em redes de larga escala (ROSA; SILVA, 2022a).

Uma abordagem similar a técnica HBM é apresentada por Guo et al. (2021), referenciada aqui como NDN-P4-SDN. Contudo, os autores focam no mecanismo de roteamento por meio de um controlador SDN onde os comutadores programáveis são capazes de reconhecer e encaminhar pacotes *Named-data Link State Routing* (NLSR) para o plano de controle. Assim como em HBM, em Guo et al. (2021) a tabela FIB é armazenada no *pipeline* de ingresso e a chave para realizar a operação de *lookup* na FIB é obtida fazendo-se a concatenação de *hashes*. Desse modo, Guo et al. (2021) herda as mesmas limitações de HBM no que diz respeito ao consumo de memória e colisões de *hash*.

Embora o foco permaneça na rede centrada em conteúdo (CCNx) (IETF, 2019a; IETF, 2019b), em Ooka e Asaeda (2023) os autores propõem a U-Table, uma tabela que unifica a FIB, PIT e a CS em uma única tabela. A U-Table é baseada em FPGA e pode armazenar até 10 milhões de prefixos nomeados por meio de uma estrutura de dados baseada em tabela de *hash*. No entanto, os prefixos nomeados são armazenados na DRAM que, embora tenha grande capacidade, possui velocidade de acesso inferior à memórias do tipo SRAM.

3.3.2 FIB baseada em *Trie*

As *Tries* são estruturas de dados semelhantes à árvores binárias e são tipicamente usadas para armazenar e recuperar *strings* e dicionários. Em uma *Trie*, cada nó armazena um caractere do nome ou um componente nomeado de um prefixo. Os nós são organizados hierarquicamente, de modo que prefixos comuns compartilhem os mesmos caminhos na árvore. Isso permite uma busca eficiente por correspondências prefixadas de nomes de dados. Assim, as *Tries* podem reduzir o consumo de memória da FIB agregando subprefixos comuns e suportando naturalmente a operação de LNPM.

Uma das primeiras implementações da FIB a usar *trie* é o *Parallel Name Lookup* (PNL) (WANG et al., 2011). Nessa solução os autores desenvolvem um protótipo em *hardware* para a FIB que se baseia na estrutura de dados *Name Prefix Trie* (NPT). De forma simplificada, a NPT é uma estrutura de dados do tipo *trie*, desenvolvida em *software*, onde cada componente do nome é representado por uma aresta e cada nó indica um estado de *lookup*. A NPT é a estrutura de dados padrão quando se trata de armazenar prefixos da FIB usando *trie*. Na NPT, o *lookup* é realizado através de uma busca em profundidade na *trie*. A NPT consegue reduzir o consumo de memória mesmo fazendo o uso de ponteiros uma vez que nomes que compartilham os mesmos prefixos terão os componentes comuns armazenados apenas uma vez. No entanto, como toda estrutura do tipo *trie*, a profundidade média da NPT é relativamente alta, o que influencia negativamente a velocidade de *lookup*. A solução PNL utiliza o paralelismo de *hardware* do roteador para justamente reduzir o tempo de *lookup*. Nesse esquema, os diferentes níveis da NPT são armazenadas em múltiplos módulos físicos. Assim, esses vários módulos são capazes de buscar prefixos em paralelo, aumentando o desempenho. Quando múltiplos estados transitam para estados no mesmo módulo físico ao mesmo tempo, ocorre o que os

autores chamam de conflito. Para reduzir os conflitos, alguns estados são duplicados em outros módulos físicos. Contudo, o consumo de memória é impactado devido aos estados duplicados.

O método *Name Component Encoding* (NCE) (WANG et al., 2012) é proposto para reduzir o consumo de memória da tabela FIB através da atribuição de identificadores numéricos únicos a cada componente nomeado de forma a permitir que o LNPM seja aplicado sobre esses identificadores usando a mesma semântica. Os nomes codificados em NCE são então armazenados na FIB que é implementada por meio de uma *trie*. O LNPM é realizado partindo-se do primeiro componente armazenado na raíz até o nó folha. As principais limitações da técnica NCE é o frequente acesso à memória lenta, o tempo extra para realizar a codificação e a profundidade da *trie* que tende a ser alta. Soluções como CONSERT (DAI; LIU, 2016) e *compactTrie* (BOUK; AHMED; KIM, 2015) podem ser usadas para reduzir a profundidade da *trie*. No entanto, essas soluções são baseadas em *software* e difíceis de implementar em *hardware*. A análise teórica e experimental da solução NCE utilizando *datasets* reais mostra que ela é capaz de comprimir uma FIB que contém 3 milhões de prefixos para cerca de 272 MB, o que representa uma taxa de compressão de 32.45% em comparação com a NPT.

Song et al. (2015) apresentam uma outra solução para a FIB baseada na estrutura *trie*. O principal objetivo dessa estrutura de dados é compactar uma FIB com alguns milhões de prefixos à uma taxa suficiente para que ela seja armazenada na memória SRAM. Para isso, Song et al. (2015) propõem usar duas *tries Patricia* (*dualPatricia*) para minimizar a informação redundante armazenada. Essa representação binária provê mais oportunidades para comprimir partes compartilhadas entre diferentes prefixos. Por outro lado, estruturas do tipo *tries Patricia* tendem a aumentar a profundidade da árvore, impactando negativamente a velocidade de *lookup*. Além disso, eles introduzem a ideia de encaminhamento especulativo, o qual usa o método de classificação de prefixo mais longo (LPC) ao invés da tradicional compatibilização do prefixo mais longo (LPM). Diferente do LPM, o *lookup* na técnica LPC garante que o pacote será encaminhado para o próximo salto mesmo se na tabela não existe prefixo que compatibiliza. Porém, o efeito colateral do encaminhamento especulativo é que ele pode causar *loops* infinitos. A avaliação experimental mostra que um *dataset* contendo 3.7 milhões de prefixos requer cerca de 31 MB de memória, 50% menos que a *trie Patricia* convencional.

OnChipFIB (SAXENA et al., 2019) é uma solução para a FIB baseada em *trie* que foi desenvolvida para a plataforma FPGA (WANG et al., 2017). Na solução OnChipFIB, o nome NDN é representado como uma coleção de *strides* de mesmo tamanho. Por exemplo, considerando um *stride* de tamanho quatro, o nome */ufu/facom* é representado por */ufu*, */fac*, e *com*. Esses *strides* são inseridos em uma árvore de busca binária e o LNPM é realizado através de uma busca em profundidade. Em relação ao consumo de memória, a complexidade da solução OnChipFIB é $O(n)$, onde n é o número de nomes

armazenados na FIB. Todavia, no pior caso, a proposta OnChipFIB é $O(nk)$, onde k é o número de *strides*. Em relação a velocidade de *lookup*, a complexidade temporal da solução OnChipFIB é $O(n)$, sendo n o número de componentes do nome.

O *NDN Forwarding Daemon* (NFD) (NDN PROJECT TEAM, 2019) é o encaminhador NDN de referência baseado em *software*. Ele implementa a FIB usando uma estrutura de dados chamada *NameTrie* (GHASEMI et al., 2018), que é projetada como um tipo especial de *trie*. O objetivo da NFD é combinar a FIB, PIT e CS em uma única tabela. Para acelerar o LNPM, as entradas da FIB no NFD são organizadas em uma tabela de *hash*, além da estrutura *trie*. Uma das principais limitações do NFD é que ele usa apenas uma *thread* para processar todos os Ipkts e Dpkts, limitando significativamente a vazão de encaminhamento. Alternativamente, YaNFD (NEWBERRY; MA; ZHANG, 2021) é proposto para melhorar a vazão usando processamento *multi-thread*. No entanto, como soluções baseadas em *software*, tanto o NFD quanto o YaNFD apresentam alguns problemas de desempenho e sobrecarga que limitam a escalabilidade de tais soluções.

Conforme mencionado na seção anterior, a arquitetura ENDN (KARRAKCHOU; SAMAAN; KARMOUCH, 2020a) traz uma proposta de roteador NDN para comutadores programáveis avaliada e testada em *software*. A tabela FIB em ENDN é baseada na estrutura de dados FCTree (KARRAKCHOU; SAMAAN; KARMOUCH, 2020b). De forma geral, a FCTree comprime prefixos nomeados armazenando subprefixos comuns apenas uma vez. Porém, como a FCTree é proposta inicialmente para rodar em *software*, recursos como ponteiros, recursão e alocação dinâmica de memória podem ser difíceis de implementar em ASICs. Além disso, em virtude da FCTree usar a estrutura *trie* para armazenar os prefixos, para reduzir a latência de *lookup*, é necessário auto balancear a *trie* após cada inserção. Assim, dadas as restrições de dispositivos P4 para esse tipo de operação, é desafiador implementar a FCTree eficientemente em *hardware*.

3.3.3 FIB baseada em Filtros de *Bloom*

Os Filtros de Bloom (BF) são estruturas de dados probabilísticas usadas para consultas de associação. De certa maneira, os BFs são usados para testar se um determinado elemento é membro de um conjunto. Embora a natureza probabilística do BF implique a possibilidade de falsos positivos, como os BFs são eficientes em termos de consumo de memória, muitas soluções baseadas em BF foram propostas na literatura para implementar a NDN FIB.

O esquema NLAPB (QUAN et al., 2014) consiste em um BF de prefixo adaptativo proposto para otimizar o consumo de memória física. A principal característica do NLAPB é dividir os prefixos NDN em dois segmentos e realizar a operação de *lookup* combinando um filtro de *bloom* contador (CBF) e uma *trie* em uma abordagem de *Co-Design* de *hardware* e *software*. Os prefixos no NLAPB são segregados de acordo com seus comprimentos. Desse modo, o primeiro segmento na NLAPB contém prefixos com m

componentes (B-prefix) armazenados em um BF enquanto o segundo (T-suffix) contém prefixos de comprimento variável armazenados na *trie*. O NLAPB suporta comutadores equipados com oito interfaces. Os experimentos realizados indicam que o NLAPB é escalável em termos de consumo de memória mesmo quando grandes *datasets* são utilizados. A limitação do NLAPB inclui a possibilidade de falsos positivos no encaminhamento de pacotes.

MaFIB (LI et al., 2017) é uma outra estrutura de dados baseada em BF para a FIB. Ela utiliza um mecanismo chamado *Mapping Bloom Filter* (MBF) proposto por Li et al. (2014). O filtro MBF consiste em um BF regular e um vetor de *bits* armazenados na SRAM. O BF é utilizado para verificar se os elementos existem ou não na MBF. O vetor de *bits* é utilizado como endereço para acessar as interfaces de saída armazenadas na memória DRAM. Em comparação com NCE e métodos similares, MaFIB provê uma melhor taxa de compressão da FIB. No entanto, sua principal limitação é a alta frequência de acesso à memória DRAM para extrair as interfaces de saída. Além disso, falsos positivos podem ocorrer, assim como na NLAPB, comprometendo a acurácia do encaminhamento.

Para melhorar a MaFIB em relação ao consumo de memória, Li et al. (2018) propõem a solução B-MaFIB. Essa solução herda as mesmas características da MaFIB, porém os autores mudam a estrutura MBF usando um BF de mapeamento de *bits*, que eles chamam de B-MBF. A ideia central do B-MBF é possibilitar alocação dinâmica de memória para reduzir o consumo de memória. Contudo, o B-MaFIB ainda necessita acessar a memória DRAM para ler as informações como as portas de saída e tempo de vida.

Quando se trata de implementar a FIB em dispositivos programáveis (p. ex.: ASICs, FPGAs, SmartNICs, etc), o uso de BF sem estruturas de dados auxiliares são incomuns na literatura (LI et al., 2019). Seguindo essa abordagem, Yu e Pao (2019) propõem o *fAccelerator*, uma solução baseada em FPGA para a FIB que utiliza BF como estrutura de dados principal. Nessa proposta, os prefixos da FIB são divididos em dois grupos distintos, sendo o primeiro formado por prefixos com até quatro componentes e o segundo formado pelos demais prefixos (cinco ou mais componentes). Os prefixos do primeiro grupo são armazenados em uma tabela de *lookup* externa que roda em *software*. Já os prefixos do segundo grupo são armazenados na memória *on-chip* da FPGA. O motivo para essa divisão está na limitação da quantidade de memória *on-chip* das placas FPGAs modernas. A principal estrutura de dados utilizada é a BF combinada com a função de *hash* H3 para armazenar os prefixos da FIB na FPGA. A principal limitação dessa proposta é a alta latência causada pelo acesso à memória externa para a realização do *lookup*.

As Tabelas 3.2 e 3.3 mostram as soluções para a NDN FIB em comutadores tradicionais bem como soluções baseadas em P4 com foco em dispositivos programáveis. Assim como as propostas de estruturas para a FIB em comutadores tradicionais, há uma predominância de soluções baseadas em HT quando se trata de soluções para a FIB

Tabela 3.2 – Principais soluções NDN FIB para comutadores tradicionais.

Método	Características	Limitações
Caesar	• Distribui a FIB em vários *line cards* • FIB usa *crc64*	• Alto número de operações de comutação (latência)
fatTree/hash	• Foca na operação de LNPM • Transforma nome em chave compacta	• Informação redundante aumenta o consumo de memória
SACS	• Armazena moldes de prefixos nomeados • *Framework* de busca de conteúdo • Baixo número de acessos à memória	• Necessidade de manter duas tabelas de *hash* aumenta o consumo de memória
PNL/NPT	• Processamento em paralelo • Divisão da NPT em módulos físicos	• Conflitos podem ocorrer • Estados duplicados aumentam o consumo de memória
NCE	• Comprime prefixos nomeados • *Trie* baseada em componente • Utiliza vetores de transição de estados	• Acesso frequente à memória lenta • O processo de codificação aumenta a latência
dualPatricia	• Encaminhamento especulativo • Classificação Prefixo mais Longo	• *Loop* de encaminhamento pode ocorrer • Alta profundidade na árvore (latência)
NLAPB	• Segmenta prefixos • *Lookup* baseia-se em popularidade	• Falsos positivos ocorrem • Alocação dinâmica de memória é difícil em *hardware*
MaFIB	• Baixo custo de memória *on-chip* • Suporta operações de atualização • Armazena grandes *datasets* de nomes	• Acesso frequente à DRAM • CBFs no plano de controle consome muita memória • Muitas funções de *hash* no plano de dados
B-MaFIB	• Executa somente uma função de *hash* • Suporta alocação dinâmica de memória • Usa mapa de *bits*	• Acesso frequente a DRAM para ler portas de saída aumenta a latência

Tabela 3.3 – Principais soluções NDN FIB baseadas em P4.

Método	Características	Limitações
NDN.p4	• FIB como uma única tabela P4 • Entradas na tabela proporcional ao nº de componentes • LNPM executada a taxa de linha	• Desperdiça memória • Muitas entradas inseridas para cada prefixo • Não escalável para milhões de prefixos
HBM	• FIB como uma única tabela P4 • FIB com maior escalabilidade • Trilhas de *hash* na TCAM	• Uso de crc16 causa muitas colisões, requerendo mais TCAM
NDN-P4-SDN	• Idem HBM • Foca em roteamento	• Idem HBM • Tabela P4 grande de ingresso desperdiça TCAM/SRAM no egresso
NDN-Tofino	• FIB armazenada na DRAM • Otimiza memória dividindo a PIT	• LNPM na DRAM aumenta a latência
Pegasus	• Combina *switches* P4 com servidores • Lógica do *parsing* no plano de dados	• LNPM na DRAM aumenta a latência
OnChipFIB	• Nome representado como *strides* • Explora paralelismo de *hardware*	• Complexidade no pior caso é $O(nk)$ • Atravessar a *trie* consome tempo
ENDN	• *Extern* para processar *strings* • Suporta busca coringa • Comprime subprefixos comuns	• Operações são $O(n)$ (>latência) • Suportar árvores balanceadas tem custo alto em P4
fAccelerator	• Prefixos são divididos em dois grupos	• *Lookup* externo aumenta a latência
U-Table	• Unifica FIB, PIT, e CS • Foca em CCNx	• LNPM na DRAM aumenta a latência

Tabela 3.4 – Comparação entre as principais soluções para a NDN FIB existentes na literatura implementadas em *software* e em *hardware*.

Solução para NDN FIB	**HW**	**P4**	**SDN**	**E.D**	**Características de Hardware**						**Co-Design de HW/SW**	**Latência de LNPM**	**Escalável em Hardware**
					Uso de DRAM	Uso de TCAM	Uso de SRAM	Otimização de memória	Otimização de vazão	Suporte à Paralelismo			
NPT	○	○	○	Trie	●	○	○	○	○	○	○	ms	○
PNL	○	○	○	Trie	●	○	○	○	○	●	○	µs	○
NCE	○	○	○	Trie	●	○	○	●	○	○	○	ms	○
Caesar	●	○	○	HT	○	○	●	○	○	●	○	µs	○
fatTree	○	○	○	Trie,HT	●	○	○	●	○	○	○	ms	○
NLAPB	○	○	○	BF,Trie	●	○	○	●	○	○	●	ms	●
dualPatricia	◐	○	○	Trie	◐	○	◐	○	●	○	●	µs	○
compactTrie	○	○	○	Trie	●	○	○	●	○	○	○	ms	○
CONSERT	○	○	○	Trie	●	○	○	●	●	○	○	ms	○
NDN.p4	●	●	○	HT	○	●	○	○	○	●	○	ns	○
MaFIB	●	○	○	BF	◐	○	◐	○	○	●	●	ms	○
B-MaFIB	●	○	○	BF	◐	○	◐	●	●	●	●	ms	○
SACS	●	○	○	HT,Trie	○	●	●	○	●	○	○	µs	○
HBM	●	●	○	HT	○	●	○	◐	○	●	○	ns	○
fAcellerator	●	○	○	BF	●	○	●	○	●	●	●	µs	○
OnChipFIB	●	○	○	Trie	○	○	●	○	○	●	○	µs	●
NDN-DPDK	○	◐	○	HT	●	○	○	○	●	○	○	µs	○
FCTree	○	○	●	Trie	●	○	○	●	○	○	○	ms	○
NFD	○	○	○	Trie,HT	●	○	○	○	○	○	○	ms	○
YaNFD	○	○	○	Trie	●	○	○	○	●	●	○	µs	○
NDNFab	○	◐	●	HT	●	○	○	○	○	●	●	µs	○
NDNTofino	○	◐	○	HT	●	○	○	○	○	●	●	µs	○
Pegasus	○	◐	○	HT	●	○	○	○	○	●	●	µs	○
U-Table	●	◐	○	HT	○	○	●	○	○	●	○	µs	○

E.D = 'Estrutura de Dados'. HT = Hash Table. BF = Bloom-Filter. O símbolo (◐) indica suporte parcial para uma determinada característica.

compatíveis com comutadores programáveis. Possivelmente isso deve-se a boa taxa de compressão proporcionada por certas funções de *hash* e a simplicidade de executar tais funções em dispositivos físicos que contam com *chips* dedicados. A Tabela 3.4 compara qualitativamente as principais soluções para a NDN FIB discutidas nesse capítulo.

3.4 Considerações

Esse capítulo apresentou os principais trabalhos relacionados às arquiteturas mais prevalentes para encaminhamento de tráfego NDN baseadas em SDN e as estruturas de dados para a NDN FIB projetadas tanto para *software* como para *hardware*. As soluções em ambas vertentes foram apresentadas destacando suas características, vantagens e limitações. Foram apresentadas também duas classificações e comparações qualitativas envolvendo as propostas em ambas vertentes.

De maneira geral, as soluções apresentadas nesse capítulo possuem três limitações principais: (i) alto número de colisões de *hash* para uma FIB com milhões de prefixos armazenados na memória do ASIC de comutadores programáveis, (ii) subutilização de recursos como SRAM/TCAM e capacidade de processamento ao armazenar a FIB no plano de dados de comutadores programáveis e (iii) falta de mecanismos para equilibrar o consumo de memória e a latência de *lookup*, além da falta de mecanismos de otimização de recursos de *hardware*. Para endereçar essas questões, o Capítulo 4 apresenta o plano de controle da arquitetura FANTNet e o Capítulo 5 traz os detalhes do plano de dados, que inclui a estrutura de dados CoFIB. A viabilidade e o desempenho de tais soluções são apresentadas no Capítulo 6.

Capítulo 4

FANTNet: Plano de Controle

4.1 Introdução

A arquitetura NDN foi projetada para ser totalmente descentralizada. Através de uma estrutura de encaminhamento que mantém estados em roteadores, a NDN é capaz de suportar nativamente transmissões *multicast* por meio de encaminhamento baseado em nomes. Devido ao seu caráter distribuído, as redes NDN são altamente escaláveis. Todavia, a convergência de estratégias de roteamento no plano de controle bem como o conhecimento sobre a distribuição de nomes para fins de compactação de tabelas no plano de dados, como por exemplo a FIB, são fatores limitantes em um modelo descentralizado. Em uma direção oposta, o paradigma SDN provê controle logicamente centralizado para uma rede sob um mesmo domínio administrativo. Esse controle centralizado incorre em problemas de escalabilidade, mas por outro lado favorece o encaminhamento rápido de pacotes através de uma visão holística de todos os elementos da rede. Desse modo, a proposta apresentada nessa tese busca trazer o controle logicamente centralizado e flexibilidade na programação de rede presentes na SDN para a NDN mantendo suas características nativas.

A arquitetura FANTNet proposta nessa tese contém comutadores de borda que recebem o tráfego NDN e realiza o encaminhamento baseado em nomes para os comutadores de núcleo com base no paradigma SDN. Esse capítulo foca nos elementos do plano de controle da arquitetura FANTNet necessários para viabilizar esse encaminhamento rápido de pacotes NDN no plano de dados do domínio. Nesse sentido, a Seção 4.2 traz uma visão geral dos principais componentes da FANTNet. A Seção 4.3 descreve a tabela RIB, seguida pela Seção 4.4 que apresenta a tabela de encaminhamento nos comutadores de núcleo. A Seção 4.5 apresenta a lógica para a extração de prefixos canônicos da RIB e a Seção 4.6 apresenta a estrutura de dados usada para gerar as entradas do plano de dados. Por fim, a Seção 4.7 apresenta as fases de operação da arquitetura FANTNet seguida pelas considerações parciais na Seção 4.8.

4.2 Visão Geral

A Figura 4.1 mostra os principais componentes da arquitetura FANTNet. Como se pode obervar, a arquitetura é dividida em três planos horizontais distintos. O plano de aplicação, no topo, contém funções de rede (*Network Function* (NF)) que manipulam estruturas de dados de propósito específico. Já o plano de controle contém as estruturas de armazenamento para as tabelas padrão da arquitetura NDN (ex.: CS, RIB) bem como para estruturas de dados projetadas especificamente para a FANTNet. Por fim, o plano de dados da FANTNet, na base, é formado por uma malha de comutadores de núcleo (*Core Switch* (CSw)) interligados com comutadores de borda programáveis (*Edge Switch* (ESw)).

Baseado no modelo *Fabric* (CASADO et al., 2012), cada comutador de borda da FANTNet é conectado a um elemento de rede chamado *NDN Packet Converter* (NPC). De maneira geral, um elemento NPC desempenha três funções principais. A primeira delas consiste em extrair um ou mais nomes de Dpkts e enviá-los para o plano de controle como forma de anúncios de prefixos. A segunda função é a de converter pacotes NDN nativos que cruzam o domínio SDN para pacotes com nomes codificados em formatos específicos. Os pacotes NDN podem ser traduzidos ou não dependendo da quantidade e comprimento dos componentes do nome. Caso não seja possível realizar a tradução, o NPC desempenha a função de enviar o pacote em formato TLV ao controlador SDN para que esse receba um tratamento específico.

Existem duas abordagens possíveis para a implementação do NPC. A primeira é utilizar *gateways* FPGAs nas extremidades do domínio NDN, conforme mostra a Figura 4.1.

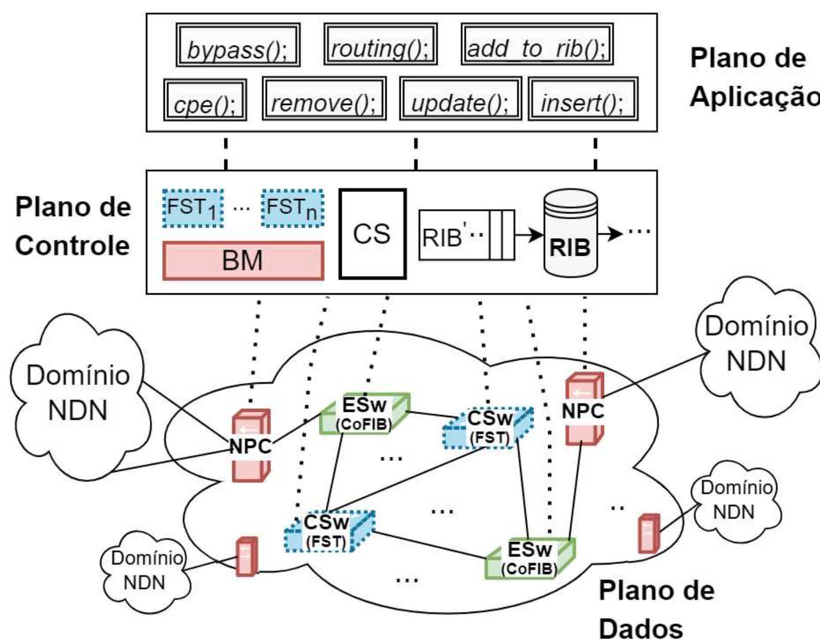

Figura 4.1 – Visão geral da arquitetura FANTNet.

Esses *gateways* tem a função de traduzir os blocos TLV que representam o prefixo para um formato específico, mantendo os demais campos na codificação original. Trabalhos como (TROSSEN et al., 2015), (WHITE; RUTZ, 2016) e (GUIMARÃES et al., 2019) empregam estratégia semelhante sem grandes impactos na latência fim-a-fim. A segunda abordagem é definir *externs* dentro dos comutadores programáveis da borda do domínio NDN. Esses *externs* são responsáveis pela conversão dos pacotes NDN nativos em tempo real para o formato proposto. Isso é possível em comutadores programáveis compatíveis com o ASIC Intel Tofino graças a arquitetura de código-aberto TNA (KUMAR, 2021). Ambas abordagens são empregadas no núcleo de uma rede NDN de forma que a comunicação entre consumidores e produtores seja totalmente transparente.

A lógica de conversão empregada no NPC varia de acordo com o tipo de pacote NDN. Caso seja Ipkt, apenas o nome será traduzido do formato TLV para o formato descrito na Seção 5.2.1. A principal motivação para a nova representação do nome em Ipkts é facilitar a operação de LNPM na CoFIB. Por outro lado, caso seja Dpkt, o NPC acrescentará um bloco TLV especial no início do pacote. A finalidade desse bloco é uniformizar a identificação de Ipkts e Dpkts nos comutadores programáveis de borda e facilitar as consultas nas tabelas PIT e CS para Dpkts com *hashes* de nomes pré-calculados. Os detalhes sobre o formato de nomes proposto serão vistos no Capítulo 5.

O plano de controle é responsável por manter tabelas gerais da arquitetura NDN (por exemplo, a CS) e estruturas de dados projetadas para desempenhar funções específicas dentro do domínio SDN. A tabela *Bypass Memory* (BM) é uma dessas estruturas de propósito específico. Ela é implementada como uma fila FIFO e armazena Ipkts e Dpkts nativos que não puderam ser traduzidos pelo NPC em razão da quantidade ou comprimento de componentes do nome exceder o valor máximo suportado. Esses pacotes são removidos da BM iterativamente e são difundidos para todos os demais NPC do domínio através da função *bypass()* do plano de aplicação, conforme mostra o fluxo operacional da

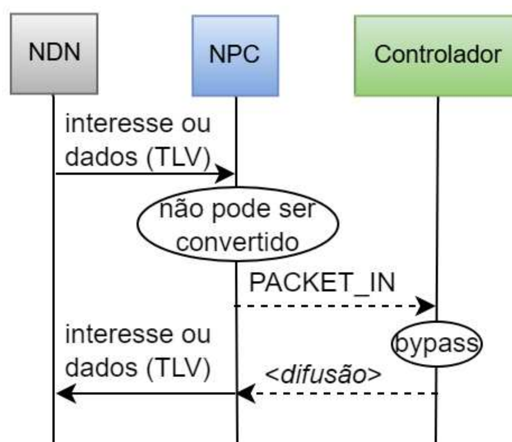

Figura 4.2 – Fluxo operacional de pacotes NDN que não podem ser convertidos.

Figura 4.2. Ao receberem esses pacotes, os nós NPC os encaminharão para as interfaces ligadas aos domínios NDN nativos.

A sobrecarga de processamento no controlador SDN gerada pela operação de *bypass()* está relacionada com a quantidade de pacotes recebidos pela BM. De forma generalizada, a taxa com que pacotes NDN serão enviados para a BM é proporcional ao número de pacotes que chegam ao NPC e não podem ser traduzidos. Assumindo uma quantidade máxima de 8 componentes por nome e cada componente podendo ter até 31 caracteres, a taxa esperada de pacotes NDN não convertidos será baixa, segundo análises estatísticas apresentadas no trabalho de Wang et al. (2012), o que não impactará a escalabilidade da solução proposta.

4.3 Routing Information Base

O plano de controle da arquitetura FANTNet armazena também a RIB, criada por meio da RIB′, uma estrutura de dados intermediária implementada como uma fila do tipo FIFO. Na FANTNet, a RIB′ armazena prefixos nomeados adicionados estaticamente pelo operador bem como prefixos anunciados dinamicamente por nós NDN de fora do domínio. Os prefixos anunciados são os nomes enviados pelo NPC ao controlador e também os Dpkts provenientes de protocolos de roteamento específicos como o NLSR (HOQUE et al., 2013).

Cada comutador de borda está conectado à um NPC e cada NPC pode estar conectado à um ou mais roteadores NDN nativos, conforme mostra a Figura 4.1. Para cenários onde mais de um roteador NDN de fora do domínio se conecta à um NPC, a função de identificar para qual desses roteadores enviar Ipkts ou Dpkts é delegada ao NPC e está fora do escopo dessa tese. Assim, cada entrada na RIB′ é composta apenas pelo prefixo anunciado e pelo *id* do comutador que recebeu esse anúncio. Essas informações são armazenadas na RIB′ através da tupla $v = \langle prefixo; swId \rangle$.

Para otimizar o consumo de memória *on-chip* nos comutadores de borda, o campo *swId* de cada entrada RIB′ é codificado em 8 *bits*. Existem duas abordagens possíveis para interpretar os *bits* de *swId*. Na primeira delas, o *swId* pode representar o conjunto de comutadores de borda que respondem por um dado prefixo, garantindo o suporte à transmissão *multipath*. Para exemplificar, se o *swId* for 00010001, então os comutadores 1 e 5 respondem pelo prefixo. Assim, o *swId* pode estar associado à um grupo *multicast* que engloba um determinado conjunto de comutadores de borda. Em razão do *swId* ser codificado em 8 *bits*, essa abordagem possibilita até 8 comutadores de borda no domínio, onde cada ESw_i é identificado pelo valor 2^i. Por outro lado, em uma segunda abordagem, o *swId* pode identificar de forma unívoca um comutador de borda que responde pelo prefixo. Nesse caso, os Ipkts serão encaminhados para apenas um comutador de borda mesmo havendo mais de um que responde pelo prefixo. Todavia, ao invés de 8, essa abordagem possibilita até 255 comutadores de borda no domínio. A escolha de qual

abordagem será utilizada depende do padrão de projeto. Nessa tese, será utilizada a primeira abordagem, muito embora seja possível alterar a interpretação dos *bits* de *swId* dinamicamente no controlador SDN dependendo de onde os anúncios estão chegando.

A RIB′ é uma estrutura de dados temporária que tem a finalidade de compatibilizar as taxas com que anúncios são recebidos e processados no plano de controle. A partir dela, gera-se a RIB principal responsável por viabilizar a inserção dos prefixos na CoFIB. A RIB é implementada como uma tabela de *hash* $H : K \rightarrow V$, onde K representa os prefixos da RIB e V os *swIds* que apontam para onde os dados identificados por aquele prefixo pode ser alcançado. A representação da RIB por meio de tabela de *hash* possibilita que os parâmetros *swIds* sejam calculados de forma rápida, tendo em vista que a complexidade de acesso à um elemento da tabela *hash* é $O(1)$. A função *add_to_rib()* no plano de aplicação, detalhada no Algoritmo 1, é responsável por retirar elementos da RIB′ iterativamente e armazená-los na RIB. Através de operações de '*and*' e adições binárias simples, os valores de *swId* associados a cada prefixo são rapidamente formados e são capazes de identificar o subconjunto de comutadores de borda que respondem por aquele prefixo.

