

UNIVERSIDADE DE PASSO FUNDO
Programa de Pós-Graduação em
Computação Aplicada

Dissertação de Mestrado

**VISUAL-INT: COLETA,
PROCESSAMENTO E
VISUALIZAÇÃO GRANULAR DE
METADADOS INT PARA GERÊNCIA
SDN**

ALISSON BORGES ZANETTI



UNIVERSIDADE DE PASSO FUNDO
INSTITUTO DE CIÊNCIAS EXATAS E GEOCIÊNCIAS
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO APLICADA

**VISUAL-INT: COLETA, PROCESSAMENTO E
VISUALIZAÇÃO GRANULAR DE
METADADOS INT PARA GERÊNCIA SDN**

Alisson Borges Zanetti

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre em Computação Aplicada na Universidade de Passo Fundo.

Orientador: Prof. Dr. Marco Antônio Sandini Trentin

Coorientador: Prof. Dr. Ricardo de Oliveira Schmidt

Passo Fundo

2020

CIP – Catalogação na Publicação

Z28v Zanetti, Alisson Borges
Visual-INT : coleta, processamento e visualização granular
de metadados INT para gerência SDN / Alisson Borges
Zanetti. – 2020.
124 f. : il. ; 30 cm.

Orientador: Prof. Dr. Marco Antônio Sandini Trentin.
Coorientador: Prof. Dr. Ricardo de Oliveira Schmidt.
Dissertação (Mestre em Computação Aplicada) –
Universidade de Passo Fundo, 2020.


1. Redes de computadores - Gerência. 2. Telemetria.
3. Processamento eletrônico de dados. I. Trentin, Marco
Antônio Sandini, orientador. II. Schmidt, Ricardo de Oliveira,
coorientador. III. Título.

CDU: 004.72


ATA DE DEFESA DO TRABALHO DE CONCLUSÃO DE CURSO DO ACADÊMICO

ALISSON BORGES ZANETTI

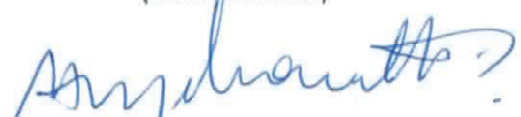
Aos vinte dias do mês de dezembro do ano de dois mil e dezenove às 14 horas, realizou-se, no Instituto de Ciências Exatas e Geociências, prédio B5, da Universidade de Passo Fundo (UPF), a sessão pública de defesa do Trabalho de Conclusão de Curso: "Visual-INT: coleta, processamento e visualização granular de metadados int para gerência SDN", de autoria de Alisson Borges Zanetti, acadêmico do Curso de Mestrado em Computação Aplicada do Programa de Pós-Graduação em Computação Aplicada – PPGCA. Segundo as informações prestadas pelo Conselho de Pós-Graduação e constantes nos arquivos da Secretaria do PPGCA, o aluno preencheu os requisitos necessários para submeter seu trabalho à avaliação. A banca examinadora foi composta pelos doutores Marco Antônio Sandini Trentin, Ricardo de Oliveira Schmidt, Marcelo Trindade Rebonatto e Lisandro Zambenedetti Granville. Concluídos os trabalhos de apresentação e arguição, a banca examinadora considerou o candidato APROVADO. Foi concedido o prazo de até quarenta e cinco (45) dias, conforme Regimento do PPGCA, para o acadêmico apresentar ao Conselho de Pós-Graduação o trabalho em sua redação definitiva, a fim de que sejam feitos os encaminhamentos necessários à emissão do Diploma de Mestre em Computação Aplicada. Para constar, foi lavrada a presente ata, que vai assinada pelos membros da banca examinadora e pela Coordenação do PPGCA.



Prof. Dr. Marco Antônio Sandini Trentin – UPF
Presidente da Banca Examinadora
(Orientador)




Prof. Dr. Ricardo de Oliveira Schmidt – UPF
(Coorientador)



Prof. Dr. Marcelo Trindade Rebonatto – UPF
(Avaliador Interno)



Prof. Dr. Lisandro Zambenedetti Granville – UFRGS
(Avaliador Externo)



Prof. Dr. Rafael Rieder
Coordenador do PPGCA

AGRADECIMENTOS

Agradeço primeiramente a minha família, em especial a minha companheira Kelli, pela paciência dispensada durante todo este período de estudos, onde muitas vezes tive que me abster do convívio em prol deste projeto. Agradeço aos meus pais pelo dom da vida, como também pela leveza de suas palavras que certamente me incentivaram nesta caminhada. Agradeço minha avó Ada Maria, matriarca da família, que com seu apoio me proporcionou a oportunidade de dar o primeiro passo em direção a conclusão de mais esta etapa. Também agradeço a meus irmãos e amigos pelo convívio fraterno e compreensão de minha ausência.

Agradeço à Universidade Internacional da Flórida, em especial ao Jeronimo Bezerra, por todo o apoio técnico, ajuda e dedicação prestada na elaboração deste trabalho. Certamente, sem esta parceria nada do que fora realizado seria possível. Agradeço ao Instituto Federal Catarinense campus Luzerna, do qual estive afastado para a realização deste mestrado, bem como aos colegas de trabalho pela divisão de esforços e tarefas enquanto estive fora. Agradeço à Universidade de Passo Fundo pela oportunidade de crescimento pessoal e profissional durante este período, bem como ao PPGCA (Programa de Pós-Graduação em Computação Aplicada) em especial aos professores Dr. Marco Antônio Sandini Trentin e Dr. Ricardo de Oliveira Schmidt pelas correções, troca de experiências e lapidações de pensamento.

“Quanto mais nos elevamos, menores parecemos aos olhos daqueles que não sabem voar.”

(Friedrich Nietzsche)

VISUAL-INT: COLETA, PROCESSAMENTO E VISUALIZAÇÃO GRANULAR DE METADADOS INT PARA GERÊNCIA SDN

RESUMO

Em redes de computadores, o monitoramento é um importante aliado na busca do gerenciamento de ativos e recursos. A evolução tecnológica e o agigantamento da escala das redes trouxeram novos desafios e possibilidades para este tipo de tarefa. Aplicada ao modelo de redes definidas por software (*Software-Defined Networking*), a telemetria de rede em banda (*In-band Network Telemetry*) visa o monitoramento de equipamentos programáveis por intermédio da captura, encapsulamento e envio, via o tráfego passante, de metadados que qualificam seus estados internos. Todavia, a telemetria de rede em banda não conceitua a forma como estes metadados serão tratados após sua extração, provocando o surgimento de algumas iniciativas de coleta e processamento. A maioria destas, porém, não possui emprego prático imediato, estando ambientadas em emuladores/simuladores, além de possuírem modos de visualização generalistas e pouco dinâmicos. Dado o exposto e tendo como caso de uso a implementação da telemetria na rede real definida por software AmLight-Exp, este trabalho apresenta o Visual-INT, um coletor versado para o hardware que abrange a apuração, a análise, a persistência e, por consequência, a visualização dos metadados de telemetria. O Visual-INT promove o entendimento de metadados suportados, como a latência por dispositivo e o caminho do fluxo, úteis no monitoramento de atrasos e identificação de pontos de falha. Para isto, esta pesquisa emprega técnicas de programação de processamento de pacotes, como o *extended Berkeley Packet Filter* e o *eXpress Data Path*, em uma SmartNIC Netronome que captura o tráfego de telemetria proveniente de outros dois dispositivos similares operados como *switches*. Principal proposta do Visual-INT, o uso das referidas técnicas em modo *offload* (Descarregado na SmartNIC) provou-se, mediante a validação e comparação com o modo nativo (*driver do kernel*), superior em níveis de computação e assertividade. Implementando tecnologias equivalentes às aplicadas no caso de uso, esta proposta pretende aprimorar, mediante a execução no hardware, as técnicas de coleta INT, ao passo em que fornece uma solução aplicável a ambientes reais.

Palavras-Chave: Redes de Computadores, Processamento de Pacotes, In-band Network Telemetry, extended Berkeley Packet Filter, eXpress Data Path.

VISUAL-INT: COLLECTION, PROCESSING, AND GRANULAR VISUALIZATION FOR SDN MANAGEMENT

ABSTRACT

In computer networks, monitoring is an important tool to meet resource management. Together, the technological evolution and the growth of the networks make new possibilities for this type of task. Applied to recent technological environments like Software-defined Networking, In-band Network Telemetry aims to watch programmable devices by catching, encapsulating, and sending their metadata through the network flow. However, the use of data after its extraction is not defined by telemetry. Because of it, a lot of solutions addressed to this gap started emerging. Nevertheless, most of them are in simulated environments, do not have an immediate practical use, also bear on generic views from the collected data. To overcome that scenario, this work presents Visual-INT, a hardware-versed collector that captures, analyzes, stores, and response telemetry metadata. To do this, Visual-INT employs packet processing programming techniques, such as extended Berkeley Packet Filter and eXpress Data Path, in a Netronome card that catches telemetry traffic generated by other similar cards. Visual-INT promotes the understanding of the metadata now available on the cards, such as device latency and flow path, useful in monitoring delays and identifying points of failure as well as achieving higher hardware computation levels. The main propose of this work is to improve the telemetry gather techniques by offloading collector programs into SmartNIC, along with to offer a broad solution that is suitable for real environments.

Keywords: Computer Network, Packet Processing, In-band Network Telemetry, extended Berkeley Packet Filter, eXpress Data Path.

LISTA DE FIGURAS

| | | |
|----|--|----|
| 1 | Diferenças entre rede tradicional e SDN | 29 |
| 2 | Comutação com Switch OpenFlow | 30 |
| 3 | Escrita, extração e reportação dos metadados INT | 32 |
| 4 | Diagrama de funcionamento do eBPF | 34 |
| 5 | Comparativo entre taxas de processamento na pilha Linux | 35 |
| 6 | Caminho normal em relação ao XDP | 36 |
| 7 | Diagrama de funcionamento do projeto AmLight-INT | 40 |
| 8 | Esquema de operação Visual-INT | 41 |
| 9 | SmartNIC Netronome Agilio CX | 43 |
| 10 | Comparação de processamentos em Mpps entre arquiteturas | 43 |
| 11 | Topologia da rede P4 complementar ao Visual-INT | 44 |
| 12 | Tráfego de telemetria até a chegada ao coletor do Visual-INT | 45 |
| 13 | Formato do <i>report</i> aplicado ao Visual-INT | 45 |
| 14 | Formato do cabeçalho INT aplicado ao Visual-INT | 46 |
| 15 | Arquitetura interna do Visual-INT | 47 |
| 16 | Divisão dos programas no coletor do Visual-INT | 48 |
| 17 | Arquitetura NFP para <i>offload</i> de objetos eBPF | 49 |
| 18 | Modelo de programação eBPF na SmartNIC | 49 |
| 19 | <i>Parser</i> Externo do Coletor do Visual-INT | 50 |
| 20 | <i>Parser</i> Interno do Coletor do Visual-INT | 51 |
| 21 | <i>Parser</i> INT do Coletor do Visual-INT | 52 |
| 22 | <i>Struct</i> enviada para o <i>userspace</i> pelo <i>perf_event</i> | 52 |
| 23 | Modelo de visualização clássico | 55 |
| 24 | Modelo de visualização clássico | 55 |
| 25 | Sankey atestando o caminho do fluxo e a latência por <i>hop</i> | 56 |
| 26 | <i>Plotly.js</i> contendo a latência e sua mediana para um dispositivo | 56 |
| 27 | FDP acerca do uso de CPU nos modos <i>kernel</i> (A) e <i>offload</i> (B) para o teste somente com INT | 61 |
| 28 | FDA acerca do uso de CPU nos modos <i>kernel</i> (A) e <i>offload</i> (B) para o teste somente com INT | 62 |
| 29 | Comparativo entre médias de CPU e erro padrão (modos <i>kernel</i> e <i>offload</i>) para o teste somente com INT | 62 |

| | | |
|----|---|----|
| 30 | FDA acerca do uso de CPU nos modos <i>kernel</i> (A) e <i>offload</i> (B) para o teste INT com <i>iPerf</i> | 64 |
| 31 | Comparativo entre médias de CPU e erro padrão (modos <i>kernel</i> e <i>offload</i>) para o teste INT com <i>iPerf</i> | 64 |
| 32 | FDP acerca do uso de CPU nos modos <i>kernel</i> (A) e <i>offload</i> (B) para o teste INT com bloqueio total | 66 |
| 33 | FDA acerca do uso de CPU nos modos <i>kernel</i> (A) e <i>offload</i> (B) para o teste INT com bloqueio total | 67 |
| 34 | Comparativo entre médias de CPU e erro padrão (modos <i>kernel</i> e <i>offload</i> para o teste INT com bloqueio total | 67 |
| 35 | <i>Boxplot</i> estratificado entre as médias de CPU dos testes INT, INT+ <i>iPerf</i> e INT+ <i>block</i> | 69 |
| 36 | Média de uso de CPU e desvio padrão entre os testes INT, INT+ <i>iPerf</i> e INT+ <i>block</i> | 69 |
| 37 | <i>Boxplot</i> estratificado entre as médias de pacotes <i>report</i> dos testes INT, INT+ <i>iPerf</i> e INT+ <i>block</i> | 70 |
| 38 | Média de pacotes <i>reports</i> e desvio padrão entre os testes INT, INT+ <i>iPerf</i> e INT+ <i>block</i> | 71 |
| 39 | Fluxograma do teste com INT simulado | 72 |
| 40 | Uso de CPU mediante a quantidade de fluxos | 73 |
| 41 | Entradas persistidas no <i>userspace</i> mediante a quantidade de fluxos | 74 |
| 42 | Uso de CPU mediante a quantidade de <i>switches</i> | 75 |
| 43 | <i>Bytes</i> processados mediante a quantidade de <i>switches</i> | 76 |
| 44 | Uso de CPU mediante a quantidade de metadados INT | 77 |
| 45 | <i>Bytes</i> processados mediante a quantidade de metadados INT | 77 |
| 46 | Modelo de regressão linear para o uso de CPU para o método <i>kernel</i> nos testes simulados | 78 |
| 47 | Modelo de regressão linear para o uso de CPU para o método <i>offload</i> nos testes simulados | 79 |
| 48 | Modelo de <i>measurement</i> no InfluxDB contendo o caminho de fluxo | 80 |
| 49 | Médias da quantidade de entradas no banco de dados e erro padrão (modos <i>kernel</i> e <i>offload</i>) para detecção de eventos | 82 |
| 50 | Comparativo entre médias de CPU e erro padrão (modos <i>kernel</i> e <i>offload</i>) para detecção de eventos | 83 |
| 51 | <i>Bytes</i> processados mediante a quantidade de metadados INT | 84 |
| 52 | Interface do gráfico Sankey no utilitário de visualização | 85 |

53 Interface do gráfico *plotly.js* no utilitário de visualização 85

LISTA DE TABELAS

| | | |
|----|--|----|
| 1 | Resumo histórico das redes programáveis | 28 |
| 2 | Plataformas SDN que empregam o OpenFlow | 30 |
| 3 | Comparação entre os trabalhos relacionados e o Visual-INT | 37 |
| 4 | Lista das métricas INT adotadas | 51 |
| 5 | Relação entre as métricas usadas e as medições praticáveis | 53 |
| 6 | Panorama da Era (<i>Epoch</i>) UNIX quanto à granularidade) UNIX | 54 |
| 7 | Médias obtidas com IC de 95% para o teste somente com INT | 59 |
| 8 | Médias obtidas com IC de 95% para o teste INT com <i>iPerf</i> | 63 |
| 9 | Médias obtidas com IC de 95% para o teste INT com bloqueio total | 65 |
| 10 | Médias obtidas com IC de 95% para detecção de eventos | 82 |

LISTA DE SIGLAS

ACL – Access Control List
API – Application Programming Interface
ASIC – Application Specific Integrated Circuit
ATM – Asynchronous Transfer Mode
BGP – Border Gateway Protocol
CPU – Central Processing Unit
FPGA – Field Programmable Gate Array
ICMP – Internet Control Message Protocol
JIT – Just In Time
LSF – Linux Socket Filter
MPLS – Multi Protocol Label Switching
NIC – Network Interface Controller
NPU – Network Processor
PCE – Path Computation Element
PCIE – Peripheral Component Interconnect Express
RCP – Remote Control Protocol
RFC – Request for Comments
SNMP – Simple Network Management Protocol
SO – Sistema Operacional
TCP – Transmission Control Protocol

SUMÁRIO

| | | |
|----------|---|-----------|
| 1 | INTRODUÇÃO | 23 |
| 2 | REVISÃO DE LITERATURA | 27 |
| 2.1 | SDN | 27 |
| 2.1.1 | Principais Conceitos SDN | 28 |
| 2.1.1.1 | OpenFlow | 29 |
| 2.1.1.2 | Switch OpenFlow | 30 |
| 2.2 | PROGRAMABILIDADE NO PLANO DE DADOS | 31 |
| 2.2.1 | P4 | 31 |
| 2.2.2 | INT | 32 |
| 2.3 | CONCEITOS DO VISUAL-INT | 33 |
| 2.3.1 | eBPF | 33 |
| 2.3.2 | XDP (<i>eXpress Data Path</i>) | 34 |
| 2.4 | TRABALHOS RELACIONADOS | 36 |
| 3 | PROCEDIMENTOS METODOLÓGICOS | 39 |
| 3.1 | AMLIGHT-INT | 39 |
| 3.1.1 | SDN e OpenFlow no AmLight-INT | 40 |
| 3.2 | VISUAL-INT | 40 |
| 3.2.1 | SDN e OpenFlow no Visual-INT | 42 |
| 3.2.2 | Arquitetura Externa | 42 |
| 3.2.2.1 | SmartNICs e Máquinas Físicas | 42 |
| 3.2.2.2 | Switches P4 e Topologia | 44 |
| 3.2.2.3 | Cabeçalhos de Telemetria | 44 |
| 3.2.3 | Arquitetura Interna | 47 |
| 3.2.3.1 | Coletor INT | 47 |
| 3.2.3.2 | InfluxDB | 53 |
| 3.2.3.3 | Visualizador INT | 54 |
| 4 | APRESENTAÇÃO E DISCUSSÃO DE RESULTADOS | 57 |
| 4.1 | VALIDAÇÃO DO COLETOR | 57 |
| 4.1.1 | Testes com tráfego obtido do INT real | 58 |

| | | |
|----------|---|------------|
| 4.1.1.1 | Teste 1: Processamento de tráfego INT/P4 | 58 |
| 4.1.1.2 | Teste 2: Processamento de tráfego INT/P4 com concorrência | 63 |
| 4.1.1.3 | Teste 3: Processamento INT e bloqueio total de tráfego | 65 |
| 4.1.1.4 | Avaliação dos testes com tráfego obtido do INT real | 68 |
| 4.1.2 | Testes com tráfego obtido do INT simulado | 71 |
| 4.1.2.1 | Saturamento de fluxos de rede no Visual-INT | 72 |
| 4.1.2.2 | Saturamento de <i>switches</i> no Visual-INT | 74 |
| 4.1.2.3 | Saturamento de métricas INT no Visual-INT | 75 |
| 4.1.2.4 | Avaliação dos testes com tráfego obtido do INT simulado | 78 |
| 4.2 | VALIDAÇÃO DO ARMAZENAMENTO DOS DADOS EM SÉRIES TEMPO- RAIS | 80 |
| 4.2.1 | Abordagem ingênua | 80 |
| 4.2.2 | Detecção de Eventos | 81 |
| 4.3 | VALIDAÇÃO DO UTILITÁRIO DE VISUALIZAÇÃO | 83 |
| 4.4 | DISCUSSÃO DOS RESULTADOS | 86 |
| 5 | CONCLUSÃO | 89 |
| 5.1 | CONSIDERAÇÕES FINAIS | 89 |
| 5.2 | RESSALVAS E LIMITAÇÕES | 89 |
| 5.3 | TRABALHOS FUTUROS | 90 |
| | REFERÊNCIAS | 93 |
| | ANEXO A – Saída do terminal para o caminho de fluxo | 99 |
| | ANEXO B – Utilitário de visualização | 101 |
| | ANEXO C – Analisador do Coletor (ebpf_prog_user.c) | 107 |
| | ANEXO D – Coletor (ebpf_prog_kern.c) | 119 |

1. INTRODUÇÃO

Apesar das mudanças significativas no uso das redes de computadores, das quais estimativas recentes atestam para um crescimento mundial do tráfego de Internet acima dos 3,2 ZB¹ até 2021 (representando uma ampliação de 24% desde 2016) [1], aliada à transformação dos hábitos e das aplicações, cujos padrões de exigência de desempenho e eficiência encontram-se em escala crescente, o monitoramento dos ativos de rede ainda abarca conceitos tradicionais das redes de outrora. Técnicas com uso do SNMP, ICMP, *traceroute* e outros, afora datados, configuram-se como mecanismos reativos de resposta a falhas, sendo insatisfatórios para o cenário que se apresenta.

A fim de solucionar limitações e vícios de modelos antigos, já não suficientes aos requisitos atuais [2], bem como propiciar a criação de diferentes paradigmas, se faz necessária a proposição de novos conceitos dentro da temática de redes de computadores. Neste cenário, a concepção de redes definidas por software, também tratadas como SDN (*Software-Defined Networking*), se constitui um exemplo bem-sucedido de tecnologia emergente de redes de próxima geração. Por ser concebida em meio às novas demandas, o SDN surge como um arquétipo de rede bem organizado, adaptado às rápidas mudanças exigidas e que busca simplificar o controle de tráfego ao menor custo operacional possível [3] [4].

Desde 2008, o SDN reorienta o conceito de redes programáveis, cuja tratativa inicial remonta à década de 80 [5]. Por conta do estabelecimento de programas customizáveis que intermedeiam (e ditam) as ações do hardware, o paradigma SDN popularizou-se nos últimos anos ao provisionar a separação das camadas de controle (gerência) e dados (hardware) e, paralelamente, a centralização do gerenciamento dos ativos e a visualização global do estado da rede [3]. A contar de 2014, com o surgimento de sua mais recente colaboração, a saber a linguagem P4 (*Programming Protocol-Independent Packet Processors*), o modelo SDN evoluiu, deslocando a programabilidade diretamente para o plano de dados (hardware), admitindo-se assim a execução, ao mais baixo nível, de tarefas outrora atreladas unicamente aos controles superiores (gerência).