Algoritmo 1 Transfere prefixo da RIB′ para a RIB

```
1: função add_to_rib() :
2:   enquanto RIB′ não estiver vazia faça
3:     p ← RIB′.pop();
4:     v ← RIB.get(p.name);
5:     se v existe então
6:       se p.swId operação AND com v for 0 então
7:         v ← v + p.swId;
8:         RIB.replace(p.name, v);
9:     senão
10:      RIB.put(p.name, p.swId);
```

4.4 Fast Switching Table

Um dos princípios de projeto da FANTNet é implementar a CoFIB apenas nos comutadores de borda. Essa estratégia reduz o consumo de memória nos comutadores de núcleo e acelera o encaminhamento de pacotes dentro do domínio em razão de não ser necessário armazenar a FIB no núcleo e realizar o LNPM a cada salto. De forma semelhante, (MADUREIRA et al., 2021) propõem uma arquitetura baseada no modelo *Fabric* que emprega o roteamento por segmentos para acelerar o encaminhamento no núcleo. No entanto, a estratégia proposta nessa tese é encapsular em cabeçalhos específicos os pacotes NDN traduzidos pelo NPC, fazer o encaminhamento salto a salto com base nesse cabeçalho até que os pacotes chegem aos roteadores de borda de saída e desencapsular os pacotes nesses comutadores de borda da saída do domínio. Esses cabeçalhos contém a tupla $\langle marcador, swId \rangle$, onde o marcador é uma sequência padronizada de *bits* utilizada

para distinguir entre pacotes que já passaram por algum comutador de borda e pacotes que acabaram de chegar no domíno FANTNet.

Para possibilitar o encaminhamento rápido de pacotes na malha, a tabela *Fast Switching Table* (FST) é proposta para cada comutador de núcleo do domínio. Como os comutadores de núcleo podem ser programáveis ou tradicionais, a chave de compatibilização exata para determinar a interface de saída do pacote varia de acordo com o tipo de comutador. Caso o comutador de núcleo (S_i) seja tradicional (não programável), o controlador SDN selecionará aleatoriamente um dos comutadores de borda indicados em *swId* (S_k) e calculará o caminho de menor custo entre S_i e S_k. A tabela FST de S_i será adaptada na tabela de endereços MAC, através da *Content Addressable Memory* (CAM), e será populada com a tupla $\langle swId, port \rangle$, onde o valor *swId* é acomodado de forma adaptada no campo de endereço MAC de destino do quadro *Ethernet* e o valor *port* é a interface de S_i ligada ao caminho de menor custo entre S_i e S_k. Já no caso do comutador de núcleo ser programável (P4 ou OpenFlow), o *swId* é representado em 8 *bits* e o parâmetro *port* representa um grupo *multicast* de 16 *bits* que inclui os comutadores de borda indicados em *swId*. É importante observar que o suporte à transmissão *multicast* na FANTNet depende da utilização de comutadores programáveis no núcleo.

4.5 Segregação de Prefixos

Para reduzir o espaço gasto ao armazenar prefixos nomeados na SRAM do comutador de borda, essa tese introduz o conceito de prefixo nomeado canônico, que será definido formalmente no Capítulo 5, Seção 5.3.5. De forma simplificada, um prefixo nomeado canônico tem seus componentes em uma mesma posição em todos os prefixos da FIB. Essa característica reduz a quantidade de *bits* necessários para codificar a posição de cada componente. Em razão da arquitetura NDN deixar em aberto questões relacionadas a convenções e acordos sobre quais espaços de nomes devem ser utilizados em um dado domínio (KHELIFI et al., 2020; TARIQ; REHMAN; KIM, 2020), é possível assumir um espaço de nomes totalmente canônico. Nomes que representam localizações hierárquicas usando o padrão /[*país*]/[*estado*]/[*cidade*]/[*logradouro*], como em Yang, Chen e Liu (2020), ou aplicações baseadas em IoT com nomes na forma /[*categoria*]/[*fabricante*]/[*dispositivo*]/[*timestamp*], como em Gündoğan et al. (2021), são exemplos de espaço de nomes canônicos. Todavia, em casos onde não é possível fazer tal suposição, propõe-se nessa tese o *Canonical Prefix Extractor* (CPE)[1], descrito no Algoritmo 2, usado para extrair prefixos nomeados canônicos a partir da RIB.

A lógica do CPE é associar um vetor de mapeamento de posições $v = [v_1, ..., v_8]$ para cada componente nc_j em todos os prefixos da RIB, onde v_k é a frequência de nc_j na posição

[1] O termo CPE proposto nessa tese difere do *Customer Premises Equipment* (CPE) comumente utilizado em telecomunicações para se referir aos equipamentos do usuário utilizados na recepção de sinais de comunicação.

Algoritmo 2 Extrator de Prefixos Canônicos (CPE)

```
1: Entrada: Conjunto de prefixos P = {p_1, ..., p_m} da RIB
2: Saída: Conjunto de prefixos canônicos D e não canônicos C com P = D ∪ C
3: função cpe():
4:     Cria tabela hash H em NCH;
5:     para cada p_j ∈ P faça {populando H}
6:       para cada nc_i in p_j faça
7:           v ← H.get(nc_i);
8:           se v == null então H.put(nc_i, [0, .., 0]);
9:           senão v[i] ← v[i] + 1; H.replace(nc_i, v);
10:    para cada p_j ∈ P faça {calculando pesos}
11:        w ← 0
12:        para cada nc_i in p_j faça
13:          v ← H.get(nc_i); w ← w + v[i];
14:        associa peso w em p_j
15:        adiciona p_j na lista ordenada L com base nos pesos
16:    para cada p_j em L faça {extraindo nomes canônicos}
17:        se p_j é não canônico então
18:           C.put(p_j);
19:           para cada nc_i em p_j faça
20:              v ← H.get(nc_i);
21:              se v[i] > 0 então
22:                 v[i] ← v[i] − 1; H.replace(nc_i, v);
23:        senão D.put(p_j)
24:    para cada p_j em D faça {removendo subprefixos comuns}
25:        se p_j compartilha prefixo comum em C então
26:           D.remove(p_j); C.put(p_j)
```

k. Um dado prefixo é canônico se os vetores de seus componentes nc_j contém zero em todos os índices exceto o índice k. O CPE calcula o peso para cada prefixo $/nc_1/../nc_k$ da RIB somando-se v_k nos vetores de cada nc_k, para $1 \leq k \leq n$. Após isso, os prefixos são ordenados de forma crescente e os canônicos são inseridos sequencialmente em uma lista enquanto os não canônicos são armazenados em uma lista paralela. O Algoritmo 2 detalha a operação do CPE.

A estrutura de dados utilizada pelo CPE para dividir o conjunto de prefixos da RIB entre canônicos e não canônicos é a *Name Component Hash Table* (NCH), conforme mostra a Figura 4.3. Essa estrutura é criada entre as linhas 4-9 do Algoritmo 2 e é definida como uma tabela de *hash* $H : K \rightarrow V$, sendo K o conjunto de todos os componentes nomeados extraídos dos prefixos da RIB e V o conjunto de todos os vetores de mapeamento de posições associados aos componentes dos prefixos da RIB. Durante a inicialização, os vetores de mapeamento de posições são zerados e, iterativamente, à medida em que os componentes nomeados são extraídos dos prefixos da RIB e inseridos na NCH, os índices desses vetores são incrementados. Por usar tabela de *hash*, a complexidade temporal do CPE é $O(kn)$, onde k é o número máximo de componentes suportado e n a quantidade de

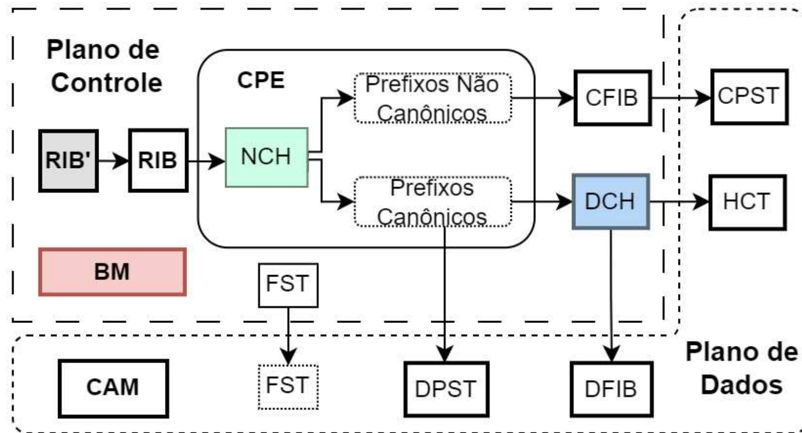

Figura 4.3 – Elementos do plano de controle da FANTNet associados aos comutadores de borda para a geração das tabelas do plano de dados.

prefixos da RIB. Em razão de na FANTNet o valor de k ser constante, a complexidade do CPE se reduz à $O(n)$.

Os prefixos não canônicos extraídos da RIB pelo CPE serão armazenados na memória *off-chip* no plano de controle. A razão disso é a existência de componentes em diferentes posições no conjunto de prefixos que requer mais espaço e inviabiliza o armazenamento desses prefixos na memória *on-chip* dos comutadores de borda. Assim, os prefixos não canônicos serão armazenados em uma estrutura secundária no plano de controle chamada *Control Plane Forwarding Information Base* (CFIB), conforme mostra a Figura 4.3. Quando não há prefixos que compatibilizam na CoFIB no plano de dados, os Ipkts que chegam aos comutadores de borda podem ser encaminhados ao plano de controle para que seus nomes sejam consultados na CFIB a fim de determinar os NPCs que encaminharão o pacote para fora do domínio. Para evitar que um prefixo contendo pelo menos dois componentes compatibilize na CoFIB mas existe um prefixo maior que compatibiliza na CFIB, ao final do processo de extração de prefixos, o algoritmo CPE remove do conjunto de prefixos canônicos todos os prefixos que possuem subprefixos comuns na CFIB. Essa operação é detalhada nas linhas 24-26 do Algoritmo 2.

4.6 Dual Component Hashmap

Os prefixos canônicos extraídos pelo CPE serão inseridos na *Dual Component Hashmap* (DCH) antes de serem descarregados na CoFIB. A DCH é uma estrutura de dados intermediária projetada para formatar as entradas na CoFIB e suportar as operações de inserção, atualização e remoção de prefixos nos planos de controle e de dados, conforme descritas nos Algoritmos 3, 4 e 5, respectivamente. Cada operação executada na DCH é espelhada para as tabelas da CoFIB através de APIs como a *P4Runtime* (ONF, 2020). Para suportar essas operações, a DCH encapsula duas tabelas de *hash* H_1 e H_2 implemen-

Algoritmo 3 Inserção de Prefixo na DCH

1: **Entrada**: Prefixo nomeado P
2: **função** $insert_prefix()$:
3:   **se** P é canônico e $P \notin DCH$ **então**
4:    **para cada** componente $nc_i \in P$ **faça**
5:      $s \leftarrow s + nc_i$; elemento $\leftarrow H_1.get(nc_i)$;
6:      **se** elemento == null **então**
7:        cad $\leftarrow$ ⟨con=1, end=0, cfl=0, pos=i, esw=0⟩; cp←1; cc←ce←cf←0;
8:        **se** i é última posição em P **então**
9:          $ce \leftarrow 1; cad \leftarrow \langle con = 0; end = 1, .., esw =$ESwId⟩
10:        **senão**
11:          $cc \leftarrow 1; cad \leftarrow \langle con = 1; end = 0, ..\rangle$
12:        **se** $i > 0$ **então** $hs \leftarrow F(s)$;
13:        elemento← $\langle cp, cad, cc, ce, 0, hs\rangle$; $H_1.put(nc_i$,elemento);
14:      **senão**
15:        elemento.cp++; cad←elemento.cad;
16:        **se** i é última posição em P **então** cad.e←1; elemento.ce++;
17:        **senão** cad.c←1; elemento.cc++;
18:        **se** cad.cf==1 **então**
19:          $hs \leftarrow F(s)$;
20:          **se** i é última posição em P **então**
21:            **se** $H2.put(hs$, ESwId) ==null **então** elemento.cf++;
22:            **senão** $H2.replace(hs$, ESwId);
23:          **senão**
24:            **se** $H2.put(hs$, 0) ==null **então** elemento.cf++;
25:        **senão**
26:          **se** $i > 1$ **então**
27:            $k \leftarrow F(s)$;
28:            **se** $k \neq$ elemento.hs **então**
29:              cad.cf←1; $hs \leftarrow k$;
30:              **se** i é última posição em P **então**
31:                ESwIdHCT←ESwId;
32:              **se** $H2.put$(elemento.hs, elemento.cad.out)==null **então**
33:                elemento.cf++;
34:              **se** $H2.put(hs$,ESwIdHCT)==null **então** elemento.cf++;
35:            **senão**
36:              **se** i é última posição em P **então** cad.out←ESwId;
37:          **senão**
38:            **se** i é última posição em P **então** cad.out←ESwId;
39:        elemento.cad←cad; $H1.replace(nc_i$,elemento);

tadas em *software*. A tabela H_2 armazena componentes conflitantes que serão discutidos no Capítulo 5, Seção 5.3.5. Já tabela H_1 é responsável por armazenar componentes de prefixos canônicos e é definida como $H_1 : K_1 \rightarrow V_1$, onde cada componente nomeado $nc_i \in K_1$ é mapeado para uma tupla de seis valores ⟨cad, cp, cc, ce, cf, hs⟩ pertencente ao conjunto V_1. Por utilizarem tabelas de *hash*, as operações de inserção, atualização e remoção de prefixos nos planos de controle e de dados não são influenciadas pelo número

Algoritmo 4 Atualiza Prefixo na DCH

```
1: Entrada: Prefixo p e identificador de saída swId.
2: função update_prefix() :
3:     nc ← ultimo componente de p;
4:     cpe ← H1.get(nc);
5:     se cpe ≠ null então
6:         currentHash ← h(p);
7:         se cpe.cad.cf == 1 então
8:             se currentHash == cpe.hs então
9:                 cpe.cad.out ← swId; H1.replace(nc, cpe);
10:            se H2.get(currentHash) ≠null então
11:                H1.replace(currentHash, swId);
12:        senão
13:            se currentHash == cpe.hs então
14:                cpe.cad.out ← swId; H1.replace(nc, cpe);
```

de prefixos e têm complexidade temporal $O(k)$ no número de componentes, o que acaba sendo reduzida à $O(1)$ em função de k ser limitado na solução proposta.

A tupla de seis valores associada a cada componente nomeado (nc_i) de DCH é responsável por formatar as entradas das tabelas P4 e por contabilizar a frequência com que tais componentes aparecem em diferentes posições em todos os prefixos. Essas frequências são necessárias para viabilizar as operações de inserção, remoção e atualização. O primeiro parâmetro da tupla de seis valores é o *cad* (*Component Action Data* (CAD)). Esse parâmetro será usado como *action data* ao armazenar esses componentes nas tabelas P4 dos comutadores de borda programáveis. Isso será visto com mais detalhes no Capítulo 5, Seção 5.3.5. Os demais parâmetros são interpretados da seguinte maneira:

- **Parâmetro** *cp*: Representa a frequência de nc_i em sua posição específica considerando todos os prefixos do conjunto. Para exemplificar, supondo o conjunto de prefixos P={*/a/b/c*, */x/b/t/n*, */y/b*} e nc_i='b', então $nc_i.cp$ = 3 tendo em vista que o componente 'b' aparece em todos os três prefixos.

- **Parâmetro** *cc*: Indica a frequência com que nc_i não aparece na última posição em todos os prefixos do conjunto. Considerando o conjunto P anterior e nc_i='b', então $nc_i.cc$ = 2 pois existem dois prefixos em P nos quais 'b' não aparece na última posição (*/a/b/c* e */x/b/t/n*).

- **Parâmetro** *ce*: Representa a frequência com que nc_i aparece na última posição em todos os prefixos do conjunto. Supondo o mesmo conjunto anterior P e nc_i='b', então $nc_i.ce$ = 1, tendo em vista que apenas o prefixo */y/b* contém o componente 'b' na última posição.

- **Parâmetro** *cf*: Representa o número de vezes que nc_i aparece em prefixos conflitantes. Esses tipos de prefixos serão abordados no Capítulo 5 na Seção 5.3.5.

Algoritmo 5 Remove Prefixo da DCH

```
1: Entrada: Prefixo p
2: função remove_prefix() :
3:   se p pode ser removido de DCH então
4:     para cada componente nc_i de p faça
5:       s ← s + nc_i; cpe← H_1.get(nc_i);
6:       se cpe ≠ null então
7:         se cpe.pos > 1 então
8:           cpe.pos← cpe.pos-1;
9:           se i <p.numComponentes então
10:            se (cpe.cc ←cpe.cc − 1)== 0 então
11:              cpe.cad.c← 0;
12:          senão
13:            se (cpe.ce ←cpe.ce − 1)== 0 então
14:              cpe.cad.e← 0; cpe.cad.out← 0;
15:          se i > 1 então
16:            se cpe.cad.cf == 1 então
17:              hashed ← h(s)
18:              se H_2.remove(hashed) ≠ null então
19:                se (cpe.cf ← cpe.cf − 1) == 1 então
20:                  cpe.cf ← cpe.cf − 1;
21:                  se H_2.remove(cpe.hs) ≠ null então
22:                    cpe.cad.cf ← 0;
23:        senão
24:          H_1.remove(nc_i);
```

- **Parâmetro** hs: Está associado ao *hash* da primeira ocorrência do subprefixo de nc_i. Considerando novamente o conjunto P e nc_i='b', então $nc_i.hs = h(/a)$, onde $h(x)$ representa uma função de *hash* e $/a$ é o subprefixo do primeiro prefixo de P que contém 'b' ($/a/b/c$).

4.7 Fases de Operação

Conforme será visto no Capítulo 5, os prefixos armazenados na CoFIB de cada comutador de borda devem ser canônicos. Dessa forma, para maximizar a quantidade de prefixos que podem ser armazenados, é necessário conhecer sobre como os nomes são distribuídos e entender a posição predominante de cada componente nos nomes anunciados antes deles serem armazenados na memória *on-chip* dos comutadores de borda. Por essa razão, o processamento de pacotes NDN na FANTNet é realizado em quatro fases distintas apresentadas a seguir.

- **Fase 1 - Amostragem:** Para cenários onde existe a possibilidade de nomes não canônicos, essa fase tem como objetivo coletar uma quantidade suficiente de prefixos na RIB′ para que seja possível conhecer sobre a distribuição de nomes que

chegam ao domínio. Nessa fase, os anúncios de prefixos são acumulados na RIB′ e as tabelas CoFIB, PIT e FST permanecem vazias. Os Ipkts e Dpkts que chegam aos comutadores de borda durante essa fase são descartados[2] de acordo com o fluxo normal de processamento de pacotes na arquitetura NDN (ver Figura 2.4).

- **Fase 2 - Segregação:** Nessa fase, o CPE é executado para extrair os prefixos canônicos da RIB, onde as tabelas NCH, CFIB e DCH são geradas. Para manter os prefixos da RIB consistentes, os anúncios que chegam ao controlador durante essa fase permanecem armazenados na RIB′ até o término da execução do CPE. Assim como ocorre na Fase 1, os Ipkts e Dpkts são descartados em razão das tabelas PIT e CoFIB estarem vazias.

- **Fase 3 - Descarregamento:** Após a fase de segregação, os prefixos acumulados na RIB′ são transferidos para a RIB por meio do Algoritmo 1 e em seguida são inseridos na DCH por meio do Algoritmo 3. Nessa fase, as tabelas FST nos comutadores de núcleo são populadas de acordo com as interfaces ligadas aos caminhos de menor custo entre os comutadores de núcleo e de borda. Paralelamente, os prefixos canônicos armazenados na DCH, os prefixos conflitantes e os moldes de prefixos canônicos e não canônicos, conforme será visto no Capítulo 5, são descarregados nas tabelas do plano de dados.

- **Fase 4: Transmissão:** Nessa fase, os Ipkts e Dpkts que chegam aos comutadores de borda são encaminhados para o núcleo de acordo com os conteúdos das tabelas CoFIB e PIT, respectivamente. No núcleo, esses pacotes são encaminhados de acordo com os conteúdos das tabelas FST. Os anúncios de prefixos que chegam à RIB durante essa fase são inseridos na DCH e espelhados na CoFIB caso sejam canônicos ou são inseridos na CFIB caso contrário. A Figura 4.4 mostra o *workflow* para Ipkts e de Dpkts na fase de transmissão.

As fases de amostragem, segregação e descarregamento retardam o armazenamento de prefixos no plano de dados e, consequentenente, causam o descarte prematuro de Ipkts e Dpkts. Além disso, dependendo da quantidade de prefixos canônicos na RIB e da capacidade de processamento do controlador SDN, essas três fases podem demorar de alguns segundos até minutos para serem concluídas. No entanto, essas fases ocorrem apenas uma vez e não são necessárias em cenários onde existem somente prefixos canônicos. Na fase de transmissão, os prefixos canônicos já estão armazenados na CoFIB e o encaminhamento de Ipkts contendo esses prefixos passa a ser realizado de forma rápida no plano de dados.

[2] Ao invés de serem descartados, esses pacotes podem, alternativamente, ser encaminhados em *multicast* para todos os comutadores de borda do domínio através de uma rota pré-definida calculada após o *bootstrapping* do controlador

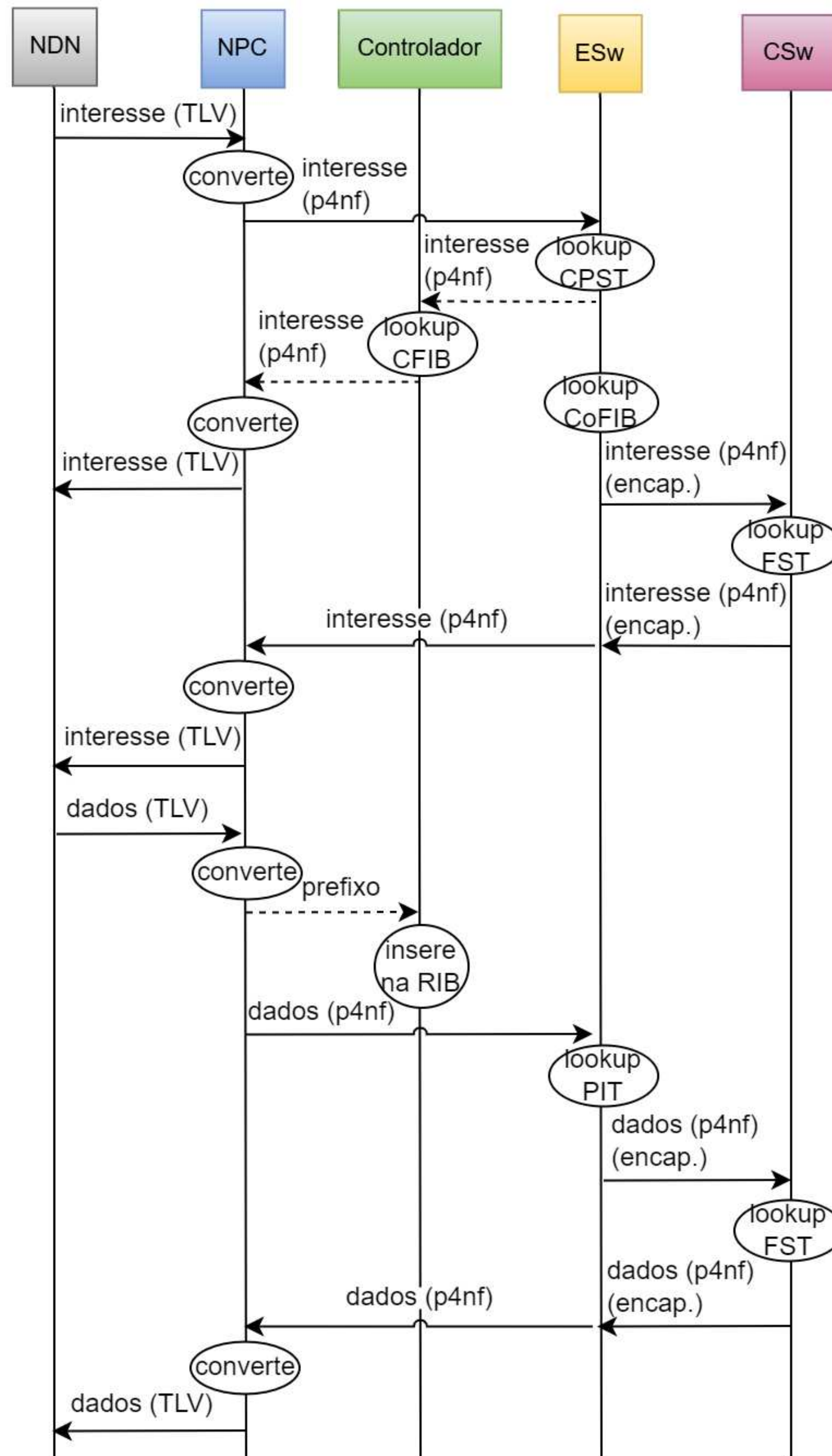

Figura 4.4 – *Workflow* para pacotes de interesse e de dados.

4.8 Considerações

Esse capítulo forneceu uma descrição detalhada do plano de controle da arquitetura FANTNet. Embora baseada no modelo *Fabric*, a FANTNet contém comutadores de borda programáveis e comutadores convencionais que são gerenciados utilizando apenas um con-

trolador logicamente centralizado. Além de descrever as funções desses comutadores, esse capítulo forneceu também uma especificação em alto nível de um elemento de rede utilizado para adaptar o tráfego NDN nativo antes de entrar na *Fabric*. Foram apresentadas as estruturas de dados do plano de controle necessárias para configurar o plano de dados a fim de garantir o encaminhamento de pacotes NDN. Essas estruturas de dados incluem os algoritmos para a inserção, atualização e remoção de prefixos nomeados. As fases de operação da FANTNet também foram descritas. O próximo capítulo detalha os aspectos do plano de dados da FANTNet.

Capítulo 5

FANTNet: Plano de Dados

5.1 Introdução

A tabela FIB é crucial na arquitetura NDN para garantir que Ipkts cheguem até nós produtores. Dentre as diversas maneiras de se implementar a FIB em um roteador NDN, as tabelas de *hash* surgem como alternativas viáveis, principalmente devido a eficiência dessas estruturas no que diz respeito ao consumo de memória. No entanto, a possibilidade de encaminhamentos de pacotes para portas de saída incorretas, devido a colisões de *hash*, torna o desenvolvimento dessas estruturas mais complexo, principalmente quando elas são implementadas em *hardware*. Assim, esse capítulo apresenta os principais elementos do plano de dados da arquitetura FANTNet. Dentre esses elementos, esse capítulo apresenta a proposta de uma estrutura de dados específica para a FIB, chamada CoFIB, empregada em comutadores de borda programáveis, projetada para lidar com as restrições de *hardware* e equilibrar o consumo de memória com velocidade de comutação.

O plano de dados na arquitetura FANTNet é responsável por realizar o encaminhamento de pacotes NDN entre comutadores de borda. Dessa forma, esse capítulo se concentra nos principais componentes do plano de dados em comutadores de borda programáveis tendo em vista que o encaminhamento de pacotes nos comutadores do núcleo é simplificado pelo uso das tabelas FST. Assim, a Seção 5.2 apresenta a proposta de uma nova representação de nomes para pacotes NDN que tem a finalidade de simplificar o processamento nos comutadores P4. Na Seção 5.3, as principais estruturas de dados que compõem a CoFIB são descritas, seguida pela Seção 5.4 que apresenta as técnicas de otimização de memória *on-chip* e de reposicionamento de tabelas nos *pipelines* de ingresso e egresso para fins de aumento de vazão. A Seção 5.5 mostra a lógica de extração de cabeçalhos e de processamento nos *pipelines* de entrada e saída para a realização da operação de LNPM. Finalmente, a Seção 5.6 apresenta as considerações parciais do capítulo.

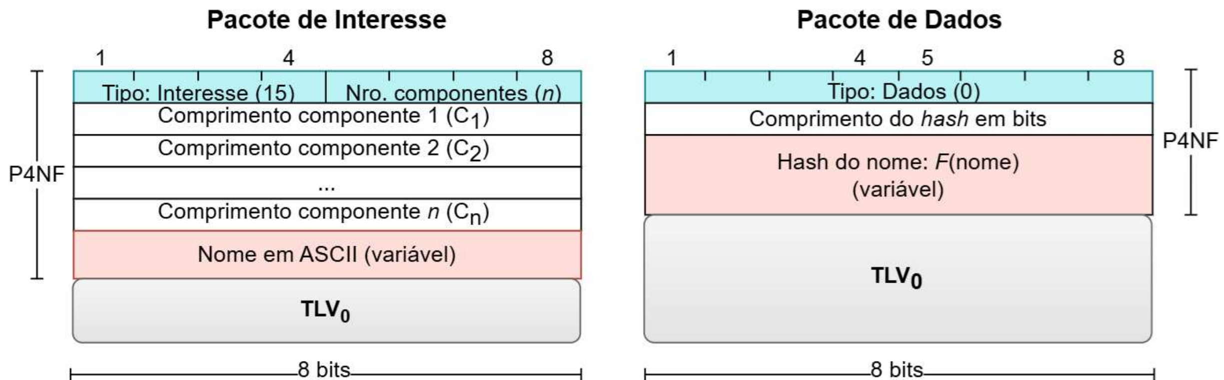

Figura 5.1 – Representação de pacotes de interesse e de dados no formato P4NF.

5.2 Representação de Nomes

5.2.1 Formato P4NF

O primeiro desafio ao projetar uma estrutura de dados para a FIB que rode em comutadores programáveis é como manipular nomes de comprimento variável em P4 tendo em vista que a linguagem não suporta processamento nativo de *strings*. O formato TLV, descrito no Capítulo 2, impõe certa complexidade no processo de extração de cabeçalhos em razão da possibilidade de haverem sub-blocos aninhados. Além disso, para que cada componente do prefixo nomeado seja identificado separadamente, um nome NDN com n componentes é codificado utilizando n blocos TLV. Como um nome NDN é formado por um prefixo e um sufixo, várias operações de atribuição devem ser feitas no bloco de controle do *pipeline* de ingresso para juntar todos componentes que fazem parte do prefixo em uma única chave para que seja possível realizar a operação de *lookup* corretamente, tendo em vista que o sufixo é ignorado para fins de busca na FIB.

Para simplificar o processamento de nomes NDN em P4, propõe-se nessa tese o *P4 Name Friendly Format* (P4NF), uma nova codificação para nomes NDN de forma aderente às restrições de *targets* P4 e que visa facilitar o *parsing* e a operação de *lookup*. O formato proposto é híbrido no sentido de conter campos específicos para a representação do nome bem como blocos TLV nativos para a representação dos demais campos do pacote. Conforme mostra a Figura 5.1, no formato proposto, os comprimentos de cada componente do prefixo são colocados em sequência e precedem os demais campos que continuam no formato TLV.

Para Ipkts, os quatro primeiros *bits* no formato P4NF tem valores 1 (representam o valor 15 em decimal) e os quatro *bits* seguintes representam a quantidade de componentes do prefixo. Por convenção, embora quatro *bits* possibilitem representar até 15 componentes, o número máximo de componentes suportado na FANTNet é 8, o que não é problema visto que prefixos NDN maiores são incomuns, de acordo com análises estatísticas rea-

lizadas em *datasets* de nomes como em Rosa e Silva (2022a). Desse modo, o segundo grupo de quatro *bits* em Ipkts contém valores que variam entre 1 e 8, onde qualquer valor diferente disso torna o Ipkt inválido. Os n campos seguintes (C_i) armazenam os tamanhos de cada componente do prefixo e é representado em P4 através de uma estrutura do tipo *header_stack*, onde cada campo C_i é representado por um valor de 8 *bits*. Apesar de 8 *bits* possibilitar a representação de componentes de até 255 caracteres, para reduzir o consumo de memória *on-chip* ao armazenar os prefixos na CoFIB, o número máximo de caracteres por componente suportado na FANTNet é 31. O próximo campo armazena o nome NDN codificado em ASCII com caracteres de 8 *bits*. Esse campo é representado em P4 por meio do tipo *varbit*. O comprimento desse campo (em *bytes*) é obtido iterativamente durante o processo de extração de cabeçalhos pelo somatório $L = \sum_{i=1}^{n} c_i$, onde $1 \leq L \leq 248$.

Como não existe necessidade de processar nomes em Dpkts para fins de LNPM, a codificação desse tipo de pacote no formato P4NF é mais simples e consiste na adição de um bloco TLV especial no início do pacote, conforme mostra a Figura 5.1. O campo *type* desse bloco é representado como uma constante de 8 *bits* contendo uma sequência de 0s. O campo *length* é também representado em 8 *bits* e indica a quantidade de *bits* do *hash* do prefixo pré-calculado no NPC. Assim, o formato proposto suporta *hashes* de nomes de até 255 *bits*. Por convenção, nessa tese o tamanho do *hash* de nomes em Dpkts é 32. O campo *value* seguinte armazena o *hash* propriamente dito do nome pré-calculado no NPC.

5.2.2 Extração de Cabeçalhos

Conforme mencionado, a linguagem P4 não foi projetada para processar *strings* de forma eficiente. Uma primeira tentativa de processar nomes de comprimento variável em P4 seria utilizar o tipo nativo *varbit*. No entanto, a especificação da linguagem P4 não permite o uso de campos *varbit* como chaves de busca em tabelas. Uma possibilidade seria copiar o valor contido no campo *varbit* para um campo de comprimento fixo e utilizar esse campo fixo para executar a operação de busca na tabela. Porém, a semântica da linguagem P4 também restringe esse tipo de operação de atribuição. Para resolver essa limitação, propõe-se utilizar a estrutura do tipo *header_union* para agrupar todas as combinações possíveis de cabeçalhos de comprimento fixo. Nessa abordagem, cada campo de comprimento fixo dentro da *header_union* armazena componentes do nome ou seus valores de *hash*, conforme apresentado na subseção 5.3. Uma vez que em uma *header_union* apenas um campo é válido por vez, o *parser* é capaz de extrair o primeiro componente do nome e armazená-lo no correspondente campo de comprimento fixo da *header_union*, fazendo com que os demais campos fiquem inválidos.

O fluxograma para extração de cabeçalhos pode ser visto na Figura 5.2. Detalhes sobre a extração dos campos *Ethernet* e demais protocolos são ignorados por não fazerem parte do escopo dessa tese. Após a leitura do primeiro *byte*, o *parser* primeiramente verifica

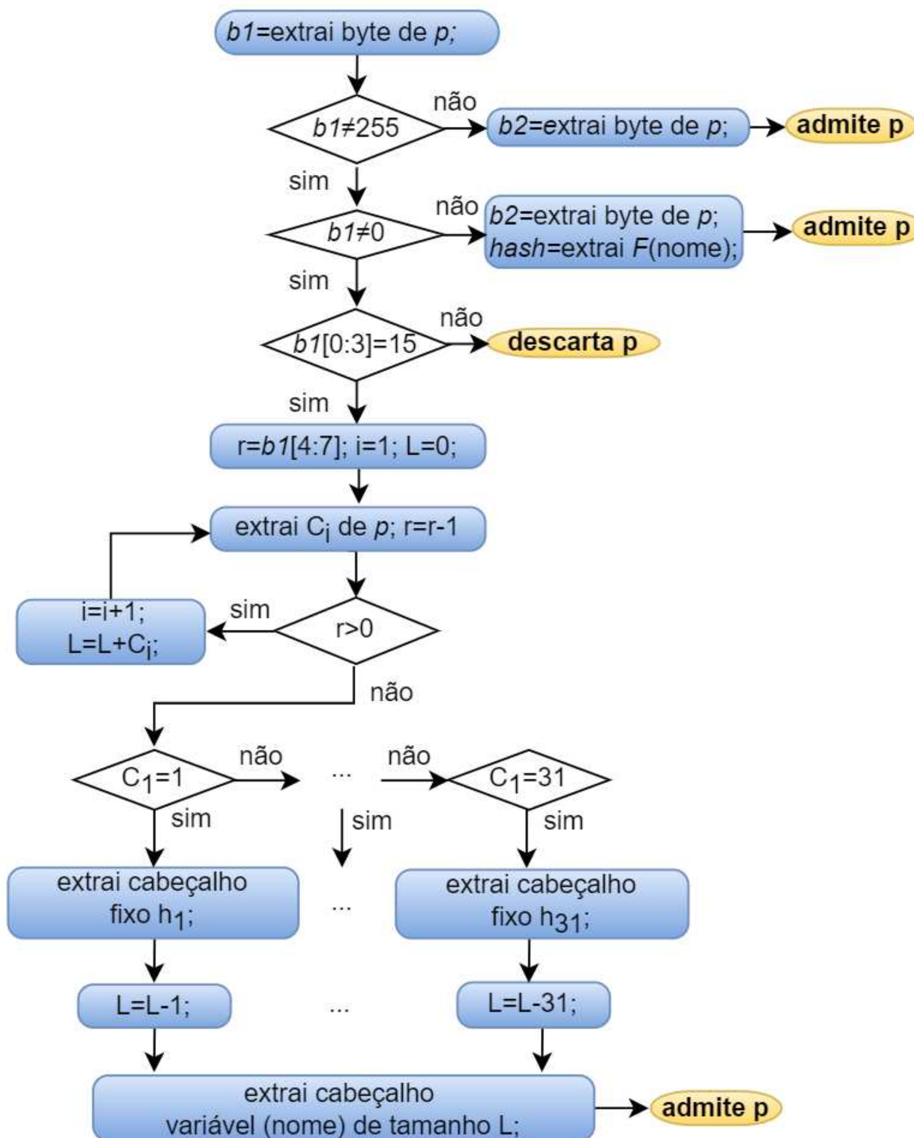

Figura 5.2 – Fluxograma para extração de cabeçalhos.

se o pacote NDN é de dados (primeiro byte igual à 0x00) proveniente do domínio NDN ou se o pacote foi comutado pelo núcleo e está saindo do domínio SDN (primeiro byte igual à 0xFF). Caso o pacote seja de dados, o *hash* do nome é extraído e o pacote segue para o *pipeline* de ingresso. Se o pacote for proveniente de algum comutador de núcleo e o primeiro byte for 0xFF, o segundo *byte* do pacote, que contém o *swId*, é extraído e o pacote é admitido no *pipeline* de ingresso, onde esses campos serão eliminados antes do pacote ser encaminhado para o NPC. Caso essas duas condições não sejam atendidas, o *parser* verifica se o pacote é de interesse e extrai a quantidade de componentes que existem no nome. A partir disso, o comprimento de cada componente C_i é lido sequencialmente para a estrutura *header_stack*. Quando o último elemento é lido, o primeiro elemento da *header_stack* é usado como índice para a leitura do próximo campo que corresponde ao primeiro componente do nome. Esse valor é lido para o respectivo campo de comprimento

fixo no *header_union*. O campo *varbit* final é então usado para armazenar o restante do pacote. É imporante observar que o primeiro componente é armazenado em um campo de comprimento fixo e, portanto, pode ser utilizado na operação de busca na tabela posteriormente. Após a leitura do último campo, o pacote é então aceito e transferido para o *pipeline* de entrada.

5.3 Estruturas de Dados

Conforme apresentado no Capítulo 2, Seção 2.3.1, existem diversas arquiteturas de dispositivos P4 existentes atualmente. O que todas elas tem em comum é um *pipeline* onde os pacotes iniciam sua jornada de processamento após terem os cabeçalhos extraídos pelo *parser*. As subseções seguintes descrevem as estruturas de dados existentes no plano de dados da CoFIB, as técnicas de otimização propostas para reduzir o consumo de memória e aumentar a vazão, além da lógica da operação LNPM.

5.3.1 Tabela CS

A implementação da estrutura de dados CS está fora do escopo dessa tese. No entanto, essa subseção descreve brevemente como essa tabela pode ser implementada na FANTNet e onde cada componente pode ser desenvolvido.

A CS pode ser implementada em *software* no plano de controle em razão da falta de memória *on-chip* para armazenar Dpkts diretamente na SRAM do dispositivo programável. Perino et al. (2014) empregam essa abordagem. Para evitar que Ipkts consultem constantemente o controlador SDN para verificar a existência de cópia de Dpkts na CS, é possível implementar um filtro de *bloom* no comutador de borda programável utilizando a memória disponível na forma de *registers*. Como os *registers* são memórias que mantém estados entre pacotes e que permitem a leitura e escrita *on-the-fly*, é possível utilizar o *hash* pré-calculado pelo NPC para indexar o filtro de *bloom* a fim de verificar a existência de determinado Dpkt na CS. Como em filtros de *bloom* não existe a possibilidade de falsos negativos, é possível ter a garantia de que determinado Dpkt não existe na CS sem ter que consultar o plano de controle.

5.3.2 Tabela PIT

Assim como a CS, a implementação da tabela PIT também está fora do escopo dessa tese. No entanto, essa subseção descreve de maneira geral como a PIT pode ser implementada nos comutadores de borda programáveis da FANTNet.

A quantidade de armazenamento requerida pela PIT é menor que a quantidade de memória requerida pela CS. Isso ocorre porque a PIT não armazena Dpkts e sim Ipkts, tipicamente menores. Por essa razão, é possível e viável implementar a PIT no plano de

dados mesmo diante da escassez de memória *on-chip*. De maneira geral, a forma mais usual de implementar a PIT em comutadores programáveis é por meio dos *registers*, como em Signorello et al. (2016) e Miguel, Signorello e Ramos (2018). A utilização de bancos de *registers* possibilita que entradas na PIT possam ser atualizadas por Ipkts e Dpkts *on-the-fly*, sem a interferência do plano de controle.

5.3.3 Tabelas de moldes

Para evitar processar Ipkts contendo prefixos nomeados que não existem, utiliza-se nessa tese uma adaptação do conceito de moldes de prefixo, introduzido inicialmente por Huang e Wang (2018). No entanto, os moldes usados nessa tese são de comprimento fixo. A ideia é evitar o LNPM desnecessário quando o prefixo não existe na memória *on-chip* no plano de dados ou na CFIB no plano de controle. De forma simplificada, o molde de prefixo representa a sequência dos comprimentos de cada componente do prefixo nomeado em suas respectivas posições. Por exemplo, o molde do prefixo */br/ufu/facom* é */2/3/5*. Assim, duas tabelas complementares são usadas no plano de dados além daquelas necessárias para armazenar os prefixos nomeados canônicos. A primeira tabela é chamada *Data Plane Shape Table* (DPST) e armazena os moldes de todos os prefixos canônicos existentes no plano de dados. Já a segunda é chamada de *Control Plane Shape Table* (CPST) e armazena os moldes dos prefixos existentes no plano de controle, especificamente na CFIB. Tanto a DPST quanto a CPST são armazenadas na memória TCAM onde cada entrada é dada pelo molde do prefixo como chave e o número de componentes como parâmetro. Esse número de componentes representa a quantidade máxima de operações de combinação-ação que serão realizadas durante a operação LNPM.

Tendo em vista que as entradas nas tabelas CPST e DPST precisam ter comprimento fixo em função dos requisitos da linguagem P4, é necessário converter os moldes de tamanho variável em moldes de tamanho fixo. Essa conversão é realizada concatenando-se os comprimentos dos componentes individuais e completando-os com zeros até um limite máximo. Para ilustrar, o molde do prefixo $/nc_1/../nc_k$ é dado por uma chave de 40 *bits* obtida fazendo-se a concatenação $l(nc_1){+}{+},..,{+}{+}l(nc_k){+}{+}p(k)$, onde k é o número de componentes, $l(x)$ é uma função que retorna um valor de 5 *bits* representando o número de caracteres no componente x, e $p(k)$ uma função que retorna uma *string* de *bits* 0s cujo comprimento é dado por $40{-}5k$. Assim, se a busca na tabela DPST por um molde, usando o LPM, resultar em sucesso, o número máximo de componentes para prefixos com aquele molde é obtido calculando-se $max{=}8{-}k$ em uma ação definida em P4.

Quando um Ipkt chega ao roteador NDN, as tabelas DPST e CPST são consultadas para verificar a existência de algum molde que compatibiliza. Caso o molde do prefixo contido no Ipkt não exista nessas tabelas, o pacote é descartado de acordo com o procedimento padrão da arquitetura NDN. Isso evita a execução desnecessária do LNPM e contribui para a diminuição da latência de processamento por pacote. Por outro lado,

caso o prefixo exista apenas na DPST ou na CPST, o procedimento de LNPM é executado recirculando o pacote no *pipeline* ou o pacote é enviado para o plano de controle para ser consultado na CFIB, respectivamente. A ideia central de se utilizar essas tabelas é reduzir o número de recirculações desnecessárias no *pipeline* de processamento.

5.3.4 Tabelas para armazenamento de prefixos

A estrutura de dados CoFIB é representada por meio de um conjunto de tabelas $T_1, .., T_k$ no *pipeline* de entrada. Esse conjunto de tabelas é denonimado *Dataplane FIB* (DFIB). Cada tabela T_i é responsável por armazenar componentes individuais contendo i caracteres, com $1 \leq i \leq k$, cada qual sendo configurada com uma chave de compatibilização exata (*exact-match key*) para forçar o compilador a alocar memória SRAM ao invés de TCAM. Em razão de cada componente do nome na FANTNet ter no máximo 31 caracteres, temos $k = 31$ no modelo proposto.

Para reduzir o consumo de memória, a largura em *bits* de cada tabela T_i é dada pela Equação 1. Assim, a tabela T_i armazena componentes codificados em ASCII de 8 *bits* quando o componente tiver no máximo quatro caracteres ou o *hash* crc32 do componente se este tiver comprimento entre 5 e 31 caracteres. Para exemplificar, o prefixo nomeado */br/ufu/facom* é armazenado na CoFIB inserindo o componente '*br*' na tabela T_2, o componente '*ufu*' na tabela T_3 e $h(facom)$ na tabela T_5, onde $h(x)$=$\mathbf{crc32}(x)$.

$$W(T_i) = \begin{cases} 8i, & \text{se } 1 \leq i \leq 4. \\ 32, & \text{caso contrário.} \end{cases} \tag{1}$$

Como se pode observar, a CoFIB armazena componentes do nome de forma individual em diferentes tabelas conforme seu comprimento. Uma questão que surge a partir disso é como associar cada componente de forma que seja possível identificar cada prefixo nomeado. A subseção seguinte descreve como esse problema é resolvido.

5.3.5 Estrutura CAD

Para garantir que componentes individuais estejam conectados, cada tabela T_i está associada ao valor CAD gerado no plano de controle, brevemente discutido na Seção 4.6. A estrutura CAD inclui informações essenciais para possibilitar a operação de LNPM no plano de dados. Assim, cada entrada na estrutura CoFIB é representada por uma tupla $\langle nc, cad \rangle$, onde nc é o componente nomeado em ASCII quando $1 \leq |nc| \leq 4$, ou $h(nc)$, caso contrário.

Embora a estrutura CAD deva incluir todas as informações necessárias para possibilitar a operação de LNPM, ela deve usar a menor quantidade possível de *bits* para reduzir o consumo de memória, tendo em vista que a quantidade de TCAM/SRAM em comutadores programáveis é escassa. Uma primeira abordagem seria considerar o CAD com 16

bits. Como a arquitetura NDN não restringe em qual posição cada componente aparece no prefixo, para codificar as 8 posíveis posições em que cada componente pode aparecer seriam necessários no mínimo 8 *bits* por componente, aumentando o consumo de memória significativamente. A razão disso é que, no pior caso, um componente x poderia aparecer em 8 posições distintas, o que exigiria um valor de 8 bits para indicar as posições em que x aparece no prefixos da FIB. Para resolver esse problema, essa tese introduz o conceito de prefixo nomeado canônico, conforme especificado na Definição 1 e brevemente discutido na Seção 4.5. É importante salientar que o prefixo canônico apresentado aqui é diferente do conceito de nomes canônicos (*Canonical Name* (CNAME)) usado no *Domain Name System* (DNS) (IETF, 1987).

Definição 1 (Prefixo Nomeado Canônico)**.** *Seja* $P = \{\rho_1, , .., \rho_m\}$ *um conjunto de* m *prefixos nomeados. O prefixo* $\rho_j \in P$*, representado pela sequência de componentes* $/nc_1/../nc_k$*, é canônico se todas as possíveis ocorrências de seus componentes* nc_i *em todos os prefixos de* P*, para* $1 \leq i \leq k$*, aparecem na posição* i*.*

Para ilustrar, considere dois conjuntos de prefixos: P=\{*/a/b/c*, */x/b*, */f/b/c/t*\} e P'=\{*/a/b/c*, */b/c*, */x*\}. Como se pode observar, todos os prefixos em P são canônicos uma vez que cada componente em todos os prefixos aparecem somente em uma única posição. Já em P', o prefixo */b/c* não é canônico visto que o componente '*b*' aparece na segunda posição em */a/b/c* e na primeira posição em */b/c*. Assumindo que os prefixos da FIB são canônicos, é possível codificar a posição de qualquer componente usando apenas três *bits* ao invés de oito *bits*, reduzindo o consumo de memória.

A estrutura CAD nessa primeira abordagem é dada pela 5-tupla cad=$\langle c, e, cf, p, out\rangle$, representada internamente por um valor de 16 *bits*. Assim, uma entrada FIB é dada pela tupla $\langle nc, cad\rangle$, conforme mencionado anteriormente. O *bit cad.c* indica se existem prefixos na FIB que contenham *nc* e pelo menos um componente nomeado nc' em uma posição maior que a posição de *nc*. Por exemplo, para o conjunto de prefixos canônicos P=\{*/a/b/c*, */x/b*, */f/b/c/t*\}, o *bit cad.c* associado com o componente c e t são 1 e 0, respectivamente. O *bit cad.e* indica se o componente *nc* está localizado na última posição em pelo menos um prefixo da FIB. Tomando o conjunto P como exemplo, o *bit cad.e* associado aos componentes c e a são 1 e 0, respectivamente. O próximo *bit cad.cf* indica se o prefixo que contém *nc* é conflitante. Considerando o conjunto P como exemplo, o *bit cad.cf* associado aos componentes b e t são 1 e 0, respectivamente. A definição formal de um prefixo nomeado conflitante é apresentada na Definição 2.

Definição 2 (Prefixo Nomeado Conflitante)**.** *Seja* P *um conjunto de prefixos canônicos e* $\rho \in P$*, com* ρ*=* $/nc_1/../nc_k$ *sendo* nc_i*=componente nomeado na posição* i*. Considera-se* ρ *um prefixo nomeado conflitante se, e somente se,* $\exists \rho' \in P$ *com* $\rho \neq \rho'$ *e* ρ'*=*$/nc'_1/nc'_2/../nc'_q$*, tal que* $\exists i$ *onde* $2 \leq i \leq k$ *com* $nc_i = nc'_i$ *e* $(/nc_1/../nc_{i-1}) \neq (/nc'_1/../nc'_{i-1})$*.*

Um prefixo nomeado conflitante contém pelo menos um componente nc_j tal que $nc_j.cad.cf{=}1$. Nesse caso, uma tabela com chaves de compatibilização exata, chamada *Hash Conflicting Table* (HCT), é usada para armazenar a tupla $\langle key, swId \rangle$, onde key é um valor de 32 *bits* dado pela relação de recorrência $F(\rho, nc_i)$ descrita na Equação 2, considerando m um número primo suficientemente grande e $swId$ armazenando o conjunto de comutadores de borda que respondem por prefixos que contenham nc_i, caso $nc_i.cad.e = 1$, ou $swId = 0$ se $nc_i.cad.e = 0$.

Para entender a necessidade de utilizar a função de recorrência ao invés de calcular o *hash* do subprefixo a cada iteração, é preciso compreender como é realizada a compatibilização de prefixos na CoFIB. A operação de LNPM é executada iterativamente para cada componente do nome. Dessa forma, a cada iteração, é necessário que o componente nomeado seja lido individualmente para um cabeçalho de tamanho fixo, conforme mostra a Figura 5.2. Isso torna possível a realização da busca do componente (ou seu *hash*) na tabela P4 correspondente. Todavia, o valor do *hash* do componente em uma dada posição não é suficiente para identificar o subprefixo inteiro. Assim, é necessário realizar a combinação sucessiva de valores de *hash* a medida em que eles vão sendo calculados a cada passagem no *pipeline* para execução da operação LNPM. A função de recorrência apresentada na Equação 2 tem a finalidade de justamente fazer essa combinação.

Como cada prefixo inserido nas tabelas P4 é canônico, existe somente uma posição i na qual qualquer componente pode aparecer, onde $1 \leq i \leq 8$. Essa informação é codificada no campo de 3 *bits* *cad.p*, onde *p*=000 equivale à primeira posição e *p*=111 à última. Considerando o conjunto P apresentado anteriormente, o campo *cad.p* associado com os componentes '*b*' e '*c*' são 2 e 3, respectivamente. Finalmente, o campo *cad.out* armazena um valor de 10 *bits* que indica as portas de saída. Tal configuração suporta transmissões *multicast* com dez possíveis interfaces de saída e prefixos com até oito componentes. Como exemplo, *cad.out*=1100000101 indica que o Ipkt deve ser enviado para as portas 1, 3, 9, e 10. Isso é possível porque em comutadores programáveis as interfaces são do mesmo tipo.

A estrutura CAD de 16 *bits* é uma boa abordagem para reduzir o consumo de memória. No entanto, isso pode ocasionar encaminhamentos incorretos devido a falsos positivos. Como exemplo, vamos assumir o seguinte conjunto de prefixos nomeados com seus respectivos valores de *swId*: $P{=}\{/a/b/c \rightarrow 3,\ /a/b \rightarrow 4,\ /x/b/l \rightarrow 5,\ /k/d \rightarrow 6\}$. Suponha agora que um Ipkt contendo o prefixo */k/b* chegue ao roteador NDN. Como se pode observar, o prefixo */k/b* é conflitante com o prefixo */a/b*. Essa situação é conhecida como o problema do prefixo cruzado pois, embora os componentes k e b aparecem em suas posições corretas, a DFIB não contém o prefixo */k/b*. Sendo assim, o Ipkt contendo o

$$F(\rho, nc_i) = \begin{cases} m * F(\rho, nc_{i-1}) + h(nc_i), & \text{se } i > 1. \\ h(nc_1), & \text{caso contrário.} \end{cases} \tag{2}$$

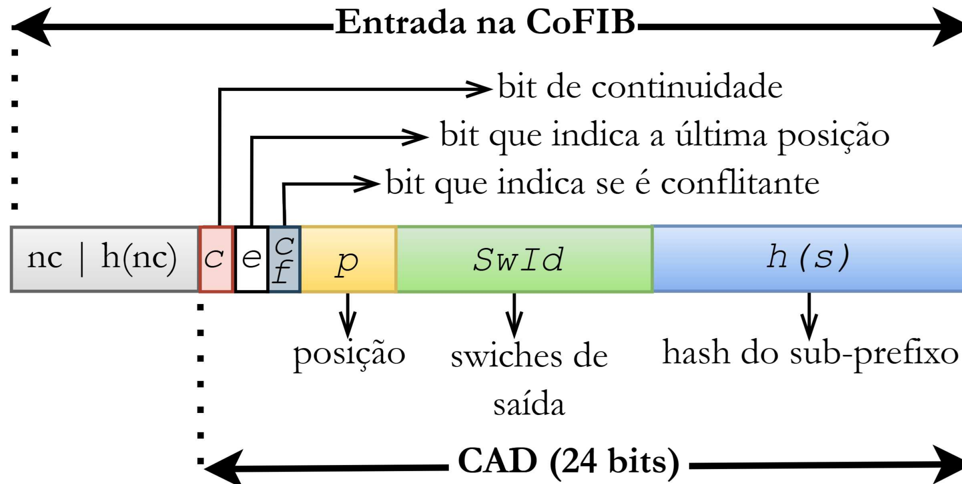

Figura 5.3 – Formato de entrada em uma tabela da CoFIB.

prefixo */k/b* será enviado incorretamente para o comutador de borda dado por $swId$=4 ao invés de ser descartado.

Para resolver o problema do prefixo cruzado, um novo campo de 10 *bits* ($cad.hs$) é adicionado à estrutura CAD para identificar de forma unívoca o subprefixo associado ao componente nomeado corrente $nc_i|h(nc_i)$ (nc_i refere-se ao componente em ASCII caso $|nc_i| \leq 4$ ou $h(nc_i)$ caso contrário). Como o campo $cad.hs$ contém 10 *bits*, seu valor é dado pelos dez *bits* mais significantes obtidos por $F(\rho, nc_{i-1})$. Por exemplo, se ρ=*/a/abcde*, então nc_2=*abcde* com $nc_2.cad.hs$=$(F(/a/abcde, a) \gg 22$ e nc_1=*a* com $nc_1.cad.hs$=0, tendo em vista que não há subprefixo para componentes na primeira posição. Com essa configuração, o campo *cad.out* passa a ter oito *bits* ao invés de dez a fim de que o tamanho da CAD seja múltiplo de oito. Consequentemente, é possível garantir o suporte a transmissões *multipath* para até oito possíveis comutadores de borda que respondem pelo prefixo. Porém, esse custo é compensado pela adição de um ponteiro para o subprefixo associado ao componente, evitando-se assim o problema do prefixo cruzado.

Em um primeiro momento, é possível argumentar que a utilização de apenas 10 *bits* para o campo *cad.hs* seria insuficiente para eliminar os falsos positivos tendo em vista que, para cada componente nomeado nc_i da FIB, existem apenas 1024 possibilidades de subprefixos ($2^{|cad.hs|}$). Todavia, isso não representa um problema, pois esse campo identifica somente a primeira ocorrência do subprefixo associado ao correspondente componente. As demais ocorrências são armazenadas na tabela HCT. A quantidade de *bits* do campo *cad.hs* somente impacta a probabilidade de colisão entre um dado prefixo em um Ipkt e os prefixos na DFIB. Tal probabilidade é baixa considerando que ambas tabelas DPST e CPST evitam o processamento de muitos prefixos não existentes na DFIB. Além disso, mesmo se um prefixo não existente na DFIB passar pelo filtro das tabelas de moldes, todos os componentes do nome devem compatibilizar nas tabelas da DFIB, o que é pouco provável. Para exemplificar, suponha o conjunto de prefixos P=$\{/a/b/c$,

/x/b, */f/b/c/t*} e nc_2='b'. Como nc_2 é conflitante, então $nc_2.cad.hs$ armazenará apenas a primeira ocorrência de subprefixo de 'b', nesse caso $h(/a)$, e todas as demais ocorrências ($h(/x)$, $h(/f)$) são armazenadas na HCT utilizando todos os 32 *bits* do *hash* calculado no plano de controle. Essas operações podem ser vistas mais detalhadamente no Algoritmo 3.

5.4 Otimização de Recursos

5.4.1 Otimização de Memória *On-Chip*

O uso de *hashes* crc32 para compactar alguns componentes nomeados, o conceito de prefixos canônicos e o número reduzido de *bits* na estrutura CAD são algumas das técnicas de otimização de memória usadas nessa tese para permitir que a CoFIB seja armazenada no espaço limitado da memória SRAM/TCAM. Como as tabelas P4 necessárias para armazenar a CoFIB contêm apenas uma ação, é possível otimizar ainda mais o consumo de memória usando *action profiles* e *action selectors* de acordo com a especificação da linguagem P4 (ONF, 2023).

Para ilustrar, uma tabela P4 tradicional com M entradas normalmente consumirá M*W *bits* no plano de dados, onde W é o tamanho dos parâmetros de ação. Se os parâmetros de ação das M entradas forem diferentes entre si, talvez não seja possível reduzir o consumo de memória. No entanto, para uma determinada tabela T que requer no máximo N diferentes conjuntos de valores de parâmetros de ação, com $N < M$, o elemento *action profile* pode reduzir o consumo de memória *on-chip* ao possibilitar o agrupamento de múltiplas ações e usando tabelas indiretas. Como as tabelas da CoFIB requerem que apenas uma ação seja executada, independentemente dos *hits* e *misses* que ocorrem nas operações de *lookups* nas tabelas, implementa-se nessa tese uma versão da CoFIB que utiliza *action profiles* e *action selectors* em vez de tabelas P4 tradicionais. O consumo de memória de uma tabela particular T usando esta abordagem é dado pela Equação 3:

$$T_{mem} = M * log_2(N) + N * W \quad (3)$$

Vale ressaltar que, em alguns casos, o uso de *action profiles* e *action selectors* pode aumentar o consumo de memória de uma determinada tabela. Isto inclui cenários onde existem muitos conjuntos distintos de parâmetros de ação. Por exemplo, assumindo W=96 *bits*, uma FIB com M=1.000 entradas e 900 parâmetros de ação distintos (N=900), então o consumo de memória usando uma tabela P4 convencional será de 96.000 *bits* enquanto usar um *action profile* exigirá 96.400 *bits* de armazenamento. Portanto, para usar *action profiles* em vez de tabelas P4, a desigualdade dada pela Equação 4 deve ser verdadeira, onde o tamanho da chave em *bits* (K) é $8i$ para a tabela T_i com i<5 ou K=32 para

$i > 4$. De acordo com as características dos *datasets* de nomes usados nesta tese, que serão apresentados no Capítulo 6, todas as tabelas da DFIB são adequadas para serem implementadas usando *action profiles*.

$$M * (K + log_2(N)) + 24 * N < (K + 24) * M \quad (4)$$

5.4.2 Otimização de Vazão

A quantidade de memória TCAM/SRAM disponível nos ASICs programáveis atuais é normalmente distribuída igualmente entre os *pipelines* de entrada e saída. Nessa arquitetura, seria muito mais simples colocar as tabelas da CoFIB apenas no *pipeline* de entrada a fim de simplificar a lógica da operação LNPM. No entanto, tal abordagem desperdiçaria metade da memória *on-chip* disponível. Assim, o objetivo de projeto da estrutura CoFIB é distribuir as tabelas pelos *pipelines* de entrada e saída. Uma solução simples é colocar as tabelas linearmente ao longos dos dois *pipelines*. Essa abordagem nessa tese é chamada de posicionamento de tabela linear, onde as primeiras 14 tabelas da DFIB ($T_{i<15}$) são colocadas na entrada (*ingress pipeline*) enquanto as tabelas restantes ($T_{i \geq 15}$) são colocadas na saída (*egress pipeline*).

A estratégia de posicionamento de tabela linear tem algumas implicações. Uma delas é que a tabela HCT deve ser duplicada nos *pipelines* de entrada e saída. Isso ocorre porque a operação de LNPM precisa identificar prefixos conflitantes sempre que ocorre uma operação de combinação-ação. No entanto, como será visto no Capítulo 6, como o tamanho da tabela HCT é algumas ordens de magnitude menor do que as tabelas da DFIB, tal redundância não afeta significativamente o consumo de memória considerando os experimentos realizados nessa tese.

Além de potencialmente utilizar toda a memória *on-chip* disponível no ASIC do comutador programável, a abordagem de posicionamento de tabela linear permite duas operações de combinação-ação por passagem no *pipeline* durante a operação de LNPM. Isto é significativo, pois é possível reduzir o número de passagens de *pipeline* para alguns Ipkts, reduzindo-se a latência média por pacote e aumentando a vazão ao longo do tempo. Por exemplo, Ipkts contendo dois componentes nomeados podem experimentar apenas uma passagem de *pipeline* caso o primeiro componente corresponda na entrada e o segundo componente corresponda na saída. Duas passagens no *pipeline* seriam necessárias para completar o LNPM se as tabelas fossem colocadas apenas na entrada.