Dentre essas tarefas, destaca-se o monitoramento por meio da telemetria de rede em banda, designada INT (*In-band Network Telemetry*). Ensaiado em 2015, o *framework* INT, por meio da linguagem P4 e de *chips* programáveis, permite a supervisão do estado da rede em tempo real ao possibilitar que pacotes inquiram dados adicionais sobre a condição interna de cada dispositivo (como tamanho da fila, utilização de *link*, dentre outros). As respostas, encabeçadas no corpo do próprio pacote, seguem o fluxo até seu destino, enquanto os dados de cada equipamento são extraídos e analisados por estruturas à parte [6]. De avaliações de desempenho à prevenção de falhas, a observação de dados internos dos

¹1 Zebibyte (ZB) corresponde a 1.024 EB (Exbibyte), sendo 1 EB igual a um bilhão de GB (Gigabyte).

ativos aprofunda o conhecimento e a visibilidade do estado da rede, suplantando métodos antigos de monitoramento.

Como caso de uso, esta pesquisa utiliza-se das diretrizes do projeto parceiro AmLight-INT (*America's Lightpaths In-band Telemetry*), em desenvolvimento pela FIU (Universidade Internacional da Flórida). Aplicado à rede SDN de apoio científico AmLight-ExP (*AmLight Express e Protect*), que interliga a América do Norte aos principais pontos de agregação de redes de computadores na América do Sul (Brasil, Chile e Panamá), o AmLight-INT busca o feito inédito de implementação INT em uma rede de escala continental. A rede AmLight-ExP é fomentadora de várias propostas de investigação científica via interligação física de fluxo de dados [7], como o empreendimento LSST (*Large Synoptic Survey Telescope*), um potente telescópio localizado no Chile que transferirá suas imagens para estações de processamento distribuídas nos Estados Unidos [8].

Contando com a maior lente já construída pelo homem (aproximadamente do tamanho de um carro popular), o LSST será capaz de mapear todo o céu visível a partir de sua área de construção, em Cerro Pachón. O equipamento fornecerá mapas tridimensionais inéditos de massa no Universo, ademais das tradicionais imagens de galáxias luminosas. Não menos importante, também disponibilizará um censo abrangente do nosso sistema solar, incluindo alerta de asteroides potencialmente perigosos [8]. Com cerca de 12,8 GB de dados gerados a cada 30 segundos, a transmissão de informações do telescópio chileno para servidores norte-americanos deverá ocorrer dentro de um intervalo de cinco segundos, exatamente o tempo de fechamento do obturador de sua câmera. Paralelo às imagens, catálogos de recursos e comandos de controle deverão utilizar-se da mesma estrutura, sendo transportados em sincronia [7].

O principal desafio de rede enfrentado pelo LSST é o alto atraso de propagação em seu caminho de ponta à ponta. Atualmente, os tempos de *Round Trip Time* (ida e volta de pacotes) do Chile ao Illinois (nos Estados Unidos) são considerados elevados para os padrões da aplicação (variando de 144 a 280 ms), podendo ser afetado pelos mecanismos de controle e congestionamento do TCP. Somada à perda de pacotes (estimada em 0,001%), a equação de Mathis [9] (que descreve o impacto da perda de pacotes na taxa de transferência) antevê um rendimento teórico máximo possível inferior ao recomendado, representando 377 MB de informação a menos por dia (quase 12 GB por mês). Conjuntamente, estes fatores são responsáveis pelo aumento da expectativa do tempo de transmissão em até dez segundos, extrapolando o limite estipulado (de cinco segundos) [7].

As dificuldades expostas causam impacto na consistência e assertividade dos dados, dada a incompletude nas informações processadas. Tratando-se de uma aplicação de caráter científico, a busca pela exatidão é um dos primeiros limiares a ser observado na estruturação do LSST. Literariamente, alguns dos motivos de atraso em redes são bem conhecidos [10]. A perda de pacotes pode ser resultado de componentes de rede danificados, de gargalos nos enlaces de compartilhamento da rede, de *traffic burst* de pacotes,

dentre outros. O modo de solucionar estes problemas também é habitual aos administradores. Por exemplo, em *links* de baixa capacidade, uma condução cabível é conceder maior amplitude de banda. Apesar disso, localizar os pontos de falha em redes complexas acaba por dificultar as tarefas de *troubleshooting* e isolamento imediatos [7].

Por intermédio do monitoramento INT, pretende-se dirimir o impasse na tomada de decisões quanto à gerência da rede. Ao transmitir informações acerca do estado de cada dispositivo, pode-se determinar, em tempo real, os principais pontos de falha presentes na estrutura. Todavia, a maioria das implementações INT tratam do emprego da telemetria em redes LAN (*datacenters*), quando não, ambientadas em cenários virtualizados. O AmLight-INT, por sua vez, pela demanda LSST, expõe o uso do INT em redes reais de grande escala (WAN).

Outrossim, apesar de situar o modo como a escrita, a coleta e a extração dos metadados deva ocorrer dentro do contexto P4, o *framework* INT não se ocupa da forma como essas informações serão processadas ou apreciadas após a reportação. Isso advém do fato do INT ser uma ferramenta de monitoramento de propósito geral num ambiente programável independente de alvo ², cuja finalidade é definida pelo usuário. Características intrínsecas de cada implementação, como a arquitetura dos *chips*, configurações, topologias, disponibilidade de recursos e demais peculiaridades podem ressignificar as instâncias de cada metadado coletado [11].

Adido a isto está a escassez, na literatura acadêmica, de ferramentas de visualização propriamente direcionadas ao contexto do *framework* INT, posto que a particularidade de cada implementação coleta muitas das vezes requer uma forma específica e customizada de representação dos dados.

Devido a estes fatores, aventa-se o seguinte problema de pesquisa: De que forma desenvolver um sistema de coleta aplicado em hardware em uma rede P4 real que permita a obtenção do histórico dos estados da rede e, conseguinte, o fomento de uma visualização interativa e granular dos metadados INT, de modo que consiga-se determinar, por meio destes, a latência dos equipamentos e o caminho de fluxo dos pacotes em tempo real?

Por conta disso, o presente trabalho propõe como objetivo geral a implementação do Visual-INT, uma ferramenta que abrange a coleta das reportações de metadados INT provenientes de *switches* P4, o processamento dessas referências e a abstração do estado da rede por meio de diagramas dinâmicos granulares portados em uma interface gráfica interativa.

Caracterizado como uma das tentativas iniciais de provisionamento de um coletor INT dentro do projeto AmLight-INT, o Visual-INT segue a mesma abordagem de uso real desta ao desenvolver um coletor versado para o hardware, utilizando-se de tecnologias equivalentes (resguardadas as proporções). O Visual-INT estabelece a análise e signifi-

²Independente de plataforma de implementação, podendo ser compilado em diferentes tipos de hardware e software (alvos).

cação das condições da rede em tempo real, especialmente no que se refere à latência individual dos ativos e do caminho de fluxo dos pacotes, promovendo a identificação dos pontos de falha e, por conseguinte, as ações administrativas cabíveis, ajudando, assim, na resolução de problemas reais enfrentados pelo projeto parceiro AmLight-INT.

Como objetivos específicos essenciais e antecedentes à consumação desta proposta, elencam-se:

- Emprego de um sistema coletor de metadados INT, a nível de hardware, que capture pacotes de reportagem oriundos de *switches* P4 (executados também em hardware), bem como o processamento e a significação destes dentro do contexto da rede investigada;
- Implementação de um sistema de banco de dados integrado à ferramenta Visual-INT para conservação dos dados de coleta;
- Desenvolvimento de uma interface gráfica interativa, com acesso via *web*, que apresente de forma clara e intuitiva os dados que foram coletados, significados e persistidos nos passos anteriores;
- Análise e validação dos resultados obtidos pelo coletor da ferramenta Visual-INT em modo nativo (*kernel*), habitual às propostas similares de coletores INT, e modo hardware (*offload*), principal elemento introduzido nesta proposta, a fim de estabelecer as vantagens e desvantagens de ambas as abordagens.

Dito isto, constitui-se esta pesquisa: de uma revisão de literatura e trabalhos relacionados (capítulo 2), onde os conceitos elementares deste trabalho são expostos juntamente com um comparativo entre obras relevantes de propósito similar; da elaboração metodológica adotada no Visual-INT (capítulo 3), na qual os detalhes de desenvolvimento e implementação são tratados; da apresentação e discussão de resultados (capítulo 4), em que validam-se os retornos obtidos pela aplicação Visual-INT; da conclusão (capítulo 5) e de ponderações relacionadas às resultâncias do processo metodológico adotado.

2. REVISÃO DE LITERATURA

Este capítulo apresenta, por meio de uma revisão de literatura, um compêndio dos fundamentos das tecnologias SDN (seção 2.1), da programabilidade no plano de dados (seção 2.2) e dos principais conceitos abarcados nos sistemas de coleta e visualização do Visual-INT (seção 2.3). Ao final do capítulo, trabalhos relacionados à proposta (seção 2.4) são apresentados, bem como pontos de diferenciação entre esta pesquisa e os retornos mais relevantes referenciados pela literatura.

Para revisão, avaliou-se diversos artigos provenientes de bases de dados distintas, como Springer Link, ACM Library e IEEE. Após aplicados filtros acerca da pertinência (como atualidade, considerando-se os últimos seis anos, além de aderência à proposta, utilizando-se os termos "SDN", "Software-Defined Networking" e "P4", "Programming Protocol-Independent Packet Processors" intercalados com "Survey" e "Networking"), a pesquisa retornou mais de 33 resultados, sendo a maioria composta por *surveys*. Trabalhos clássicos, que marcaram descobertas em redes programáveis [5] [12] também foram consultados. Similarmente, foram consideradas informações relevantes obtidas em palestras e outras apresentações esquematizadas, disponíveis em diferentes tipos de mídia [13] [14].

2.1 SDN

O paradigma SDN foi desenvolvido por cientistas da Universidade de Stanford [15], sendo seus principais conceitos provenientes de trabalhos anteriores de egressos de outras universidades americanas (Universidade da Califórnia em Berkeley, Universidade de Carnegie Mellon e Princeton). Mais tarde, da maturação dos processos, nasceria a Nicira Networks, uma das maiores empresas em pesquisa SDN no mundo [3].

O modelo SDN, apesar de recente, é baseado em ideias de longo histórico [17]. Sua base fundamenta-se nos trabalhos sobre redes programáveis, chamadas de redes ativas [16] e redes ATM programáveis [4] [12]. A programabilidade foi o principal objetivo dessas duas escolas de pensamento que dominaram a agenda de pesquisa em redes de computadores, por volta da década de 90 até o início dos anos 2000 [12]. Surgiram também dessa época as primeiras iniciativas de separação entre dados e controle [5]. Neste ínterim, podemos citar o NCP (*Network Control Point*)³, desenvolvido pela AT&T, que buscava a segmentação por meio de técnicas de sinalização (troca de informações entre nós e controles sem mediação).

³Técnica empregada em serviços de telefonia *toll-free* (como 0800), que se utiliza de geolocalização para sua concessão.

Similares aos NCPs, demais tentativas de separação dos planos de controle e de dados, como o Tempest (virtualização de redes ATM), ForCES (redes *Ethernet*), RCP (protocolo BGP) e PCE (tagueamento MPLS) [4] tornaram-se populares no período. Como iniciativas recentes de desacoplamento entre os planos estão o OpenFlow (detalhado na seção 2.1.1.1), SANE [17], Ethane [18], NOX, POX [3] e o FlowVisor. Apresentando uma perspectiva histórica, a Tabela 1, baseada em Kreutz [4], traz as diferentes instâncias relacionadas às redes programáveis, retratando o antes e depois da consolidação do modelo SDN.

Tabela 1. Resumo histórico das redes programáveis. Fonte: KREUTZ D. et al. [4].

| CATEGORIA | INICIATIVAS PRÉ-SDN | INICIATIVAS SND RECENTES |
|--|--|---------------------------------------|
| Plano de dados programável | xbind, IEEE P1520, smart packets, ANTS, SwitchWare, Calvert, High Performance Router, NetScript, Tennenhouse | ForCES, OpenFlow, POF |
| Separação dos planos de dados e controle | NCP, GSMP, TEMPEST, ForCES, RCP, SoftRouter, PCE, 4D, IRSCP | SANE, ETHANE, OpenFlow, NOX, POF |
| Virtualização de rede | Tempest, MBone, 6Bone, RON, VINI PlanetLab, Impasse, GENI | Open vSwitch, Mininet, FlowVisor, NVP |
| SOs de rede | Cisco IOS, JUNOS, ExtremeXOS, SR OS | NOX, Onix, ONOS |
| Tecnologias <i>pull</i> * | Open Signaling | ONF |

*Modelo em que a comunicação parte do cliente para o servidor.

2.1.1 Principais Conceitos SDN

O paradigma SDN possui quatro características definidoras. A primeira é a capacidade de dissociar o plano de dados do plano de controle. Isso dá-se ao tornar o plano de controle o responsável pela decisão de roteamento, analisando o pacote recebido e resolvendo a melhor forma de lidar com o tráfego entre os equipamentos disponíveis. Já o plano de dados apenas encaminha os pacotes de acordo com a decisão tomada pelo plano de controle [3]. A camada de controle regula, globalmente, os estados da rede, por meio de políticas centralizadas ou distribuídas. Devido ao acesso irrestrito a elementos e recursos da rede global, essas políticas de rede podem ser atualizadas em tempo hábil para reagir às atividades de fluxo atuais [19].

A segunda distinção reside na própria ideia de fluxo, onde baseiam-se as decisões de encaminhamento (ao contrário do destino, presente nas redes tradicionais). De forma ampla, fluxos são definidos como um conjunto de valores que atuam pelos critérios de correspondência e ação (*match-action*). Em SDN, os qualificamos como uma sequência de pacotes entre uma origem e um destino. Todos os pacotes de um fluxo recebem políticas de serviço idênticas nos dispositivos de encaminhamento. Essa abstração permite uniformizar o comportamento dentro dos diferentes tipos de dispositivos de rede, incluindo roteadores, *switches*, *firewalls* e *middleboxes* [4].

O terceiro aspecto é o provisionamento de um plano de controle unificado, de forma que um único programa de software possa gerir planos de dados com múltiplos elementos. A autoridade direta sobre os elementos do plano de dados da rede, como *switches* e interfaces, são realizadas por meio de protocolos como o OpenFlow [3]. O quarto e último traço é o fornecimento de uma rede administrável, de visão global, que permite que as alterações sejam feitas em larga escala e de uma só vez, ao invés de pontuais e graduais [3]. A Figura 1 representa todas essas vertentes, contrastando-as com a arquitetura tradicional. Percebe-se a forma consolidada com que as relações SDN ocorrem, onde tanto as aplicações de alto nível quanto as operações de baixo têm a supervisão do controlador.

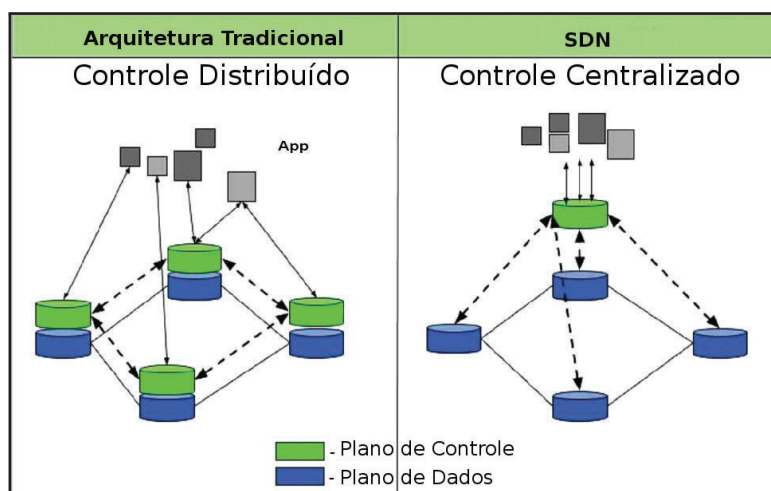


Figura 1. Diferenças entre rede tradicional e SDN. Fonte: Autor.

2.1.1.1 OpenFlow

O OpenFlow é um protocolo aberto de comunicação que opera entre o plano de encaminhamento (hardware) e o plano de controle (dados). Conforme mencionado, a centralização dos dispositivos é obtida mediante o uso deste e demais protocolos similares [20], sendo estes responsáveis pelo encorajamento dos fornecedores na implementação de alguns conceitos SDN em produtos de rede [4] [15]. Proposto por McKeown et al. [21], o OpenFlow é utilizado na maioria das implementações SDN, possuindo diversas aplicações *open source* de controladores, conforme especificado na Tabela 2. Experimentos iniciais utilizando-se OpenFlow foram criados para separar a rede em uma fatia de software controlável, com foco no encaminhamento de pacotes. O protocolo aproveita-se do fato de *switches* e roteadores modernos conterem tabelas de fluxo para funções de roteamento, máscaras de sub-rede, proteção de *firewall* e análise estatística do fluxo de dados [2].

Tabela 2. Plataformas SDN que empregam o OpenFlow. Fonte: SUZUKI, K. et al. [22].

| PLATAFORMA | LINGUAGEM | MAIOR CONTRIBUIDOR |
|-----------------|------------|---------------------|
| NOX | C++/Python | Nicira Networks |
| POX | Python | James McCauley |
| Floodlight [23] | Java | Big Switch Networks |
| Trema [24] | Ruby/C | NEC |

2.1.1.2 Switch OpenFlow

Dependendo das regras instaladas pelo aplicativo controlador, um *switch* OpenFlow pode se comportar como um roteador, comutador, *firewall*, *load balancer*, *traffic shaper* ou quaisquer outras funções que caracterizam um *middlebox* de rede [4]. A Figura 2 traz de modo simplificado o funcionamento da comutação OpenFlow. Os *switches* OpenFlow suportam a operação de tabelas de fluxo, por onde o protocolo OpenFlow combina e processa pacotes de rede nelas estabelecidas. Cada entrada da tabela constitui-se de três elementos (cabeçalho, instrução e estatísticas) em vez da entrada de roteamento tradicional (quíntupla IP). Os pacotes são correspondidos (*matching*) por seus campos de cabeçalho (*header*) e, em seguida, processados de acordo com a instrução (*action*) no fluxo de entrada. As estatísticas indicam o *status* da rede, incluindo prioridade, contadores, e assim por diante [25].

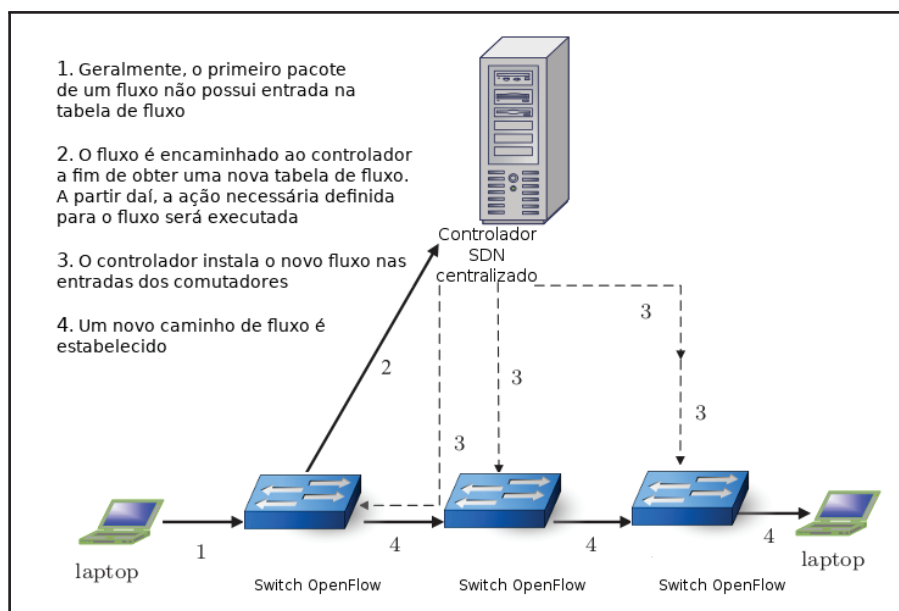


Figura 2. Comutação com Switch OpenFlow. Fonte: SOOD, K; XIANG, Y. [15].

2.2 PROGRAMABILIDADE NO PLANO DE DADOS

Consoante o já exposto, caracterizam-se as camadas de controle e dados como fisicamente separadas [26], o que permite a evolução de ambas de forma distinta [27]. Essa separação permitiu que os dispositivos de encaminhamento atingissem níveis maiores de programabilidade. Não obstante essa capacidade, houve um enredamento destes ao plano de controle. Ao utilizar-se de uma interface comum, aberta e independente de provedor (tal como o OpenFlow), o plano de controle promoveu o gerenciamento dos equipamentos de rede de diferentes fornecedores de hardware e software, revelando-se imprescindível ao plano de dados [26].

No início, a simplicidade da interface OpenFlow, onde uma única tabela de regras abstraía diversos campos de cabeçalho (como a quintupla IP), atendeu satisfatoriamente a demanda. Porém, nos últimos anos, para permitir que dispositivos expusessem mais de seus recursos ao controlador, a especificação OpenFlow (diretiva do uso do protocolo) tornou-se mais complexa, acrescentando campos de cabeçalho e estágios de regras [26]. Embora continuamente expandida, a catalogação fica aquém da proliferação de novos campos de cabeçalho. Em outras palavras, a padronização da programabilidade no plano de controle produziu um plano de dados “fixo”, no sentido de que os usuários restringiram-se a trabalhar com protocolos identificados na especificação OpenFlow [28].