A operação de combinação-ação na CoFIB funciona da seguinte maneira. Sempre que ocorre uma consulta na tabela, seja na entrada ou na saída, uma ação é invocada para extrair o *SwId* e os outros campos da estrutura CAD. Em implementações FIB tradicionais, como (KARRAKCHOU; SAMAAN; KARMOUCH, 2020b), (YI et al., 2014), a FIB é implantada em todos os roteadores NDN, e as interfaces de saída para encaminhar o pacote são recuperadas da memória quando ocorre uma correspondência. No entanto, a

Algoritmo 6 Otimização de Posicionamento de Tabela

1: ENTRADA: Prefixos $P = \{p_1, ..., p_m\}$ da RIB
2: SAÍDA: Tabelas de *hash* de Ingresso (IG) e Egresso (EG).
3: **função** *table_placement*() :
4: Seja M uma matriz tal que $M \in \mathbb{Z}^{32\times 9}$ com $M[i][j]$ armazenando o # de componentes com i caracteres na posição j
5: Seja H um array de tabelas de *hash* onde $H[i]$ armazena valores *crc32* de componentes com i caracteres
6: **para cada** $p_j \in P$ **faça**
7: **para cada** nc_i in p_j **faça**
8: $len \leftarrow$length(nc_i); $h \leftarrow$crc32(nc_i)
9: $H[len]$.putIfAbsent(h)
10: $M[len][i] \leftarrow M[len][i] + 1$
11: Seja V um array contendo a matriz M serializada onde $V[i]$ é a 3-triple *<frequency, position, size>* associada com cada componente nc_i.
12: Ordene o array V em ordem descendente com base na frequência
13: **para** i de 1 até $|V|$ **faça**
14: **se** $V[i]$.frequency == 0 **então**
15: *break*;
16: tableName ← "T"+ $V[i]$.size;
17: **se** (i mod 2) == 1 **então**
18: **se** tableName $\notin (IG \cup EG)$ **então**
19: IG.put(tableName);
20: **senão**
21: **se** tableName $\notin (IG \cup EG)$ **então**
22: EG.put(tableName);
23: **return** IG, EG

CoFIB foi projetada para funcionar apenas nos comutadores de borda. Portanto, o *SwId* não aponta diretamente para as interfaces de saída, mas para os *switches* de borda que podem recuperar os Dpkts correspondentes. As portas de saída para enviar os Ipcks aos comutadores de núcleo são selecionadas aleatoriamente, onde esses Ipcks serão encaminhados utilizando as tabelas FST. Esta abordagem é impossível em implementações FIB tradicionais porque a decisão sobre quais portas enviar os pacotes deve ser tomada apenas no *pipeline* de entrada.

Apesar do benefício de usar a estratégia de posicionamento de tabela linear, uma questão que surge neste ponto é: *É possível reorganizar as tabelas da CoFIB nos blocos de ingresso e egresso de forma a reduzir o número de passagens no pipeline em contraste com a abordagem de posicionamento de tabela linear*? Para responder a essa questão, é necessário analisar as características dos prefixos canônicos, como o tamanho dos componentes nomeados e suas respectivas posições, antes de serem inseridos na CoFIB.

Com base nos prefixos canônicos extraídos da RIB, propõe-se nessa tese um método para reposicionar as tabelas no *pipeline* de maneira otimizada. A ideia é calcular a frequência de cada componente nomeado e colocar iterativamente as tabelas da DFIB

nos *pipelines* de entrada e saída de acordo com os componentes nomeados que são mais frequentes, seus respectivos tamanhos e posições. Por exemplo, o componente nomeado '*com*' é o mais frequente entre todos os prefixos canônicos e aparece na posição 1. Portanto, a estratégia de otimização coloca a tabela T_3 no *pipeline* de entrada. Então, supondo que o segundo componente mais frequente seja '*data*', a tabela T_4 pode ser colocada no *pipeline* de saída. Consequentemente, Ipkts que carregam prefixos contendo ambos os componentes experimentarão duas correspondências em uma única passagem no *pipeline* em contraste com a estratégia de posicionamento linear de tabela. O Algoritmo 6 mostra a estratégia otimizada de posicionamento de tabela em mais detalhes.

5.5 Lógica do LNPM

A lógica de processamento de Ipkts no pipeline de entrada é apresentada na Figura 5.4. Cada pacote que entra no *pipeline* carrega algumas informações adicionais na estrutura de *metadados* provida pelo *target* P4. Essas informações são usadas para guiar o processo de LNPM e permitem identificar se o pacote está entrando no *pipeline* pela primeira vez (pacote normal) ou se está sendo recirculado ou resubmetido. A razão disso é que certas operações serão realizadas apenas para pacotes normais, enquanto outras serão realizadas para todos os tipos de pacotes, incluindo aqueles que passaram múltiplas vezes no *pipeline*.

Como se pode observar na Figura 5.4, caso o pacote seja normal, os moldes são construídos e duas operações de *lookup* são realizadas, uma na tabela DPST e outra na CPST. Caso o *lookup* na DPST tenha sucesso, o *hash* do componente do nome atual nc_i é calculado bem como a função $F(p, nc_i)$, onde o pacote segue para ter cada um de seus componentes consultados nas tabelas da DFIB. Por outro lado, se o *lookup* na DPST falha, o pacote é enviado para o plano de controle (*Central Processing Unit* (CPU)), caso o *lookup* na CPST tenha tido sucesso, para que seja realizado o *lookup* na CFIB. Se o *lookup* nas duas tabelas de moldes falharem, o pacote é descartado. É importante observar que, ao final de cada operação de *lookup* bem sucedida em alguma tabela de DFIB, o valor de *swId* é extraído do CAD é armazenado no *metadados* até a finalização da operação LNPM.

A execução do algoritmo de LNPM começa quando ocorre um *lookup* bem-sucedido na tabela DPST. Como as tabelas da DFIB são distribuídas nos *pipelines* de entrada e de saída, a operação de LNPM verifica inicialmente se a tabela contendo o primeiro componente do prefixo existe no *pipeline* de entrada. Se isso ocorrer, uma operação combinação-ação é executada na tabela correspondente da DFIB, tendo o primeiro componente nomeado (ou seu *hash*) como chave. Caso contrário, é possível que a tabela esteja no *pipeline* de saída e, nesse caso, o pacote é enviado diretamente para esse estágio.

Durante o percurso no *pipeline*, sempre que uma consulta em qualquer tabela da DFIB for bem-sucedida, o *bit* conflitante da estrutura CAD será verificado. Caso o *bit* esteja

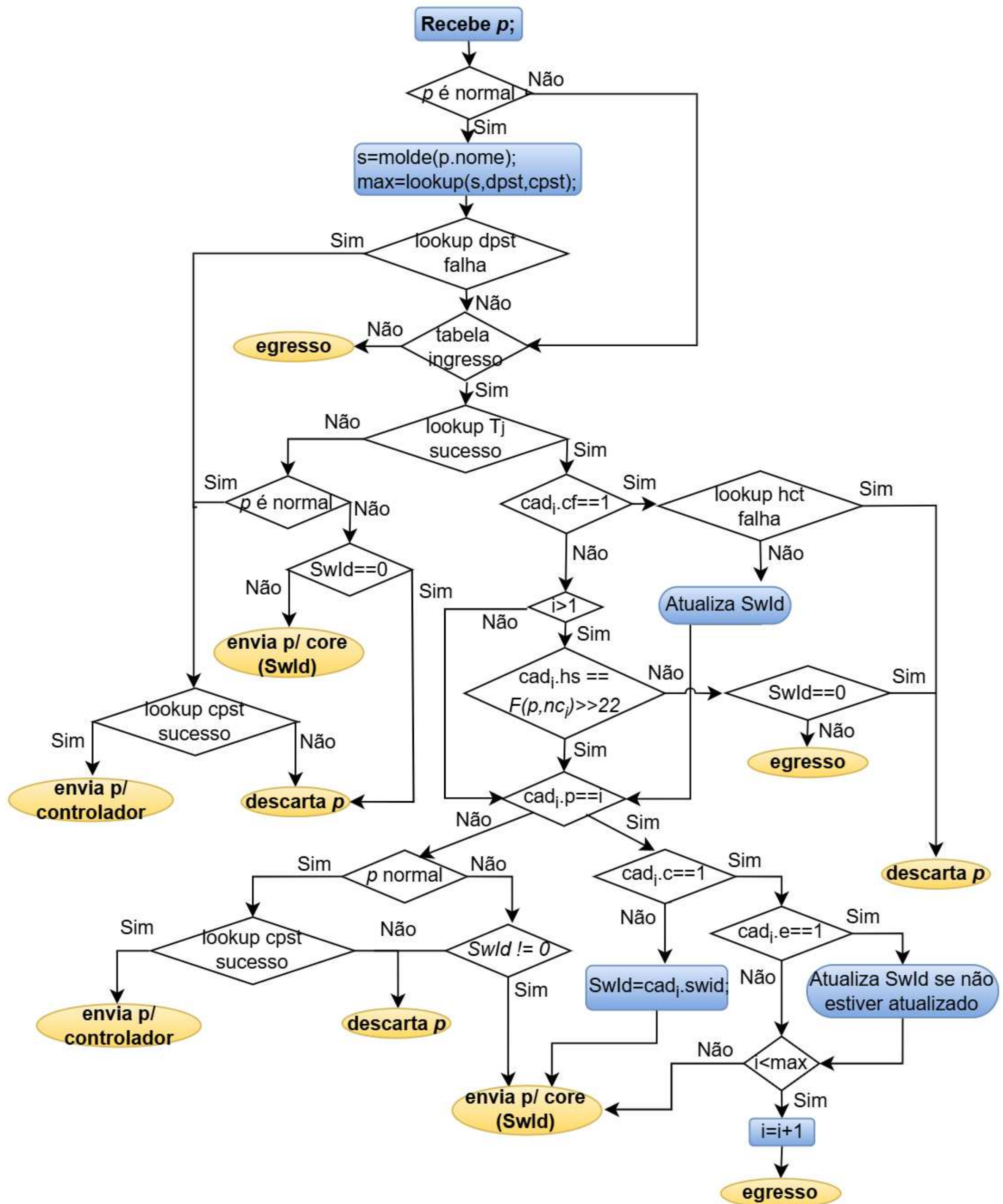

Figura 5.4 – Fluxograma de processamento de pacotes de interesse no *pipeline* de entrada do comutador de borda.

definido, então existe pelo menos um prefixo conflitante armazenado na tabela HCT. Neste caso, é realizada uma consulta na tabela HCT e o resultado é armazenado no *SwId* corrente. Se o *lookup* na HCT falhar mesmo quando o *bit* conflitante estando definido, isso indicará um erro e o pacote será descartado. Por outro lado, se o *bit* conflitante não estiver definido, o valor de cad_i.hs é comparado com os 10 *bits* mais significativos da

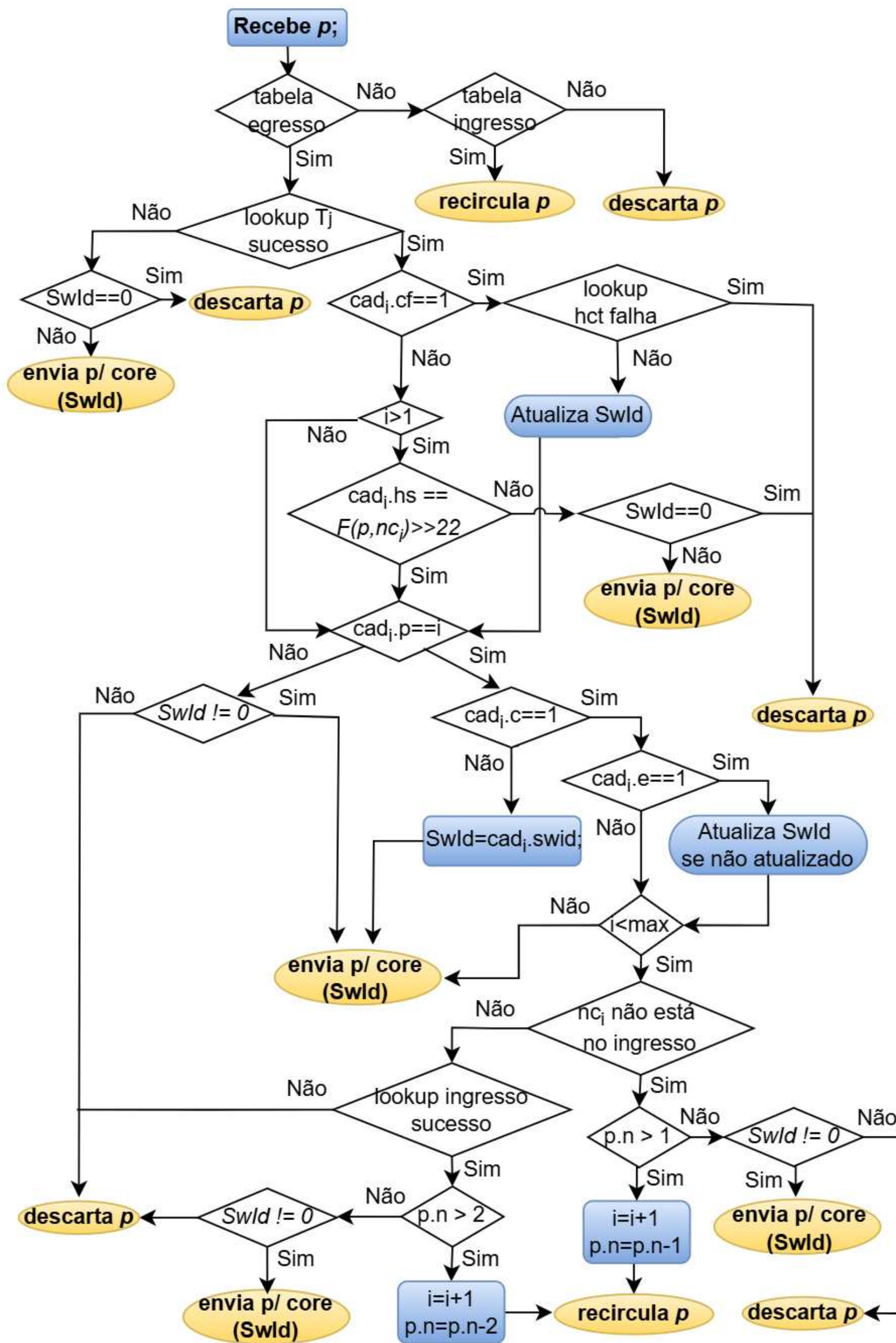

Figura 5.5 – Fluxograma de processamento de pacotes de interesse no *pipeline* de saída do comutador de borda.

função de recorrência $F(p, nc_i)$ para verificar se o componente nomeado atual é associado a qualquer subprefixo anterior existente na DFIB. Isto evita o problema do prefixo cruzado, conforme discutido anteriormente.

Após o Ipkt passar na verificação do *bit* conflitante, é necessário comparar a posição do componente nomeado atual com a posição na estrutura CAD, já que os prefixos na

DFIB são canônicos. Caso a posição seja diferente, o LNPM deverá ser finalizado. Pode-se observar na Figura 5.4 e na Figura 5.5 que é possível enviar um Ipkt para o núcleo mesmo quando a posição do componente nomeado atual difere da posição no CAD, desde que $SwId \neq 0$. Assim, o LNPM pode encaminhar Ipkts que carrega nomes não-canônicos. Por outro lado, se a posição for a mesma, o *bit* de continuidade e o *bit* de final de prefixo da estrutura CAD são usados para decidir se o pacote deve ser enviado para o núcleo (o LNPM está completo) ou deve ser enviado para o *pipeline* de saída para continuar a operação de LNPM.

A figura 5.5 mostra a lógica do LNPM no pipeline de saída. Como pode-se observar, a maioria das operações neste estágio são iguais às do ingresso. A principal diferença é o que acontece com o Ipkt quando o processamento de saída termina, o que pode incluir o envio do Ipkt de volta ao bloco de controle de ingresso para continuar o LNPM. Nesse sentido, o Ipkt será recirculado sempre que a consulta em qualquer tabela da DFIB for bem-sucedida, o *bit* de continuidade for igual a 1 e a posição correspondente do componente nomeado no prefixo for menor que o valor *max*. O valor *max* é obtido durante a operação de pesquisa na tabela DPST que ocorre durante a primeira passagem do Ipkt pelo pipeline de ingresso. Quando a operação LNPM for concluída, o pacote será encaminhado ao núcleo através de uma porta de saída aleatória carregando o valor *swId* mais recente.

Uma observação importante é que, caso o Ipkt seja marcado para recirculação no *pipeline* de saída pela primeira vez, uma cópia dos cabeçalhos originais extraídos pelo *parser* será realizada no *deparser*. Esta operação visa manter os cabeçalhos dos pacotes inalterados para posterior conversão de volta ao formato TLV no próximo nó NPC. Ao invés de duplicar o cabeçalho, uma outra possibilidade é manter todas as informações necessárias para o LNPM nas estruturas de *metadados* disponíveis pelo *target* P4. Todavia, optou-se por não utilizar essa abordagem em razão de sua complexidade. Caso o Ipkt chegue ao o *pipeline* de saída marcado para ser enviado para uma ou mais portas de saída ($SwId \neq 0$), os cabeçalhos originais serão invalidados no *deparser* para que o Ipkt saia do comutador de borda inalterado.

Embora as operações em ambos *pipelines* pareçam complexas à primeira vista, a complexidade computacional de um programa P4 não depende do número de estados nos blocos de controle nem do tamanho do estado acumulado durante o processamento dos pacotes, conforme a especificação da linguagem (ONF, 2023). Assumindo um custo fixo para operações de consulta de tabelas e interações com objetos *extern*, todos os programas P4 executam consistentemente um número fixo de operações para cada byte de um pacote de entrada recebido pelo *parser* e processado nos blocos de controle. Estas garantias proporcionam uma latência previsível e que varia apenas em casos de recirculações, sendo independentes da complexidade do fluxo operacional dos blocos de controle.

5.6 Considerações

Esse capítulo descreveu os componentes do plano de dados da FANTNet, com foco na estrutura de dados CoFIB. O formato P4NF foi apresentado juntamente com a lógica para a extração dos cabeçalhos customizados dos Ipkts. Nos *pipelines* de entrada e saída, foram apresentadas as tabelas para armazenamento moldes (DPST e CPST), as tabelas para armazenamento dos componentes nomeados (DFIB) e a tabela para armazenamento dos componentes conflitantes (HCT). A operação de LNPM ao longo dos blocos de ingresso e egresso foi descrita em função dos fluxogramas apresentados. A distribuição dos prefixos da CoFIB tanto no *pipeline* de ingresso como no de egresso resultou na necessidade de desenvolver uma heurística para posicionamento de tabelas que tem como premissa aumentar a vazão, onde foram empregadas algumas técnicas de compressão de memória utilizando recursos nativos da linguagem P4. O próximo capítulo traz a avaliação experimental da FANTNet e CoFIB bem como fornece uma discussão e comparação qualitativa em relação aos trabalhos na literatura.

Capítulo 6

Avaliação Experimental e Discussão

6.1 Introdução

Essa tese propõe a FANTNet, uma arquitetura inspirada no modelo *Fabric* para viabilizar o encaminhamento de pacotes NDN por meio de uma rede SDN. Os roteadores de borda são programáveis e encaminham Ipkts para outros roteadores de borda do domínio através de uma malha de roteadores de núcleo que podem ser tradicionais ou programáveis. Para maximizar a quantidade de prefixos nomeados que são armazenados na FIB, essa tese propõe a CoFIB, uma estrutura de dados compacta que armazena prefixos canônicos na borda da rede. Tendo em vista que a CoFIB desempenha um papel crucial na arquitetura FANTNet, esse capítulo apresenta os principais resultados da avaliação experimental da estrutura de dados CoFIB utilizando uma instância da FANTNet que serve como prova de conceito.

Esse capítulo está estruturado da seguinte maneira. A Seção 6.2 descreve o método usado para a avaliação da FANTNet e da CoFIB. A Seção 6.3 descreve os *datasets* de nomes que foram utilizados nesse estudo. Esses *dataset*s incluem nomes de domínio reais adaptados para NDN e nomes sintéticos. A Seção 6.4 apresenta uma análise quantitativa, onde os resultados referentes à taxa de descarregamento de prefixos da RIB, a probabilidade de encaminhamento incorreto devido a colisões de *hash* e o consumo teórico de memória *on-chip* são apresentados. A Seção 6.5 apresenta o *testbed* experimental desenvolvido como prova de conceito da FANTNet e a Seção 6.6 descreve os experimentos realizados nesse *testbed*, provendo uma análise dos resultados obtidos. A Seção 6.7 traz uma discussão comparando qualitativamente a FANTNet e a CoFIB com seus pares na literatura. Por fim, a Seção 6.8 apresenta as considerações parciais.

6.2 Método para a Avaliação

A avaliação da arquitetura FANTNet e da estrutura de dados CoFIB é realizada por meio de análises quantitativas, qualitativas e via experimentos de simulação. O desem-

penho do CPE (Algoritmo 2) é medido em função da taxa de *offloading*, que indica o percentual de prefixos canônicos extraídos da RIB. Já a eficiência da estrutura CoFIB é avaliada em relação à probabilidade de falso encaminhamento, consumo de memória, e vazão. O código-fonte da FANTNet, bem como os elementos do plano de dados e controle da CoFIB, está disponível em repositório do GitHub[1].

Na avaliação analítica apresentada nesse capítulo, a CoFIB foi comparada com as soluções FCTree (KARRAKCHOU; SAMAAN; KARMOUCH, 2020b), NDN.p4 (SIGNORELLO et al., 2016) e HBM (MIGUEL; SIGNORELLO; RAMOS, 2018) em relação ao consumo de memória. A FCTree foi escolhida para comparação porque é uma estrutura de dados relativamente recente, desenvolvida para a NDN FIB e adequada para ser implementada em comutadores programáveis. A FCTree é a estrutura de dados subjacente na arquitetura ENDN (KARRAKCHOU; SAMAAN; KARMOUCH, 2020a). Por outro lado, NDN.p4 e HBM foram incluídos na comparação porque, até a escrita dessa tese, essas soluções eram as únicas propostas de FIB baseadas em *hardware* totalmente compatíveis com P4, com foco em ASICs programáveis (ver Tabela 3.4). Implementações recentes da arquitetura NDN baseadas em P4, como NDNFab (MADUREIRA et al., 2021), NDNTofino (TAKEMASA; KOIZUMI; HASEGAWA, 2021) e Pegasus (LONG et al., 2023), não implementam a FIB no ASIC mas no *software* do plano de controle.

Já na avaliação experimental, uma instância da arquitetura FANTNet foi simulada em um ambiente virtualizado e alguns experimentos foram realizados para avaliar o comportamento da CoFIB em relação a vazão e ao número de passagens de *pipeline*. A análise do número de passagens no *pipeline* foi realizada devido ao impacto de possíveis recirculações de pacotes durante o LNPM quando a CoFIB é implementada em ASIC programáveis.

6.3 Datasets de Nomes

Diante da dificuldade em encontrar *datasets* específicos de nomes NDN, utiliza-se nos experimentos descritos nesse capítulo *datasets* de nomes de domínio disponíveis publicamente como o DMOZ (DMOZ, 2019) e o DomCop.com (DOMCOP, 2021). O primeiro, chamado aqui de 440K, foi obtido por meio de um *web crawler* desenvolvido pelo *Open Directory Project* (ODP). O segundo *dataset*, chamado aqui de 10M, contém os 10 milhões de *websites* mais populares da Internet. Foi sintetizado um terceiro *dataset* de nomes, denominado 2M, contendo 2 milhões de prefixos criados usando componentes aleatórios extraídos dos *datasets* 440K e 10M. Para aumentar o comprimento médio de cada prefixo nomeado, o número de componentes no *dataset* 2M segue a distribuição de *Weibull* com parâmetros μ=0.1, α=0.3, β=2. Para cada *dataset*, as URLs foram convertidas em nomes NDN invertendo-se a ordem dos componentes, onde os prefixos com mais que 8 componentes ou contendo componentes com mais que 31 caracteres foram removidos. Adicionou-se

[1] <https://github.com/castilhorosa/ndn-cofib>

Tabela 6.1 – Características de cada *dataset* de nomes utilizado nessa tese.

ID	**Descrição**	**Total de Prefixos**	**Prefixos pós filtro**	**Tam. Médio de Comp.**	**No. Médio de Comps.**	**No. Máx de Comps.**
0.5K	*Dataset* aleatório contendo nomes canônicos cujos componentes compatibilizam nos *pipelines* de ingresso e egresso de todas formas possíveis.	510	510	15.90	7.02	8
180K	Prefixos canônicos extraídos do *dataset* 440K usando o algoritmo CPE.	184,684	184,684	7.23	2.08	6
440K	*Dataset* DMOZ obtido via *crawler* do Open Directory Project (ODP).	447,169	292,079	6.41	2.27	6
1M	Prefixos canônicos extraídos do *dataset* 2M usando o algoritmo CPE	1,216,410	1,216,410	12.00	2.65	8
2M	*Dataset* canônico aleatório gerado extraindo-se componentes de 440K e 10M.	2,000,000	2,000,000	12.00	3.13	8
4M	Prefixos canônicos extraídos do *dataset* 10M usando o algoritmo CPE.	4,407,427	4,407,427	7.65	2.19	6
10M	*Dataset* DomCop.com contendo os 10 milhões de *websites* mais populares.	10,000,000	9,981,418	6.71	2.44	8

também em cada *dataset*, com probabilidade Pr, todos os possíveis subprefixos de um dado nome. Por exemplo, se o prefixo é */a/b/c* e Pr=1, então foi adicionado também ao *dataset* os subprefixos */a/b* e */a*. Além disso, três *datasets* contendo apenas nomes canônicos foram derivados dos *datasets* 440K, 2M, e 10M por meio do algoritmo CPE, chamados aqui de 180K, 1M, e 4M, respectivamente. A Tabela 6.1 mostra as principais características de cada *dataset* utilizado nesse estudo.

Como visto na seção 5.4.2, as tabelas na DFIB são colocadas nos *pipelines* de entrada e saída de maneira otimizada para reduzir o número de passagens do *pipeline*. Portanto, para um determinado prefixo contendo k componentes e assumindo o pior cenário, que é quando o LNPM precisa realizar k correspondências, existem 2^k maneiras diferentes dos k componentes corresponderem nas tabelas dependendo de onde elas são colocadas (*pipeline* de entrada ou *pipeline* de saída). Por exemplo, se tivermos um prefixo com dois componentes nomeados, existem quatro maneiras diferentes pelas quais as correspondências podem ocorrer no LNPM: *entrada-entrada*, *entrada-saída*, *saída-entrada*, *saída-saída*. Assim, foi sintetizado outro *dataset*, denominado 0,5K, que contém 2^1 combinações de prefixos com um componente nomeado, 2^2 combinações de prefixos com dois componentes e assim por diante até 2^8 combinações de prefixos com oito componentes nomeados. O número total de prefixos no conjunto de dados de 0,5K é dado pela soma $2^1 + 2^2 + ... + 2^8$, que resulta em 510 prefixos distintos.

6.4 Análise Quantitativa

Essa seção avalia o algoritmo CPE em função da taxa de *offloading* de prefixos da RIB. Além disso, uma análise comparativa quantitativa da CoFIB em relação as soluções mencionadas na seção anterior é também apresentada. Os parâmetros avaliados na análise quantitativa incluem a probabilidade de falso encaminhamento devido a colisões de *hash* e a quantidade de memória consumida por cada solução, além de uma análise sobre o consumo de memória *on-chip* no plano de dados considerando diferentes *datasets* de nomes.

6.4.1 Taxa de *Offloading*

O desempenho do algoritmo CPE é avaliado em relação à taxa de *offloading*. Essa taxa representa o percentual de prefixos nomeados canônicos que podem ser extraídos da RIB. Foram inseridos os *datasets* 440K, 2M e 10M na RIB para medir quantos prefixos canônicos podem ser extraídos. Em seguida, o algoritmo CPE foi executado sob esses *datasets* 10 vezes para a obtenção de um intervalo de confiança de 95%. Para cada *dataset*, foi adicionado, com probabilidade de Pr, todos os subprefixos possíveis para cada prefixo conforme mencionado anteriormente. Como na arquitetura NDN o nome do conteúdo pode ser construído iterativamente, a FIB armazena prefixos de nomes parciais. Assim, o parâmetro P_r é usado para analisar o comportamento do CPE quando a RIB contém prefixos e seus subprefixos.

Como pode ser observado na Figura 6.1, a taxa de *offloading* do CPE tende a diminuir quando o número de prefixos em cada *dataset* cresce. A razão disso é que *datasets* maiores tem chances igualmente maiores de conter componentes em diferentes posições nos nomes quando comparado à *datasets* menores. Quando compara-se diferentes valores de P_r

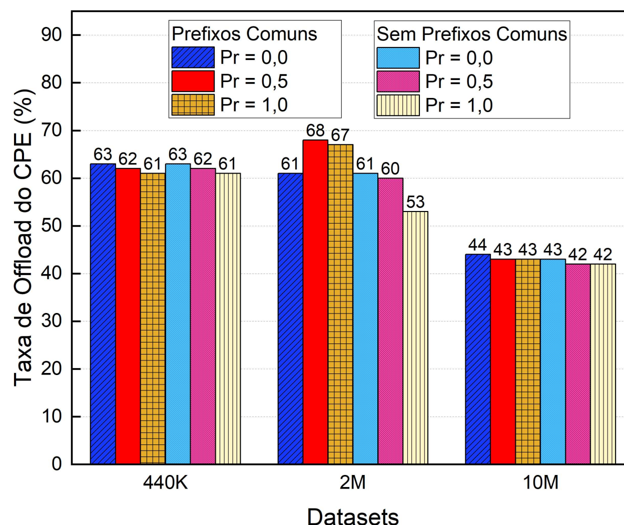

Figura 6.1 – Porcentagem de prefixos canônicos extraídos da RIB usando o CPE.

usando *datasets* reais, como o 440K e o 10M, é possível observar que não há um grande impacto na taxa de *offloading*. Porém, é possível notar maiores variações na taxa ao usar o *dataset* sintético 2M, especialmente com valores de P_r maiores. De fato, o *dataset* 2M contém componentes nomeados uniformemente extraídos dos *datasets* 440K e 10M. Tal abordagem faz o CPE ser mais sensível à variações nos valores P_r tendo em vista que o *dataset* 2M uniformemente dispersa os componentes aleatórios em diferentes posições. Contrariamente, os *datasets* reais contém nomes somente em algumas posições distintas, o que faz com que eles sejam menos sensíveis às variações de P_r.

Como descrito na Seção 4.5, para resolver o problema da ocultação de *cache*, o algoritmo CPE move prefixos que compartilham subprefixos da CoFIB para a CFIB. A Fig. 6.1 mostra que a transferência dos prefixos compartilhados da CoFIB não impacta severamente a taxa de *offloading*, especialmente para os *datasets* de nomes reais (440K, 10M).

O *dataset* 10M contém nomes de domínios referentes aos *websites* mais acessados na Internet. Assim, cada prefixo está associado com uma métrica de *ranking* que varia de 0 a 10 indicando a popularidade de tal prefixo. Ao calcular a média dos *ranking* dos 10 milhões de prefixos e comparar com a média dos prefixos canônicos, observa-se uma variação de apenas 0.58% no nível de popularidade, que corresponde à 3.42 para o *dataset* 10M e 3.40 para o *dataset* canônico obtido por meio do CPE. Isso significa que o CPE não altera o nível de popularidade média dos prefixos canônicos. Em outras palavras, o CPE extrai um subconjunto uniforme de prefixos da RIB.

6.4.2 Probabilidade de Encaminhamento Incorreto

Embora as tabelas de *hash* sejam poderosas estruturas de dados para a FIB, funções de *hash* são suscetíveis a colisões. Quando não tratada adequadamente, as colisões de *hash* na FIB podem levar a falhas na operação de *lookup*, o que destroi a precisão e integridade do esquema de encaminhamento (WANG et al., 2012). Assim, para avaliar o impacto que colisões de *hash* podem causar na FIB em comutadores programáveis, foi calculada analiticamente a probabilidade de um dado prefixo nomeado colidir no mínimo uma vez em qualquer posição após nomes de diferentes *datasets* serem armazenados na FIB. Além dos *datasets* apresentados na Tabela 6.3, para essa análise foi gerada o *dataset* 10K extraindo 10.000 prefixos aleatórios do *dataset* 1M.

A probabilidade de colisão em componentes nomeados na CoFIB pode ser estimada da seguinte maneira. Seja $P_C(T_w)$ a probabilidade de colisão na tabela T_w da DFIB, $P_W(i, w)$ a probabilidade de um componente nomeado na posição i ter w caracteres e $P(i)$ a probabilidade de ocorrer uma colisão na posição i. Assim, temos:

$$P(i) = \sum_{w=1}^{31} P_W(i, w) \times P_C(T_w)$$

Desse modo, $\overline{P(i)}$ denota a probabilidade de não ocorrer colisão na posição i, que é calculada com base na probabilidade de não ocorrer colisão na tabela T_w, dada por meio do complemento $1 - P_C(T_w)$:

$$\overline{P(i)} = \sum_{w=1}^{31} P_W(i, w) \times (1 - P_C(T_w))$$

Disso, é possível estimar $PrC(i)$ como sendo a probabilidade de pelo menos uma colisão ocorrer em um prefixo com i componentes:

$$PrC(i) = 1 - \prod_{j=1}^{i} \overline{P(j)}$$

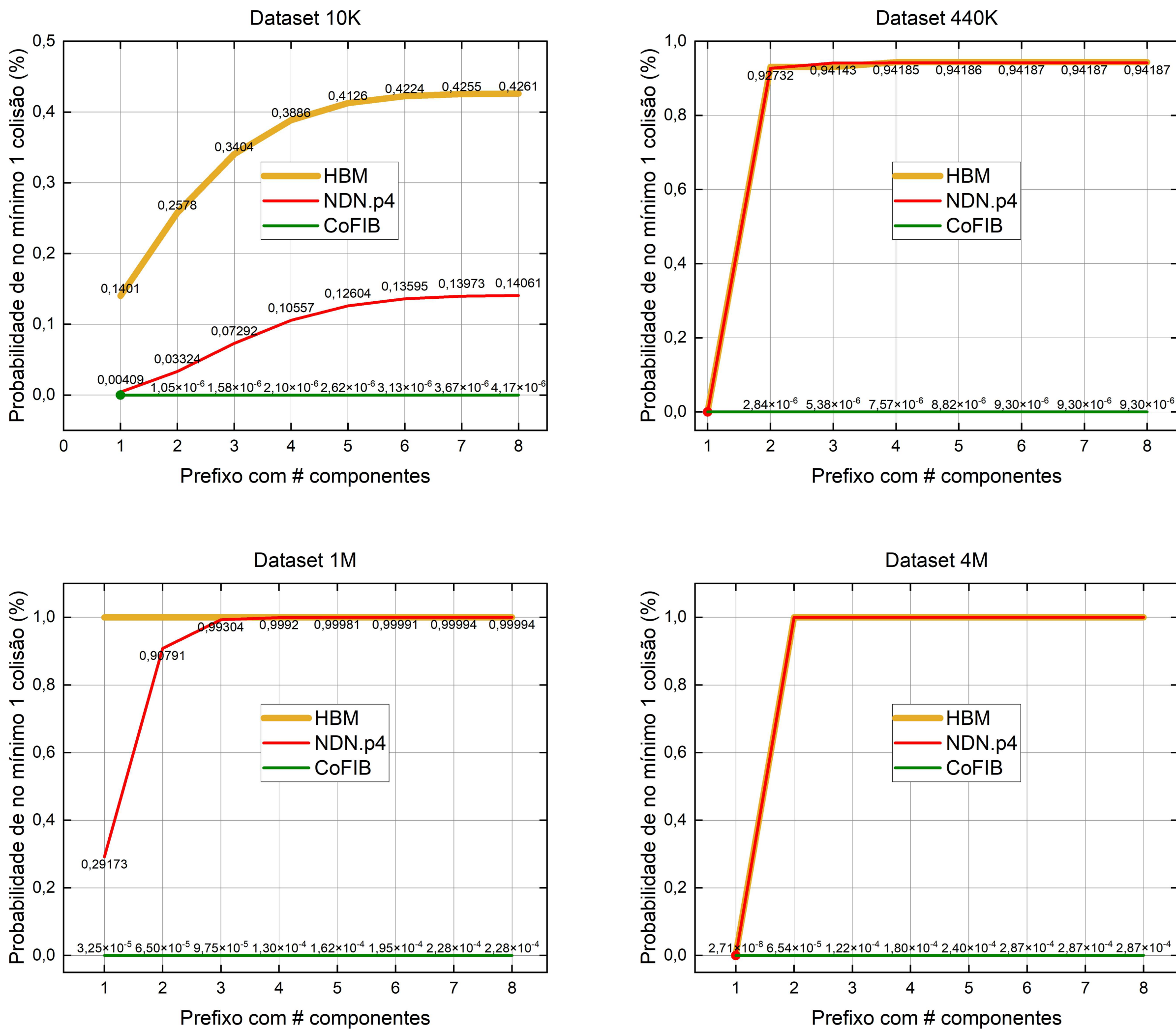

Figura 6.2 – Colisão de *hash* de componentes nomeados considerando quatro *datasets* de nomes distintos.

A Figura 6.2 mostra que a CoFIB mantém uma probabilidade de colisão baixa quando comparada com o HBM e NDN.p4 independentemente do tamanho do *dataset* e do número

de componentes em cada prefixo. Assim, a CoFIB é capaz de escalar para milhões de prefixos nomeados sem se preocupar com técnicas complexas para lidar com colisões, como as operações de *rehashing* e a sondagem linear, por exemplo.

Duas razões explicam o motivo da probabilidade de colisão na CoFIB ser menor. Primeiramente, o uso de diferentes tabelas P4 para armazenar componentes diminui o domínio de colisão tendo em vista que componentes nomeados que colidem são permitidos desde que seus comprimentos sejam diferentes. Em segundo lugar, por utilizar resumos de 32 *bits*, a função crc32 gera menos colisão que a função crc16 utilizada em HBM e NDN.p4. É possível observar que prefixos com menos componentes tem menos chances de colidir. A razão disso é que, após armazenar todos os prefixos canônicos dos quatro *datasets*, a probabilidade de um dado prefixo colidir no mínimo uma vez em qualquer posição cresce a medida em que o número de componentes aumenta. Para exemplificar, um prefixo com 8 componentes é mais suscetível a colidir em pelo menos uma posição que um prefixo com somente um componente tendo em vista que as colisões são eventos independentes.

6.4.3 Consumo de Memória

Nessa seção, o consumo de memória da estrutura de dados CoFIB foi estimado usando os *datasets* canônicos de 180K, 1M e 4M mostrados na Tabela 6.1. O consumo teórico de memória *on-chip* da CoFIB foi comparado com as soluções FCtree (KARRAKCHOU; SAMAAN; KARMOUCH, 2020b), NDN.p4 (SIGNORELLO et al., 2016) e HBM (MIGUEL; SIGNORELLO; RAMOS, 2018). Para estimar o consumo de memória de cada método, os *bytes* necessários para armazenar um determinado prefixo foram somados, assumindo que a memória SRAM/TCAM seja usada. Para as tabelas na DFIB que armazenam os componentes nomeados em *American Standard Code for Information Interchange* (ASCII) (tabelas $T_{i \leq 4}$), assume-se que uma entrada de x *bytes* consome x *bytes* na memória SRAM. Isso é possível graças a um truque arquitetural que possibilita ASICs programáveis atuais combinar conjuntos de entradas pequenas em palavras de memória suficientes para reduzir a fragmentação sem causar impacto na função de combinação-ação (BOSSHART et al., 2013).

As entradas nas tabelas P4 da CoFIB, NDN.p4 e HBM consistem em uma chave de compatibilização exata e parâmetros utilizados em ações. Na NDN.p4, cada entrada requer 104 *bits* de espaço. No entanto, sendo conservador, é possível considerar cada entrada como 96 *bits* visto que é possível combinar o valor de prioridade e a porta de saída em um único *byte* em vez de dois. A chave de compatibilização no método HBM consiste em 128 *bits*, formada por uma sequência de oito *hashes* crc16. Na CoFIB, conforme descrito na seção 5.3.4, cada chave de correspondência consiste em x *bytes* se o componente contém x caracteres e $x \leq 4$, ou quatro *bytes* caso a quantidade de caracteres no componente for maior que quatro (*hashes* crc32). Os parâmetros da ação na NDN.p4 e HBM são

um valor de 8 *bits* que representam a porta de saída associada à chave de correspondência. Finalmente, os dados de ação para cada entrada na CoFIB são um valor de 24 *bits* correspondente à estrutura de dados CAD descrita na seção 5.3.5.

Em relação a FCTree, é importante observar que tal estrutura foi proposta inicialmente para funcionar em *software*. Assim, seu consumo de memória é estimado considerando as características de cada *dataset* bem como a sobrecarga adicionada em cada nó da árvore (ex., ponteiros, pilha, etc). Infelizmente, até o momento da escrita dessa tese, não há implementação da FCTree em P4 disponível publicamente. Desse modo, foi necessário realizar uma comparação qualitativa considerando a FCTree com parâmetro $\beta = 10$, que é a versão que provê a melhor taxa de compressão entre todas as variantes da FCTree.

Nos experimentos para analisar o consumo de memória, foram utilizadas duas versões da CoFIB. A primeira versão considera a CoFIB implementada como um conjunto de tabelas P4 convencionais. Já a segunda versão incorpora as otimizações descritas na seção 5.4.1, onde as tabelas P4 são concebidas como *action profiles*. A Figura 6.3 apresenta o consumo de memória teórico da CoFIB em comparação com cada método, considerando diferentes *datasets* e valores de P_r. Como esperado, pode-se observar que a versão otimizada da CoFIB supera as demais soluções em todos os cenários. Porém, é importante destacar que as *action profiles* são funções externas que podem não estar disponíveis em algums dispositivos programáveis, embora fazem parte da especificação da linguagem P4.

Ao observar diferentes valores de P_r aplicados nos três *datasets*, pode-se constatar que o consumo de memória de todas as estruturas de dados não muda significativamente quando os *datasets* 180K e 4M são usados. No entanto, é possível notar um aumento importante no consumo de memória à medida que os valores de P_r variam no *dataset* 1M. A razão é que o número médio de componentes nomeados em 2M e 1M é maior do que nos demais *datasets*, conforme mostrado na Tabela 6.1. Conseqüentemente, mais subprefixos extras são adicionados em 2M e 1M em contraste com os outros dois *datasets* quando usa-se P_r=$0,5$ e P_r=$1,0$. Como esperado, com mais prefixos adicionados ao *dataset* 1M, o consumo de memória tende a aumentar proporcionalmente.

Curiosamente, a Fig. 6.3 mostra que a CoFIB que armazena o *dataset* 1M com $P_r = 1,0$ requer mais memória do que a CoFIB que armazena o *dataset* 4M. Isso ocorre porque o *dataset* canônico 1M, incluindo os subprefixos extras devido ao parâmetro $P_r = 1,0$, torna-se um conjunto de dados com 3.6 milhões de prefixos no total. Como o número de prefixos é próximo ao número de prefixos no *dataset* 4M e o número médio de prefixos em 1M é maior do que no 4M, é possível observar essa diferença no consumo de memória.

A Figura 6.3 mostra que a versão da CoFIB otimizada com *action profiles* supera as demais soluções na maioria dos cenários. Mesmo a versão da CoFIB que utiliza tabelas P4 tradicionais requer menos memória que as outras soluções. De maneira geral, observa-se que a versão otimizada da CoFIB requer até 13.64× menos memória que NDN.p4, até 16.58× menos memória que FCTree, e até 2.83× menos memória que HBM considerando

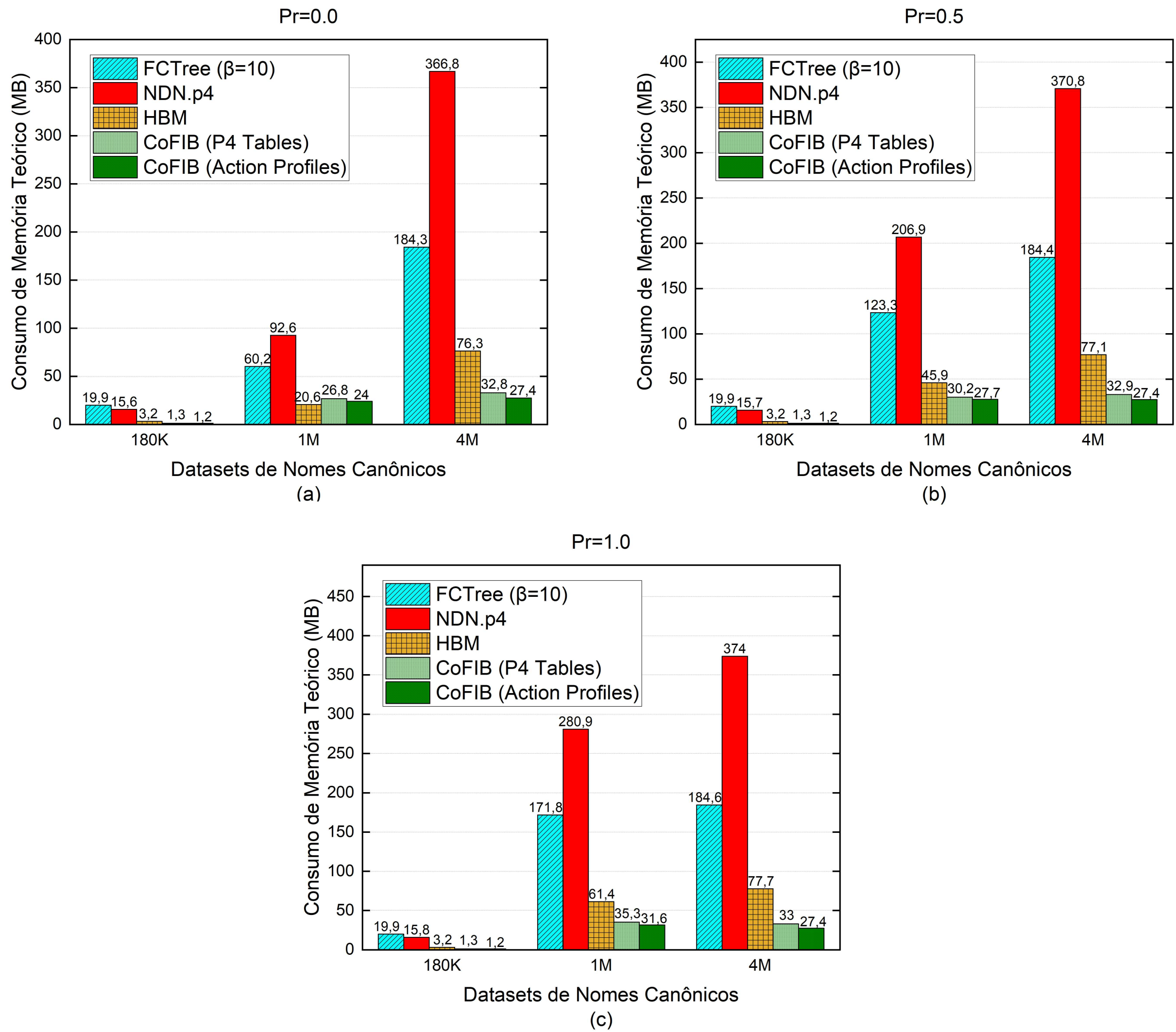

Figura 6.3 – Consumo de memória com diferentes valores de P_r.

datasets de nomes reais (440K e 4M) independentemente do valor de P_r. Ainda, é possível notar que, mesmo usando o *dataset* sintético (1M), a CoFIB provê uma melhor taxa de compressão que as demais soluções para a FIB com P_r=1 e P_r=0.5. O consumo de memória da CoFIB pode ser reduzido ainda mais, em aproximadamente 33%, se o problema do prefixo cruzado for ignorado e a estrutura CAD for implementada utilizando 16 *bits* ao invés de 24 *bits*.

Existem algumas razões que explicam o porquê a CoFIB provê uma taxa de compressão da FIB maior que as soluções FCTree, NDN.p4 e HBM. Primeiramente, a CoFIB armazena cada componente nomeado individualmente ao invés do prefixo inteiro como em NDN.p4 e HBM. Isso evita o armazenamento de entradas FIB com subprefixos comuns múltiplas vezes. Em segundo lugar, os componentes com mais que quatro caracteres são comprimidos usando *hashes* crc32. Assim, os componentes com menos de quatro caracteres são armazenados usando a menor quantidade de *bits* possível. É importante ressaltar

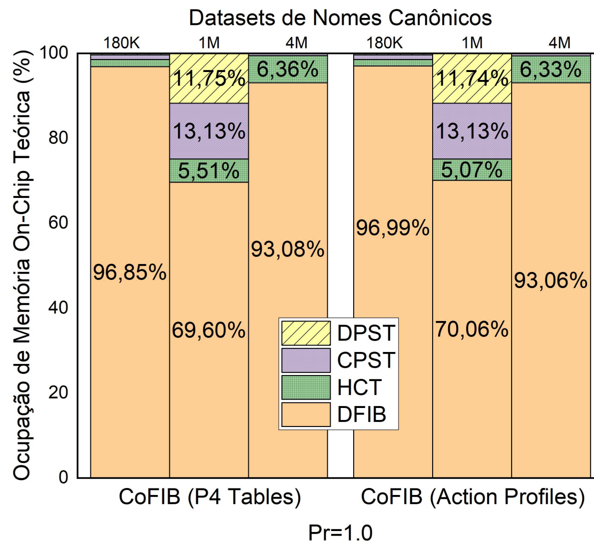

Figura 6.4 – Taxa de ocupação de memória *on-chip* da CoFIB.

que a CoFIB é otimizada com *action profiles*, e isso reduz o consumo de memória em contraste com o uso de tabelas P4 visto que somente uma ação é invocada quando o *lookup* na tabela ocorre. Em relação a solução FCTree, a CoFIB não inclui sobrecarga de ponteiros, que aumenta o consumo de memória. Quando a CoFIB usa o *dataset* 1M com P_r=0, é possível observar que ela consome 1.16x mais memória que a HBM. A razão disso é devido o *dataset* 1M não compartilhar subprefixos comuns tanto quanto os demais *datasets*.

Embora o consumo de memória na CoFIB exceda o do HBM em alguns cenários, esta avaliação não considerou o efeito de colisão de *hash*. Como tanto NDN.p4 quanto HBM usam *hashes* crc16, ao armazenar componentes nomeados individuais em HBM e CoFIB, espera-se uma probabilidade de colisão muito maior em HBM do que na CoFIB para a mesma quantidade de componentes, já que a CoFIB usa *hashes* crc32. Para resolver o problema de colisão, técnicas como sondagem linear, encadeamento e *rehashing* precisam ser usadas às custas do aumento do consumo de memória. É natural argumentar que a CoFIB também está sujeito à colisão de *hash*. No entanto, o uso de tabelas P4 diferentes para armazenar componentes nomeados diminui o domínio de colisão consideravelmente, uma vez que *hashes* de componentes nomeados colididos são permitidos, desde que o comprimento de tais componentes nomeados seja diferente. Portanto, é possível armazenar na CoFIB todos os *datasets* de nomes canônicos usados nesse estudo sem que haja qualquer colisão de *hash*, o que é impossível em soluções como NDN.p4 e HBM.

Em relação a ocupação teórica de memória *on-chip*, a Figura 6.4 mostra como a memória do comutador é alocada entre os principais elementos da CoFIB. Como pode-se observar, o uso de tabelas auxiliares como a DPST, CPST e HCT não impacta significativamente o consumo de memória visto que, no pior cenário, menos de 30% da memória é ocupada por tais tabelas considerando diferentes valores de P_r e *datasets*. Ao obser-

var *datasets* de nomes reais, nota-se que as tabelas auxiliares ocupam menos que 4% do total de memória *on-chip* utilizada. De fato, conforme mostra a Tabela 6.1, a média de componentes nomeados no *dataset* 1M é maior que nos demais *datasets* e isso aumenta o número de prefixos conflitantes e a quantidade de moldes na DPST e CPST. Além disso, as soluções NDN.p4 e HBM utilizam extensivamente a memória TCAM, que embora possibilite a LNPM em um único ciclo, é um tipo de memória com menos disponibilidade, cara e que consome mais energia quando comparado a SRAM. Por outro lado, a Figura 6.4 mostra que as tabelas DFIB na CoFIB, que representam pelo menos 68% de toda a memória *on-chip* consumida, são armazenadas na memória SRAM, que é mais abundante e barata.

Da avaliação analítica da CoFIB em relação ao consumo de memória surgem duas questões importantes. 1) *A CoFIB é capaz de armazenar os 4.3 milhões de prefixos do dataset 4M na memória SRAM e TCAM disponível em um comutador programável real?* 2. *É possível armazenar os 4.3 milhões de prefixos do dataset 4M na TCAM considerando a FIB das soluções NDN.p4 e HBM?* Para responder à essas questões, é necessário comparar a quantidade de memória SRAM e TCAM tipicamente disponível em *switches* programáveis modernos com os resultados obtidos analiticamente.

Atualmente, os ASICs da série Intel Tofino são o estado da arte quando se fala em comutadores programáveis físicos. Desse modo, como forma de responder as questões supracitadas, pode-se utilizar as versões Intel Tofino 2 e Intel Tofino 3 como exemplo. O total de memória TCAM por *pipe* em ambos ASICs é 10.3 Mb, com 4 *pipes* ao todo (LANNER ELECTRONICS, 2012). Assim, a quantidade total de memória TCAM disponível no Intel Tofino 2 é 41.2 Mb, o que representa 5.15 MB. Essa quantidade é insuficiente para armazenar os 4.3 milhões de prefixos usando o projeto de FIB da NDN.4 e HBM mesmo quando o problema de colisão de *hash* é ignorado. A razão disso é que tais soluções dependem exclusivamente da memória TCAM e o consumo de memória teórico indica 374 MB e 77.7 MB para NDN.4 e HBM, respectivamente.

Quanto ao tamanho da SRAM, pode-se, de maneira conservadora, assumir 100 MB com base em valores comumente encontrados em *whitepapers* e em trabalhos como (MIAO et al., 2017). Portanto, com base nos resultados obtidos, a CoFIB é a única solução que pode ser dimensionada para 4.3 milhões de prefixos, considerando um comutador real programável. Na verdade, a CoFIB pode armazenar até 4.5 milhões de prefixos na SRAM ao usar o *dataset* 4M com P_r=1. Isso requer 25.5 MB de SRAM para armazenar prefixos em DFIB (25.5% do total de SRAM disponível no *chip*) e 1.9 MB de TCAM para armazenar as tabelas HCT, CPST e DPST (36.8% do total TCAM disponível no *chip*), deixando alguma memória livre para acomodar mais prefixos e as demais aplicações que rodam concomitantemente no plano de dados.

Além disso, as tabelas na CoFIB são distribuídas igualmente pelos *pipelines* de entrada e saída, o que permite que a CoFIB use toda a memória SRAM disponível no ASIC.

Por outro lado, em NDN.p4 e HBM, todas as tabelas P4 são colocadas no *pipeline* de entrada, desperdiçando os recursos do *pipeline* de saída, que representa em torno de 50% da capacidade total de memória *on-chip*.

6.5 TestBed Experimental da FANTNet

Para viabilizar a condução dos experimentos descritos nesse capítulo, foi criado um *testbed* experimental da arquitetura FANTNet como prova de conceito. O *testbed* foi implementado em uma `VM Ubuntu 20.04` com 4 GB de RAM e 4 vCPUs. Essa VM foi hospedada em uma máquina com o `Microsoft Windows 11` equipada com CPU `Intel Core i5-1135G7` e 16 GB de RAM. Foi utilizada uma imagem *vagrant* pré-construída e configurada conforme disponível em (OPEN NETWORKING FOUNDATION, 2019). O *testbed* criado simula as fases de operação 2, 3 e 4 da FANTNet, conforme mostra a Figura 6.5. A fase 1 foi simulada de maneira estática utilizando uma RIB já preenchida com os prefixos contidos nos *datasets* de nomes, partindo-se do pressuposto de que a amostragem ocorrera *a priori*.

Como pode-se obervar na Figura 6.5, o *testbed* foi implementado utilizando uma topologia customizada no *software* de emulação de redes `mininet` versão `2.3.1b4` (MININET DEVELOPMENT TEAM, 2024). A topologia representa uma instância do plano de dados da FANTNet, que consiste em três *hosts* simulando um nó consumidor e outros dois *hosts* simulando nós produtores. Para simplificar os experimentos, os nós NPCs foram implementados em C e embutidos como uma aplicação dentro de cada *host*. A topologia conta também com dois comutadores de núcleo (CSw_1 e CSw_2) e três comutadores de borda

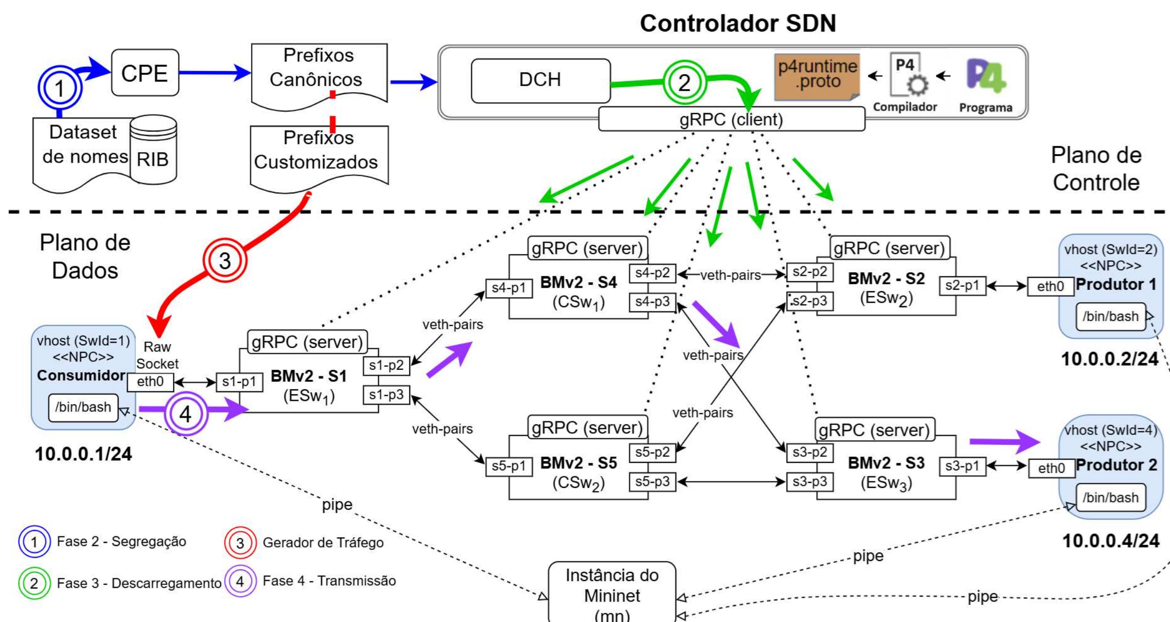

Figura 6.5 – Testbed experimental da FANTNet.

(ESw_1, ESw_2, e ESw_3), todos implementados usando o *switch* de *software Behavioral Model Version 2* (BMv2) (BMV2 DEVELOPMENT TEAM, 2024), versão `1.15.0`. Como o BMv2 não foi concebido para ser um *switch* de *software* de nível de produção, diferentemente do *Open vSwitch* (OVS) (PFAFF et al., 2015), a versão do BMv2 utilizada nesse *testbed* foi clonada e reconstruída com *flags* customizadas de acordo com (FINGERHUT, 2024). Essas *flags* incluem os parâmetros `CXXFLAGS=-g -O3` e `CFLAGS=-g -O3`, que habilitam o modo de depuração e ativam o nível máximo de otimização do compilador para obter o melhor desempenho possível. Além disso, as *flags* de otimização incluem também e os parâmetros `-disable-logging-macros` e `-disable-elogger`, que desabilitam os registros de *logging*, o que aumenta o desempenho do BMv2 em razão da diminuição do número de operações de I/O durante sua execução.

As estruturas de dados e algoritmos do plano de controle da FANTNet, como o CPE, NCH, e DCH, são implementadas e desenvolvidos em Java. O arquivo executável *fantnet.jar*, criado a partir do repositório da FANTNet no GitHub[2], possui várias funções embutidas para processar os arquivos de *datasets* de nomes. Antes de se tornar parâmetro de entrada do algoritmo CPE, conforme mostra a etapa 1 da Figura 6.5, a aplicação *fantnet.jar* é utilizada para produzir os prefixos filtrados a partir dos *datasets* de nomes, conforme mostra a Tabela 6.1. Após isso, os prefixos filtrados são submetidos ao CPE e isso gera a estrutura DCH, que fica armazenada na memória DRAM. Em razão da alta quantidade de prefixos e a disponibilidade limitada de memória virtual da *Java Virtual Machine* (JVM) em sua configuração padrão, foi necessário executar o arquivo *fantnet.jar* utilizando o parâmetro `-Xmx2048m` para aumentar a quantidadede memória alocada para a JVM em tempo de execução.

A fase de descarregamento no *testbed* desenvolvido, ilustrada na Figura 6.5 como etapa 2, é realizada carregando os prefixos canônicos para a DCH e depois utilizando a API `gson 2.10.1` para gerar o arquivo contendo as entradas P4RT a partir das tabelas de *hash* contidas na DCH. Paralelamente, as tabelas FST dos comutadores de núcleo bem como as tabelas DPST, CPST e HCT são também populadas via P4RT durante essa fase. As entradas P4RT são enviadas para as tabelas do BMv2 através do `gRPC` cliente, que usa o `protobuf` como formato padrão para a serialização dos dados que são enviados para o `gRPC` servidor, que ouve na porta TCP 9559.

O tráfego NDN enviado pelo nó consumidor é gerado utilizando como base o arquivo de prefixos canônicos produzido pelo CPE, um *template* de prefixos customizados para simular diferentes tipos de tráfego ou uma combinação de ambos. A escolha sobre qual fonte de dados utilizar no gerador de tráfego depende do experimento a ser realizado, conforme será detalhado na Seção 6.6. Foi realizado um teste empírico entre os geradores de pacotes `Scapy`, disponível no Python, a API `Raw Socket` em C e a ferramenta `iperf`, tendo essa última apresentado melhor desempenho em termos de vazão. No entanto, em

[2] <https://github.com/castilhorosa/ndn-cofib>

razão da dificuldade de gerar pacotes customizados no `iperf`, como os pacotes NDN codificados em P4NF, optou-se por utilizar a API `Raw Socket` como gerador de tráfego. Embora fora do escopo dessa tese, o *testbed* implementado possibilita também o envio de tráfego IP convencional, tendo em vista que cada *host* no `mininet` implementa nativamente a pilha TCP/IP e a coexistência entre tráfego IP e não IP é simples de ser realizada em P4.

6.6 Experimentos de Simulação

Nesta seção, apresenta-se os resultados de vários experimentos conduzidos para avaliar o desempenho da CoFIB em relação à vazão e ao número de passagens no *pipeline* de processamento. Todos os experimentos são realizados utilizando o *testbed* descrito na Seção 6.5. Nesses experimentos, a CoFIB foi comparada com a HBM apenas em relação à taxa de transferência, uma vez que a NDN.p4 não escala para milhões de prefixos. Já em relação ao número médio de passagens no *pipeline*, a CoFIB foi comparada com NDN.p4 e HBM. Como nenhuma implementação P4 da FCTree está disponível publicamente, tal solução não foi incluída nos experimentos aqui descritos. De fato, o objetivo desses experimentos é comparar a CoFIB com soluções que fornecem processamento de pacotes à taxa de linha quando implantados em comutadores programáveis reais, que é o caso da NDN.p4 e HBM, mas não da FCTree.

Em todos os cenários avaliados, foram simulados ambientes apenas com prefixos canônicos armazenados no plano de dados. Portanto, o CPST está vazio em todos os experimentos. O *dataset* 0.5K foi armazenado na CoFIB em ESw_1 nos três primeiros experimentos. No experimento 4, o *dataset* canônico 4M foi armazenado na CoFIB em ESw_1. Cada prefixo em ambos *datasets* contém um valor de *SwId* aleatório, que pode assumir os valores 2 ou 4, representando ESw_2 ou ESw_3, respectivamente.

Todos os experimentos consistem no envio de Ipkts do nó consumidor para os nós produtores utilizando o gerador de tráfego descrito na Seção 6.5. Nos experimentos 1 e 2, utiliza-se a ferramenta *tcpdump* para capturar pacotes em ESw_2 e ESw_3 através de suas interfaces virtuais de entrada. Os arquivos *pcap* resultantes foram mesclados usando a ferramenta *Wireshark* para auxiliar na obtenção da vazão média ao longo do tempo. Por outro lado, nos experimentos 3 e 4, o CDF do número de passagens do *pipeline* foi calculado considerando a carga de tráfego de Ipkts injetada em ESw_1. Como todos os experimentos apresentados nesta seção visam medir a influência da CoFIB na vazão final, as estruturas de dados CS e PIT foram ignoradas em todos os comutadores. Desse modo, a CoFIB foi implementada apenas no comutador de borda ESw_1.