2.2.1 P4

Um plano de dados verdadeiramente programável deve ser livre de tais restrições, permitindo a reconfiguração de acordo com a sintaxe e semântica do protocolo a ser implementado [28]. Para contornar as deficiências do OpenFlow, o emprego de linguagens de programação para o processamento de pacotes popularizou-se tanto na academia quanto na indústria [27]. Abordagens como o P4 permitem descrever um comportamento agnóstico de encaminhamento no plano de dados. Por ser uma linguagem de alto nível, o P4 fornece um dialeto de rede simples para descrever o processamento dos datagramas, podendo trabalhar em conjunto com protocolos de controle SDN como o OpenFlow [26].

Um dispositivo habilitado para P4 é independente de protocolo, no sentido de que nenhum suporte inerente a nenhum protocolo é assumido. Em vez disso, programas escritos em P4 definem cabeçalhos de pacotes e especificam comportamentos de análise e processamento dos mesmos [28]. Com a proposta da linguagem P4, busca-se a evolução do modelo por meio da:

- **Reconfigurabilidade no campo:** os programadores devem ser capazes de mudar a forma como os *switches* processam os pacotes depois de implementados;

- **Independência do protocolo:** os computadores não devem estar vinculados a nenhum protocolo de rede específico;
- **Independência do alvo:** os programadores devem ser capazes de descrever a funcionalidade de processamento de pacotes independentemente das especificidades do hardware subjacente.

2.2.2 INT

O INT é um *framework* que propicia a coleta e a reportação de informações do estado da rede pelo plano de dados (hardware), sem a necessidade de intervenção do plano de controle. Para isso, o modelo INT utiliza-se de cabeçalhos que são interpretados como instruções de telemetria pelos dispositivos de rede. Por meio dessas instruções, os equipamentos que suportam o INT compreendem quais estados de rede devem ser colhidos ou escritos nos pacotes que movimentam o tráfego [11]. Com intenção de alcançar seus objetivos, o *framework* INT integra-se ao código P4 que define o comportamento do dispositivo.

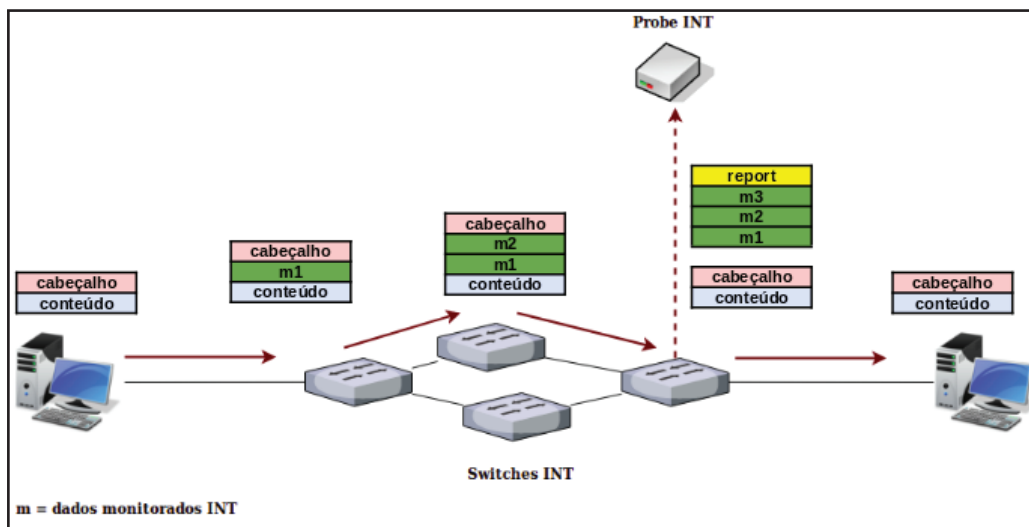


Figura 3. Escrita, extração e reportação dos metadados INT. Fonte: Adaptado de TU, N. V. [29].

Representado na Figura 3 está o modo de funcionamento do *framework* INT. Considerando a topologia fictícia descrita, dispõem-se dois *hosts* que se comunicam entre si mediante quatro *switches* P4 habilitados com INT. Admitindo-se que um pacote seja composto por cabeçalho e conteúdo e que o fluxo (caminho entre os *hosts* de origem e destino) perfaça três equipamentos, tem-se o aproveitamento do caminho habitual do pacote para o transporte de cabeçalhos INT (encapsulados entre o cabeçalho original e o conteúdo) incluindo informações sobre o estado de cada equipamento passante. Ao fim do percurso, o último *switch* (denominado *sink*) desacopla os metadados do cabeçalho e conteúdo ori-

ginais, organizando-os em uma reportagem (*report*) enviada para um dispositivo de coleta externo. Novamente montado, o pacote (cabeçalho e conteúdo) é encaminhado para seu destino final.

2.3 CONCEITOS DO VISUAL-INT

Esta seção sumariza, via referencial teórico, conceitos importantes utilizados no Visual-INT, especificamente no tocante à ferramenta de coleta de metadados INT, a saber o eBPF e XDP. Detalhes metodológicos e de implementação destes elementos são expostos no capítulo 3.

2.3.1 eBPF

O eBPF é uma ferramenta integrada ao sistema Linux que oportuniza ao usuário o desenvolvimento de aplicações de análise de elementos do *kernel* com baixo *overhead*. Um programa eBPF é capaz de anexar-se a um caminho específico (também denominado soquete) dentro do *kernel*. Deste modo, quando um soquete (caminho) é percorrido, qualquer programa eBPF anexado a este será executado. O eBPF é especialmente utilizado em programas de rede, sendo estes anexando aos soquetes para filtrar e classificar o tráfego, bem como efetuar quaisquer outras ações definidas pelo usuário. O eBPF também possibilita a modificação das configurações de soquetes de rede, além de, em conjunto com o XDP (particularizado na seção 2.3.2), executar o processamento de alto desempenho dos pacotes no nível mais baixo da pilha de rede, imediatamente após o recebimento.

Originalmente, sua versão clássica, o BPF (ou cBPF, sendo c de *classic*) foi projetada para capturar e filtrar pacotes de rede que correspondiam a regras específicas, tornando-se responsável pela filtragem de soquete (LSF) que evitava que cópias inúteis de pacotes do *kernel* passassem a integrar o *userspace*⁴. À época, essa situação era comum ao *kernel* que, ao ser consultado pelo modo usuário, entregava diversas cópias do tráfego passante, ainda que o dado requerido fosse de um tipo específico (o que degradava o desempenho). Nesse ínterim, algumas ferramentas tradicionais de avaliação de pacotes beneficiaram-se de sua simplicidade e robustez, como no caso do *tcpdump* (ferramenta em C para leitura de cabeçalhos de pacotes) [30].

Redesenhado em 2013, o eBPF (assim designado desde então), introduziu novos recursos, como mapas e chamadas de cauda⁵, além de uma codificação mais próxima da linguagem de máquina nativa. Além do filtro de soquete, os programas eBPF podem ser conectados a interfaces de ingresso ou egresso e executar uma variedade de tarefas de

⁴Modo de execução onde o processador ocupa-se apenas de instruções não privilegiadas.

⁵Se excedido os 4096 bytes, esse mecanismo permite que um novo programa eBPF assumo o controle.

processamento de pacotes de modo eficiente. A Figura 4 denota essa transformação, evidenciada nas interações entre o programa eBPF no *userspace* (à esquerda) e o *kernel* do sistema (à direita). Após compilado no *userspace*, o programa eBPF produz um *bytecode* que é carregado e verificado pelo *kernel*. Uma vez validado, o programa pode atuar em diferentes níveis de *tracing* (rastreamento) nos *data sources* Linux. Por exemplo, se anexado a *kprobes* (pontos de observação no *kernel*), o programa passa a supervisionar determinadas partes do *kernel* [31]. O mesmo vale para *uprobes* (pontos de observação no *userspace*) e *tracepoints* (ambos os modos, sem dependência de versionamento do *kernel*).

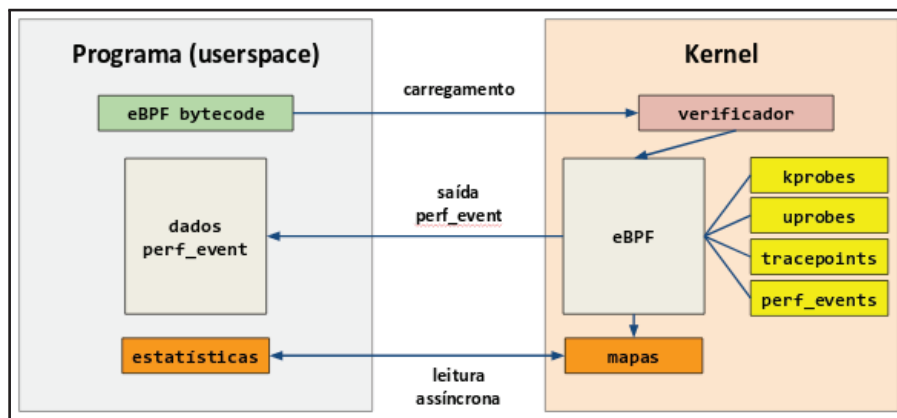


Figura 4. Diagrama de funcionamento do eBPF. Fonte: Autor.

A Figura 4 também destaca alguns elementos que se comunicam diretamente com o *userspace*, formando os principais meios de acesso às informações do *kernel*. Dentre eles está o mecanismo de coleta *perf_event*, considerado o instrumento central para a reportagem de ocorrências no *kernel*. Como um *buffer* que atua sobre os *data sources*, o *perf_event* pode monitorar eventos significativos para o *userspace*, como a análise de entrada e saída de disco, por exemplo. Outras unidades bastante utilizadas para retornar dados ao *userspace* são os mapas. Citados anteriormente, os mapas são arranjos arbitrários de chave e valor (*key-value*) definidos pelo usuário que, através de *syscalls*, fornecem informações salvas em suas estruturas de forma assíncrona.

2.3.2 XDP (*eXpress Data Path*)

O XDP, desenvolvido pelo projeto IOVisor, tem por finalidade fornecer um caminho rápido para o processamento de pacotes dentro do Linux. Os programas eBPF têm seu desempenho alavancado através da interoperabilidade com esta tecnologia, que confia ao eBPF a responsabilidade de execução da computação dos datagramas [30]. Formulado em 2016, o XDP atua no nível mais baixo da pilha de rede, tendo sua programabilidade inicialmente projetada para aplicações de alta demanda. Difundiu-se recentemente no contexto

de *bare metal*⁶, suplantando métodos para o aumento de desempenho conhecidos, como o *bypass*.

Ao longo do tempo, a técnica de *kernel bypass* foi favoritada por diversas aplicações especializadas. Sua metodologia consiste em ignorar a camada de rede do *kernel* e realizar todo o processamento de pacotes no *userspace*, que passa a ser o gerenciador do *driver*. Essencialmente, a rede do *userspace* alcança altas velocidades fora do domínio do *kernel*. Apesar da redução de *overhead*, o emprego dessa técnica exclui as demais abstrações do *kernel* para recursos de hardware, causando retrabalho na implementação. Os *drivers* de *userspace*, embora funcionais, são geralmente menos testados e reutilizáveis do que os fornecidos pelo *kernel*. O comprometimento da integração com as demais partes do SO, tal qual a segurança, são algumas das objeções para a adoção deste método.

Em alternativa, o XDP emprega técnica oposta: com o auxílio do eBPF, move os programas de rede (filtros, mapeadores, roteamento, entre outros) do *userspace* para o domínio do *kernel*. Isso permite executar a função de rede tão logo o pacote atinja a NIC, resultando num aumento significativo do desempenho de processamento de pacotes [32]. Corroborando a afirmativa anterior, a Figura 5 representa as taxas de processamento dentro do Linux. Nota-se que, quanto mais próxima do hardware de rede a aplicação for executada, mais elevada será a quantidade de pacotes processados em Mpps (milhões de pacotes por segundo), garantindo-se os melhores coeficientes nos níveis mais baixos da pilha de rede. Por exemplo, o XDP (quando orquestrado no *driver* Linux) atinge índices próximo aos 20 Mpps. Seu uso no Visual-INT é detalhado na seção 3.

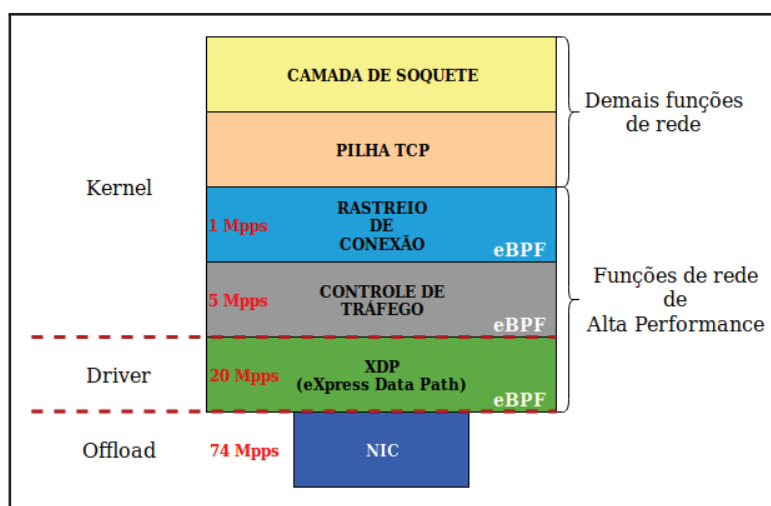


Figura 5. Comparativo entre taxas de processamento na pilha Linux. Fonte: Adaptado de VILJOEN, N. [33].

Como exemplo prático, a Figura 6 sintetiza o funcionamento do XDP/eBPF ante o caminho normal de rede, trabalhando o caso hipotético de rejeição de pacotes. Na composição à esquerda, representando o caminho normal de rede, percebe-se o trajeto percorrido

⁶Execução de instruções diretamente no hardware lógico, sem a intervenção de um Sistema Operacional.

(retângulos destacados) por um pacote inadequado à aplicação (L7). Até ser recusado, o pacote entra pelo meio físico (RX), percorre a pilha TC (*Traffic Control*) e aproxima-se da camada de transporte (L4). Por sua vez, o caminho expresso XDP, à direita na imagem, toma a decisão sobre o pacote a nível do plano de dados (por meio de um programa eBPF), rejeitando o datagrama antes que ele atinja os níveis mais altos da pilha de rede.

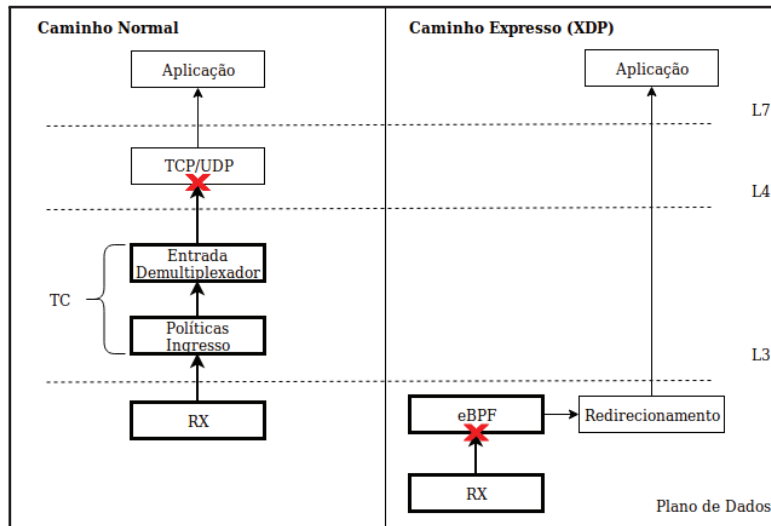


Figura 6. Caminho normal em relação ao XDP. Fonte: Autor.

2.4 TRABALHOS RELACIONADOS

Durante a pesquisa bibliográfica, diversos trabalhos relacionados à análise de fluxos de rede foram identificados. Dentro da área de monitoramento, elencam-se trabalhos clássicos pré-SDN como NetFlow [34] e SFlow [35], ambos não idealizados para aplicações em tempo real, possuindo ciclos de consolidação de *status* próximos a 1/4 de minuto. O primeiro, introduzido pela Cisco em 2004, anexa aos pacotes passantes informações sobre o fluxo, ao custo do aumento de carga de trabalho e uso de memória nos *switches*. De mesma época, o SFlow adota a utilização de *samples*⁷ que, apesar do pouco *overhead*, demonstram menor acuracidade. As estratégias de obtenção dos dados de fluxo variam de técnicas *polling*⁸ a despachos diretos a *probes*, respectivamente.

Relacionado ao NetFlow, destacam-se duas ferramentas, NFDUMP e NfSen [36]. A primeira funciona de forma similar ao *tcpdump*, lendo informações sobre o NetFlow por meio de um *daemon* (*nfcapd*). Seu objetivo é analisar dados do fluxo de rede, além de rastrear padrões interessantes de tráfego. O NfSen, por seu turno, cria gráficos do tráfego de rede com base nas informações extraídas pelo NFDUMP, utilizando-se para tal de tecnologias *open source* como PHP, Perl e RRD (*Round-Robin Database*).

⁷Amostras sobre o estado da rede obtidas de tempos em tempos.

⁸Envio de amostras sobre o estado da rede por uma arquitetura cliente-servidor.

Em SDN, propostas como OpenNetMon [37], OpenSample [38] e FlowSense [39] demarcam tentativas de monitoramento de fluxo nessas redes. O OpenNetMon utiliza-se da mesma técnica de *polling* supramencionada, com a distinção de realizá-la nos *edges* (*switches* de borda) POX (apresentado na Tabela 2) com frequências variáveis que adaptam-se à velocidade do fluxo. O OpenSample, por sua vez, opera em *sampling* (também citado) em redes Floodlight (Tabela 2), focando em grandes tráfegos de rede. Destarte o mais efetivo, empregando mensagens de fluxos diretamente ao controlador SDN, o FlowSense padece de limitações análogas a seus pares, tais quais o não-provimento de informações fim a fim, granulares e em tempo real sobre o estado da rede.

Das iniciativas que utilizam INT, até o início da elaboração desta proposta, destacam-se o Prometheus INT [40], INTMon [41] e INTCollector [29]. O primeiro emprega a coleta INT de dados recentes, cujo nome advém do uso de uma *suite open source* de monitoramento homônima (Prometheus), provendo a visualização por meio de interfaces próprias e/ou ferramentas como o Grafana (ferramenta *open source* para visualização de métricas). O INTMon provê a implementação INT em conjunto com o SO de rede ONOS (presente na Tabela 1), que a controla. Mesmo não enredando-se a um controlador em específico, o INTCollector também associa seu mecanismo de visualização ao ONOS (e paralelamente ao Grafana), questão que baliza a forma de apresentação dos dados.

A obra que mais se assemelha a esta pesquisa é o INTCollector. Pesa contra ela a realização eBPF/XDP (bem como P4/INT) em ambiente emulado/simulado pelo Mininet e modelo BMv2, a ausência da elaboração do *timestamp* de ingresso (necessário para determinar a latência corretamente) e a falta de uma interface própria de visualização de fluxos. Em relação ao INTMon, sua integralização com o ONOS impede que sua solução se estenda aos demais cenários possíveis. Tal como o anterior, a principal desvantagem do Prometheus INT é o alto *overhead* (toda iteração INT é reportada para um *gateway* central), além de basear-se em informações de curta duração, não promovendo um histórico do comportamento da rede.

Tabela 3. Comparação entre os trabalhos relacionados e o Visual-INT. Fonte: Autor.

| PROPOSTA | AMBIENTE REAL | SUPORTE P4/INT | OVERHEAD | HIST/VIS |
|-------------------|---------------|----------------|----------|----------|
| OpenNetMon | ✓ | X | ALTO | X |
| OpenSample | ✓ | X | ALTO | X |
| FlowSense | ✓ | X | BAIXO | X |
| Prometheus INT | X | ✓ | MÉDIO | VISUAL |
| INTMon | X | ✓ | ALTO | VISUAL |
| INTCollector | X | ✓ | BAIXO | AMBOS |
| Visual-INT | ✓ | ✓ | BAIXO | AMBOS |

A Tabela 3 contempla as principais diferenças entre o Visual-INT e os trabalhos expostos (excluindo-se as obras de valor histórico), relacionando-os com ambiente real (se

está aplicado em hardware), suporte à programabilidade no plano de dados e telemetria de rede em banda (P4/INT), estimativas do valor de *overhead* entre baixo, médio e alto (com base na regressão linear $C \times R + b$ - lendo-se C como CPU, R como taxa de envio e b como coeficiente - apresentada em TU, N. V. [29]) e a disponibilização de histórico e visualização dos fluxos (assumindo X em caso negativo, VISUAL para suporte à visualização, HISTÓRICO para fornecimento de histórico e AMBOS para provimento de visualização e histórico).

3. PROCEDIMENTOS METODOLÓGICOS

Este capítulo versa sobre os procedimentos metodológicos incorporados ao Visual-INT, objetivo geral desta pesquisa. Neste, descreve-se brevemente o projeto de referência AmLight-INT (seção 3.1), seguido do detalhamento da proposta Visual-INT (seção 3.2) por meio de uma abordagem geral acerca da arquitetura externa (seção 3.2.2) e interna (seção 3.2.3), esta última composta da aplicação coletora (seção 3.2.3.1), do sistema de banco de dados de séries temporais (seção 3.2.3.2) e do utilitário de visualização (seção 3.2.3.3).

3.1 AMLIGHT-INT

Concebido em 2018, o projeto AmLight-INT (assim como a rede AmLight-Exp) descende da iniciativa AmLight (*America's Lightpaths*), empreendimento que opera desde 2014 uma rede SDN homônima que interliga pontos de intercâmbio acadêmicos e NRENs (Redes Nacionais de Pesquisa e Educação) da América do Sul ao AMPATH (*America's Path*)⁹ [42]. Para alcançar seus objetivos de monitoramento com o INT, o AmLight-INT reúne em sua equipe especialistas em redes acadêmicas e empresas de comutadores P4 (como a NoviFlow), com o propósito de alterar o plano de dados da rede AmLight-Exp sem afetar o seu plano de controle.