Em todos os experimentos realizados, os pacotes foram enviados do consumidor para o NPC_1 à taxa constante. Para determinar a taxa máxima que pode ser obtida considerando as limitações do BMv2 em nosso cenário, foi estimado o menor intervalo de tempo

possível entre pacotes subsequentes para o qual o BMv2 não sofre perda de pacotes ao usar o HBM como *baseline*. Esse cálculo foi realizado empiricamente através de sucessivas execuções do HBM com pequenos decrementos no intervalo entre pacotes até que ocorressem descartes na interface virtual de entrada. Além disso, para reduzir os efeitos do tempo de processamento de pacotes não determinístico do BMv2, todos os experimentos foram repetidos dez vezes para obter a vazão média.

O tráfego injetado em ESw_1 nos três primeiros experimentos consiste em 160K Ipkts criados usando *raw socket* em *C* com base em 160K prefixos nomeados canônicos extraídos do *dataset* 0.5K. No experimento 4, o tráfego consiste 4.3M de Ipkts criados em *C* com base em prefixos canônicos do *dataset* 4M. Todos os pacotes foram armazenados em um arquivo binário no nó consumidor. Cada Ipkt contém um prefixo nomeado acrescido de uma média de 150 *bytes* em campos como *selectors*, *nonce*, e *guiders*. Nos três primeiros experimentos, os 160K Ipkts correspondem a oito grupos, cada um contendo exatamente 20K prefixos com um determinado número de componentes. No experimento 4, os 4.3M Ipkts são embaralhados antes de serem armazenados no arquivo de tráfego. Então, em cada experimento, os Ipkts são lidos um por um do arquivo de tráfego e são enviados sequencialmente, espaçados por um intervalo de tempo fixo, para ESw_1 através de um *raw socket* em *C*.

6.6.1 Experimento 1: Pacotes de Interesse Ordenados

Uma característica da CoFIB que pode impactar a vazão é o número de recirculações de pacotes no *pipeline* durante o algoritmo LNPM. Disso, uma importante questão de pesquisa que surge é: *Qual é o comportamento da vazão no pior caso, quando a LNPM compatibiliza com todos os componentes do prefixo contido em cada Ipkt?*. Para responder a esta questão, este experimento foi montado para avaliar a vazão do HBM e do CoFIB no pior cenário possível. A CoFIB foi implementada usando a estratégia de posicionamento de tabela linear padrão.

O racional deste experimento é aumentar gradualmente o número de componentes nomeados nos prefixos que compatibilizam na CoFIB para observar o impacto na vazão ao processar prefixos com o número máximo de componentes nomeados. Para atingir esse objetivo, o arquivo de tráfego foi criado com os oito grupos classificados em ordem crescente no número de componentes. Assim, os primeiros 20K Ipkts são enviados para ESw_1 contendo prefixos com um componente nomeado, seguidos por 20K Ipkts contendo prefixos com dois componentes nomeados, e assim por diante até os últimos 20K Ipkts contendo prefixos com oito componentes nomeados.

Seguindo a metodologia descrita anteriormente, a taxa máxima que se pôde obter sem ocorrer perdas de pacotes foi em torno de 2.5 Mbps, correspondendo a uma média de 1500 pkts/s. A Fig. 6.6 descreve a vazão obtida na CoFIB e HBM. Pode-se notar um aumento no número de *bits* por segundo ao longo do tempo em ambas as soluções,

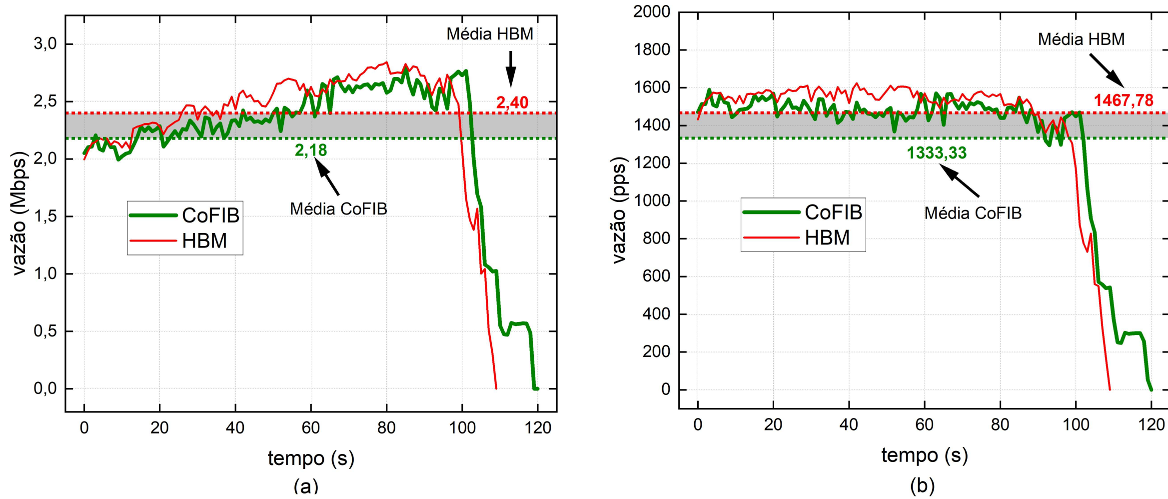

Figura 6.6 – Vazão em *bits* por segundo (a) e em pacotes por segundo (b) ao enviar Ipkts ordenados pelo número de componentes.

embora a taxa de transferência em pacotes por segundo pareça constante. Pode-se fazer duas observações a partir deste resultado. Primeiro, o aumento em *bits* por segundo é devido os tamanhos dos pacotes se tornarem maiores à medida que temos prefixos de nomes com mais componentes, já que o tamanho médio de cada Ipkt é o mesmo. Segundo, este resultado indica que a vazão ao longo do tempo não muda significativamente quando o CoFIB processa Ipkts cujos prefixos contêm o número máximo de componentes, como visto no momento ∼80 segundos em diante. O comportamento da vazão ao longo do tempo segue uma tendência semelhante em CoFIB e HBM.

Ao observar esse resultado, é natural argumentar porquê a vazão ao usar a solução HBM é similar ao CoFIB, considerando que o HBM processa cada pacote em uma única passagem de *pipeline*. O motivo é que, embora o HBM possa fornecer processamento de taxa de linha quando implantado em um comutador programável real, a latência do *parser* é influenciada pelo número de componentes nomeados. Assim, realizar o *parsing* de nomes com mais componentes tende a aumentar a latência por pacote, o que explica a ligeira diminuição da vazão observada entre 80 e 100 segundos.

Na Fig. 6.6, quando a vazão média em ambas soluções são comparadas, nota-se que a vazão ao usar a CoFIB é cerca de 9.16% menor do que quando se usa a HBM. Isso ocorre porque a CoFIB recircula Ipkts algumas vezes no *pipeline* para executar o LNPM, enquanto que em HBM e em NDN.p4 o LNPM é executado em uma única passagem no *pipeline*. No entanto, como o algoritmo CPE prioriza prefixos mais curtos sobre os mais longos, conforme mostrado na Tabela 6.1, a CoFIB tende a armazenar prefixos contendo poucos componentes nomeados. Em contraste, os prefixos mais longos têm mais probabilidade de serem armazenados na CFIB no plano de controle. Portanto, o número médio de recirculações por pacote tende a ser baixo, o que não afeta significativamente o desempenho geral da CoFIB.

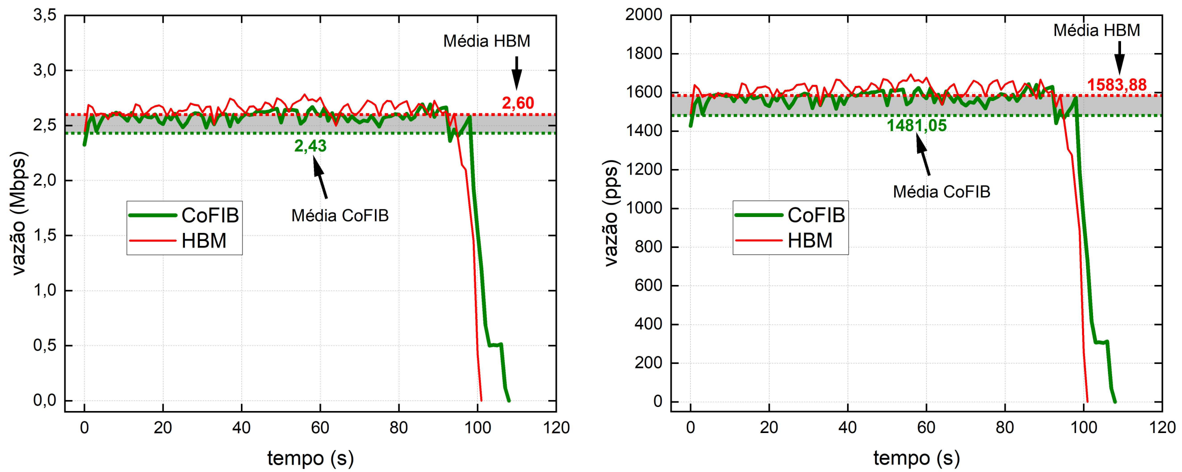

Figura 6.7 – Vazão em *bits* por segundo (a) e em pacotes por segundo (b) ao enviar Ipkts em ordem arbritária.

6.6.2 Experimento 2: Pacotes de Interesse Não Ordenados

Em um cenário mais realístico, espera-se que os Ipkts que chegam aos comutadores de borda contenham prefixos com um número arbitrário de componentes. Assim, este experimento visa avaliar o comportamento da vazão ao usar a CoFIB no caso médio, representando um cenário onde não se pressupõe ordem específica nos Ipkts. Consequentemente, este experimento visa responder à pergunta: *Como a vazão na CoFIB se comporta em contraste com o HBM em um cenário onde os Ipkts contém prefixos com um número arbitrário de componentes*?

Nesse experimento, utiliza-se o mesmo arquivo de tráfego usado no experimento 1. No entanto, os Ipkts agora são enviados sem que haja qualquer ordem em relação ao número de componentes em cada prefixo. A taxa máxima possível obtida neste experimento sem que haja perdas de pacotes foi em torno de 2.6 Mbps, correspondendo a uma média de 1600 ptks/s.

A Fig. 6.7 resume os resultados obtidos nesse experimento. Os gráficos mostram que a taxa de transferência em *bits* por segundo e em pacotes por segundo não muda ao longo do tempo tanto na CoFIB quanto na solução HBM. A vazão em *bits* por segundo ao longo do tempo se comporta conforme o esperado, uma vez que o tamanho médio do pacote não aumenta durante a execução do experimento. Ao comparar a vazão média da CoFIB em relação ao HBM nota-se que o desempenho do CoFIB é cerca de 6.5% menor que o observado no HBM. Esta degradação da vazão também se deve às recirculações de pacotes, como no experimento anterior. No entanto, ao possibilitar o envio de Ipkts sem que haja quaisquer mecanismos de ordenação, observa-se uma melhora da vazão em relação à obtida no experimento 1, que representa 9.16% de degradação.

Outra observação importante é que, como o tempo entre pacotes é constante e definido em um valor suficiente para evitar perdas de pacotes, a fila na interface de entrada

em ESw_1 pode absorver tanto o tráfego novo quanto o tráfego extra devido aos pacotes recirculados. Isto é possível porque os Ipkts contendo prefixos com apenas alguns componentes nomeados são distribuídos igualmente ao longo do tempo. Além disso, como não existem grandes sequências de pacotes que recirculam várias vezes, a ocupação da fila é menor do que no experimento anterior, o que contribui para uma vazão maior.

É importante destacar que o BMv2 recircula pacotes pela mesma porta de entrada em que eles chegam inicialmente. Um algoritmo de escalonamento baseado em *Priority Queuing* (PQ) é definido para cada porta para processar primeiro os pacotes recirculados para depois encaminhar os pacotes originais. Esse detalhe de implementação reduz a vazão, pois a possibilidade de paralelismo é limitada em *software*. Por outro lado, em ASICs programáveis reais, como o Intel Tofino, é possível dedicar portas para recirculação de forma a aumentar a vazão.

Também é preciso considerar que a CoFIB é implementada apenas nos comutadores de borda. Dessa forma, os Ipkts enviados para o núcleo são encaminhados usando apenas correspondências exatas nas tabelas FST. Por outro lado, tanto a HBM quanto a NDN.p4 são implementadas em todos os comutadores do domínio. Conseqüentemente, embora forneçam processamento de taxa de linha quando implantados em ASICs programáveis reais, o *parsing* dos cabeçalhos TLV em toda a rede adiciona alguma latência por pacote, comprometendo a vazão final em contraste com a CoFIB.

No geral, os resultados obtidos nos experimentos 1 e 2 mostram que a abordagem interativa para realizar o LNPM através de recirculações de pacotes em *software* não degrada consideravelmente a vazão fim-a-fim. Mesmo no pior cenário, onde o ESw_1 recebe Ipkts contendo prefixos que correspondem no DFIB usando 8 componentes nomeados, a possibilidade de alguns pacotes recircularem 7 vezes, passando 8 vezes no *pipeline*, não afetou significativamente a vazão média.

6.6.3 Experimento 3: Passagens no *pipeline* (*Dataset* 0.5K)

Apesar de todas as otimizações feitas no ambiente de simulação, sabe-se que o BMv2 não foi concebido para ser um *switch* de *software* adequado para uma rede de produção. Além disso, vários fatores podem impactar a vazão em um *switch* de *software*, como o número de entradas nas tabelas, a quantidade de núcleos de CPU, a plataforma de experimentação (Máquina Virtual Linux ou Máquina Linux *bare-metal*) e assim por diante. Portanto, os experimentos 3 e 4 visam avaliar a CoFIB sob outra perspectiva, ignorando os efeitos do processamento de *software* não determinístico.

Desta forma, o objetivo principal dos experimentos 3 e 4 é avaliar a CoFIB em termos de quantas passagens de *pipeline* cada Ipkt experimenta durante o LNPM. Esta abordagem nos permite inferir o comportamento da vazão com mais precisão ao implementar a CoFIB em um comutador programável real, uma vez que o principal fator que pode impactar a vazão em tais dispositivos é o número de recirculações de pacotes.

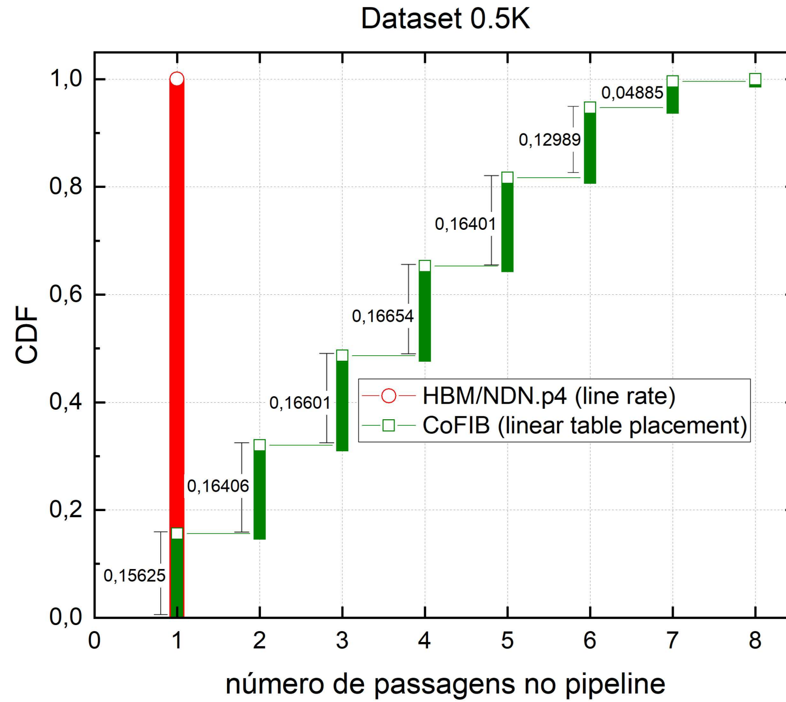

Figura 6.8 – CDF do número de passagens no *pipeline* usando o *dataset* 0.5K.

Nesse experimento, são enviados para ESw_1 um total de 160K Ipkts em ordem arbitrária, exatamente como no experimento 2. A única diferença é que aqui calcula-se o número de passagens no *pipeline* para cada pacote durante o LNPM. A figura 6.8 apresenta o CDF do número de passagens no *pipeline* quando a CoFIB armazena o *dataset* 0.5K e processa os 160K Ipkts do arquivo de tráfego. Não é novidade as soluções NDN.p4 e HBM processarem 100% dos pacotes em uma passagem no *pipeline*. Por outro lado, nota-se que o número de passagens no *pipeline* da CoFIB varia de 1 a 8 vezes. Uma vez que os valores discretos fornecem o número de passagens do *pipeline*, o gráfico CDF em forma de escada mostrado na Fig. 6.8 é uma aproximação de uma distribuição uniforme discreta, como esperado. Isso ocorre porque o arquivo de tráfego possui a mesma quantidade de pacotes (20K) para cada número possível de componentes.

Curiosamente, a altura de cada degrau do gráfico varia à medida que o número de passagens do *pipeline* aumenta, o que não deveria acontecer em um gráfico do tipo escada baseado em uma distribuição discreta e uniforme. No entanto, tal comportamento é esperado, pois os pacotes que passaram x vezes no *pipeline* não contêm necessariamente prefixos com exatamente x componentes. Mais precisamente, os pacotes que cruzaram o *pipeline* x vezes apenas podem conter prefixos com y componentes, onde $x \leq y \leq 2x$ se $x \leq 4$ ou $x \leq y \leq 8$ caso contrário. Isso ocorre porque a CoFIB suporta prefixos com até 8 componentes nomeados e permite duas operações de combinação-ação de componentes por passagem do *pipeline*, dependendo de onde as tabelas são colocadas.

A possibilidade de realizar duas operações de combinação-ação em uma única passagem de *pipeline* é benéfica para prefixos mais longos. Por exemplo, embora o arquivo de

tráfego tenha 20K Ipkts com prefixos contendo 8 componentes, apenas uma fração desses pacotes experimentou exatamente 8 passagens no *pipeline*. Isso representa o pior cenário, onde apenas 1 combinação-ação de componente ocorre por passagem do *pipeline*. O gráfico CDF na Fig. 6.8 indica que 99.561% dos pacotes experimentaram no máximo 7 passagens de *pipeline*, o que resulta em apenas 0.439% pacotes (altura mais baixa entre os 8 degraus no gráfico) que experimentaram exatamente 8 passagens de *pipeline* (100% − 99.561%). Isso equivale a apenas 702 dos 20 mil pacotes enviados.

Por outro lado, quando observa-se o degrau mais alto do gráfico da CDF, nota-se que este valor é 0.16654 e corresponde ao número de pacotes que cruzaram o *pipeline* 4 vezes. Na verdade, os pacotes que atravessam o *pipeline* 4 vezes incluem pacotes contendo prefixos com 4 componentes e alguns contendo prefixos com 5, 6, 7 e 8 componentes. Isso significa que o número de recirculações é reduzido para alguns grupos de pacotes, contribuindo positivamente para uma melhor vazão.

Vale ressaltar que a ordem dos Ipkts enviados para ESw_1 não influencia o gráfico da CDF, pois o número de passagens do *pipeline* é o mesmo, estejam os pacotes ordenados ou não. Portanto, não foi necessário conduzir experimentos usando Ipkts ordenados, pois os resultados serão equivalentes aos obtidos aqui.

6.6.4 Experimento 4: Passagens no *Pipeline* (*Dataset* 4M)

O Experimento 4 também visa avaliar a CoFIB em relação ao número de passagens no *pipeline*. No entanto, esse experimento inclui a estratégia de posicionamento otimizado das tabelas nos *pipelines* de ingresso e egresso bem como o *dataset* 4M ao invés do 0.5K. Assim, o arquivo de tráfego injetado no ESw_1 consiste em 4.3M de Ipkts em ordem aleatória contendo os prefixos canônicos reais do *dataset* 4M com P_r=1.0, conforme mostrado na Tabela 6.1.

Este experimento considera o pior cenário possível. O pior cenário aqui significa que todos prefixos contidos nos Ipkts que forem processados na CoFIB executarão as operações de combinação-ação para todos os seus componentes. Em outras palavras, as tabelas CPST e DPST permanecem vazias e, portanto, não reduzem o número de operações combinação-ação. Evidentemente, no cenário médio, espera-se que alguns pacotes sejam processados à taxa de linha quando os moldes de seus prefixos não existirem na CPST e/ou na DPST.

De maneira geral, esse experimento visa responder à duas questões. Primeiro, *qual é a porcentagem de redução no número de passagens no pipeline quando a estratégia de otimização de posicionamento de tabelas é empregada*?. Segundo, *em que grau a vazão pode ser reduzida devido às recirculações de pacotes quando a CoFIB armazena 4.3 milhões de prefixos em uma comutador programável real*?.

A Fig. 6.9 mostra o CDF do número de passagens no *pipeline* quando a CoFIB armazena o *dataset* 4M e processa os 4.3 milhões de Ipkts do arquivo de tráfego. Em

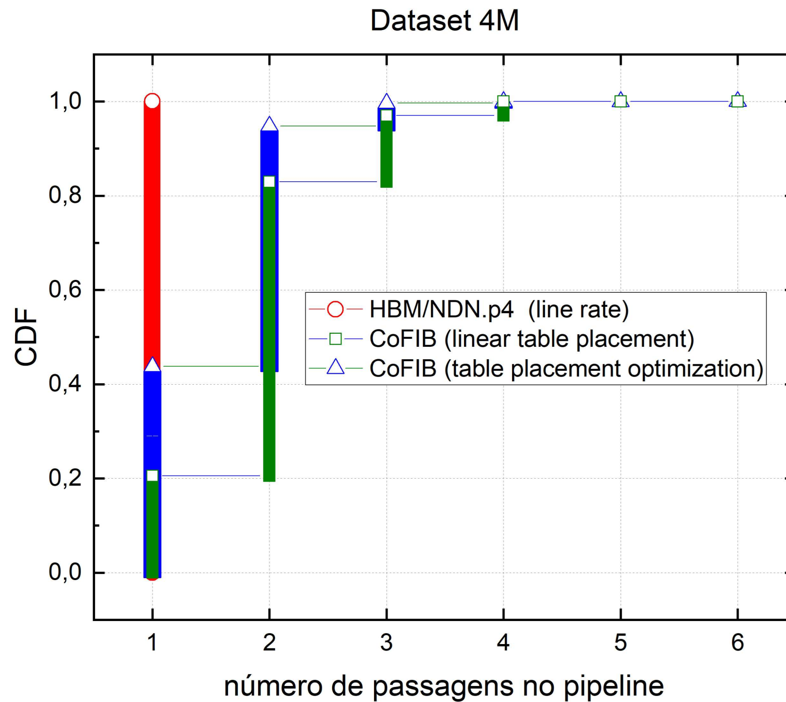

Figura 6.9 – CDF do número de passagens no *pipeline* usando o *dataset* 4M.

contraste com os resultados obtidos no Experimento 3, pode-se observar que o gráfico da CDF não se aproxima de uma distribuição uniforme discreta. O motivo é que o número médio de componentes no *dataset* 4M, conforme mostrado na tabela 6.1, é menor que no *dataset* 0.5K. Além disso, como mencionado anteriormente, o algoritmo CPE prioriza prefixos mais curtos em detrimento aos mais longos. Isso explica o por quê do número médio de passagens no *pipeline* ser menor do que o observado no experimento anterior.

Surpreendentemente, pode-se notar que a porcentagem de Ipkts processados à taxa de linha (uma única passagem no *pipeline*) aumenta de 20.57% para 43.74% ao usar a estratégia de otimização de posicionamento de tabela. Para Ipkts que cruzaram o *pipeline* 2 vezes (experimentaram 1 recirculação), a porcentagem aumentou de 82.95% para 94.77%. É possível observar a mesma tendência para pacotes que experimentaram um número maior de passagens no *pipeline*. Por exemplo, de 96.98% para 99.65% para pacotes que experimentaram 3 passagens no *pipeline* e de 99.98% para 100% para pacotes que experimentaram 4 passagens no *pipeline*. Esse resultado é importante porque mostra que a CoFIB é capaz de prover vazão próxima à taxa de linha ao reduzir o número de recirculações por pacote através da técnica de otimização de posicionamento das tabelas nos *pipelines* de ingresso e egresso.

Infelizmente, não é possível responder completamente a segunda pergunta, pois não foi possível implementar a CoFIB em um comutador programável real. No entanto, é possível traçar algumas observações sobre como a taxa de transferência seria afetada com base nos resultados obtidos neste experimento e em dados técnicos disponíveis na literatura.

Primeiramente, um estudo recente e abrangente de perfil de latência do ASIC progra-

mável Intel Tofino, apresentado por Franco et al. (2024), sugere uma latência média por pacote de 385.8ns ao usar pacotes com comprimentos semelhantes aos utilizados nos experimentos descritos nesse capítulo. Segundo os autores do estudo, esse valor é obtido ao configurar as portas do comutador para velocidade de 100G e considerar vários cenários de caso de uso semelhantes ao que o CoFIB experimentaria em uma eventual implementação física. Assim, com base nos resultados obtidos nesta seção, pode-se dizer que a latência, para a maioria dos Ipkts, está limitada a 771.7ns considerando nosso cenário, o que nos dá alguma confiança para hipotetizar que a latência na CoFIB ficaria dentro da escala de tempo de nanossegundos ao ser implementada em um comutador programável real.

Em segundo lugar, sabe-se que o tempo de busca na TCAM é 2.7ns para uma consulta, enquanto que na SRAM cada tempo de acesso é 0.47ns (SONG et al., 2015). Embora a CoFIB exija múltiplos acessos na SRAM para realizar o LNPM, no pior cenário, 94.77% de todos os *Ipckts* terão a latência do LNPM limitada a 1.88ns (0.47×2 para o lookup na tabela e 0.47×2 para extrair as informações da CAD na SRAM). Portanto, ao ignorar os efeitos de múltiplos *parsings*, de operações relacionadas às recirculações, e ao considerar apenas a latência LNPM, a CoFIB supera o NDN.p4 e o HBM em relação à latência por pacote, uma vez que são soluções baseadas em TCAM com a FIB implementada em todos os comutadores do domínio.

Além disso, para minimizar os efeitos de múltiplas recirculações de pacotes no *pipeline*, uma vez que a CoFIB foi projetada para usar 8 portas de saída, é possível dedicar 50% das portas do comutador para operações de *loopback*, ganhando-se 8× mais vazão, como em Chen (2020). Além disso, como cada componente aparece em apenas uma posição no prefixo e componentes nomeados duplicados no mesmo prefixo não são permitidos, ao recircular e receber Ipkts ao mesmo tempo em ASICs programáveis reais, é possível realizar operações de combinação-ação de forma paralela nas tabelas da DFIB, já que o comutador possui portas dedicadas para recirculação.

Como consequência dos resultados apresentados nesta seção e das observações acima mencionadas, é razoável supor que, ao rodar em um ASIC programável real, a latência de processamento por pacote na CoFIB permanecerá dentro da escala de tempo de nanossegundos, dado o pequeno número de passagens no *pipeline* experimentadas para a maioria dos pacotes. Assim, estas observações sugerem que a recirculação de alguns dos Ipkts não afetará a escalabilidade da CoFIB para um *dataset* de nomes em grande escala.

6.7 Discussão

Os resultados apresentados nesse capítulo ajudam a responder algumas das questões de pesquisas levantadas no Capítulo 1. Esses resultados envolvem a avaliação da CoFIB e a viabilidade da arquitetura FANTNet, que foi implementada como uma prova de conceito no `mininet`. Essa seção procura traçar uma correlação entre a FANTNet, a CoFIB e os

principais trabalhos na literatura já discutidos no Capítulo 3, focando nos *gaps* existentes envolvendo tanto o plano de controle quanto o plano de dados.

6.7.1 Análise Qualitativa

A Tabela 6.2 traz novamente as principais arquiteturas baseadas em SDN para facilitar o encaminhamento NDN, comparando-as qualitativamente com a FANTNet. Como se pode observar, a FANTNet é a única arquitetura que escala a FIB para armazenar milhões de prefixos na memória *on-chip* de comutadores programáveis. Isso é possível graças as técnicas de compressão de memória como a utilização de prefixos canônicos, utilização de funções de hash crc32 e recursos nativos da linguagem P4 como os *actions profiles*. Além disso, é importante destacar que, embora as arquiteturas NDNFab, ENDN, Pegasus, e NDN-Tofino utilizem comutadores programáveis para acelerar o tráfego NDN, apenas a ENDN possibilita que a FIB seja armazenada na memória interna do comutador, por meio da estrutura de dados FCTree. Mesmo assim, a FCTree é disponibilizada em Karrakchou, Samaan e Karmouch (2020b) como uma solução em *software*, o que torna dificil sua implementação em P4 considerando que os autores não fornecem o *codebase* de como a FIB é implementada.

A arquitetura NDNFab, embora empregue a FIB no plano de controle, utiliza o roteamento por segmentos (*Segment Routing*) para comutar o tráfego entre os roteadores de borda programáveis. A FANTNet, por outro lado, emprega uma estratégia de encaminhamento baseada em *Multi Protocol Label Switching* (MPLS) que viabiliza nativamente o suporte a encaminhamento IP e NDN de forma simultânea. O NDNFlow, embora utilize comutadores híbridos assim como na FANTNet, utiliza a tecnologia legada do OpenFlow, que dificulta o processamento eficiente de Ipkts. A FANTNet é uma arquitetura que tem como premissa preencher as lacunas existentes na literatura, conforme mostra a Tabela 6.2.

No plano de dados, para garantir que pacotes NDN sejam processados eficientemente em *hardware*, um dos critérios a serem observados é o suporte ao processamento paralelo. Em comutadores físicos, as soluções propostas para a NDN FIB apresentadas no Capítulo 3 são capazes de realizar o processamento de pacotes em paralelo, com excessão das soluções NDN-Tofino (TAKEMASA; KOIZUMI; HASEGAWA, 2021) e Pegasus (LONG et al., 2023), na qual a FIB é implementada em *software* no plano de controle. No entanto, algumas propostas para a NDN FIB foram projetadas para *hardware* mas não prevêem a possibilidade de paralelismo, o que acaba impactando o desempenho do encaminhamento. A Tabela 6.3 compara qualitativamente a CoFIB com as principais propostas da literatura considerando diferentes características.

Quando se trata de armazenar prefixos nomeados na NDN FIB, devido o tamanho dessa tabela ser algumas ordens de magnitude maior que as tabelas de roteamento convencionais, um problema ainda não tratado na literatura é a impossibilidade de armazenar

Tabela 6.2 – Comparação entre a FANTNet e as principais arquiteturas baseadas em SDN para o encaminhamento de tráfego NDN.

Arquitetura	**Características Gerais**	**Características de Desempenho**				
		FIB na SRAM	Utiliza P4	Co-Design HW/SW	Latência por Pacote (*ns*)	Escalável (HW)
NDNFab	• Comutadores programáveis na borda • Comutadores híbridos no núcleo • FIB na DRAM no plano de controle • Encaminhamento via *Segment Routing* (SR)	○	●	●	○	○
ENDN	• Comutadores programáveis em todo domínio • FIB emprega a estrutura FCTree • Utiliza *externs* para processar *strings*	●	●	○	○	○
NDNFlow	• Comutadores OpenFlow em todo domínio • Canal de comunicação separado para ICN e IP • Tunelamento de tráfego entre comutadores não OpenFlow	●	○	○	●	○
Pegasus	• Comutadores programáveis em todo domínio • Descarrega o *parser* no plano de dados • Implementa a FIB no plano de controle	○	●	●	○	○
NDN-Tofino	• Comutadores programáveis em todo domínio • Somente Dpkts são processados no plano de dados • A FIB é armazenada na DRAM	○	●	●	○	○
FANTNet	• Comutadores programáveis na borda • Comutadores híbridos no núcleo • FIB na SRAM/TCAM e DRAM • Encaminhamento no núcleo baseado em MPLS • Otimização de memória *on-chip* e vazão	●	●	●	●	●

todos os prefixos nomeados na memória SRAM/TCAM do comutador dada as limitações na quantidade de memória disponível atualmente. Várias propostas utilizam uma abordagem de *Co-Design*, conforme mostra a Tabela 6.3, onde uma parte dos prefixos da NDN FIB ficam no plano de dados e são processados em *hardware* enquanto a outra parte é processada em *software* no plano de controle. Considerando cenários em que pode haver

Tabela 6.3 – Comparação entre as principais soluções para a NDN FIB existentes na literatura implementadas em *software* e em *hardware*.