A Figura 7 representa um panorama geral das interações entre atores e componentes dentro do projeto. Os atores principais, simbolizados pelas figuras humanas, aludem aos times da NCSA (*National Center for Supercomputing Applications*)¹⁰ (ao topo) e do LSST (na parte inferior), respectivamente. Já as caixas azuis e a vermelha figuram os *switches* NoviFlow e o servidor SDN *Looking Glass*¹¹ (SDN-LG), na devida ordem. Dentro do servidor SDN-LG encontram-se a aplicação de coleta e a API de visualização dos dados (peças centrais no Visual-INT, visto mais adiante). Por fim, as *probes* designam os dispositivos de coleta INT.

Pela Figura 7 também é possível perceber o relacionamento entre os atores e os componentes. A linha alaranjada qualifica o fluxo de dados entre o LSST, no Chile, e a sede da NCSA, em Illinois. Numa menor escala, evidencia-se a comunicação dos *switches* SDN da NoviFlow entre si, via rede AmLight-Exp, e destes com as *probes* INT. Abastecidas com os dados coletados, as *probes* por sua vez descarregam as informações dentro do dispositivo coletor localizado no servidor SDN-LG. Processados, estes dados são exibidos por uma API aos agentes interessados, possibilitando o monitoramento em tempo real.

⁹Conecta redes de pesquisa das Américas do Sul e Central à rede norte-americana Internet2.

¹⁰Entidade-chave em grandes projetos de infraestrutura de redes nos EUA.

¹¹Servidor de rotas público, separado da rede real, que apenas exibe a tabela de roteamento.

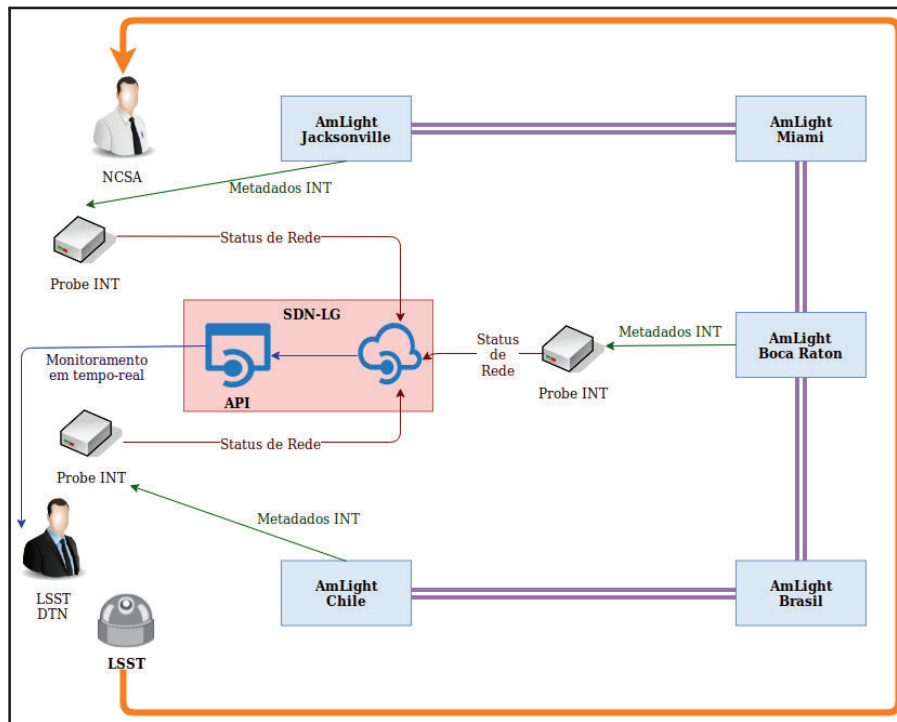


Figura 7. Diagrama de funcionamento do projeto AmLight-INT. Fonte: Autor.

3.1.1 SDN e OpenFlow no AmLight-INT

Dentro da iniciativa AmLight-INT, tanto o uso do protocolo OpenFlow quanto de outras prerrogativas do paradigma SDN são contempladas. Como exemplo estão os *switches* da rede AmLight-ExP, que possuem suporte à versão 1.3 do OpenFlow. Apesar disso, por motivos técnicos, o atual versionamento tornou-se um inviabilizador do monitoramento via P4/INT. Para resolver esse problema, a empresa canadense NoviFlow, parceira da pesquisa AmLight-INT, proverá novos *switches* OpenFlow com suporte à programação no plano de dados, permitindo assim a implantação da telemetria em seus equipamentos.

3.2 VISUAL-INT

Conforme o exposto, constata-se o grande alcance e complexidade do projeto AmLight-INT. Para obter êxito na tarefa de reprodução do cenário vislumbrado anteriormente, bem como na implementação e apuração dos metadados INT, optou-se pela divisão dos esforços em dois projetos distintos: um deles, destacado desta pesquisa, tornou-se o responsável pelos elementos fundamentais da estruturação da rede de testes (topologia, conectividade, transporte, dentre outros), da qual as reportações INT são provenientes. O outro é o Visual-INT, objetivo geral desta pesquisa, que ocupa-se da coleta, armazenamento e visualização dos metadados remetidos.

Desta feita, o Visual-INT, circunstanciado daqui em diante, é uma colaboração com o AmLight-INT, sendo sua primeira tentativa de provisionamento de coleta e visualização de metadados INT oriundos de SmartNICs programadas em P4. Os projetos estruturam-se de acordo com a distribuição contida na Figura 8. No bojo da estrutura, concentram-se as camadas de controle (marcada em fundo roxo) e dados (ilustrada em fundo amarelo). Na parte inferior da imagem, vemos a porção do plano de dados onde, por meio da programação das SmartNICs (expressa nas seções 3.2.2.1 e 3.2.2.2), a configuração de metadados INT acontece (detalhado na seção 3.2.2.3).

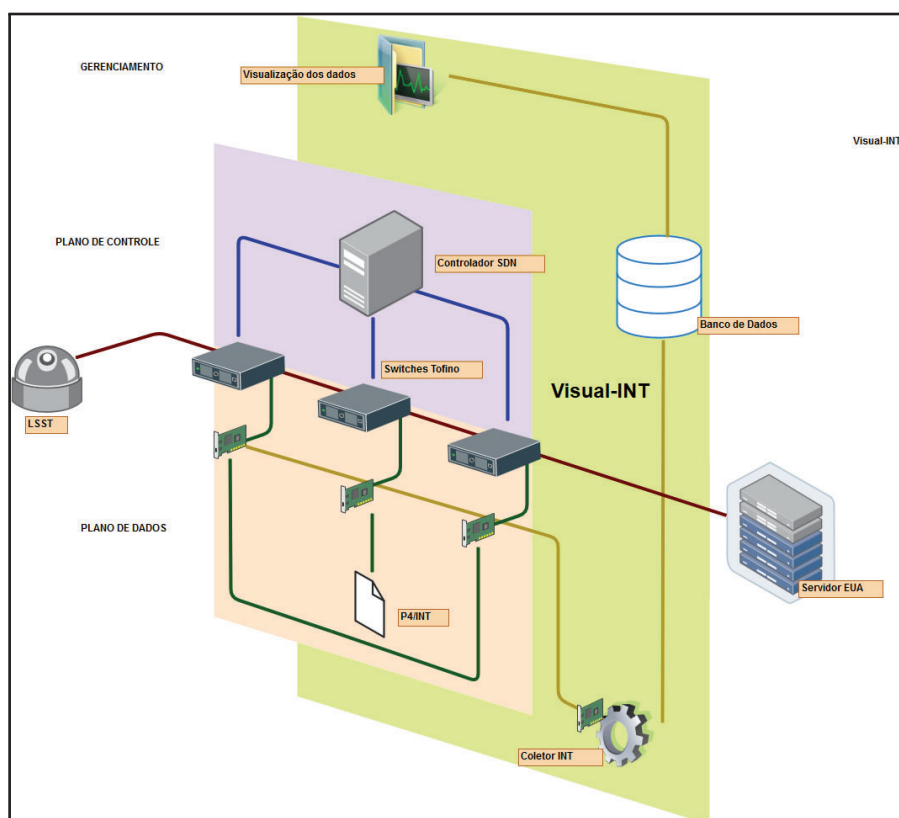


Figura 8. Esquema de operação do Visual-INT. Fonte: Autor.

Concerne ao Visual-INT o arranjo externo (destacado em fundo verde) responsável pela visualização de metadados INT, dispondo de um sistema coletor (operado dentro de uma SmartNIC), um sistema de banco de dados e um utilitário para demonstração visual dos dados coletados. Por intermédio de um programa eBPF aplicado à SmartNIC coletora (expressa na seção 3.2.2.1), o Visual-INT obtém as informações dos metadados reportados pelos *switches* P4. Após a coleta, os metadados são pré-processados em expressões comuns contendo os estados da rede em tempo real, sendo encaminhados para o banco de dados InfluxDB (abordado na seção 3.2.3.2). Por meio do utilitário de visualização (explicitado na seção 3.2.3.3), os dados são recuperados do banco e demonstrados às equipes de gerenciamento da aplicação. O estrato de gerenciamento, por sua vez, estabelece ao controle as ações a serem realizadas, de acordo com o estado da rede obtido pela aplicação

de visualização. Depreende-se o uso agnóstico de um controlador, imbuído de executar as diretrizes impostas pela camada de gerenciamento.

3.2.1 SDN e OpenFlow no Visual-INT

Diferentemente do AmLight-INT, que necessita do emprego de equipamentos robustos providos pela NoviFlow e Barefoot (disponibilizadora do processador P4 Tofino), bem como de um controlador SDN padrão (Kytos), o projeto Visual-INT parte de uma perspectiva agnóstica quanto a essas concepções. Por exemplo, por não ter um controlador definido, espera-se que o programa, por meio de APIs, enquadre-se em quaisquer outros ambientes SDN, generalizando sua solução. Além disso, a estrutura utiliza-se inicialmente de NPUs Netronome (SmartNICs), tornando-se mais barata e flexível para os tipos de testes a serem executados. Com foco no plano de dados, pretende-se que a ferramenta adéque-se ao maior número de cenários possíveis.

3.2.2 Arquitetura Externa

A seguir, abordam-se os tópicos cruciais da contextura externa ao Visual-INT, dentre os quais os pontos de intersecção entre este e a rede P4 associada. As seções seguintes ensaiam apenas sobre o entendimento comum existente nas duas propostas, desconsiderando-se aspectos não triviais à aplicabilidade do presente trabalho.

3.2.2.1 SmartNICs e Máquinas Físicas

Entre os fundamentos da arquitetura adotada estão as SmartNICs Netronome, também denominadas NPUs. Acoplado fisicamente a três máquinas Linux reais (com distribuições CentOS para as duas de função *switch* e Ubuntu 18.04 para a Visual-INT), o modelo Agilio CX [43], representado na Figura 9, é utilizado em toda a estrutura, desde a implementação à coleta de metadados INT, possuindo conexão PCIe, memória de 2GB DDR3 e duas portas com 40GbE (*Gigabit Ethernet*) de capacidade cada. A adesão de tal tecnologia justifica-se pelo seu custo acessível (quando comparada a *appliances* programáveis de uso similar), versatilidade de aplicação (com suporte ao P4 e eBPF) e capacidade de *throughput* (superior às propostas virtualizadas). Em menor escala, as placas fornecem características parecidas com os dispositivos empregados em produção na rede AmLight-Exp, permitindo métricas comparativas quanto ao desempenho.

Atendo-se à perspectiva da aplicação coletora, maior beneficiária do uso de SmartNICs, observa-se o exposto na Figura 10. Ao comparar a aplicabilidade da técnica de *offloading* (descarregamento) do programa *bpfilter* (software *firewall* escrito em eBPF) dentro

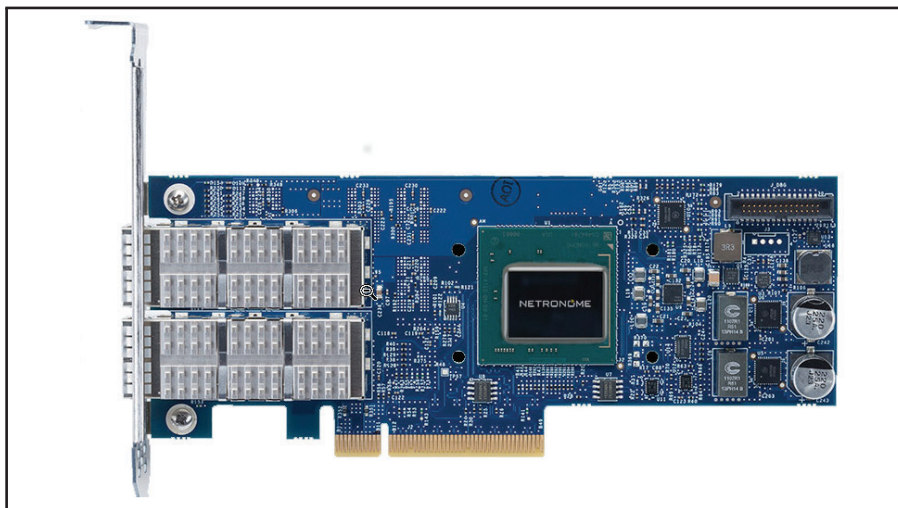


Figura 9. SmartNIC Netronome Agilio CX. Fonte: NETRONOME, 2019 [44].

de uma SmartNIC Agilio CX 1x40GbE (compatível com o modelo priorizado nesta proposta) quanto aos demais métodos de filtragem de pacotes (*iptables*, *nftables* e o próprio *bpfiler* em ambiente virtualizado), percebe-se que a taxa de transferência do primeiro desbancas as demais em proporções relevantes (considerando-se em Mpps - milhões de pacotes por segundo) [44]. Para o aumento de desempenho, o recurso de *offloading* também é implementado dentro do coletor do Visual-INT, sendo seus detalhes expostos na seção 3.2.3.1.

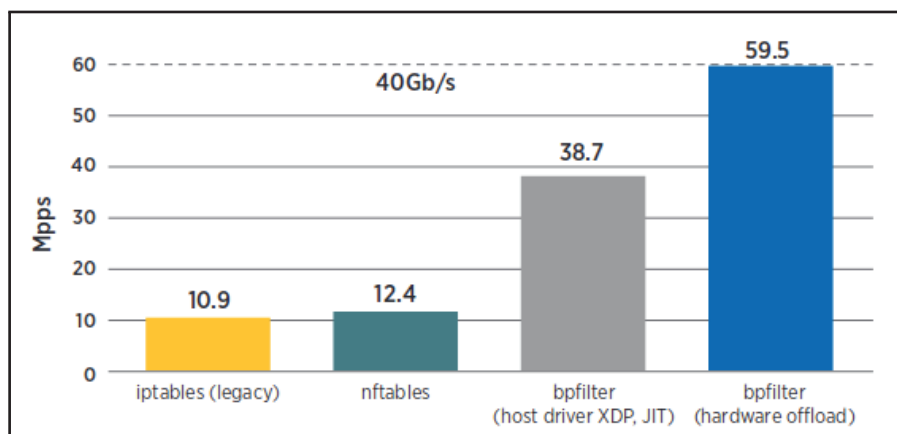


Figura 10. Comparação de processamentos em Mpps entre arquiteturas. Fonte: NETRONOME, 2019 [44].

Para este cálculo, considera-se a taxa máxima de 40Gbps para cada porta e o padrão para taxa de encaminhamento em Mpps (a saber 64 *bytes* por *frame* + 8 *bytes* de *header frame* + 12 *bytes* de intervalo entre *frames*), aproximando-se, assim, de um coeficiente de 59 Mpps (conforme a Figura 10), totalizando índices teóricos de 118 Mpps em caso de uso simultâneo das duas portas.

3.2.2.2 Switches P4 e Topologia

As possibilidades decorrentes da adoção do P4 são substanciais. Por ser independente de alvo, a linguagem permite a construção de *switches* que consigam lidar com protocolos próprios ou ainda não homologados em RFC (trâmite que pode levar anos). Essas características são aproveitadas para a elaboração da rede acessória que disponibiliza os dados INT para o coletor do Visual-INT. Conforme a Figura 11, utilizando-se de SmartNICs Netronome (Switch 1, Switch 2 e Coletor Visual-INT), é formulada a topologia básica implementada neste trabalho. Ainda na imagem, observa-se a estrutura de *links* e suas respectivas capacidades, onde as conexões perfazem 10Gb/s (entre *hosts* e *switches*) e 40Gb/s (entre a borda da rede e o coletor Visual-INT).

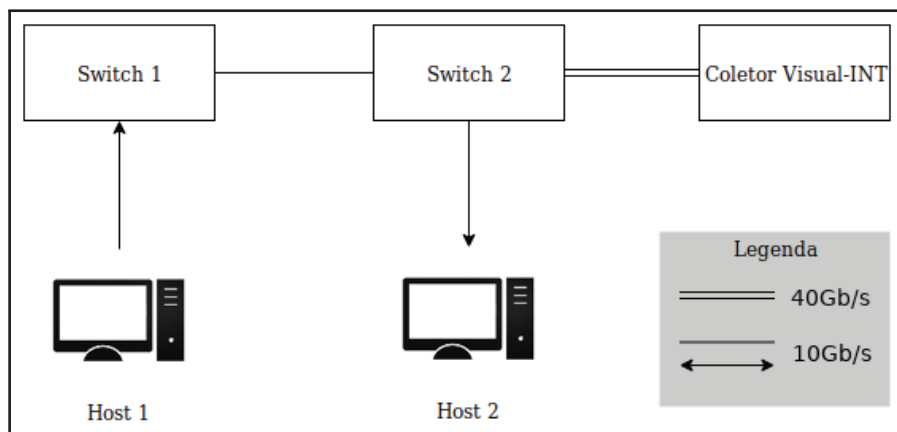


Figura 11. Topologia da rede P4 complementar ao Visual-INT. Fonte: Autor.

3.2.2.3 Cabeçalhos de Telemetria

Codificados em P4, coexistem dentro das SmartNICs tanto as especificações usuais para o funcionamento do equipamento (controle de portas, roteamento, ACLs e outros) quanto as regras imanentes ao *framework* INT (formatos, cabeçalhos, reportagem e equivalentes). Dentro da rede na qual o Visual-INT insere-se, o fluxo resultante da sinergia entre essas duas rotinas, a saber os metadados INT, são discriminados na Figura 12. Num primeiro momento, o Host 1 envia tráfego ao *switch* P4 (Switch 1) com a telemetria habilitada (A). Dentro do Switch 1, as diretrizes INT são apuradas e escritas nos pacotes (B), sendo subsequentemente enviadas ao fluxo normal (C) com seus cabeçalhos modificados. No próximo salto (*hop*), o cabeçalho INT é lido por outro *switch* P4 (Switch 2) com o INT implementado, sendo suas informações de estado empilhadas num novo segmento de metadados (D). Pela limitação de recursos, o Switch 2 desempenha também a função de reportador, designada como o *sink*. Nele, os pacotes de rede são separados dos metadados

INT, seguindo os primeiros ao seu destino final (E) enquanto os últimos são encabeçados em formato *report* e entregues ao coletor do Visual-INT.

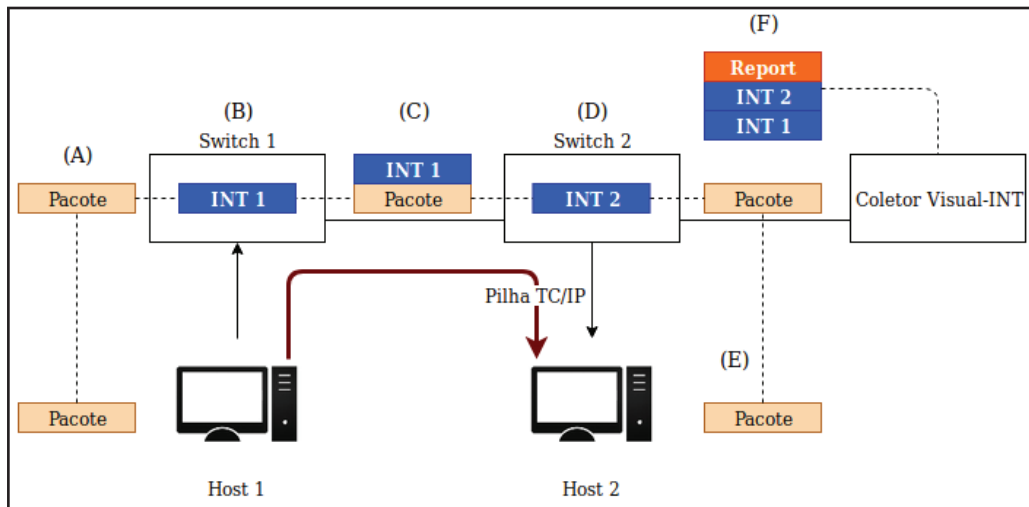


Figura 12. Tráfego de telemetria até a chegada ao coletor do Visual-INT. Fonte: Autor.

Arranjo imprescindível para obtenção das medições, importa ao coletor do Visual-INT saber lidar com o *report*, que encapsula o cabeçalho INT contendo os metadados. De construção mais simples que o cabeçalho de referência INT (evidenciado na Figura 14), o *report* (pormenorizado na Figura 13) é composto pelos campos *Ver* e *NProto* (ambos 4 *bits*), significando a versão INT e o protocolo subsequente (por padrão o *Ethernet*). Os *bits* *D*, *Q* e *F* controlam se o pacote deve ser recusado (*drop*), se a fila pertencente está congestionada (*queue*) ou se fluxo está sendo monitorado (*flow*), respectivamente. As áreas *Rsvd1* e *Rsvd2* são reservadas, em compasso que *Hw_Id*, *SeqNumber* e *Ingress TimeStamp* elaboram sobre o identificador do *switch*, o número de *reports* INT relacionados ao equipamento, além do tempo em nanossegundos (*timestamp*) que o pacote atingiu a interface de entrada do *sink*.

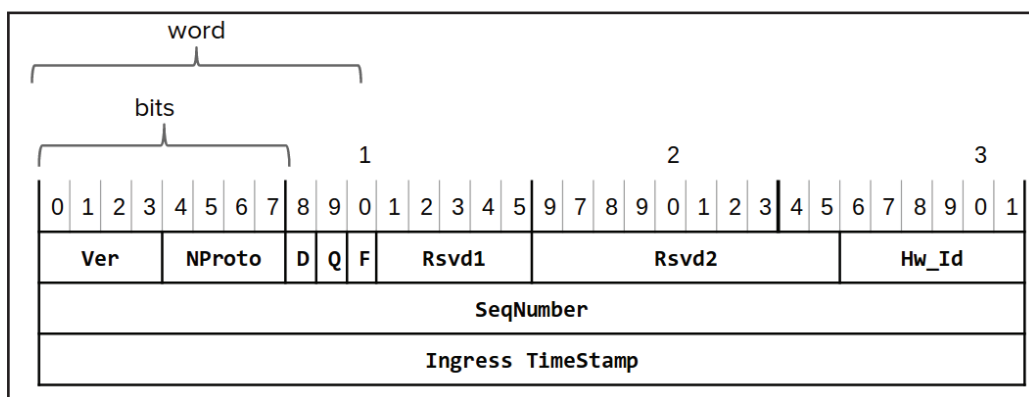


Figura 13. Formato do *report* aplicado ao Visual-INT. Fonte: Autor.

Na Figura 14, a estrutura completa do cabeçalho INT pode ser observada, contendo a quantidade de bits, *offset* e segmentação. Da esquerda para direita, os campos *Ver*

e Rep (de 4 e 2 *bits*, respectivamente) referem-se a versão INT utilizada e o uso da replicação de pacotes (útil em casos de encaminhamento para múltiplos caminhos), respectivamente. Os *bits* de controle C e E representam a habilitação de cópia e a indicação se o número máximo (predefinido) de metadados foi atingido. Rsvd1 e Rsvd2 são *bits* reservados, podendo ser utilizados na customização (deslocamento de *bits*, inserção de metadados, dentre outros) de campos. Na devida ordem, InstCnt, MaxHopCnt e TotalHopCnt (5 e 8 *bits*) são contadores básicos contendo o número de instruções ativadas (neste caso, para o Instruction Bitmap), a quantidade de saltos (*hops*) permitidos e a contagem de equipamentos passantes até o momento de reportação.

Continuando na Figura 14, o campo Instruction Bitmap (que pode chegar à 32 *bits*) organiza as regras que serão processadas em cada equipamento com suporte INT. Estas prescrições incidem sobre o elemento INT Metadata Stack, que forma a pilha de metadados contendo estritamente a disposição de itens provida pelo *bitmap*. As ordens são compostas por nove tipos de campos, sendo quatro de identificação (*switch ID*, IDs da porta de entrada/saída e ID de fila), um de tempo (*timestamp*) e outros quatro de medição (latência de *hop*, ocupação/congestionamento de filas e utilização de *link*). Teoricamente, estes campos combinados podem processar cerca de 12 medições diferentes, dependendo exclusivamente das capacidades do hardware.

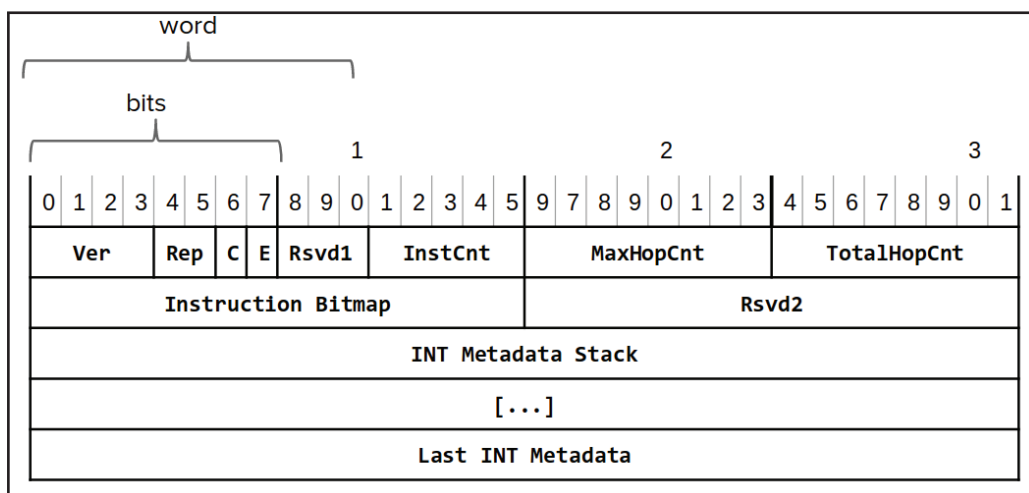


Figura 14. Formato do cabeçalho INT aplicado ao Visual-INT. Fonte: Autor.

Na implementação P4, modificou-se os campos iniciais do modelo de referência INT, sendo os campos Ver, Rep, C e E (e colateralmente o Rsvd1) deslocados para ocuparem todo o espaço do primeiro *byte*. A customização de cabeçalhos dá maior liberdade ao administrador uma vez que, dependendo da aplicação ou da capacidade do dispositivo, certas características tornam-se desejáveis e outras preteríveis.

3.2.3 Arquitetura Interna

A fim de alcançar o propósito de significação dos metadados INT, encapsulados dentro do cabeçalho *report*, propõe-se a arquitetura exposta na Figura 15. Nela, compreende-se a parte de extração, processamento e visualização dos dados de telemetria. Ao atingir a SmartNIC coletora, o pacote *report* é desmontado em três estratos de *parse* à nível de hardware (NPU) por intermédio da programação eBPF/XDP previamente descarregada (*offloaded*) na placa. Após serem sacados, os metadados são encaminhados para o *userspace* onde são aferidos e organizados por um programa em C (Analisador do Coletor), que persiste seus valores em formato JSON para dentro do banco de dados *time series* (InfluxDB) através de uma API. A aplicação visualizadora, composta por um servidor NodeJS e páginas HTML/JavaScript, recupera essas entradas (*measurements*) do banco e as exibe de forma gráfica mediante mapas e diagramas interativos concernentes ao estado da rede em tempo real. Todo o ambiente é performado em uma máquina física com o Sistema Operacional Ubuntu 18.04 que conta com processador AMD EPYC 7451, memória RAM de 128GB DDR3 e disco rígido de 1TB.

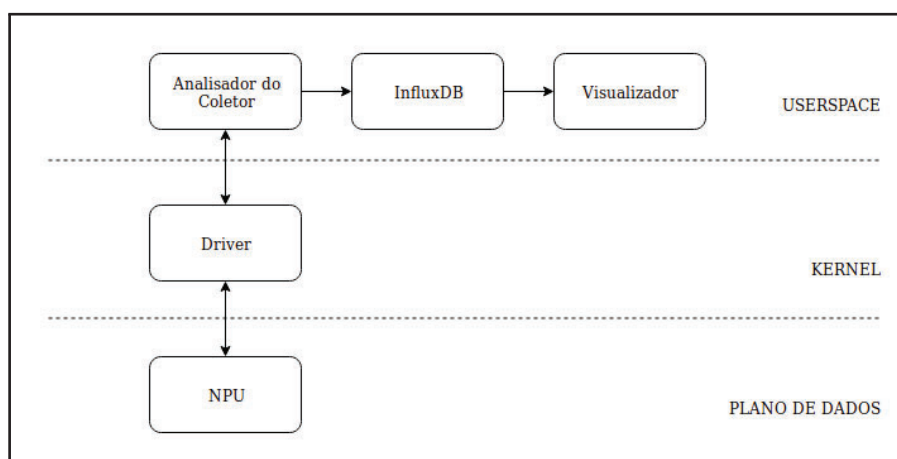


Figura 15. Arquitetura interna do Visual-INT. Fonte: Autor.

3.2.3.1 Coletor INT

A fase de extração dos dados desempenha um papel central na arquitetura interna do Visual-INT. Esta institui-se, consoante a Figura 16, de dois programas principais, sendo o principal o Coletor implementado em hardware (*ebpf_prog_kern.c*), que processa os pacotes INT advindos da rede P4, além do Analisador do Coletor empregue no *userspace* (*ebpf_prog_user.c*), responsável pela organização e transferência das informações apuradas no hardware para o banco de dados. Recapitulando os conceitos da seção 2.3.1, vê-se que a organização faz uso tanto de um mapa, cujo fluxo de entrada parte do Analisador

do Coletor para dentro da placa, quanto de um *buffer perf_event*, que comunica os dados processados na SmartNIC de forma contínua para o *userspace*.

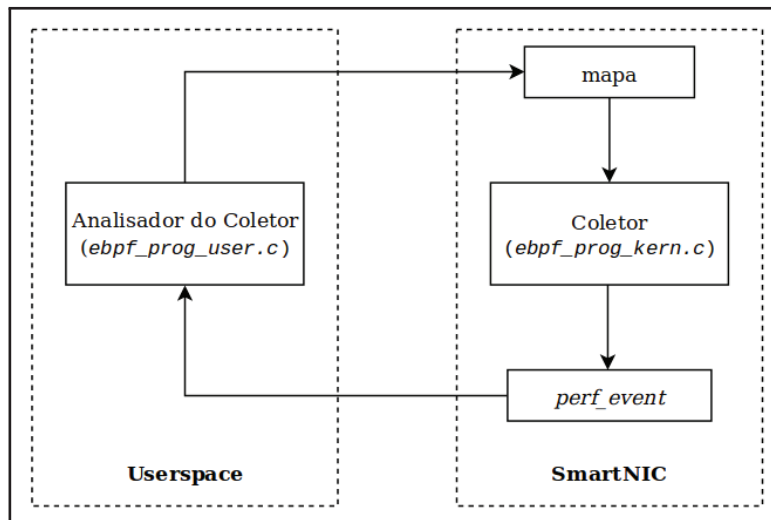


Figura 16. Divisão dos programas no coletor do Visual-INT. Fonte: Autor.

Seguindo a ordem de implementação, descreve-se primeiramente o Coletor (*ebpf_prog_kern.c*), um programa eBPF para extração de metadados INT introduzido dentro do hardware mediante a técnica de *offloading* (citada na seção 3.2.2.1). A Figura 17 ilustra a forma como o processo de *offloading* eBPF ocorre na arquitetura NFP (*Netronome Flow Processor*) dentro da pilha de rede do Linux. Por padrão, o programa eBPF (escrito em C) é anexado ao *hook* (gancho) XDP. Em seguida, esta mesma infraestrutura (de acordo com a Figura 18) compila o código C (*ebpf_prog_kern.c*) em objeto eBPF (*ebpf_prog_kern.o*) por meio do Clang¹² (por sua vez amparado pelo LLVM¹³).

Este objeto é descarregado (*offloading*) para a SmartNIC mediante a tradução (e validação) do formato *bytecode* para o código de máquina do *driver* NFP (através do *NFP JIT*). O mapa também é descarregado nesta fase, aproveitando-se da capacidade de hardware de 2GB DRAM. Isto posto, percebe-se que as decisões fundamentais XDP de aceitar (*XDP_PASS*), descartar (*XDP_DROP*) ou redirecionar (*XDP_TX*) os pacotes provenientes da rede são efetuadas no mais baixo nível da pilha de rede, proporcionando celeridade no processamento e análise dos datagramas.

Uma vez descarregado no hardware, o programa Coletor realiza a análise de pacotes por meio de três níveis de *parsers*: Externo, Interno e INT. O grau inicial, compreendido como Externo, traz o exame de *frames Ethernet* encaminhados pelo *switch sink* da rede P4 (com MTU padrão de 1.500 *bytes*). Nesta etapa, o Coletor ainda desconhece se os pacotes que atingem sua porta física são ou não portadores do cabeçalho *report* (significando que possuem metadados INT em seu *payload*). Isso se deve ao fato de que a máquina coletora, por estar conectada ao meio físico, não recebe necessariamente apenas pacotes remetidos

¹²Compilador *front-end* para a linguagem C que permite a transposição do código para o eBPF.

¹³*Low Level Virtual Machine*: Infraestrutura *back-end* utilizada para instruções em microprocessadores.

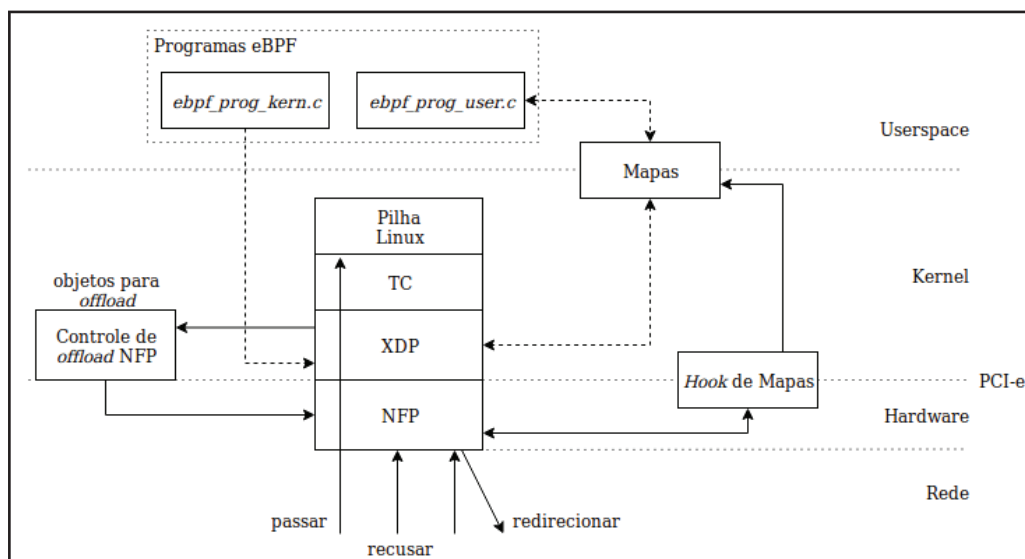


Figura 17. Arquitetura NFP para *offload* de objetos eBPF. Fonte: Adaptado de KICINSKI, J.; VILJOEN, N. [45].

pele *sink*. Comunicações de diversas camadas e *hosts*, como ICMP (*ping*) e TCP, podem concorrer aos recursos da SmartNIC.

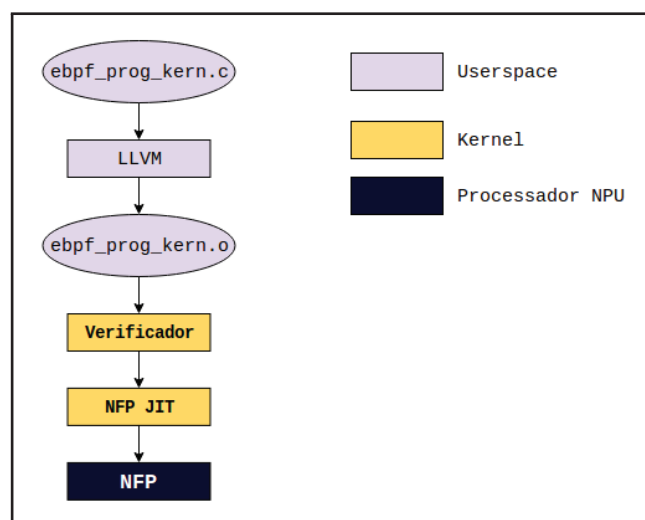


Figura 18. Modelo de programação eBPF na SmartNIC. Fonte: Adaptado de VILJOEN, N. [33].

Diante disso, o *parser* Externo determina se o pacote contém um *report* por intermédio de testes condicionais. Descrito na Figura 19 (A), a ação eventual de aceite do pacote (*XDP_PASS*) só ocorre se o tipo de protocolo do cabeçalho *Ethernet* for igual ao IPv4 (0x800 em hexadecimal). De forma análoga, os cabeçalhos IPv4 (B) e UDP (C) apenas acederão às camadas superiores da pilha se o protocolo de transporte do primeiro for o UDP (0x17 em hexadecimal) e caso a porta de destino do último seja igual a da aplicação coletora (32766). Encapsulado, o cabeçalho *report* somente é determinado ao ser aceito

em todos os estágios retratados. Por padrão, as ocorrências omissas são descartadas (*XDP_DROP*).

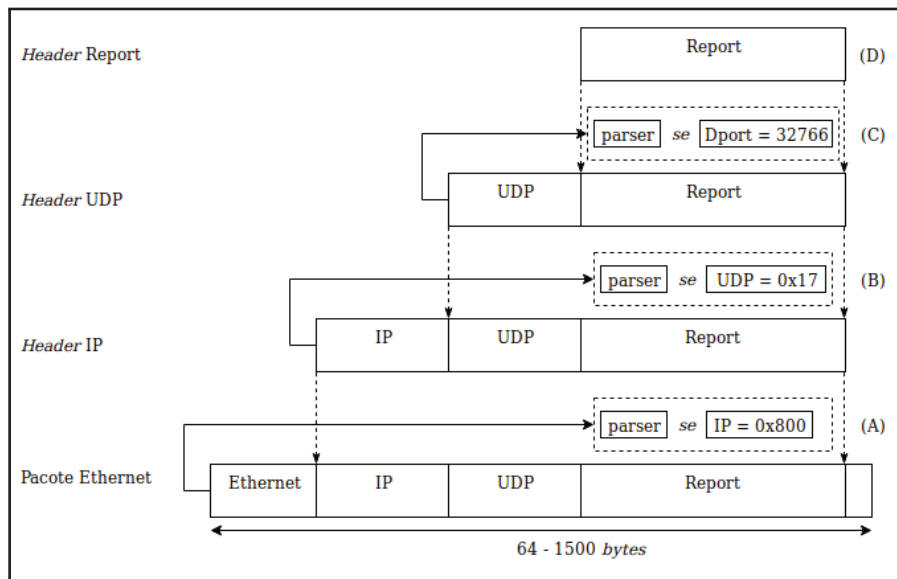


Figura 19. *Parser* Externo do Coletor do Visual-INT. Fonte: Autor.

A seguir, o Coletor está apto para analisar os dados no segundo nível. O *parser* Interno não contém somente os metadados INT, mas informações auxiliares (de acordo com o exposto na seção 3.2.2.3) relativas ao estado intrínseco do *sink*, como o *timestamp* de entrada neste, importante para se determinar a latência do equipamento. Na Figura 20, o cursor XDP expande seu alcance enquanto forem satisfeitas as condições referentes ao *Ethernet* (A) e IPv4 (B). Neste último estão as três primeiras informações da quintupla IP (protocolo e IPs de origem e destino), que serão aproveitadas mais adiante para estabelecer o caminho do fluxo.

Ao avançar (C), o *parser* utiliza-se do cabeçalho modificado (*Portas*) contendo as portas de origem e destino. Assume-se que estas podem estar tanto no formato UDP quanto no TCP, portanto, é necessária a verificação do tipo do protocolo de transporte para o correto deslocamento dos *bits* (em virtude da diferença de *bytes*). Por último, tem-se os metadados de telemetria em si (D), organizados conforme o cabeçalho INT (demonstrado anteriormente na Figura 14). Subsequentemente, o próximo *parser* incumbe-se de acessá-los.

O último *parser*, reconhecido como *parser* INT, é o mais engenhoso e envolve-se diretamente com os parâmetros de telemetria praticados. A Tabela 4 sumariza as métricas INT implementadas, contendo a ordem de instrução do *bitmap* (apresentado na seção 3.2.2.3) depreendida do 0 ao 7 (totalizando 8 métricas possíveis), assim como uma breve descrição seguida do respectivo tamanho em *bits*. Por motivos técnicos, como limitação do hardware das placas (discutido em detalhes na seção 5.2), três destes parâmetros são desconsiderados, como os IDs da fila e ocupação, a ocupação e congestionamento da fila e a taxa de utilização da porta de saída.

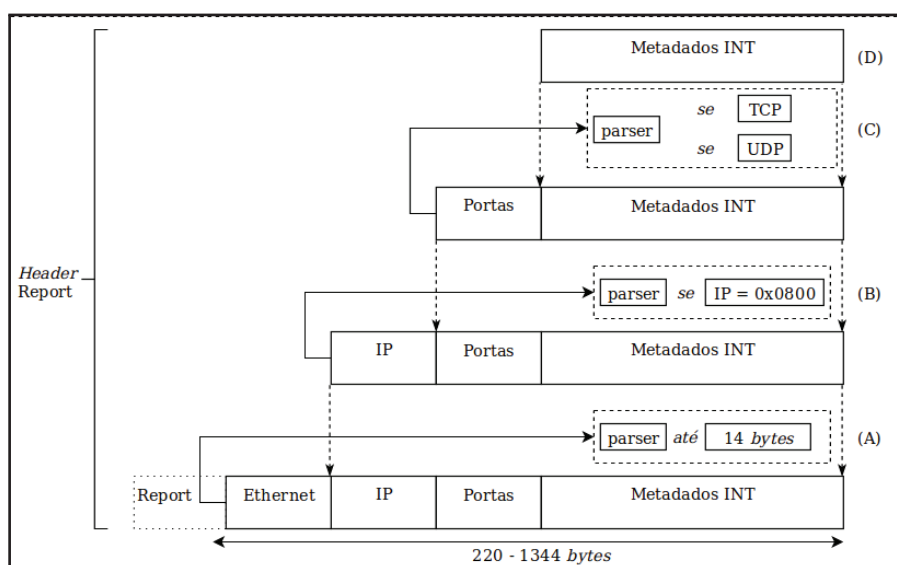


Figura 20. *Parser* Interno do Coletor do Visual-INT. Fonte: Autor.