Solução para NDN FIB	**HW**	**P4**	**SDN**	**E.D**	**Características de Hardware**						**Co-Design de HW/SW**	**Latência de LNPM**	**Escalável em Hardware**
					Uso de DRAM	Uso de TCAM	Uso de SRAM	Otimização de memória	Otimização de vazão	Suporte à Paralelismo			
NPT	○	○	○	Trie	●	○	○	○	○	○	○	ms	○
PNL	○	○	○	Trie	●	○	○	○	○	●	○	μs	○
NCE	○	○	○	Trie	●	○	○	●	○	○	○	ms	○
Caesar	●	○	○	HT	○	○	●	○	○	●	○	μs	○
fatTree	○	○	○	Trie,HT	●	○	○	●	○	○	○	ms	○
NLAPB	○	○	○	BF,Trie	●	○	○	●	○	○	●	ms	●
dualPatricia	◐	○	○	Trie	◐	○	◐	○	●	○	●	μs	○
compactTrie	○	○	○	Trie	●	○	○	●	○	○	○	ms	○
CONSERT	○	○	○	Trie	●	○	○	●	●	○	○	ms	○
NDN.p4	●	●	○	HT	○	●	○	○	○	●	○	ns	○
MaFIB	●	○	○	BF	◐	○	◐	○	○	●	●	ms	○
B-MaFIB	●	○	○	BF	◐	○	◐	●	●	●	●	ms	○
SACS	●	○	○	HT,Trie	○	●	●	○	●	○	○	μs	○
HBM	●	●	○	HT	○	●	○	◐	○	●	○	ns	○
fAcellerator	●	○	○	BF	●	○	●	○	●	●	●	μs	○
OnChipFIB	●	○	○	Trie	○	○	●	○	○	●	○	μs	●
NDN-DPDK	○	◐	○	HT	●	○	○	○	●	○	○	μs	○
FCTree	○	○	●	Trie	●	○	○	●	○	○	○	ms	○
NFD	○	○	○	Trie,HT	●	○	○	○	○	○	○	ms	○
YaNFD	○	○	○	Trie	●	○	○	○	●	●	○	μs	○
NDNFab	○	◐	●	HT	●	○	○	○	○	●	●	μs	○
NDNTofino	○	◐	○	HT	●	○	○	○	○	●	●	μs	○
Pegasus	○	◐	○	HT	●	○	○	○	○	●	●	μs	○
U-Table	●	◐	○	HT	○	○	●	○	○	●	○	μs	○
CoFIB	●	●	●	HT	◐	◐	●	●	●	●	●	ns	●

E.D = 'Estrutura de Dados'. HT = Hash Table. BF = Bloom-Filter. O símbolo (◐) indica suporte parcial para uma determinada característica.

necessidade de armazenar algumas dezenas de milhões ou até mesmo bilhões de prefixos na FIB, um critério importante a ser observado nas soluções propostas para a NDN FIB é a capacidade de realizar o *offload* de um subconjunto de prefixos seguindo algum critério. Nesse sentido, essa tese introduz o conceito de prefixo nomeado canônico como critério para a escolha de quais prefixos são descarregados na memória interna do comutador programável.

A possibilidade de processar Ipkts na escala de nanosegundos no comutador é outro um critério importante a se observar em razão disso contribuir para a diminuição da latência de processamento por pacote. Das propostas apresentadas no Capítulo 3, apenas as soluções baseadas em comutadores programáveis tem a capacidade de processar pacotes nessa escala, sendo que as demais soluções têm um enfoque maior em reduzir o consumo de memória. Os resultados apresentados nesse capítulo mostraram que a CoFIB é capaz de processar Ipkts na escala de nanosegundos mesmo diante da possibilidade de recirculações de pacotes no *pipeline*.

Por fim, considerando apenas as soluções para a NDN FIB compatíveis com comutadores programáveis, foi possível identificar três questões importantes em aberto na literatura. Primeiro, dada a predominância das tabelas de *hash* como estruturas de dados subjacentes para a NDN FIB, uma questão não tratada satisfatoriamente pela literatura é como lidar com o alto número de colisões de *hash* no *pipeline* de processamento de comutadores programáveis. Reduzir o número de colisões em *hardware* sem aumentar o consumo de memória não é trivial dadas as restrições e limitações de *targets* P4. A estimação da probabilidade de falso encaminhamento apresentada nesse capítulo indica uma baixa probabilidade de colisão de *hash* na CoFIB em comparação com o estado da arte

como consequência da segregação de tabelas e uso de funções de *hash* de 32 *bits*. Em segundo lugar, exceto as soluções NDN.p4 (SIGNORELLO et al., 2016) e HBM (MIGUEL; SIGNORELLO; RAMOS, 2018), todas as demais implementações da arquitetura NDN em comutadores programáveis empregam a FIB no plano de controle. Das soluções que implementam a FIB no plano de dados, somente os recursos de memória (TCAM/SRAM) e processamento (ALU) disponíveis no bloco de ingresso são utilizados, causando um desperdício em torno de 50% dos recursos disponíveis no ASIC programável. Além disso, as soluções propostas para a NDN FIB não provêem mecanismo algum para equilibrar o consumo de memória e a latência de *lookup*, grandezas essas tipicamente inversamente proporcionais. Portanto, a proposta de uma solução para a NDN FIB apresentada nessa tese, a CoFIB, vem de encontro à essas limitações, sendo capaz de escalar para milhões de prefixos nomeados mantendo a latência teórica de processamento de pacotes na escala de nanosegundos.

6.7.2 Questões de Pesquisa

As hipóteses e questões de pesquisas levantadas no Capítulo 1 foram comprovadas e respondidas, respectivamente, ao longo do desenvolvimento desse trabalho. Isso foi realizado por meio da definição de conceitos no Capítulo 4 e Capítulo 5 e da realização de análises quantitativas e experimentais nesse capítulo. A consolidação das respostas às principais questões de pesquisas são apresentadas a seguir:

- **Q1**: *Como processar pacotes NDN eficientemente em comutadores programáveis dadas as restrições de hardware e a complexidade da codificação TLV*?

  R1: A linguagem P4 foi projetada para programar o *pipeline* de processamento de pacotes de forma agnóstica quanto à arquitetura do equipamento de rede subjacente. Desse modo, para garantir o processamento de pacotes de forma flexível sem degradar o desempenho, a especificação da linguagem P4 restringe uma ampla variedade de operações que podem ser realizadas nos *targets* físicos. Dentre essas restrições, a que mais impacta o processamento de pacotes NDN é a dificuldade em processar eficientemente *strings* de comprimento variável. A utilização do tipo *varbit* contorna parcialmente esse problema. No entanto, a linguagem P4 também impõe limitação quanto ao uso de campos *varbit* para a realização de operações de combinação-ação em tabelas. Para contornar essa limitação, vários campos de comprimento fixos são agrupados em uma estrutura do tipo *header_union*, na granularidade de 1 *byte*, de forma a garantir o armazenamento de *strings* de comprimento arbitrário e ainda viabilizar a busca nas tabelas P4 utilizando chaves de comprimento fixo. O uso da estrutura *header_union* otimiza a alocação de memória pelo compilador tendo em vista que apenas um elemento é válido por vez. Para que fosse possível estabelecer o limite máximo de *bits* armazenados em um campo do tipo *varbit*, definiu-se que

cada componente de um nome NDN tenha no máximo 31 caracteres, onde nomes que não se enquadram nesse critério sofrem *bypass* no núcleo por meio do controlador SDN. Em relação à complexidade da codificação TLV de pacotes NDN nativos, essa tese propõe, sem perda de generalidade, o formato P4NF. Apesar de ser um formato rígido, o P4NF facilita o encaminhamento baseado em nomes nos comutadores programáveis de borda e simplifica a operação de extração de cabeçalhos ao eliminar a possibilidade de cabeçalhos aninhados.

❑ **Q2:** *Como eliminar a necessidade de armazenar a FIB em comutadores de núcleo e ao mesmo tempo garantir o encaminhamento baseado em nomes entre comutadores de borda*?

R2: A operação de LNPM na FIB é bem mais complexa que a operação LPM em tabelas de roteamento IP tradicionais. A razão disso é que a NDN FIB normalmente armazena prefixos nomeados de comprimento variável que requerem maior quantidade de espaço de armazenamento quando comparados à prefixos IP. Desse modo, para otimizar o uso de memória ao armazenar a FIB em roteadores NDN, a operação LNPM acaba impactando a latência de processamento e inviabilizando o encaminhamento de Ipkts à taxa de linha. Quando a tabela FIB é substituída por estruturas equivalentes (p. ex.: *Label Information Base* (LIB) em redes MPLS) capazes de garantir o processamento de Ipkts à taxa de linha, é possível reduzir a latência fim-a-fim por pacote. Todavia, o princípio NDN nativo de encaminhamento baseado em nomes é comprometido nessa abordagem. Para resolver esse problema, a arquitetura proposta nessa tese emprega a FIB apenas nos comutadores programáveis de borda. Isso faz com que, ao invés da operação LNPM retornar a sequência de interfaces de saída do roteador, conforme a especificação original da arquitetura NDN, a LNPM proposta nessa tese retorna a lista dos comutadores de borda programáveis pertencentes ao domínio que respondem pelo prefixo nomeado. Assim, ao passar pela CoFIB, o Ipkt é precedido por campos de comprimento fixo que serão usados pelos comutadores de núcleo para encaminhar o pacote à taxa de linha realizando o combinação-ação de forma rápida. Tendo em vista que a arquitetura proposta é baseada no paradigma SDN, as tabelas dos comutadores de núcleo são facilmente configuradas e populadas com entradas que garantem o encaminhamento entre os comutadores de borda do domínio. Essa abordagem elimina a necessidade de armazenar a FIB em todos os comutadores do domínio, contribuindo para a diminuição da latência fim-a-fim e garantindo a propriedade NDN nativa de encaminhamento baseado em nomes.

❑ **Q3**: *Em razão da impossibilidade de armazenar todos os prefixos da RIB na FIB, devido a limitação de memória on-chip, qual critério utilizar para selecionar um subconjunto de prefixos para descarregar na CoFIB*?

R3: O surgimento de *pipelines* reconfiguráveis através de linguagens de domínio específico como P4 têm possibilitado que uma ampla gama de funções de rede, originalmente desenvolvidas para rodar em *software*, sejam descarregadas em comutadores físicos para serem executadas em alta velicidade em *hardware* (p.ex.: ASIC, FPGA, etc). A maioria dessas funções utiliza tabelas do tipo combinação-ação para viabilizar a execução de suas rotinas de processamento. Ao descarregar uma determinada função de rede para o plano de dados, a manutenção da semântica original dessa função é facilmente garantida por meio da expressividade da linguagem P4. Todavia, o desafio é como descarregar as entradas dessas tabelas, originalmente armazenadas no plano de controle em memórias de maior capacidade (p.ex.: DRAM), para a memória *on-chip* dos comutadores físicos, geralmente SRAM e TCAM. Considerando a impossibilidade de armazenar todos os prefixos da RIB no plano de dados, o conceito de prefixo nomeado canônico, introduzido pela primeira vez nessa tese, é o principal critério utilizado para selecionar um subconjunto de prefixos da RIB para ser descarregado na CoFIB em tempo de execução.

- ❏ **Q4:** *Como armazenar componentes com menos de 5 caracteres de forma eficiente na SRAM?*

R4: Arquiteturas baseadas no modelo RMT possibilitam que entradas em tabelas de combinação-ação possam ser agrupadas horizontalmente para reduzir a fragmentação de memória. Assim, entradas na CoFIB com menos de 5 caracteres são armazenadas diretamente em formato ASCII. O compilador é responsável por agrupar entradas de determinados comprimentos para formar entradas de largura compatível com as palavras de memória suportadas pela arquitetura. Esses agrupamentos evitam o desperdício de memória e fazem com que entradas de 1 Byte, por exemplo, consumam em média 1 Byte de memória. A vantagem de armazenar essas palavras diretamente em ASCII e não utilizando funções de *hash* é que não existe a possibilidade de colisões e essas entradas sempre consomem menos espaço que *hashes* de 32 *bits*, por exemplo.

- ❏ **Q5:** *Quais técnicas de compressão podem ser utilizadas para reduzir a quantidade de memória on-chip necessária para armazenar prefixos nomeados na NDN FIB?*

R5: A memória *on-chip* disponível em comutadores programáveis modernos é composta por bancos de TCAM e SRAM. A TCAM garante que as operações de combinação-ação sejam realizadas em um único ciclo de *clock*. Porém, a sua disponibilidade é menor quando comparada à SRAM, o que torna impossível armazenar milhões de prefixos nomeados unicamente na TCAM. Assim, em uma direção oposta do que vem sendo utilizado na literatura, essa tese propõe a utilização da SRAM para o armazenamento dos prefixos da CoFIB. No entanto, mesmo que a disponibilidade de SRAM seja mais abundante quando comparado à TCAM, ela é

algumas ordens de magnitude menor que bancos de DRAM convencionais utilizados em diferentes propostas na literatura para a NDN FIB em *software*. Desse modo, armazenar prefixos nomeados na escala de milhões em memórias do tipo SRAM requer mecanismos de compressão. De maneira geral, essa tese emprega três técnicas para a redução do consumo de memória. A primeira é a utilização de prefixos nomeados canônicos. Nesses prefixos, um componente só pode aparecer em uma determinada posição e isso diminui a quantidade de *bits* necessários para codificar a posição do componente na estrutura CAD. A segunda técnica envolve a utilização de funções de *hash* de 32 *bits* para armazenar componentes nomeados com 5 caracteres ou mais. Por fim, essa tese explora recursos nativos da linguagem P4, como *action profiles* e *action selectors* para diminuir o espaço de memória ocupado por tabelas no *pipeline*. Em razão das tabelas da DFIB estarem associadas a apenas uma ação, os resultados experimentais demonstraram que essa técnica foi capaz de reduzir em cerca de 17% a quantidade de memória quando comparado com a utilização de tabelas P4 convencionais.

- ❏ **Q6:** *Qual critério pode ser utilizado para agrupar componentes nomeados a fim de determinar o número de tabelas P4 a serem utilizadas*?

  R6: A utilização de múltiplas tabelas P4 dá ao compilador a chance de fazer uma melhor alocação de memória para armazenar as entradas. O critério adotado nessa tese para determinar a quantidade de tabelas a serem utilizadas para armazenar os prefixos nomeados é baseado no comprimento de cada componente do prefixo. Como o comprimento máximo de cada componente nomeado é fixado em 31, são utilizadas 31 tabelas P4 para armazenar os prefixos nomeados da FIB. Cada uma dessas tabelas armazena componentes de determinado comprimento. Essa abordagem contribui para a diminuição da probabilidade de colisões de *hash* tendo em vista que componentes nomeados que colidem são permitidos caso eles tenham comprimentos distintos.

- ❏ **Q7:** *Quais mecanismos podem ser utilizados para diminuir a quantidade de recirculações de pacotes no pipeline e mitigar o efeito dessas recirculações em relação à latência de processamento*?

  R7: Dispositivos físicos compatíveis com a linguagem P4 normalmente restringem várias operações de alto nível para garantir o processamento de pacotes em alta velocidade. Dentre as operações que não são permitidas está a possibilidade de laços de repetição. A exemplo da função de LNPM no mecanismo proposto, como uma única passagem no *pipeline* não é suficiente para garantir a execução de determinadas funções, a arquitetura da maioria dos *targets* prevê a possibilidade de recircular e resubmeter pacotes múltiplas vezes em um *pipeline* de processamento. O problema dessa abordagem é o aumento da latência e a redução da vazão. Para reduzir a

quantidade de recirculações necessárias para efetuar a LNPM, essa tese propõe uma heurística para posicionamento das tabelas da DFIB nos *pipelines* de entrada e saída de acordo com a distribuição de comprimento de componentes observados em *datasets* de nomes disponíveis publicamente. Diante da impossibilidade de eliminar completamente as recirculações, para mitigar seus efeitos, essa tese apresenta três soluções. A primeira delas é a utilização de tabelas de moldes (DPST e CPST) que visam identificar o número máximo de componentes que são compatíveis com determinado prefixo. Esse mecanismo evita recirculações desnecessárias, diminuindo a latência por pacote. Uma segunda abordagem é a limitação da quantidade máxima de componentes por prefixo, sendo esse limite fixado em 8 componentes. Desse modo, no pior caso, o número de recirculações fica limitado a 7. Os experimentos apresentados no Capítulo 6 mostram que, para determinados conjuntos de *datasets* de nomes, cerca de 90% dos prefixos canônicos possuem até quatro componentes, que gera no máximo três operações de recirculações. Isso mantém a latência média de processamento na ordem de nanosegundos. Por fim, a terceira abordagem adotada é a utilização de portas específicas dedicadas à operações de recirculações. Trabalhos na literatura mostram que essa abordagem leva a um ganho de até 11x na vazão final.

6.8 Considerações

Neste capítulo, foi detalhado o *testbed* experimental da arquitetura FANTNet. Utilizando esse *testbed*, tanto os componentes do plano de controle quanto os do plano de dados foram avaliados através de análises quantitativas e qualitativas usando *datasets* de nomes reais e sintéticos. A análise quantitativa forneceu resultados sobre a taxa de descarregamento de prefixos canônicos da RIB, a probabilidade de encaminhamento incorreto, o consumo de memória e métricas de vazão, obtidas por meio de experimentos de simulação. Além disso, a FANTNet e a CoFIB foram comparadas qualitativamente com a literatura correlata e as respostas às questões de pesquisa levantadas no Capítulo 1 foram consolidadas. No próximo capítulo as considerações finais serão apresentadas, destacando novamente as contribuições e a produção bibliográfica realizada ao fim desse trabalho de doutorado.

Capítulo 7

Considerações Finais

A arquitetura NDN se apresenta como uma alternativa promissora para a consolidação da Internet do futuro. Através de características como transmissão *multicast*, *caching* em roteadores e encaminhamento baseado em nomes, as redes NDN são capazes de resolver grande parte das limitações existentes no modelo TCP/IP. No entanto, o projeto de roteadores NDN para redes de larga escala precisa lidar com inúmeros desafios de caráter técnico. Dentre esses desafios, o desenvolvimento de estruturas de dados para a FIB em *hardware* que sejam rápidas e eficientes no consumo de memória é ainda um problema de pesquisa não resolvido satisfatoriamente. Nesse sentido, a proposta apresentada nessa tese se concentra em como viabilizar o encaminhamento eficiente de pacotes NDN em uma rede de trânsito baseada no paradigma SDN através do desenvolvimento de uma estrutura de dados para a FIB, chamada CoFIB, que roda em comutadores de borda. A CoFIB é o principal elemento da arquitetura FANTNet.

O desenvolvimento desse trabalho mostrou que os objetivos estabelecidos na concepção inicial da ideia foram atingidos. De uma maneira geral, o principal objetivo desse trabalho foi contribuir com uma arquitetura para viabilizar o tráfego em alta velocidade de pacotes NDN entre múltiplos domínios, chamada aqui de arquitetura FANTNet. Para viabilizar o encaminhamento em alta velocidade na rede de trânsito, a FANTNet utiliza comutadores programáveis na borda. Para isso, foi necessário projetar e desenvolver uma estrutura de dados para a FIB, chamada aqui de CoFIB, que fosse compatível com esses comutadores e que pudesse comprimir os nomes de modo a ser possível armazenar milhões de prefixos na memória SRAM desses comutadores. O mecanimo de compressão proposto foi baseado no conceito de prefixo nomeado canônico, introduzido pela primeira vez nessa tese. Assim, desenvolveu-se um algoritmo para a identificação e extração de prefixos canônicos da RIB para serem armazenados no plano de dados. Além disso, foi proposto também uma heurística para otimizar o posicionamento de tabelas nos *pipelines* de entrada e saída a fim de diminuir o número de recirculações de pacotes durante a LNPM. No plano de controle, foi necessário também desenvolver estruturas de dados como a DCH, para suportar as operações de inserção, remoção e atualização de prefixos na CoFIB. Para

facilitar a operação de extração de cabeçalhos, foi proposto também um novo formato para a representação do nome NDN, chamado P4NF, que atende as limitações e restrições de dispositivos P4. Todos esses desenvolvimentos demonstraram que o objetivo geral e específicos foram alcançados.

Os resultados da avaliação da proposta obtidos por meio de análise quantitativa e simulação mostraram que a CoFIB apresenta um desempenho melhor que soluções alternativas em relação a chance de prefixos nomeados falharem no processo de *lookup* devido a colisões de *hash*. Além disso, a estrutura CoFIB é capaz de reduzir o consumo de memória *on-chip* em comparação com as propostas NDN.p4 e HBM. As análises também mostraram que a CoFIB consegue manter a velocidade de *lookup* na mesma ordem de magnitude que soluções baseadas em *hardware* para *datasets* com mais de 1 milhão de nomes.

7.1 Principais Contribuições

Essa tese propôs a FANTNet, uma arquitetura de rede de trânsito para viabilizar o encaminhamento de tráfego em alta velocidade entre múltiplos domínios NDN. Essa arquitetura é baseada no modelo *Fabric* e utiliza o paradigma SDN para prover um gerenciamento centralizado dos elementos da rede. Nesse sentido, as principais contribuições dessa tese incluem a especificação dos componentes dessa arquitetura no plano de controle e o desenvolvimento da estrutura de dados CoFIB e demais elementos relacionados ao plano de dados, cujo princípio de projeto é minimizar o consumo de memória *on-chip* em comutadores programáveis de borda. Além disso, essa tese introduziu o conceito de prefixo nomeado canônico e apresentou um algoritmo para a extração *offline* de prefixos canônicos da RIB em tempo de execução. As demais contribuições dessa tese incluem um algoritmo para a LNPM que realiza operações de combinação-ação tanto no bloco de controle de entrada como no de saída além da proposta de uma heurística para otimizar o posicionamento de tabelas P4 nos *pipelines* de entrada e saída a fim de reduzir a quantidade de recirculações de pacotes e melhorar a vazão. Essas abordagens são aspectos inovadores trazidos pela proposta apresentada nesse trabalho.

7.2 Trabalhos Futuros

Uma das limitações da FANTNet, em seu estágio atual, é o suporte a apenas oito comutadores de borda quando a transmissão *multicast* está habilitada. Desse modo, a escalabilidade da solução proposta fica limitada a redes de trânsito de porte intermediário. Embora seja possível reinterpretar os *bits* da estrutura CAD (*swId*) para dar suporte à até 255 comutadores de borda, essa abordagem elimina o suporte à transmissão *multicast* e *multipath* nativas da arquitetura NDN. Um dos caminhos possíveis em trabalhos

futuros é avaliar o custo-benefício de utilizar uma estrutura CAD com mais *bits*, a fim de suportar mais comutadores de borda mantendo as características NDN nativas, sem que isso impacte de forma significativa o consumo de memória.

Ainda em relação à FANTNet, de acordo com a forma com que o algoritmo CPE foi implementado, o único critério utilizado para descarregar os prefixos na CoFIB é ser considerado canônico. Essa abordagem não leva em consideração, por exemplo, a popularidade ou frequência de acesso aos prefixos. Em razão de parte dos prefixos (não canônicos) estarem armazenados na memória *off-chip* do controlador SDN (p.ex.: na CFIB), é importante minimizar os acessos ao plano de controle centralizado o tanto quanto possível. Uma direção de pesquisa futura é desenvolver mecanimos para avaliar a popularidade de determinados prefixos em tempo de execução e utilizar também esse critério para fazer o descarregamento no plano de dados.

Embora não faça parte do escopo dessa tese, a arquitetura FANTNet foi projetada para suportar o processmento de tráfego IP de forma simultânea ao tráfego NDN, tendo em vista que os comutadores programáveis de borda são flexíveis e existe memória *on-chip* disponível para acomodar tráfego extra, conforme mostrou os experimentos da Seção 6.6. De fato, o mecanismo de encapsulamento usado para carregar o *SwId* em quadros Ethernet no núcleo possibilita o transporte por rede de sobreposição. Assim, uma direção de pesquisa futura é explorar cenários de coêxistencia entre NDN e outras arquiteturas para a Internet do Futuro, como a Etarch (GUIMARãES et al., 2014) e o FIXP (SILVA et al., 2022), (GAVAZZA et al., 2020), integrando mecanismos de QoS, a fim de tornar a FANTNet agnóstica quanto a arquitetura de rede sem impactar na experiência do usuário.

Em relação à CoFIB, a principal limitação é a suposição de ter prefixos canônicos no plano de dados. De fato, devido à sua natureza descentralizada e distribuída, a arquitetura NDN não impõe qualquer suposição sobre as características dos nomes já que a definição do *namespace* depende de cada domínio NDN. No entanto, como a quantidade de memória TCAM/SRAM disponível nos ASICs programáveis atuais não é suficiente para armazenar a NDN FIB em sua totalidade, considerando uma rede com o tamanho da Internet atual, é necessário implementar mecanismos para descarregar um subconjunto de prefixos para armazená-los na memória *on-chip*. Assim, o conceito de prefixo nomeado canônico pode ser usado como uma das possíveis métricas de descarregamento. No entanto, é razoável supor que tal técnica proporcionaria melhores resultados se combinada com outros métodos, incluindo mecanismos de *cache* baseados em popularidade, sendo essa uma possível direção de pesquisa em trabalhos futuros.

A heurística de posicionamento de tabelas proposta para reduzir a quantidade de recirculações foi eficaz em relação ao posicionamento linear, conforme demonstrado nos experimentos realizados. Todavia, uma questão que ainda permanece em aberto é se existe alguma forma de tornar esse posicionamento ótimo por meio de técnicas como programação linear inteira. Caso isso se mostre inviável, em trabalhos futuros é possível

empregar técnicas metaheurísticas e aprendizado de máquina para realizar esse posicionamento de maneira a reduzir ainda mais a quantidade de recirculações de pacotes em relação à heurística proposta.

A implementação das tabelas PIT e CS tanto no plano de dados como no plano de controle da FANTNet também é uma direção de pesquisa importante. Em relação à CoFIB, em trabalhos futuros é preciso avaliar como ela se comporta implementada com alterações na RIB ao longo do tempo. Adicionalmente, avaliar os aspectos relacionados à segurança e incorporar no plano de dados recursos nativos da arquitetura NDN não tratatos nessa tese, como a estratégia de encaminhamento, pode contribuir para uma melhor integração entre as redes centradas na informação e o paradigma SDN.

Os experimentos realizados no Capítulo 6 utilizam um gerador de tráfego de taxa constante e apenas um nó consumidor transmite Ipkts. Em trabalhos futuros é importante investigar os efeitos de geredores de tráfego mais complexos (ex.: Poisson, Erlang, etc), como em (ROSA; GUARDIEIRO, 2011), explorando diferentes topologias, e analisar o comportamento da FANTNet e CoFIB em um cenário dinâmico com todos os comutadores de borda recebendo tráfego simultaneamente. Esse ambiente dinâmico abre espaço para melhor avaliar métricas de QoS como a taxa de ocupação de fila e quais algoritmos de escalonamento de pacotes são mais adequados nos comutadores de borda e de núcleo em cenários com alta carga de tráfego.

De uma perspectiva mais técnica, em trabalhos futuros é possível implementar os elementos do plano de controle da FANTNet de maneira virtualizada, usando ambientes como o *Docker* (MERKEL, 2014) ou *Kubernetes* (BURNS et al., 2022). Outra direção possível é implementar o plano de controle da arquitetura proposta em controladores SDN considerados estado da arte, como o ONOS (BERDE et al., 2014). O custo para realizar essa implementação não é alto em razão das estruturas do plano de controle estarem implementadas na linguagem Java, assim como os principais módulos do controlador ONOS. Já em relação ao plano de dados, o principal trabalho futuro é implementar a estrutura CoFIB em ASICs programáveis físicos como o Intel Tofino ou em SmartNICs programáveis como NetFPGA. Essa implementação permitirá a exploração de recursos de paralelismo de *hardware* e processamento em Gbps ou Tbps, que contribuirão para a diminuição da latência por pacote e aumento da vazão média. Além disso, essa implementação ajudará a responder algumas questões em aberto que não puderam ser tratadas nos experimentos realizados em razão do uso de *software switches* ao invés de *switches* ou SmartNICs de alto desempenho.

7.3 Produção Bibliográfica

Essa seção apresenta a produção bibliográfica gerada através da concepção, análise e experimentos realizados ao longo do desenvolvimento desse trabalho. As produções diretas

e indiretas geradas durante o desenvolvimento dessa tese são apresentadas a seguir.

Trabalhos diretamente ligados à esta tese:

- ROSA, E. C.; SILVA, F. d. O. A hash-free method for fib and lnpm in icn programmable data planes. In: 2022 International Conference on Information Networking (ICOIN). Jeju-si, Korea, Republic of: IEEE, 2022. p. 186–191. DOI: 10.1109/ICOIN53446.2022.9687201. (Qualis **A3**).
- ROSA, E. C.; SILVA, F. de O. A review on recent ndn fib implementations for high-speed switches. In: BAROLLI, L.; HUSSAIN, F.; ENOKIDO, T. (Ed.). Advanced Information Networking and Applications. Cham, Switzerland: Springer Nature, 2022. (AINA '22), p. 288–300. DOI: 10.1007/978-3-030-99619-2_28. (Qualis **A3**).
- ROSA, E.; SILVA, F. Enabling native coexistence between icn and tcp/ip architectures over the same domain. In: Anais do XI Workshop de Pesquisa Experimental da Internet do Futuro. Porto Alegre, RS, Brasil: SBC, 2020. p. 13–19. DOI: 10.5753/wpeif.2020.12469. (**Best Paper Award**). (Qualis **B4**).
- ROSA, E.; CUNHA, I.; SILVA, F. A simple approach to verify and debug data plane programs. In: Anais do XIII Workshop de Pesquisa Experimental da Internet do Futuro. Porto Alegre, RS, Brasil: SBC, 2022. (WPEIF, 22), p. 47–52. DOI: 10.5753/wpeif.2022.223586. (Qualis **B4**).

Trabalhos em colaboração com outros membros do grupo de pesquisa:

- CUNHA, I.; ROSA, E.; SILVA, F. Highly Reliable Communication Using Multipath Slices with Alternating Transmission. In: BAROLLI, L. (Ed.). Advanced Information Networking and Applications. Cham, Switzerland: Springer Nature, 2024. (AINA'24), p. 471–482. DOI: 10.1007/978-3-031-57916-5_40. (Qualis **A3**).
- CUNHA, I.; ROSA, E.; SILVA, F. Comunicação fim-a-fim altamente confiável e de baixa latência entre ues móveis no contexto de 5g/b5g via multipath slices elásticas. In: Anais do XIII Workshop de Pesquisa Experimental da Internet do Futuro. Porto Alegre, RS, Brasil: SBC, 2022. p. 41–46. DOI: 10.5753/wpeif.2022.223584. (Qualis **B4**).

Além desses trabalhos, foi submetido um artigo para o periódico *IEEE TRANSACTIONS ON NETWORK AND SERVICE MANAGEMENT* do IEEE (**Qualis A1**), intitulado "*CoFIB: A Fast and Memory-Efficient FIB Design for Programmable Edge Switches in NDN Transit Networks Based on SDN*". Esse artigo aborda os aspectos do plano

de controle e do plano de dados da CoFIB e encontra-se atualmente em **processo de peer-review**.

Referências

ADRICHEM, N. L. M. van; KUIPERS, F. A. Ndnflow: Software-defined named data networking. In: IEEE. **Proceedings of the 2015 1st IEEE Conference on Network Softwarization (NetSoft)**. London, UK: IEEE, 2015. p. 1–5. Disponível em: <https://doi.org/10.1109/NETSOFT.2015.7116131>. Acesso em: 20/07/2024.

AFANASYEV, A. et al. A brief introduction to named data networking. In: **MILCOM 2018-2018 IEEE Military Communications Conference (MILCOM)**. Los Angeles, CA, USA: IEEE, 2018. p. 1–6. Disponível em: <https://doi.org/10.1109/MILCOM.2018.8599682>. Acesso em: 20/07/2024.

_____. **Packet Fragmentation in NDN: Why NDN Uses Hop-By-Hop Fragmentation**. Los Angeles, CA, USA, 2015. 4 p. Disponível em: <https://named-data.net/publications/techreports/>. Acesso em: 20/07/2024.

AGRAWAL, B.; SHERWOOD, T. Ternary cam power and delay model: Extensions and uses. **IEEE Transactions on Very Large Scale Integration (VLSI) Systems**, IEEE, v. 16, n. 5, p. 554–564, 2008. Disponível em: <https://doi.org/10.1109/TVLSI.2008.917538>. Acesso em: 20/07/2024.