Com base nas métricas apresentadas, o Coletor consome a análise dos metadados INT em concordância com o apresentado na Figura 21. Os dados dentro do coletor são lidos em forma de pilha, sendo temporariamente armazenados na `struct_INT_data`. Acompanhando o disposto na Tabela 4, o primeiro dado (ID do *switch*) é extraído pela função *parse 1*. Invocado por esta função de desreferenciação de ponteiros, o `xdp_md` comporta-se tal qual um compasso que mede onde começa e termina cada tipo de informação, atentando-se para o tamanho de *bits* passado como parâmetro (que varia entre 16, 32 e 64). Seguindo a mesma mecânica, as funções *parses* imediatas (2, 3, 4 e 5) coletam os IDs da porta de entrada (*ingress*) e saída (*egress*), a latência do salto (*hop*) e os *timestamps* das respectivas portas de entrada e saída.

Tabela 4. Lista das métricas INT adotadas. Fonte: Autor.

| MÉTRICAS INT | | |
|-----------------------|---|------------------------|
| Posição do <i>bit</i> | Descrição | Tamanho em <i>bits</i> |
| 0 | ID do <i>switch</i> | 32 |
| 1 | ID da porta de entrada e ID da porta de saída | 16 + 16 |
| 2 | Latência do salto | 64 (nanossegundos) |
| 3 | ID da fila e ID da ocupação da fila | Não implementado |
| 4 | <i>Timestamp</i> de entrada | 64 (nanossegundos) |
| 5 | <i>Timestamp</i> de saída | 64 (nanossegundos) |
| 6 | Ocupação e congestionamento da fila | Não implementado |
| 7 | Utilização da porta de saída | Não implementado |

Ao final deste processo, resta enviar os dados capturados para o Analisador do Coletor (`ebpf_prog_user.c`) no *userspace*. Amparados pelo `perf_event`, a comunicação entre o programa inserido na placa e o programa no espaço do usuário começa a acontecer. Para isso, no entanto, é necessária a definição de uma estrutura comum para a portabilidade dos dados, que se dá mediante a `struct INT_metadados`, apresentada na Figura 22.

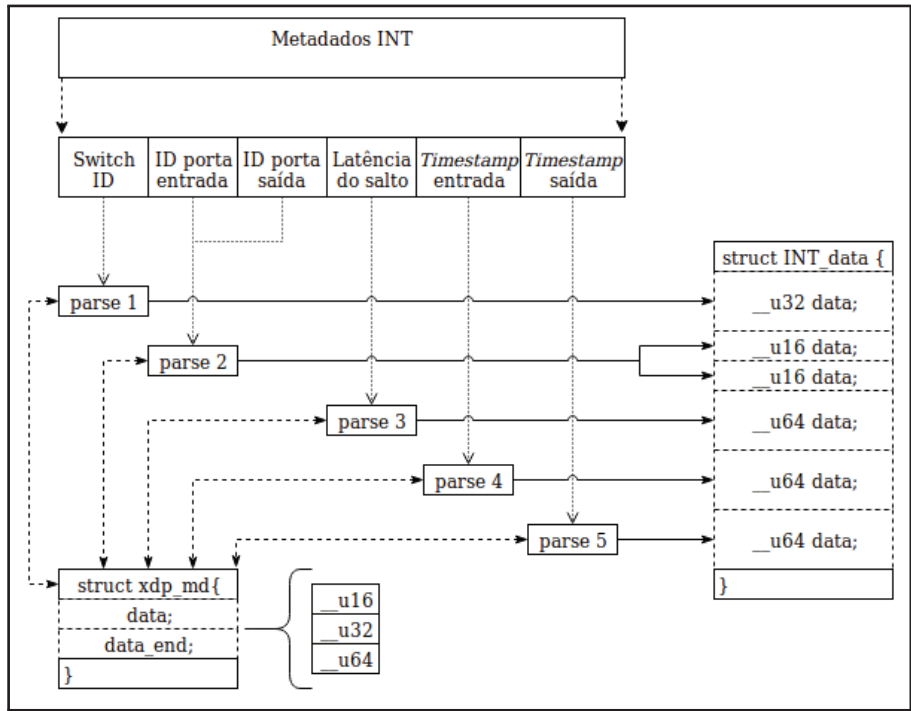


Figura 21. *Parser INT* do Coletor do Visual-INT. Fonte: Autor.

Abrangendo informações coletadas ao longo das séries de *parsers*, bem como dados de controle formulados pelo Coletor, o *INT_metadados* chega ao *userspace* em formato hexadecimal. Como primeira atitude, o Analisador do Coletor converte as informações recebidas (*casting*) para então significá-las.

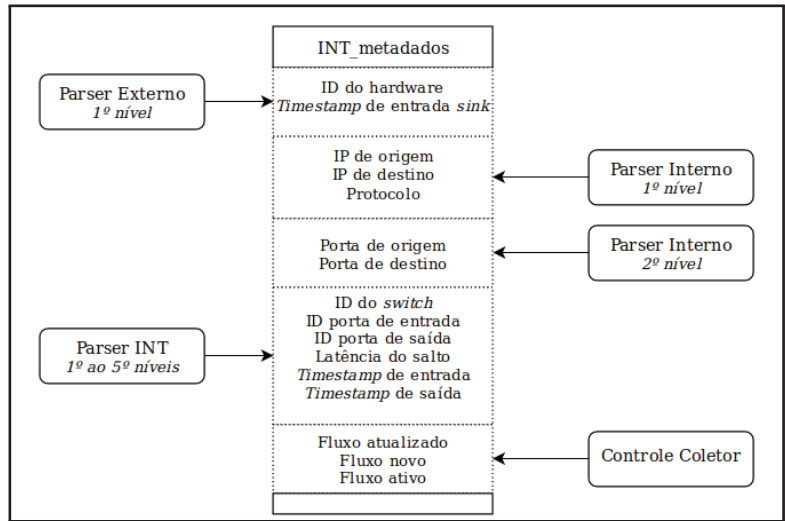


Figura 22. *Struct* enviada para o *userspace* pelo *perf_event*. Fonte: Autor.

Esta significação ocorre por meio da composição das métricas entre si, que produz o que denomina-se de medição. Sem esta, os dados coletados, ainda que humanamente inteligíveis, não possuem sentido completo para inferir-se acerca do estado da rede. De caráter complementar, a Tabela 5 abarca a relação entre as métricas implementadas e as medições exequíveis a partir destas. Por meio dos cinco parâmetros extraídos é possível

determinar o caminho do fluxo, ou seja, por quais equipamentos os pacotes de um dado fluxo (mesma origem e destino) passaram, tal qual o tempo de permanência (latência) despendidos por estes no respectivo dispositivo.

Tabela 5. Relação entre as métricas usadas e as medições praticáveis. Fonte: Autor.

| MÉTRICAS | |
|---|---|
| Tipo | Descrição |
| Identificação | ID do <i>switch</i> |
| | IDs das portas de entrada/saída |
| Tempo | <i>Timestamp</i> da latência do salto |
| | <i>Timestamps</i> portas de entrada/saída |
| MEDIÇÕES * | |
| Quíntupla IP + ID do <i>switch</i> + IDs portas | Caminho do fluxo |
| Quíntupla IP + ID do <i>switch</i> + <i>timestamp</i> salto | Latência do fluxo por salto |

**Combinação de métricas.*

Ademais, o Analisador do Coletor é responsável por organizar e persistir as informações significadas para o banco de dados InfluxDB (em formato JSON), por meio de uma API que se utiliza do método POST. Além disso, o programa do *userspace* também é encarregado de computar as *flags* de controle (fluxo novo, ativo e atualizado) e de executar as ações pertinentes que, na maioria das vezes, culminam na atualização de mapas (uma característica não tão bem implementada no hardware das SmartNICs).

Embora seja capaz de deletar e incluir novos mapas (que exercem um controle interno dos registro de fluxos), o Coletor (*ebpf_prog_kern.c*) tem seu desempenho comprometido ao lidar com modificações contínuas nessas estruturas (detalhes na seção 5.2), necessitando da ajuda do programa no *userspace* para isto. Através da biblioteca *libbpf.h*, por onde é buscado o mapa descarregado na placa a fim de adicionar novas informações computadas no *userspace*, o Analisador do Coletor coopera para a assertividade dos dados nos processos de coleta executados no hardware.

3.2.3.2 InfluxDB

O InfluxDB [46] é um banco de dados *open source* de séries temporais (*Time Series Database*) escrito na linguagem Go, possuindo diversas extensões e suporte a várias linguagens de programação clientes. Especializado em perpetrar históricos, o InfluxDB permite que seu *timestamp* seja modificado, alcançado a precisão em nanossegundos¹⁴ pela representação de 64 *bits*, além de oportunizar o encaminhamento de dados pelo método *push*, ou seja, o envio de informações a critério do cliente (neste caso em específico, o Analisador do Coletor). A baixa granularidade de tempo propicia a identificação de ocorrências

¹⁴Unidade comum em telecomunicações, sendo 96 nanossegundos o tempo mínimo de pausa entre pacotes *Gigabit Ethernet*, por exemplo.

(chamadas de *measurements*) muito mais apuradas do que as técnicas de *sampling* (geralmente, com representações de 32 *bits*). Para exemplificar a exibição das informações de tempo, a Tabela 6 traz um compêndio das grandezas de *timestamp* comumente utilizadas em telecomunicações e sua interpretação usual referentes a uma data específica.

Tabela 6. Panorama da Era (*Epoch*) UNIX quanto à granularidade. Fonte: Autor.

| ERA UNIX (<i>UNIX EPOCH</i>) | | | |
|--------------------------------|----------------|----------------------|--|
| Precisão | Tamanho | <i>Timestamp</i> | Representação |
| Segundos (s)* | 32 <i>bits</i> | 1561545824 | 26/06/19 - 07:43:44 |
| Milissegundos (ms)** | 41 <i>bits</i> | 1561545824,000 | 26/06/19 - 07hrs, 43min, 44s e 0ms |
| Nanossegundos (ns) | 64 <i>bits</i> | 1561545824,000000000 | 26/06/19 - 07hrs, 43min, 44s, 0ms, 0μs e 0ns |

* 1 segundo é igual a 1.000.000.000 de nanossegundos.

** 1 milissegundo é a 1.000.000 de nanossegundos.

3.2.3.3 Visualizador INT

O visualizador INT integra a última parte na significação dos metadados. Sua motivação dá-se pela ausência de ferramentas que exponham, dinamicamente, os dados recuperados dentro do contexto INT empregue. Deste modo, sua aplicação apenas é satisfeita ao completar-se, com êxito, os passos anteriormente descritos (a saber a coleta, o processamento e o armazenamento de metadados INT).

Todavia, reconhece-se a existência de trabalhos substanciais sobre a visualização aplicada à SDN [47]. Neste aspecto, a própria tarefa de visualização em si figura-se como um campo de pesquisa amplo e inovador, estando fora do escopo desta proposta. Embora este trabalho faça uso de alguns dos conceitos clássicos, o mesmo não visa promover uma implementação focada nos aspectos de usabilidade ou até mesmo uma análise completa dos métodos de visualização.

Nisto, a visualização do Visual-INT ampara-se em um cenário clássico, de acordo com a Figura 23, onde o dado coletado primeiramente é transferido para tabelas (neste caso ao ser preservado no banco de dados), sendo estas tabelas posteriormente manipuladas e transformadas em outras formas de representação (como na vez em que os dados são resgatados e retratados via JavaScript).

Para facilitar a análise das condições de rede e a tomada de decisões gerenciáveis com base no monitoramento dos fluxos, o Visual-INT, por meio de APIs (atestada pela Figura 24), recupera as informações salvas no banco de dados (por meio de *queries* customizadas à ferramenta Telegraf do InfluxDB) via *middleware* e as transmite ao usuário, de maneira gráfica, por meio de páginas dinâmicas em JavaScript hospedadas em um servidor NodeJS.

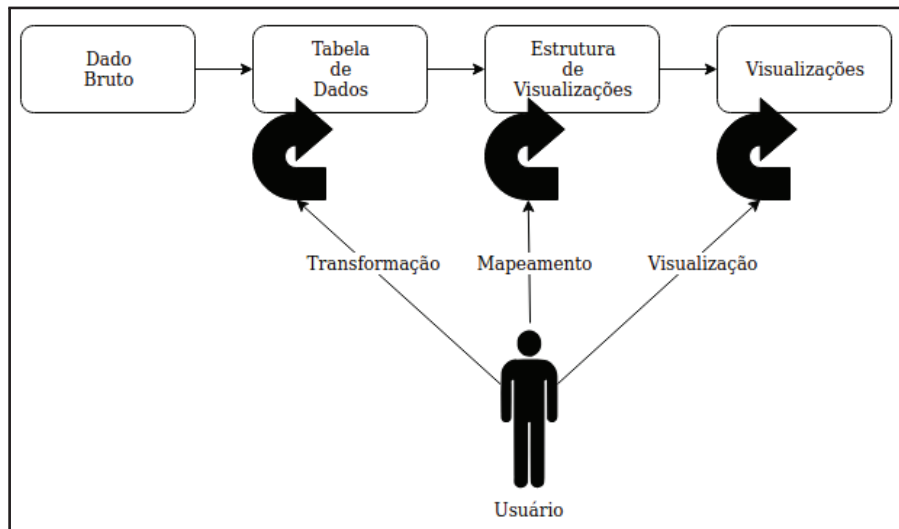


Figura 23. Modelo de visualização clássica. Fonte: Adaptado de CARD, S. et al. [48].

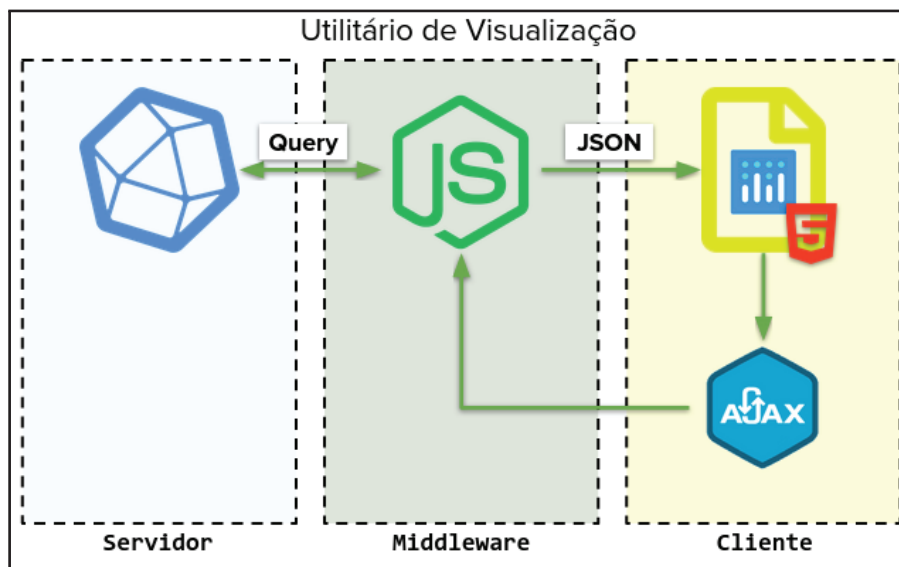


Figura 24. Modelo de visualização clássica. Fonte: Autor.

Com foco na representação do caminho dos pacotes e na latência dos equipamentos, buscou-se uma diagramação que melhor satisfizesse esses aspectos, com o uso de esquemas no estilo Sankey, um tipo de diagrama no qual a largura das setas é proporcional à taxa de fluxo, além de ferramentas tradicionais para concepção de séries temporais como o uso do *plotly.js* [49].

Referenciado na Figura 25, o gráfico Sankey, bastante utilizado na ilustração de fluxos e redes, proporciona uma aferição da granularidade mais externa, ou seja, o caminho do fluxo (por quais equipamentos os pacotes daquele fluxo adentraram) e sua respectiva latência total, enquanto o gráfico em *plotly.js* representa uma granularidade mais interna, onde a latência de cada equipamento demonstrado no gráfico anterior pode ser explorada de forma mais minuciosa.

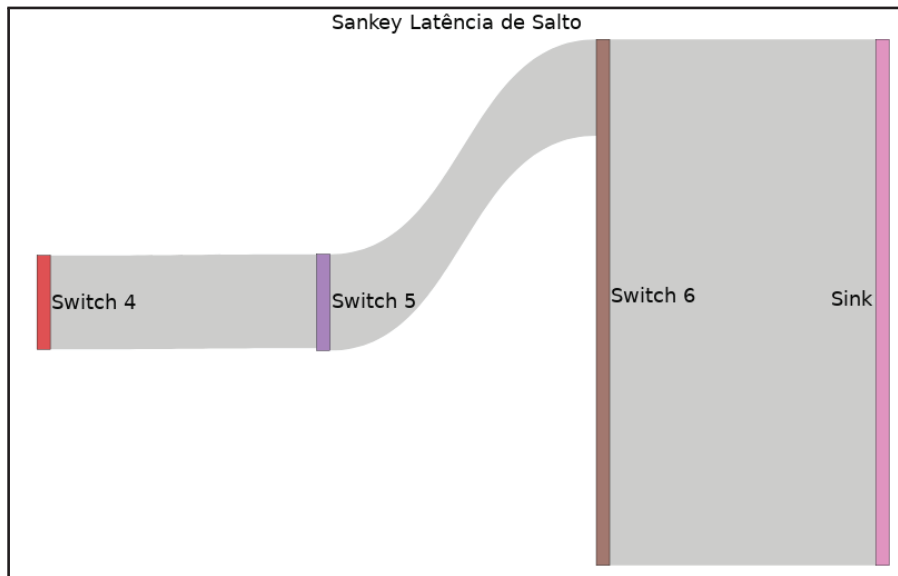


Figura 25. Sankey atestando o caminho do fluxo e a latência por *hop*. Fonte: Autor.

Nativamente, o *plotly.js*, presente na Figura 26, permite o uso de recursos de aproximação e seleção dos dados de forma interativa e cronológica, o que facilita a análise do tempo de permanência dos pacotes em cada ativo observado.

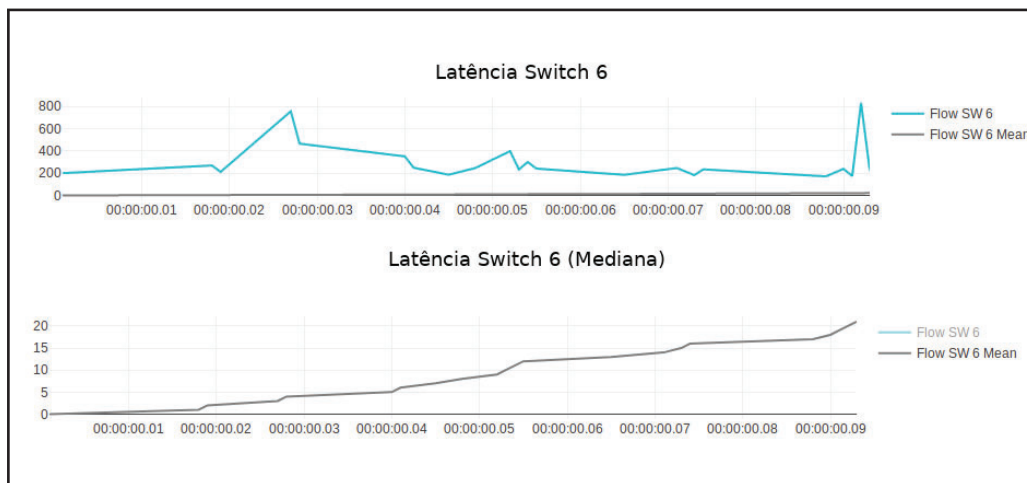


Figura 26. *Plotly.js* contendo a latência e sua mediana para um dispositivo. Fonte: Autor.

4. APRESENTAÇÃO E DISCUSSÃO DE RESULTADOS

O presente capítulo discorre sobre a validação da ferramenta Visual-INT em três estágios, abrangendo desta forma os principais elementos que compõe a aplicação, como o coletor em hardware (seção 4.1), o armazenamento dos dados *time series* no InfluxDB (seção 4.2) e o utilitário de visualização de metadados (seção 4.3). Ao final do capítulo, os resultados obtidos são discutidos (seção 4.4) face às finalidades e funcionalidades estabelecidas para este trabalho, bem como os aspectos que o diferenciam das demais propostas relacionadas.

4.1 VALIDAÇÃO DO COLETOR

A validação do coletor per fez dois grupos de testes, executados dentro do ambiente real anteriormente descrito (seção 3.2.2.2), estando estes separados entre os que priorizam apenas o tráfego INT proveniente dos *switches* P4 (seção 4.1.1) e os que formalizam o trânsito de dados INT simulados (seção 4.1.2). Respectivamente, tanto este quanto aquele intentam demonstrar a assertividade e a capacidade de processamento dos metadados INT, quer pela análise de fluxos produzidos unicamente pelo meio físico, quer pela geração artificial de pacotes que exploram cenários distintos. Ambos utilizam-se de duas formas de aferição: a primeira via *kernel*, em que o XDP posiciona-se no *driver* do SO, e a segunda pelo método *offload*, onde o programa eBPF é descarregado para dentro da SmartNIC.

Esta abordagem bimodal (*kernel* e *offload*) justifica-se pela impossibilidade de comparação direta do presente trabalho (elaborado para o hardware) com as demais metodologias adotadas em trabalhos semelhantes, visto que, dentre outras coisas, estas aderem a ambientes virtualizados. Contudo, o modo *kernel* compreende um descarregamento eBPF comum a maioria das ferramentas de análise que operam o XDP, sendo seu contraste com o método *offload* o principal meio de diferenciação desta proposta.

Totalizando 720 amostras (sendo 30 iterações para cada tipo de teste) a execução das experimentações instrumentalizou-se sob um conjunto comum de ferramentas, como o `stat_watch.py` [50], um programa Python provido pela Netronome que facilita a leitura das saídas do comando Linux `ethtool` (tais como o *status* das interfaces de entrada e saída do *host* e da própria SmartNIC), o comando nativo Linux `mpstat`, responsável por recuperar as informações de média do uso da CPU, assim como o gerenciador de terminal XDOtool [51], empregado na sincronização dos testes em multiterminais.

Anterior ao início dos testes, isolou-se o comportamento do uso de CPU da máquina coletora baseando-se nos seus índices em modo de repouso (*idle time*). Também,

foram paralisados quaisquer *daemons* ou serviços não essenciais para o funcionamento do servidor. Com isso, os valores calculados consideraram apenas a carga de processamento inerente às atividades de coleta, processamento e armazenamento dos metadados INT pelo coletor. De igual modo, garantiu-se, por exames de vazão, carga e fim-a-fim, que nenhuma outra interferência externa viesse a comprometer os resultados apurados (tais quais defeitos no meio físico, má configuração dos *switches*, perda de pacotes, dentre outros). Ainda, para certificar a validade das amostras, foi realizado o teste T de Student ¹⁵ para todas as médias depreendidas, onde, por teste estatístico, refutou-se a hipótese nula de significância entre as médias, em outras palavras, se a diferença dos resultados produzidos ocorreram por mero acaso.

4.1.1 Testes com tráfego obtido do INT real

A fim de aferir-se a aptidão na computação de pacotes *report* advindos do ambiente P4 real, considerou-se nas avaliações que se seguem a relação da quantidade de pacotes que atingem a interface da SmartNIC, a porção de pacotes aceitos e negados (marcados como *XDP_PASS* ou *XDP_DROP*) pelo programa eBPF/XDP, a fração de metadados enviados ao Analisador do Coletor e processados por ele (levando-se em conta o uso de CPU da máquina coletora para tal) e, por conseguinte, a série de entradas no banco de dados InfluxDB.

Os dois primeiros parâmetros objetivam estabelecer a razão entre a capacidade de *throughput* da interface de entrada e a proporção que os cabeçalhos *report* (encapsulados dentro do tráfego passante) são reconhecidos. A próxima métrica corresponde ao poder de processamento das informações no Analisador do Coletor no *userspace*, determinado pelo seu consumo de CPU. Nestes moldes, cada fluxo analisado corresponde a uma entrada no banco de dados, pressupondo-se, em termos de armazenamento, uma *naive approach* (abordagem ingênua, externada na seção 4.2.1), útil na comparação de resultados dos níveis antecedentes nesta primeira fase. De antemão, salienta-se o emprego de outras abordagens para tratamento dos dados em série, como a detecção de eventos (seção 4.2.2).

4.1.1.1 Teste 1: Processamento de tráfego INT/P4

Para a realização do teste com tráfego INT real, necessitou-se criar um fluxo constante entre os dois *hosts* conectados aos *switches* P4, conforme organização demonstrada previamente na Figura 12. A partir desta movimentação, geraram-se (nas SmartNICs da rede paralela) os pacotes *report* enviados para a interface de coleta (SmartNIC do coletor).

¹⁵Teste que compara duas médias e demonstra se as diferenças entre estas são significativas.

Devido ao tamanho constante de cada *report* (cerca de 178 *bytes*), a taxa de transferência entre o *switch sink* e o coletor estabilizou-se próxima aos 60Kbps para este tipo de pacote.

Com isso, executou-se as verificações de recebimento de dados INT reais pelos métodos *kernel* e *offload*. Por ambientar-se no meio físico, os resultados obtidos apresentaram pequenas variações entre si. Para garantir a confiabilidade das margens trabalhadas, os testes adequaram-se ao Intervalo de Confiança [52] (IC), que define os valores máximo e mínimo assumidos pela amostra de dados dentro de um nível de confiança percentual estabelecido, neste caso de 95%. Neste íterim, assume-se o índice 1.960 (equivalente ao nível de confiança de 95%), conforme a Equação 1, onde \bar{x} é média da amostra, σ é o desvio padrão ¹⁶ e n é o tamanho da amostra:

$$IC = \bar{x} \mp 1.960(\sigma)(\sqrt{n}) \quad (1)$$

Com base na fórmula exposta, denota-se na Tabela 7 o intervalo das médias alcançadas pelos modos *offload* e *kernel*, considerando um IC de 95%, tanto para a quantidade de *reports* processados quanto para o uso de CPU. A partir destas informações, pode-se inferir que em apenas 5% dos casos o valor do uso de CPU não estará no intermédio entre 0.46 e 0.52 para o método *kernel* e de 0.47 e 0.52 para a técnica *offload*. O mesmo é verdade para a parcela de pacotes INT processados em ambos os modos, cujos intervalos mínimo e máximo (compreendidos dentro de 2.739~2.759 e 2.788~2.789, na devida ordem), se somados, diferenciam-se em pouco mais de 20 pacotes (cerca de 0.7% sobre o maior valor).

Tabela 7. Médias obtidas com IC de 95% para o teste somente com INT. Fonte: Autor.

| Modo | Uso de CPU | Nro pacotes INT |
|---------|-------------|-----------------|
| Kernel | 0.52 ± 0.46 | 2.789 ± 2.759 |
| Offload | 0.52 ± 0.47 | 2.788 ± 2.739 |

Além disso, algumas das métricas descritas (como a carga de CPU) nos dois níveis de teste (*kernel* e *offload*) atenderam às funções de Densidade de Probabilidade (FDP) e de Distribuição Acumulada (FDA). A FDP julga, a partir de uma entrada amostral em sua função, a expectativa P de um variável aleatória X se repetir dentro de certo valor apontado x . Por sua vez, a FDA demonstra, por meio da saída de sua função (considerando-se também uma dada amostragem), a concentração de valores distribuída probabilisticamente entre todos os resultados obtidos, em outras palavras, a possibilidade cumulativa P dos valores x ocorrerem, em algum ponto, igual ou abaixo de determinado valor X .

Os dados amostrais, por sua avaliação empírica, elegeram-se à demonstração por intermédio destas duas funções (que são correlatas) ao enquadrarem-se nas três condições comuns a ambas, a saber a característica de serem estritamente crescentes ($f(x_n) \leq$

¹⁶Índice de dispersão da amostra em relação à média.

$f(x_{n-2})$), a não-assunção de valores negativos ($\neg f(-\infty)$) e a não-extrapolação do limite de 1 ($0 \leq P(x) \leq 1$), também entendido como a barreira dos 100%.

A concomitância entre FDP e FDA pode ser descrita como $\int f(x) = F(x)$ (ou ainda $\int FDP = FDA$). Por exemplo, ao assumir-se X como uma variável aleatória contínua e x como um valor presente na amostra, designam-se $f(x)$ e $F(x)$ como suas FDP e FDA, conforme exposto nas Equações 2 e 3 (sendo s a variável *dummy* ¹⁷). Consubstanciando a Equação 2, a área sob a qual a função de densidade (d) opera é constituída da probabilidade de X estar entre ∞ e x . Deste modo, pode-se afirmar, amparando-se na Equação 3, que os valores cumulativos desta área são representados pela FDA, ou seja, o valor de certo ponto x na curva do gráfico FDA corresponde a área deste mesmo ponto x no diagrama FDP.

$$P(X \leq x) = \int_{-\infty}^x f(x)dx \quad (2) \quad F(X) = P(X \leq x) = \int_{-\infty}^x f(s)ds \quad (3)$$

As Equações 4 e 5 aplicam os conceitos das funções acima, utilizando-se para isso da distribuição normal ¹⁸. Nisto, além do parâmetro x , as funções também contam com a média (μ) e o desvio padrão (σ), sendo σ^2 a variância ¹⁹ e e a base do logaritmo natural.

$$f(x|\mu\sigma) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}, \sigma > 0 \quad (4) \quad F(x) = \int_{-\infty}^x \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(s-\mu)^2}{2\sigma^2}} ds, \sigma > 0 \quad (5)$$

Aplicando-se o FDP ao consumo de CPU no tratamento *kernel*, observa-se o apresentado na Figura 27 (A), onde um histograma evidencia a distribuição das probabilidades dos valores x . Em azul, destaca-se ao centro da imagem a linha contendo a distribuição normal. Notou-se que o coletor em modo *kernel* apresenta uma tendência final inclinada positivamente, divergindo de um resultado típico esperado (curva de sino) da distribuição normal. Esse comportamento determina que os picos mais altos de processamento ocorrem em seu extremo (à direita), posicionando a pontuação média à direita do pico. Sendo assim, sua média torna-se maior que a mediana, que por sua vez supera a moda (pontuação mais exibida). Possivelmente, se desconsiderados esses picos de consumo, os elementos seguiriam a distribuição normal.

Distintivamente, a Figura 27 (B) ilustra uma distribuição normal mesocúrtica e simétrica para a técnica *offload*, aproximando-se das expectativas da distribuição normal.

¹⁷Variável binária fictícia (assumindo apenas 0 e 1), utilizada em análises de regressão.

¹⁸Denominada Gaussiana, é a distribuição contínua mais conhecida dada sua frequência em fenômenos naturais.

¹⁹Medida de dispersão que mostra o quão longe cada valor está da média.

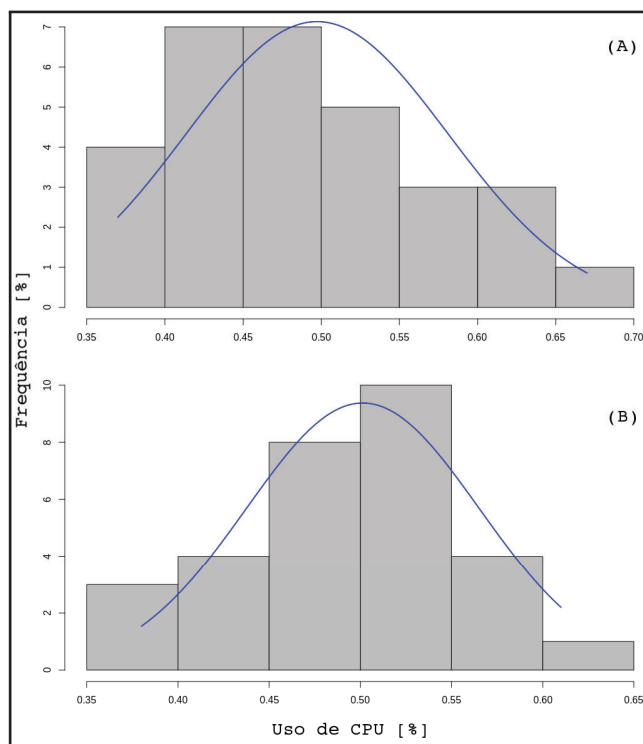


Figura 27. FDP acerca do uso de CPU nos modos *kernel* (A) e *offload* (B) para o teste somente com INT. Fonte: Autor.

Isso indica que a média, a mediana e a moda estão muito próximos uns dos outros, o que evidencia um comportamento mais estável (sem picos de consumo) em *offload*. Considerando-se que a mediana do número de pacotes *report* recebidos por ambos os modos é idêntica (2.782), infere-se que a assimetria constatada na Figura 27 (A) seja produto de certa incapacidade do *driver* do *kernel* de lidar com as repostas desejadas para aplicação (que processa 45 medições por segundo), ainda que com baixo nível de exigência (dada a exiguidade de pacotes gerados a uma taxa de 60Kbps).

Em paralelo, as Figuras 28 (A) e (B) trazem o FDA gerado a partir da probabilidade calculada anteriormente. Percebe-se que o gráfico do *kernel* (A) comporta-se igualmente ao histograma prévio, onde a mediana (linha vertical preta) aproxima-se do quadrante negativo, tendo a média a sua direita (0.50) e a moda à esquerda (0.38). A normalização, em linha azul, afasta-se das medições (em linha vermelha) à proporção que o gráfico caminha para sua finalização (alcance dos 100%). Ao contrário da técnica progressa, o método *offload* intersecciona a linha da distribuição normal em vários momentos, além de possuir um formato em "S", comum às distribuições Gaussianas. Observa-se ainda que o começo da queda cumulativa coincide com o fim da média e da mediana (0.50 e 0.51).

Por fim, a Figura 29 representa os dados sobre o uso da CPU e suas respectivas médias para o *kernel* e *offload* dentro do IC apresentado na Tabela 7. Estipulando-se o erro padrão ²⁰, concluiu-se a relevância das amostras dada as margens apresentadas, cuja

²⁰Medida que atesta a confiabilidade da média calculada para um grupo amostral.

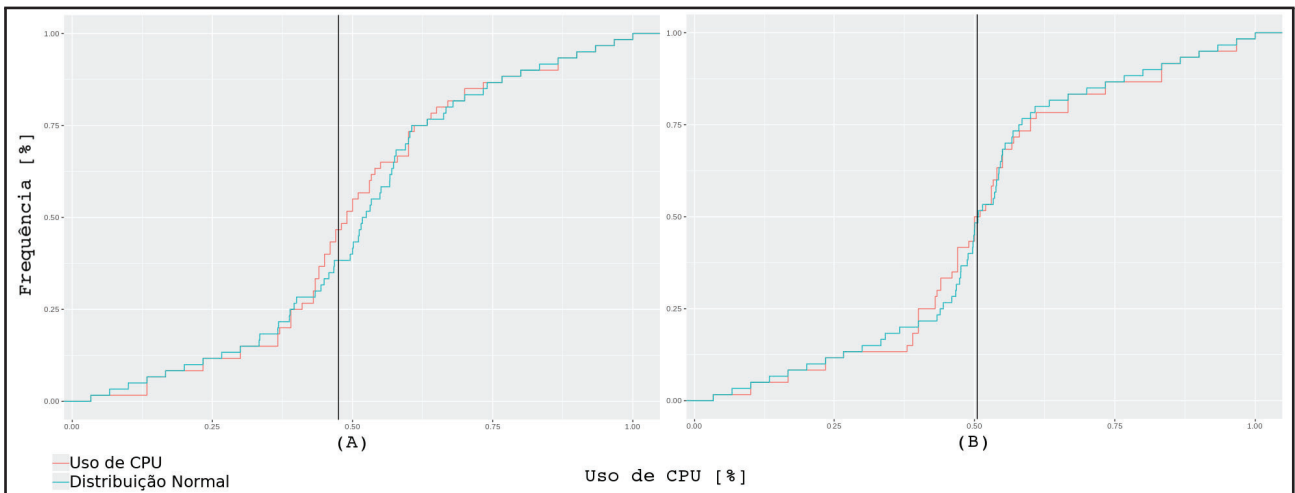


Figura 28. FDA acerca do uso de CPU nos modos *kernel* (A) e *offload* (B) para o teste somente com INT. Fonte: Autor.

eventual repetição retornará índices aproximados. Averiguou-se que o tratamento *kernel*, apesar de possuir uma ligeira vantagem em relação à média de processamento (0.48 para 0.50), acaba por exceder o desvio padrão (0.8 para 0.6) da modalidade *offload*.

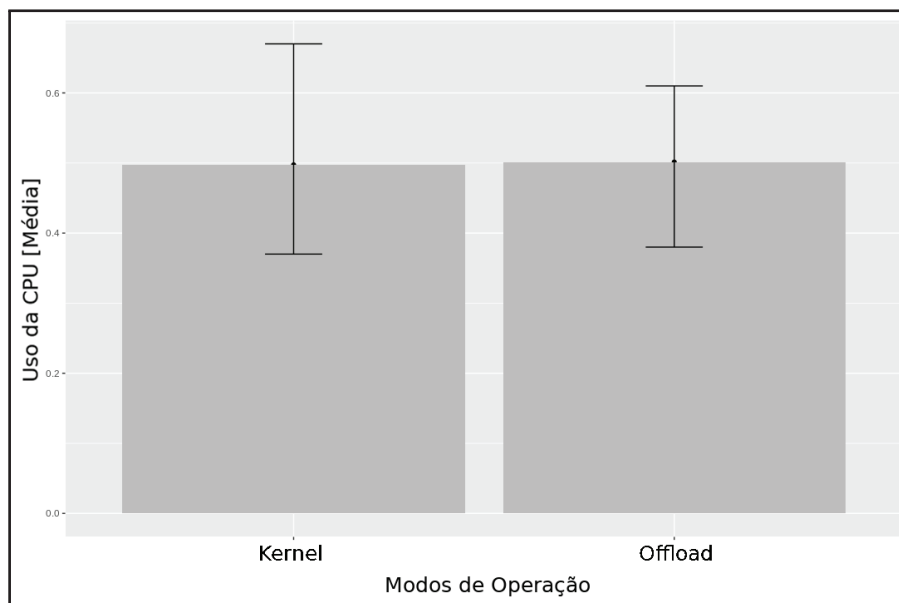


Figura 29. Comparativo entre médias de CPU e erro padrão (*kernel* e *offload*) para o teste somente com INT. Fonte: Autor.

Com números similares, constatou-se uma pequena vantagem do modo *kernel* (no que refere-se ao uso de CPU) ante o modo *offload* em um fluxo contínuo contendo apenas o tráfego INT real a uma taxa de 60Kbps. Apesar disso, as medições constataram um comportamento desviante no processamento realizado dentro do SO (modo *kernel*). Tanto a computação de pacotes quanto a análise das informações acaba por consumir recursos em comum (processador da máquina coletora), justificando-se nisto os picos de utilização registrados. A técnica *offload*, no que lhe concerne, emprega o processamento dos pacotes

no plano de dados, o que libera o processamento para outras tarefas (como a análise dos metadados). Esta característica de desoneração propicia um comportamento mais estável para aplicação, uma vez que, ao não lidar diretamente com o fluxo, estabelece-se uma constância nas demandas enviadas para o *userpace*.

4.1.1.2 Teste 2: Processamento de tráfego INT/P4 com concorrência

Não obstante sua utilidade, o teste inicial desconsidera a presença de outros fluxos concorrentes no meio físico e, conseqüentemente, torna-se insuficiente para estabelecer o impacto destes na extração e verificação de metadados. A concomitância com uma fonte externa que não se utiliza da semântica INT (programada na aplicação) torna-se um teste válido de aferição das capacidades, posto que a detecção de pacotes no mais baixo nível é uma das propostas do Visual-INT.

Valendo-se da ferramenta *iPerf* [53], gerou-se um tráfego TCP paralelo a uma taxa aproximada de 3,4Gbps. Os fluxos INT provenientes do *switch sink* também disputaram o mesmo meio. No programa eBPF/XDP, configurou-se o aceite do tráfego INT e a passagem de qualquer outro pacote distinto para a pilha Linux (*XDP_PASS*). Com isso, houve uma sobrecarga em relação ao processamento em ambos os métodos. O objetivo deste exercício foi determinar se há perda de pacotes à medida em que o programa eBPF passa a gerenciar o que deve aceder ou não aos estratos mais altos da pilha TC (*Traffic Control*). Neste sentido, ambas as abordagens proveram a passagem dos pacotes e a comunicação básica sem perdas com o servidor *iPerf* no *userspace*.

A Tabela 8 traz a relação das médias referentes ao consumo de CPU, número de pacotes INT recebidos e a quantidade de pacotes TCP encaminhados para a pilha Linux (através da *flag XDP_PASS*) provenientes do *iPerf*, considerando-se um IC de 95%. Diferentemente do teste anterior, distinguiu-se nesta abordagem uma carga de processamento menor para o modo *offload*. Novamente, ao possuir medianas muito próximas (2812 para *kernel* e 2814 para *offload*), a parcela de pacotes INT registrados equivalem-se em ambos os tratamentos. No entanto, no quesito de taxa de entrega de pacotes ao *userspace* a técnica de *offload* demonstrou menor desvio padrão (70.000 contra 225.000 pacotes), caracterizando-se como mais homogênea.

Tabela 8. Médias obtidas com IC de 95% para o teste INT com *iPerf*. Fonte: Autor.

| Modo | Uso de CPU | Nro pacotes INT | Nro pacotes aceitos |
|---------|-------------|-----------------|-------------------------|
| Kernel | 1.34 ± 1.28 | 2.826 ± 2.744 | 16.698.163 ± 16.507.679 |
| Offload | 1.21 ± 1.13 | 2.815 ± 2.789 | 16.527.458 ± 16.473.617 |

Tal como na verificação preliminar, as Figuras 30(A) e (B) sustentam padrões de uso de CPU quase idênticos. Considerando-se o FDA, o padrão de recuo negativo da mediana se manteve para o modo *kernel*, assim como sua centralização para a técnica de

offload. No mais, a distribuição normal obteve mais pontos de convergência para o *offload* do que para o *kernel*, constituindo-se uma característica reiterada.

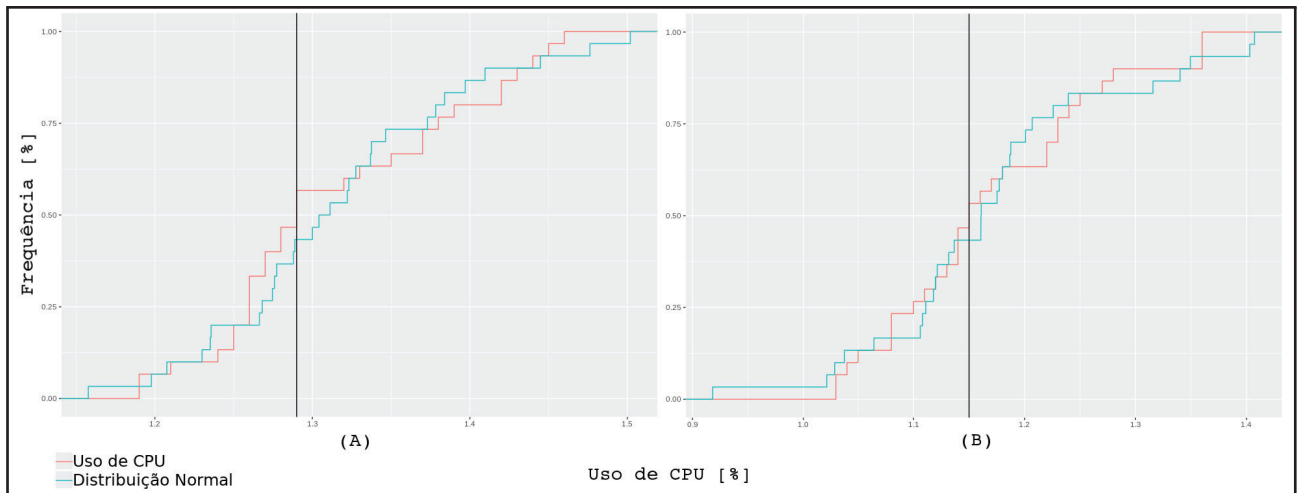


Figura 30. FDA acerca do uso de CPU nos modos *kernel* (A) e *offload* (B) para o teste INT com *iPerf*. Fonte: Autor.