AHDAN, S.; SITUMORANG, H.; SYAMBAS, N. R. Effect of overhead flooding on ndn forwarding strategies based on broadcast approach. In: **2017 11th International Conference on Telecommunication Systems Services and Applications (TSSA)**. Lombok, Indonesia: IEEE, 2017. p. 1–4. Disponível em: <https://doi.org/10.1109/TSSA.2017.8272907>. Acesso em: 20/07/2024.

AHLGREN, B. et al. A survey of information-centric networking. **IEEE Communications Magazine**, IEEE, v. 50, n. 7, p. 26–36, 2012. Disponível em: <https://doi.org/10.1109/MCOM.2012.6231276>. Acesso em: 20/07/2024.

ALDAOUD, M. et al. Leveraging icn and sdn for future internet architecture: A survey. **Electronics**, MDPI, v. 12, n. 7, p. 1–36, 2023. Disponível em: <https://www.mdpi.com/2079-9292/12/7/1723>. Acesso em: 20/07/2024.

BERDE, P. et al. Onos: Towards an open, distributed sdn os. In: **Proceedings of the Third Workshop on Hot Topics in Software Defined Networking**. Chicago, IL, USA: Association for Computing Machinery, 2014. (HotSDN ’14), p. 1–6. Disponível em: <https://doi.org/10.1145/2620728.2620744>. Acesso em: 20/07/2024.

BMV2 DEVELOPMENT TEAM. **BMv2**: The behavioral model. Online, 2024. Disponível em: <https://github.com/p4lang/behavioral-model>. Acesso em: 20/07/2024.

BOSSHART, P. et al. Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. **Computer Communication Review**, Association for Computing Machinery, v. 43, n. 4, p. 99–110, 2013. Disponível em: <https://doi.org/10.1145/2534169.2486011>. Acesso em: 20/07/2024.

BOUK, S. H.; AHMED, S. H.; KIM, D. Hierarchical and hash based naming with compact trie name management scheme for vehicular content centric networks. **Computer Communications**, Elsevier, v. 71, p. 73–83, 2015. Disponível em: <https://doi.org/10.1016/j.comcom.2015.09.014>. Acesso em: 20/07/2024.

BURNS, B. et al. **Kubernetes: Up and Running: Dive Into the Future of Infrastructure**. 3. ed. Sebastopol, CA, USA: O'Reilly Media, 2022. 326 p.

CABRAL, C. M.; ROTHENBERG, C. E.; aES, M. F. M. Reproducing real ndn experiments using mini-ccnx. In: **Proceedings of the 3rd ACM SIGCOMM Workshop on Information-Centric Networking**. Hong Kong, China: Association for Computing Machinery, 2013. (ICN '13), p. 45–46. Disponível em: <https://doi.org/10.1145/2491224.2491242>. Acesso em: 20/07/2024.

CASADO, M. et al. Fabric: A retrospective on evolving sdn. In: **Proceedings of the First Workshop on Hot Topics in Software Defined Networks**. Helsinki, Finland: Association for Computing Machinery, 2012. (HotSDN '12), p. 85–90. Disponível em: <https://doi.org/10.1145/2342441.2342459>. Acesso em: 20/07/2024.

CHEN, X. Implementing aes encryption on programmable switches via scrambled lookup tables. In: **Proceedings of the Workshop on Secure Programmable Network Infrastructure**. Virtual Event, USA: Association for Computing Machinery, 2020. (SPIN '20), p. 8–14. Disponível em: <https://doi.org/10.1145/3405669.3405819>. Acesso em: 20/07/2024.

CHOLE, S. et al. Drmt: Disaggregated programmable switching. In: **Proceedings of the Conference of the ACM Special Interest Group on Data Communication**. Los Angeles, CA, USA: Association for Computing Machinery, 2017. (SIGCOMM '17), p. 1–14. Disponível em: <https://doi.org/10.1145/3098822.3098823>. Acesso em: 20/07/2024.

CUNHA, I. et al. Highly Reliable Communication Using Multipath Slices with Alternating Transmission. In: BAROLLI, L. (Ed.). **Advanced Information Networking and Applications**. Cham, Switzerland: Springer Nature, 2024. (AINA '24), p. 471–482. Disponível em: <https://doi.org/10.1007/978-3-031-57916-5_40>. Acesso em: 20/07/2024.

CUNHA, I.; ROSA, E.; SILVA, F. Comunicação fim-a-fim altamente confiável e de baixa latência entre ues móveis no contexto de 5g/b5g via multipath slices elásticas. In: **Anais do XIII Workshop de Pesquisa Experimental da Internet do Futuro**. Porto Alegre, RS, Brasil: SBC, 2022. p. 41–46. Disponível em: <https://sol.sbc.org.br/index.php/wpeif/article/view/21491>. Acesso em: 20/07/2024.

DAI, H.; LIU, B. Consert: Constructing optimal name-based routing tables. **Computer Networks**, Elsevier, v. 94, p. 62–79, 2016. Disponível em: <https://doi.org/10.1016/j.comnet.2015.11.020>. Acesso em: 20/07/2024.

DMOZ. **DMOZ 2006 Dataset and its Wikification**. 2019. Disponível em: <https://data.mendeley.com/datasets/9mpgz8z257/1>. Acesso em: 20/07/2024.

DOMCOP. **Top 10 million Websites**. 2021. Disponível em: <https://www.domcop.com/top-10-million-websites>. Acesso em: 20/07/2024.

FEAMSTER, N.; BORKENHAGEN, J.; REXFORD, J. Guidelines for interdomain traffic engineering. **Computer Communication Review**, Association for Computing Machinery, v. 33, n. 5, p. 19–30, oct 2003. Disponível em: <https://doi.org/10.1145/963985.963988>. Acesso em: 20/07/2024.

FEAMSTER, N.; REXFORD, J.; ZEGURA, E. The road to sdn: an intellectual history of programmable networks. **Computer Communication Review**, Association for Computing Machinery, v. 44, n. 2, p. 87–98, 2014. Disponível em: <https://doi.org/10.1145/2602204.2602219>. Acesso em: 20/07/2024.

FINGERHUT, A. **High losses in performance test**. 2024. <https://github.com/p4lang/behavioral-model/issues/823#issuecomment-554687253>. Issue #823 on github. Acesso em: 20/07/2024.

FOSTER, N. et al. Frenetic: A network programming language. **ACM Sigplan Notices**, Association for Computing Machinery, v. 46, n. 9, p. 279–291, sep 2011. Disponível em: <https://doi.org/10.1145/2034574.2034812>. Acesso em: 20/07/2024.

FRANCO, D. et al. A comprehensive latency profiling study of the tofino p4 programmable asic-based hardware. **Computer communications**, Elsevier B.V, v. 218, p. 14–30, 2024. Disponível em: <https://doi.org/10.1016/j.comcom.2024.01.010>. Acesso em: 20/07/2024.

GAVAZZA, J. A. T. et al. Future internet exchange point (fixp): Enabling future internet architectures interconnection. In: BAROLLI, L. et al. (Ed.). **Advanced Information Networking and Applications**. Cham, Switzerland: Springer International Publishing, 2020. (AINA '22), p. 703–714. Disponível em: <https://doi.org/10.1007/978-3-030-44041-1_62>. Acesso em: 20/07/2024.

GHASEMI, C. et al. A fast and memory-efficient trie structure for name-based packet forwarding. In: IEEE. **2018 IEEE 26th International Conference on Network Protocols (ICNP)**. 2018. p. 302–312. Disponível em: <https://doi.org/10.1109/ICNP.2018.00046>. Acesso em: 20/07/2024.

GRPC DEVELOPMENT TEAM. **gRPC A high performance, open source universal RPC framework**. Online, 2024. Disponível em: <https://grpc.io/>. Acesso em: 20/07/2024.

GUIMARÃES, C. et al. Exploring interoperability assessment for future internet architectures roll out. **Journal of Network and Computer Applications**, Elsevier BV, v. 136, p. 38–56, 2019. Disponível em: <https://doi.org/10.1016/j.jnca.2019.04.008>. Acesso em: 20/07/2024.

GUIMARãES, C. et al. Ieee 802.21-enabled entity title architecture for handover optimization. In: **2014 IEEE Wireless Communications and Networking Conference (WCNC)**. Istanbul, Turkey: [s.n.], 2014. (WCNC '14), p. 2671–2676. Disponível em: <https://doi.org/10.1109/WCNC.2014.6952830>. Acesso em: 20/07/2024.

GüNDOgAN, C. et al. Reliable firmware updates for the information-centric internet of things. In: **Proceedings of the 8th ACM Conference on Information-Centric Networking**. Paris, France: Association for Computing Machinery, 2021. (ICN '21), p. 59–70. Disponível em: <https://doi.org/10.1145/3460417.3482974>. Acesso em: 20/07/2024.

GUO, X. et al. An efficient ndn routing mechanism design in p4 environment. In: **2021 2nd Information Communication Technologies Conference (ICTC)**. Nanjing, China: IEEE, 2021. p. 28–33. Disponível em: <https://doi.org/10.1109/ICTC51749.2021.9441639>. Acesso em: 20/07/2024.

HADDADI, H. et al. Network topologies: inference, modeling, and generation. **IEEE Communications Surveys & Tutorials**, IEEE, v. 10, n. 2, p. 48–69, 2008. Disponível em: <https://doi.org/10.1109/COMST.2008.4564479>. Acesso em: 20/07/2024.

HOQUE, A. K. M. M. et al. Nlsr: Named-data link state routing protocol. In: **Proceedings of the 3rd ACM SIGCOMM Workshop on Information-Centric Networking**. Hong Kong, China: Association for Computing Machinery, 2013. (ICN '13), p. 15–20. Disponível em: <https://doi.org/10.1145/2491224.2491231>. Acesso em: 20/07/2024.

HUANG, K.; WANG, Z. A hybrid approach to scalable name prefix lookup. In: **2018 IEEE/ACM 26th International Symposium on Quality of Service (IWQoS)**. IEEE, 2018. p. 1–10. Disponível em: <https://doi.org/10.1109/IWQoS.2018.8624182>. Acesso em: 20/07/2024.

INTERNATIONAL TELECOMMUNICATION UNION. **Recommendation ITU-T X.690**: Information technology – asn.1 encoding rules: Specification of basic encoding rules (ber), canonical encoding rules (cer) and distinguished encoding rules (der). Geneva, Switzerland, 2021. 38 p. Disponível em: <https://www.itu.int/rec/T-REC-X.690-202102-I/en>. Acesso em: 20/07/2024.

INTERNET ENGINEERING TASK FORCE. **RFC 1035**: Domain names-implementation and specification. Wilmington, DE, USA, 1987. 55 p. Disponível em: <https://www.rfc-editor.org/rfc/rfc1035>. Acesso em: 20/07/2024.

_____. **RFC 4271**: A border gateway protocol 4 (bgp-4). Wilmington, DE, USA, 2006. 104 p. Disponível em: <https://www.rfc-editor.org/info/rfc4271>. Acesso em: 20/07/2024.

_____. **RFC 6020**: Yang - a data modeling language for the network configuration protocol (netconf). Wilmington, DE, USA, 2010. 173 p. Disponível em: <https://doi.org/10.17487/RFC6020>. Acesso em: 20/07/2024.

_____. **RFC 6241**: Network configuration protocol (netconf). Wilmington, DE, USA, 2011. 113 p. Disponível em: <https://www.rfc-editor.org/rfc/rfc6241>. Acesso em: 20/07/2024.

_____. **RFC 8446**: The Transport Layer Security (TLS) Protocol Version 1.3. Wilmington, DE, USA, 2018. 160 p. Disponível em: <https://datatracker.ietf.org/doc/rfc8446>. Acesso em: 20/07/2024.

_____. **RFC 8569**: Content-centric networking (ccnx) semantics. Wilmington, DE, USA, 2019. 40 p. Disponível em: <https://datatracker.ietf.org/doc/rfc8569/>. Acesso em: 20/07/2024.

_____. **RFC 8609**: Content-centric networking (ccnx) messages in tlv format. Wilmington, DE, USA, 2019. 46 p. Disponível em: <https://datatracker.ietf.org/doc/rfc8609/>. Acesso em: 20/07/2024.

_____. **RFC 9000**: Quic: A udp-based multiplexed and secure transport. Wilmington, DE, USA, 2021. 151 p. Disponível em: <https://www.rfc-editor.org/rfc/rfc9000.html>. Acesso em: 20/07/2024.

JACOBSON, V. et al. Networking named content. In: **Proceedings of the 5th International Conference on Emerging Networking Experiments and Technologies**. Rome, Italy: Association for Computing Machinery, 2009. (CoNEXT '09), p. 1–12. Disponível em: <https://doi.org/10.1145/1658939.1658941>. Acesso em: 20/07/2024.

KARRAKCHOU, O.; SAMAAN, N.; KARMOUCH, A. Endn: An enhanced ndn architecture with a p4-programmabie data plane. In: **Proceedings of the 7th ACM Conference on Information-Centric Networking**. Virtual Event, Canada: Association for Computing Machinery, 2020. (ICN '20), p. 1–11. Disponível em: <https://doi.org/10.1145/3405656.3418720>. Acesso em: 20/07/2024.

_____. Fctrees: A front-coded family of compressed tree-based fib structures for ndn routers. **IEEE Transactions on Network and Service Management**, v. 17, n. 2, p. 1167–1180, 2020. Disponível em: <https://doi.org/10.1109/TNSM.2020.2969172>. Acesso em: 20/07/2024.

KHELIFI, H. et al. Named data networking in vehicular ad hoc networks: State-of-the-art and challenges. **IEEE Communications Surveys & Tutorials**, v. 22, n. 1, p. 320–351, 2020. Disponível em: <https://doi.org/10.1109/COMST.2019.2894816>. Acesso em: 20/07/2024.

KUMAR, S. **P416 Intel® Tofino™ Native Architecture – Public Version**. Intel - Barefoot Networks, 2021. Original-date: 2020-10-14T22:45:36Z. Disponível em: <https://github.com/barefootnetworks/Open-Tofino/blob/9b70ee58656bc32bdd144dbd1139b35ae6b0b5f2/PUBLIC_Tofino-Native-Arch.pdf>. Acesso em: 20/07/2024.

LANNER ELECTRONICS. **What is Intel Tofino? An Easy Guide to its Benefits and Use Cases — whiteboxsolution.com**. Fremont, CA, USA, 2012. Disponível em: <https://www.whiteboxsolution.com/blog/what-is-intel-tofino-an-easy-guide-to-its-benefits-and-use-cases/>. Acesso em: 20/07/2024.

LI, F. et al. Longest prefix lookup in named data networking: How fast can it be? In: **2014 9th IEEE International Conference on Networking,**

Architecture, and Storage. Tianjin, China: IEEE, 2014. p. 186–190. Disponível em: <https://doi.org/10.1109/NAS.2014.36>. Acesso em: 20/07/2024.

LI, Z. et al. Hybrid wireless networks with FIB-based Named Data Networking. **EURASIP Journal on Wireless Communications and Networking**, Springer, v. 2017, n. 1, p. 7, mar. 2017. Disponível em: <https://doi.org/10.1186/s13638-017-0836-0>. Acesso em: 20/07/2024.

_____. Mapit: An enhanced pending interest table for ndn with mapping bloom filter. **IEEE Communications Letters**, IEEE, v. 18, n. 11, p. 1915–1918, 2014. Disponível em: <https://doi.org/10.1109/LCOMM.2014.2359191>. Acesso em: 20/07/2024.

_____. 5g with b-mafib based named data networking. **IEEE Access**, IEEE, v. 6, p. 30501–30507, 2018. Disponível em: <https://doi.org/10.1109/ACCESS.2018.2844294>. Acesso em: 20/07/2024.

_____. Packet forwarding in named data networking requirements and survey of solutions. **IEEE Communications Surveys & Tutorials**, IEEE, v. 21, n. 2, p. 1950–1987, 2019. Disponível em: <https://doi.org/10.1109/COMST.2018.2880444>. Acesso em: 20/07/2024.

LONG, X. et al. Pegasus: A high-speed ndn router with programmable switches and server clusters. In: **Proceedings of the 10th ACM Conference on Information-Centric Networking**. Reykjavik, Iceland: Association for Computing Machinery, 2023. (ACM ICN ’23), p. 12–18. Disponível em: <https://doi.org/10.1145/3623565.3623713>. Acesso em: 20/07/2024.

MA, X.; AFANASYEV, A.; ZHANG, L. A type-theoretic model on ndn-tlv encoding. In: **Proceedings of the 9th ACM Conference on Information-Centric Networking**. Osaka, Japan: Association for Computing Machinery, 2022. (ICN ’22), p. 91–102. Disponível em: <https://doi.org/10.1145/3517212.3558093>. Acesso em: 20/07/2024.

MADUREIRA, A. L. R. et al. Ndn fabric: Where the software-defined networking meets the content-centric model. **IEEE Transactions on Network and Service Management**, IEEE, v. 18, n. 1, p. 374–387, mar 2021. Disponível em: <https://doi.org/10.1109/TNSM.2020.3044038>. Acesso em: 20/07/2024.

MCKEOWN, N. et al. Openflow: enabling innovation in campus networks. **Computer Communication Review**, Association for Computing Machinery, v. 38, n. 2, p. 69–74, mar 2008. Disponível em: <https://doi.org/10.1145/1355734.1355746>. Acesso em: 20/07/2024.

MEDDEB, M. et al. Afirm: Adaptive forwarding based link recovery for mobility support in ndn/iot networks. **Future Generation Computer Systems**, Elsevier, v. 87, p. 351–363, 2018. Disponível em: <https://doi.org/10.1016/j.future.2018.04.087>. Acesso em: 20/07/2024.

MERKEL, D. Docker: lightweight linux containers for consistent development and deployment. **Linux journal**, v. 2014, n. 239, p. 2, 2014. Disponível em: <https://dl.acm.org/doi/fullHtml/10.5555/2600239.2600241>. Acesso em: 20/07/2024.

MIAO, R. et al. Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics. In: **Proceedings of the Conference of the ACM Special Interest Group on Data Communication**. Los Angeles, CA, USA: Association for Computing Machinery, 2017. (SIGCOMM '17), p. 15–28. Disponível em: <https://doi.org/10.1145/3098822.3098824>. Acesso em: 20/07/2024.

MIGUEL, R.; SIGNORELLO, S.; RAMOS, F. M. V. Named data networking with programmable switches. In: **2018 IEEE 26th International Conference on Network Protocols (ICNP)**. Cambridge, UK: IEEE, 2018. p. 400–405. Disponível em: <https://doi.org/10.1109/ICNP.2018.00055>. Acesso em: 20/07/2024.

MININET DEVELOPMENT TEAM. **An Instant Virtual Network on your Laptop (or other PC)**. Online, 2024. Disponível em: <https://mininet.org/>. Acesso em: 20/07/2024.

MURALIDHARAN, S.; ROY, A.; SAXENA, N. Mdp-iot: Mdp based interest forwarding for heterogeneous traffic in iot-ndn environment. **Future Generation Computer Systems**, v. 79, p. 892–908, 2018. Disponível em: <https://doi.org/10.1016/j.future.2017.08.058>. Acesso em: 20/07/2024.

NDN PROJECT TEAM. **NDN Specification**: Ndn packet format specification. Los Angeles, CA, USA, 2016. Disponível em: <https://named-data.net/publications/techreports/>. Acesso em: 20/07/2024.

______. **NFD Specification**: Nfd overview — named data networking forwarding daemon (nfd). Los Angeles, CA, USA, 2019. Disponível em: <https://docs.named-data.net/NFD/current/overview.html>. Acesso em: 20/07/2024.

NEWBERRY, E.; MA, X.; ZHANG, L. Yanfd: Yet another named data networking forwarding daemon. In: **Proceedings of the 8th ACM Conference on Information-Centric Networking**. Paris, France: Association for Computing Machinery, 2021. (ICN '21), p. 30–41. Disponível em: <https://doi.org/10.1145/3460417.3482969>. Acesso em: 20/07/2024.

OOKA, A.; ASAEDA, H. Ccnx router on fpga accelerator achieving predictable performance. In: **Proceedings of the 10th ACM Conference on Information-Centric Networking**. Reykjavik, Iceland: Association for Computing Machinery, 2023. (ACM ICN '23), p. 1–11. Disponível em: <https://doi.org/10.1145/3623565.3623710>. Acesso em: 20/07/2024.

OPEN NETWORKING FOUNDATION. **P4 Developer Day 2019**. Stanford, CA, USA, 2019. Disponível em: <https://opennetworking.org/p4-events/p4-developer-day-2019/>. Acesso em: 20/07/2024.

P4 LANGUAGE CONSORTIUM. **P4Runtime Specification**. Palo Alto, CA, USA, 2020. 113 p. Disponível em: <https://p4.org/p4-spec/p4runtime/v1.3.0/P4Runtime-Spec.html>. Acesso em: 20/07/2024.

______. **PNA Specification**: Portable nic architecture (pna). Palo Alto, CA, USA, 2022. 22 p. Disponível em: <https://p4.org/p4-spec/docs/PNA-v0.7.html>. Acesso em: 20/07/2024.

_____. **PSA Specification**: Portable switch architecture (psa). Palo Alto, CA, USA, 2022. 73 p. Disponível em: <https://p4.org/p4-spec/docs/PSA-v1.2.html>. Acesso em: 20/07/2024.

_____. **P416 Language Specification**. Palo Alto, CA, USA, 2023. 187 p. Disponível em: <https://p4.org/p4-spec/docs/P4-16-v1.2.4.html>. Acesso em: 20/07/2024.

PERINO, D. et al. Caesar: a content router for high-speed forwarding on content names. In: **Proceedings of the Tenth ACM/IEEE Symposium on Architectures for Networking and Communications Systems**. Los Angeles, CA, USA: Association for Computing Machinery, 2014. (ANCS '14), p. 137–148. Disponível em: <https://doi.org/10.1145/2658260.2658267>. Acesso em: 20/07/2024.

PFAFF, B. et al. The design and implementation of open vSwitch. In: **12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)**. Oakland, CA: USENIX Association, 2015. p. 117–130. Disponível em: <https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/pfaff>. Acesso em: 20/07/2024.

QUAN, W. et al. Scalable name lookup with adaptive prefix bloom filter for named data networking. **IEEE Communications Letters**, IEEE, v. 18, n. 1, p. 102–105, 2014. Disponível em: <https://doi.org/10.1109/LCOMM.2013.112413.132231>. Acesso em: 20/07/2024.

REXFORD, J.; DOVROLIS, C. Future internet architecture: Clean-slate versus evolutionary research. **Communications of the ACM**, Association for Computing Machinery, v. 53, n. 9, p. 36–40, sep 2010. Disponível em: <https://doi.org/10.1145/1810891.1810906>. Acesso em: 20/07/2024.

ROSA, E.; CUNHA, I.; SILVA, F. A simple approach to verify and debug data plane programs. In: **Anais do XIII Workshop de Pesquisa Experimental da Internet do Futuro**. Porto Alegre, RS, Brasil: SBC, 2022. (WPEIF, 22), p. 47–52. Disponível em: <https://sol.sbc.org.br/index.php/wpeif/article/view/21492>. Acesso em: 20/07/2024.

ROSA, E.; GUARDIEIRO, P. An uplink scheduling and cac algorithms for ieee 802.16 networks. In: **Proceedings of The International Workshop on Telecomunications (IWT)**. Santa Rita do Sapucaí, MG: Inatel, 2011. Disponível em: <https://inatel.br/iwt/documents/documents2011/ProceedingsIWT2011.pdf>. Acesso em: 20/07/2024.

ROSA, E.; SILVA, F. Enabling native coexistence between icn and tcp/ip architectures over the same domain. In: **Anais do XI Workshop de Pesquisa Experimental da Internet do Futuro**. Porto Alegre, RS, Brasil: SBC, 2020. p. 13–19. Disponível em: <https://sol.sbc.org.br/index.php/wpeif/article/view/12469>. Acesso em: 20/07/2024.

ROSA, E. C.; SILVA, F. d. O. A hash-free method for fib and lnpm in icn programmable data planes. In: **2022 International Conference on Information Networking (ICOIN)**. Jeju-si, Korea, Republic of: IEEE, 2022. p. 186–191. Disponível em: <https://doi.org/10.1109/ICOIN53446.2022.9687201>. Acesso em: 20/07/2024.

ROSA, E. C.; SILVA, F. de O. A review on recent ndn fib implementations for high-speed switches. In: BAROLLI, L.; HUSSAIN, F.; ENOKIDO, T. (Ed.). **Advanced**

Information Networking and Applications. Cham, Switzerland: Springer Nature, 2022. (AINA ’22), p. 288–300. Disponível em: <https://doi.org/10.1007/978-3-030-99619-2_28>. Acesso em: 20/07/2024.

SALSANO, S. et al. Information centric networking over sdn and openflow: Architectural aspects and experiments on the ofelia testbed. **Computer Networks**, Elsevier, v. 57, n. 16, p. 3207–3221, 2013. Disponível em: <https://doi.org/10.1016/j.comnet.2013.07.031>. Acesso em: 20/07/2024.

SAXENA, D. et al. Scalable, high-speed on-chip-based ndn name forwarding using fpga. In: **Proceedings of the 20th International Conference on Distributed Computing and Networking**. Bangalore, India: Association for Computing Machinery, 2019. (ICDCN ’19), p. 81–89. Disponível em: <https://doi.org/10.1145/3288599.3288613>. Acesso em: 20/07/2024.

SHI, J.; PESAVENTO, D.; BENMOHAMED, L. Ndn-dpdk: Ndn forwarding at 100 gbps on commodity hardware. In: **Proceedings of the 7th ACM Conference on Information-Centric Networking**. Virtual Event, Canada: Association for Computing Machinery, 2020. (ICN ’20), p. 30–40. Disponível em: <https://doi.org/10.1145/3405656.3418715>. Acesso em: 20/07/2024.

SIGNORELLO, S. et al. Ndn.p4: Programming information-centric data-planes. In: **2016 IEEE NetSoft Conference and Workshops (NetSoft)**. Seoul, Korea (South): IEEE, 2016. p. 384–389. Disponível em: <https://doi.org/10.1109/NETSOFT.2016.7502472>. Acesso em: 20/07/2024.

SILVA, T. B. da et al. Toward next-generation and service-defined networks: A novagenesis control agent for future internet exchange point. **IEEE Network**, IEEE, v. 36, n. 3, p. 74–81, 2022. Disponível em: <https://doi.org/10.1109/MNET.008.2100555>. Acesso em: 20/07/2024.

SIVARAMAN, A. et al. Packet transactions: High-level programming for line-rate switches. In: **Proceedings of the 2016 ACM SIGCOMM Conference**. Florianopolis, Brazil: Association for Computing Machinery, 2016. (SIGCOMM ’16), p. 15–28. Disponível em: <https://doi.org/10.1145/2934872.2934900>. Acesso em: 20/07/2024.

SONG, T. et al. Scalable name-based packet forwarding: From millions to billions. In: **Proceedings of the 2nd ACM Conference on Information-Centric Networking**. San Francisco, CA, USA: Association for Computing Machinery, 2015. (ACM-ICN ’15), p. 19–28. Disponível em: <https://doi.org/10.1145/2810156.2810166>. Acesso em: 20/07/2024.

TAKEMASA, J.; KOIZUMI, Y.; HASEGAWA, T. Vision: toward 10 tbps ndn forwarding with billion prefixes by programmable switches. In: **Proceedings of the 8th ACM Conference on Information-Centric Networking**. Paris, France: Association for Computing Machinery, 2021. (ICN ’21), p. 13–19. Disponível em: <https://doi.org/10.1145/3460417.3482973>. Acesso em: 20/07/2024.

TAKI, S. U.; MASTORAKIS, S. An ndn-enabled fog radio access network architecture with distributed in-network caching. In: **ICC 2023 - IEEE International Conference on Communications**. Rome, Italy: IEEE, 2023. p. 5291–5296. Disponível em: <https://doi.org/10.1109/ICC45041.2023.10279580>. Acesso em: 20/07/2024.

TANENBAUM, A.; FEAMSTER, N. **Computer Networks**. 6. ed. London, UK: Pearson Education, 2019. 960 p.

TARIQ, A.; REHMAN, R. A.; KIM, B.-S. Forwarding strategies in ndn-based wireless networks: A survey. **IEEE Communications Surveys & Tutorials**, v. 22, n. 1, p. 68–95, 2020. Disponível em: <https://doi.org/10.1109/COMST.2019.2935795>. Acesso em: 20/07/2024.

TEHRANI, P. F. et al. Sok: Public key and namespace management in ndn. In: **Proceedings of the 9th ACM Conference on Information-Centric Networking**. Osaka, Japan: Association for Computing Machinery, 2022. (ICN '22), p. 67–79. Disponível em: <https://doi.org/10.1145/3517212.3558085>. Acesso em: 20/07/2024.

TROSSEN, D. et al. Ip over icn - the better ip? In: **2015 European Conference on Networks and Communications (EuCNC)**. Paris, France: IEEE, 2015. p. 413–417. Disponível em: <https://doi.org/10.1109/EuCNC.2015.7194109>. Acesso em: 20/07/2024.

WANG, H. et al. P4FPGA: A Rapid Prototyping Framework for P4. In: **Proceedings of the Symposium on SDN Research**. Santa Clara CA USA: ACM, 2017. p. 122–135. Disponível em: <https://dl.acm.org/doi/10.1145/3050220.3050234>. Acesso em: 20/07/2024.

WANG, Y. et al. Parallel name lookup for named data networking. In: **2011 IEEE Global Telecommunications Conference - GLOBECOM 2011**. Houston, TX, USA: IEEE, 2011. p. 1–5. Disponível em: <https://doi.org/10.1109/GLOCOM.2011.6134161>. Acesso em: 20/07/2024.

______. Scalable name lookup in ndn using effective name component encoding. In: **2012 IEEE 32nd International Conference on Distributed Computing Systems**. Macau, China: IEEE, 2012. p. 688–697. Disponível em: <https://doi.org/10.1109/ICDCS.2012.35>. Acesso em: 20/07/2024.

WHITE, G.; RUTZ, G. **Content Delivery with Content-Centric Networking**. Louisville, CO, USA, 2016. 26 p. Disponível em: <https://www.cablelabs.com/wp-content/uploads/2016/02/Content-Delivery-with-Content-Centric-Networking-Feb-2016.pdf>. Acesso em: 20/07/2024.

XIE, M.; WIDJAJA, I.; WANG, H. Enhancing cache robustness for content-centric networking. In: IEEE. **2012 Proceedings IEEE INFOCOM**. Orlando, FL, USA, 2012. p. 2426–2434. Disponível em: <https://doi.org/10.1109/INFCOM.2012.6195632>. Acesso em: 20/07/2024.

YANG, N.; CHEN, K.; LIU, Y. Towards efficient ndn framework for connected vehicle applications. **IEEE Access**, v. 8, p. 60850–60866, 2020. Disponível em: <https://doi.org/10.1109/ACCESS.2020.2981928>. Acesso em: 20/07/2024.

YI, C. et al. On the role of routing in named data networking. In: **Proceedings of the 1st ACM Conference on Information-Centric Networking**. Paris, France: Association for Computing Machinery, 2014. (ACM ICN '14), p. 27–36. Disponível em: <https://doi.org/10.1145/2660129.2660140>. Acesso em: 20/07/2024.

______. A case for stateful forwarding plane. **Computer Communications**, Elsevier, v. 36, n. 7, p. 779–791, 2013. Disponível em: <https://doi.org/10.1016/j.comcom.2013.01.005>. Acesso em: 20/07/2024.

YU, W.; PAO, D. Hardware accelerator for FIB lookup in named data networking. **Microprocessors and Microsystems**, Elsevier BV, v. 71, nov. 2019. Disponível em: <https://doi.org/10.1016/j.micpro.2019.102877>. Acesso em: 20/07/2024.

YUAN, H. **Data structures and algorithms for scalable NDN forwarding**. Tese (Doutorado) — School of Engineering & Applied Science, Washington University, Saint Louis, MO, USA, 2015. Disponível em: <https://doi.org/10.7936/K7JQ0Z92>. Acesso em: 20/07/2024.

ZHANG, L. et al. Named data networking. **Computer Communication Review**, Association for Computing Machinery, v. 44, n. 3, p. 66–73, jul 2014. Disponível em: <https://doi.org/10.1145/2656877.2656887>. Acesso em: 20/07/2024.