A Figura 31 representa o erro padrão e a média entre os métodos considerando o IC de 95% mediante um gráfico de barra de erros. As pequenas margens qualificam a análise análise do ponto de vista amostral, solidificando o mesmo comportamento para uma população maior. Em relação ao teste decorrido, o fluxo extra representou um aumento de 171% para a técnica *kernel* e 127% para o método *offload*. Neste íterim, o descarregamento eBPF na placa (*offload*) suavizou em cerca de 12% a carga de CPU imposta pelo *kernel*.

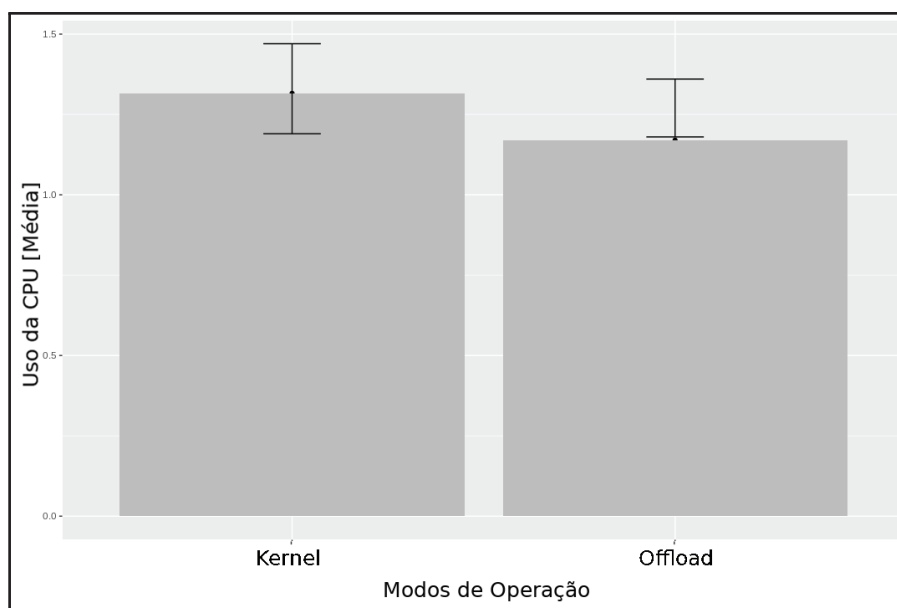


Figura 31. Comparativo entre médias de CPU e erro padrão (modos *kernel* e *offload*) para o teste INT com *iPerf*. Fonte: Autor.

Com esta avaliação, conclui-se que o programa eBPF/XDP tem capacidade para encaminhar pacotes ao *userspace* de outra natureza, não estando restrito ao mero reconhecimento de cabeçalhos *report* contendo metadados INT. Outro ponto importante fora a execução desta tarefa sem o desencadeamento de uma quebra de comunicação entre serviços de conectividade TCP, por exemplo. Comprovou-se também que a análise de fluxos paralelos de alta vazão impactam o consumo de CPU, todavia, esta apresenta certo grau de atenuidade ao ser desempenhada pelo hardware. Essa diferença justifica-se, novamente, pela não utilização dos recursos da máquina coletora no processamento de pacotes que atingem o meio físico.

4.1.1.3 Teste 3: Processamento INT e bloqueio total de tráfego

O seguinte teste visou a avaliação da capacidade de bloqueio de tráfego não resultante da computação INT dos *switches* P4. Para isso, aumentou-se o fluxo paralelo para 4,4Gbps através de uma comunicação unilateral UDP (dispensando o formato cliente-servidor do *iPerf*). Com uma vazão maior, esta validação buscou consolidar a forma como o programa eBPF/XDP lida com a *flag XDP_DROP* em situações de estresse e concorrência.

A Tabela 9 contempla um compêndio das médias auferidas para uso de CPU, quantidade de pacotes INT processados e o número de pacotes rejeitados considerando-se um IC de 95%. Repetidamente, o método *offload* apresentou uma média de carga de processamento abaixo da técnica *kernel*. Em relação à computação dos *reports*, houve um desempenho consideravelmente superior do modo *offload* em relação ao *kernel*, em oposição ao equilíbrio expresso nos testes anteriores (expondo neste medianas de 2816 e 2767, respectivamente). Embora apresente IC e desvio padrão menores, a porção de pacotes bloqueados pelo programa descarregado no hardware (*offload*) em proporção à aplicação em *kernel* foi equivalente, exibindo uma diferença de pouco mais de 0,16% entre ambos.

Tabela 9. Médias obtidas com IC de 95% para o teste INT com bloqueio total. Fonte: Autor.

| Modo | Uso de CPU | Nro pacotes INT | Nro pacotes negados |
|---------|-------------|-----------------|-------------------------|
| Kernel | 0.61 ± 0.54 | 2.688 ± 2.456 | 29.110.645 ± 27.185.316 |
| Offload | 0.44 ± 0.43 | 2.821 ± 2.806 | 28.335.214 ± 28.051.130 |

A Figura 27 (A) apresenta o histograma da distribuição do uso de CPU para o método *kernel* pelo FDP. Diferentemente do primeiro teste (seção 4.1.1.1), apresentou-se desta vez uma linha mesocúrtica próxima da distribuição normal, o que evidencia a ausência de picos de processamento ou *outliers*²¹. A Figura 27 (B), por sua vez, versa sobre o FDP do *offload*, que figurou-se pela primeira vez com as linhas de distribuição próximas ao eixo

²¹Valor atípico, afastado dos demais da série.

x do começo ao fim, uma característica desejável que enseja uma distribuição probabilística que dirime resultados extremos (*fat tail*).

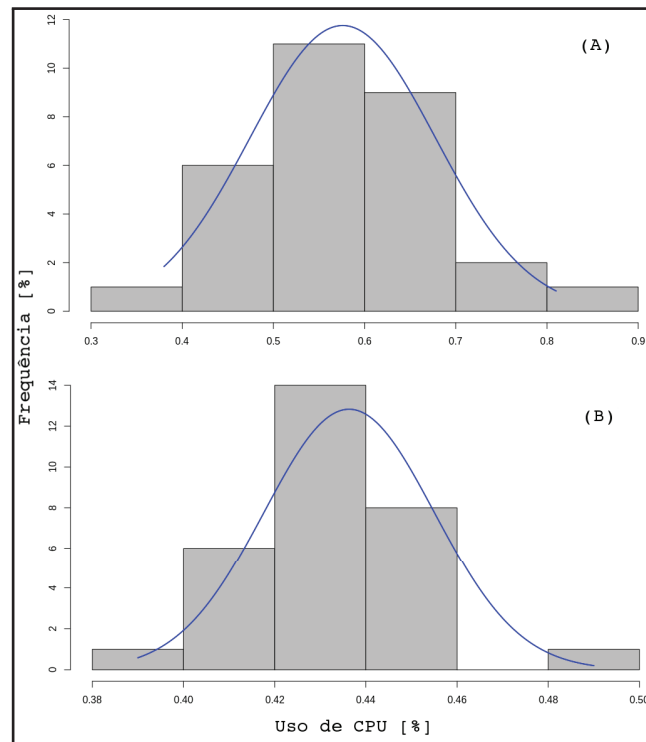


Figura 32. FDP acerca do uso de CPU nos modos *kernel* (A) e *offload* (B) para o teste INT com bloqueio total. Fonte: Autor.

As Figuras 28 (A) e (B) apontam o FDA para ambas as técnicas *kernel* e *offload*, nesta ordem. Como esperado, o traçado da função para o *kernel* é retratado no formato em "S", uma vez que alguns pontos da área de distribuição da probabilidade interseccionam a linha da distribuição normal. No mais, observa-se a equivalência entre a média e a mediana (0.58), além de baixo desvio padrão (0.10), o que propicia a queda da probabilidade logo após a passagem pela mediana (listra vertical preta). Para a técnica *offload*, os traços referentes ao consumo de CPU e à distribuição normal encurtaram-se devido a concisa extensão da série (detalhada na Tabela 9), que estabilizou seu crescimento ao ponto 0.43, quando mais de 90% dos resultados compreenderam uma faixa abaixo deste valor.

Por último, o erro padrão e a média do uso de CPU são demonstrados pela Figura 34. Notou-se que, pela pouca variabilidade, o método *offload* apresentou uma menor margem de erro, enquanto o modo *kernel* exibiu uma maior amplitude nos pontos distantes da média. Além disso, a técnica *offload* fora executada utilizando-se de uma fração muito menor de processamento, cerca de 32% a menos em relação ao tratamento *kernel*. Considerando-se o primeiro teste (seção 4.1.1.1), a aplicação em *offload* obteve uma melhora na performance, percebida em torno de 13%. Em contrapartida, a execução no *kernel* degradou-se em quase 16%.

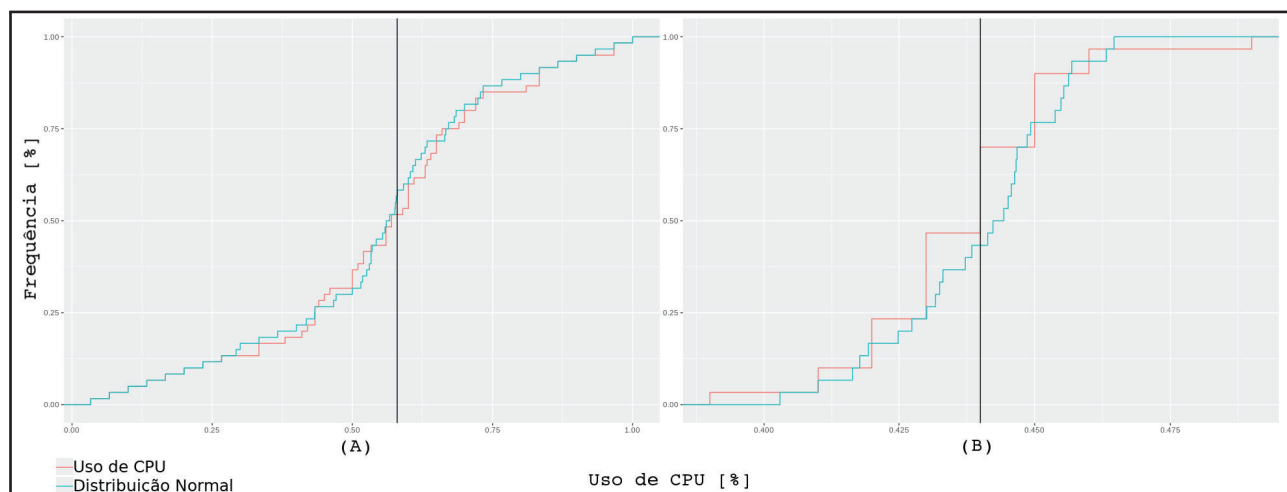


Figura 33. FDA acerca do uso de CPU nos modos *kernel* (A) e *offload* (B) para o teste INT com bloqueio total. Fonte: Autor.

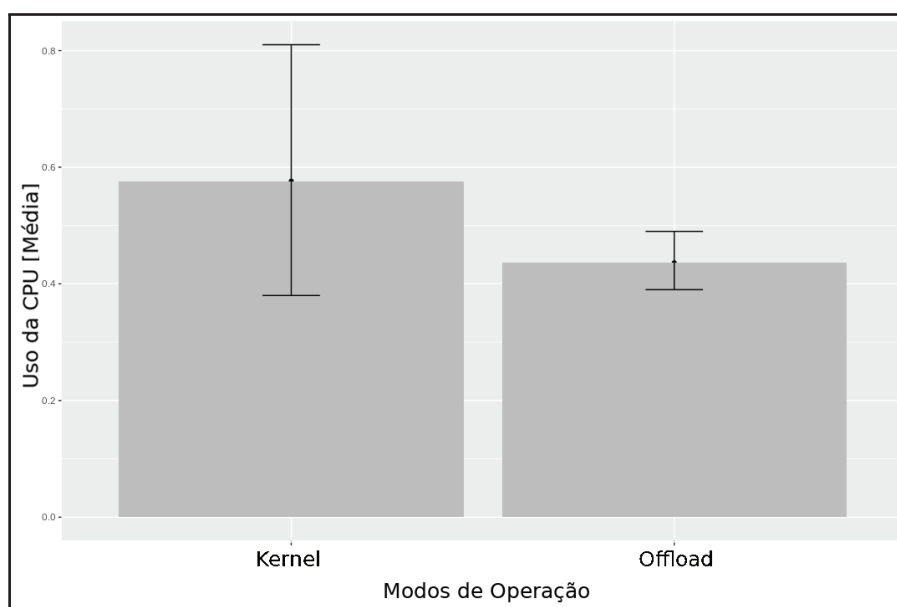


Figura 34. Comparativo entre médias de CPU e erro padrão (modos *kernel* e *offload* para o teste INT com bloqueio total. Fonte: Autor.

Desta avaliação, atesta-se que, quando aplicada a execução eBPF/XDP em modo *kernel*, o processamento de pacotes exerce uma influência maior nos recursos disponíveis, dado o compartilhamento dos meios entre o *driver* do SO e o programa no *userspace*. Como esperado, o emprego do *offload* reduziu este consumo, uma vez que as decisões de *drop* são tomadas a nível de hardware, permitindo que somente as informações relacionadas à análise de metadados INT proceda para o ambiente do usuário.

4.1.1.4 Avaliação dos testes com tráfego obtido do INT real

Esta seção busca sintetizar, através de um parecer comparativo, pontos relacionados ao comportamento apresentado pela aplicação Visual-INT nos testes com tráfego real realizados de forma bimodal (*kernel* e *offload*). A Figura 35 ilustra, por meio de um *boxplot*, o contraste entre o consumo de CPU nos três tipos de teste, a saber a avaliação ponderando apenas tráfego de metadados dos *switches* (INT), o experimento considerando o tráfego INT com concorrência TCP (INT+*iPerf*) e *flooding* UDP com o bloqueio ativo (INT+*block*), observando-se os dois modos de carregamento do programa eBPF/XDP. O gráfico atenta para o uso de *jittering*, pequenos pontos que abstraem o arredondamento das medidas, proporcionando uma leitura mais clara dos dados.

Organizada em ordem decrescente, a Figura 35 conceitua os quartis (percentils ²² de 25%, 50% e 75%) de cada amostra. Destacadas pelo traçado horizontal em cada caixa estão as medianas obtidas. As linhas verticais (*whiskers*) denominam o intervalo entre o maior e o menor valor (abaixo e abaixo da caixa, respectivamente) desconsiderando os *outliers* (pontos negros abaixo dos *whiskers*). Notadamente, vê-se que o modo *offload* angariou valores de carga de processamento inferiores ao tratamento *kernel* em todos os testes. Disto, o diagrama assente uma dispersão interquartílica (diferença entre os primeiros 25% contra os últimos 75%) para as validações INT, INT+*iPerf* e INT+*block* de 0.09, 0.13 e 0.02 no modo *offload*, e de 0.11, 0.12 e 0.14 no modo *kernel*. No geral, observa-se uma menor variabilidade no modo *offload*, cujos intervalos compreendem 0.23, 0.18 e 0.10 contra 0.30, 0.27 e 0.43 da abordagem *kernel* para os testes INT, INT+*iPerf* e INT+*block*. Pelo *kernel*, em dois terços dos casos o terceiro quartil (75% dos valores) foi superior à mediana, enquanto que no *offload*, na mesma proporção, foi apresentada simetria em relação à mediana e igualdade entre o primeiro e terceiro quartis (25% e 75%).

Corroborando a análise anterior, a Figura 36 traz a média em relação ao desvio padrão do consumo de CPU para os testes INT, INT+*iPerf* e INT+*block*, ordenando-os quantitativamente do menor para o maior. Excluindo-se o experimento INT+*iPerf* (onde estabeleceu-se um índice de 0.10 contra 0.08), a técnica *offload* registrou desvios padrões menores do que a modalidade *kernel* nos testes INT (0.06 ante 0.08) e INT+*block* (0.02 ante 0.10). Em termos práticos, isto significa um comportamento mais homogêneo do método *offload*, onde ocorreu uma aglutinação mais elevada de valores próximos a média. Desta feita, o modo *kernel* sucedeu-se com valores mais distribuídos em torno da média, o que significa uma maior influência de fatores externos no desempenho da aplicação.

A Figura 37 aplica um gráfico *boxplot* que relaciona a porção de pacotes *report* percebidos em cada um dos testes. Reiteradamente, os resultados do método *offload* convergem para um comportamento mais estável, onde as medianas (cruz vermelha no centro

²²Medida de posição que indica o percentual de elementos que estão abaixo (ou no mesmo nível) de dado valor.

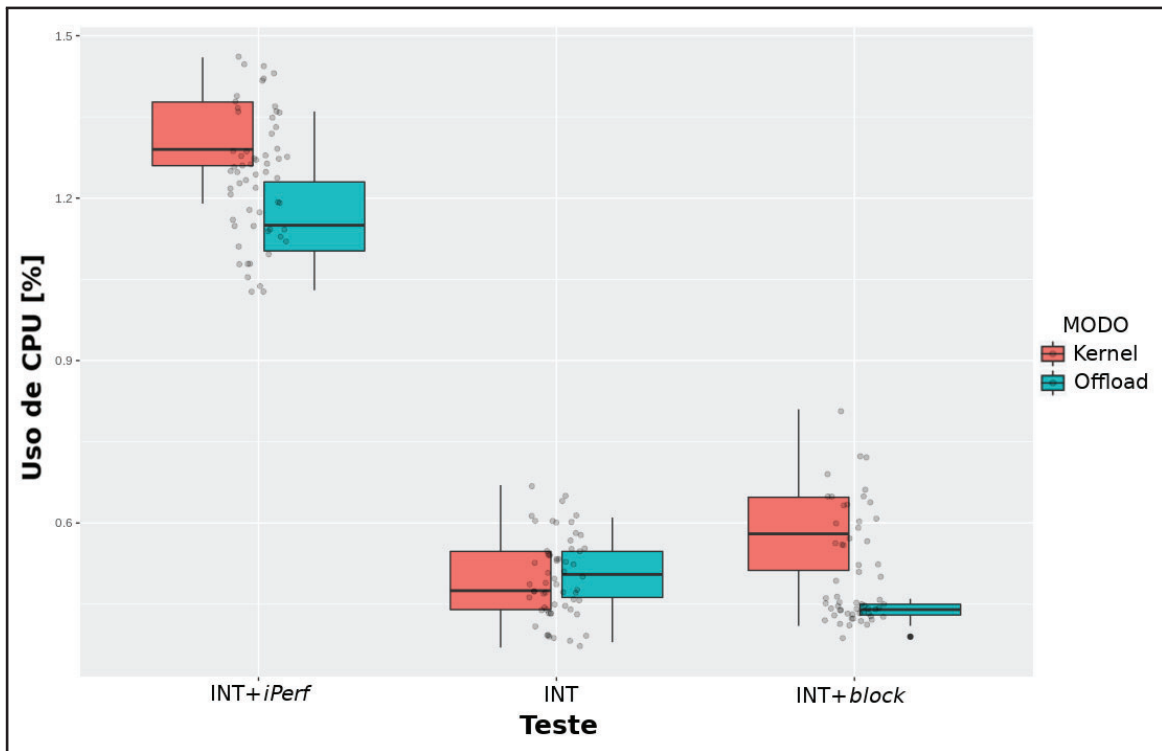


Figura 35. *Boxplot* estratificado entre as médias de CPU dos testes INT, INT+iPerf e INT+block. Fonte: Autor.

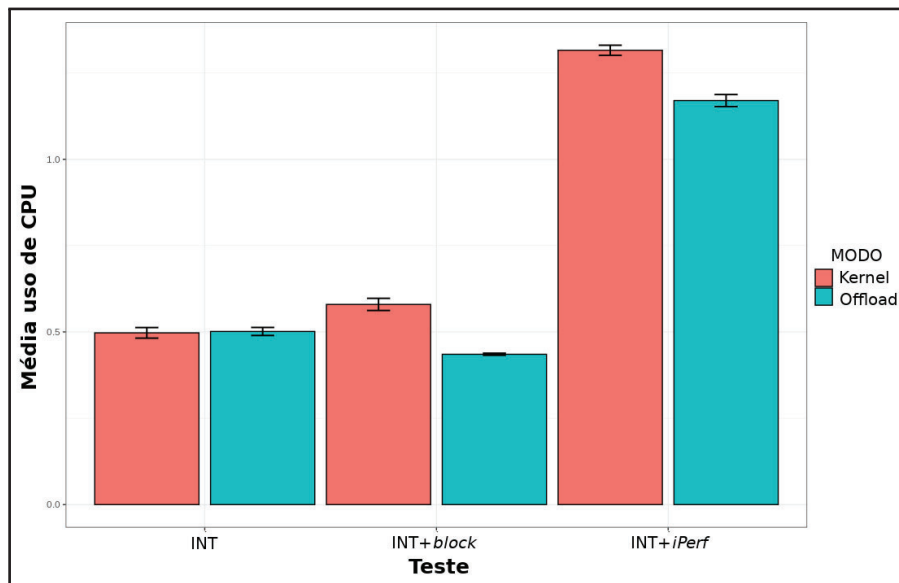


Figura 36. Média de uso de CPU e desvio padrão entre os testes INT, INT+iPerf e INT+block. Fonte: Autor.

da caixa), no geral, aproximam-se das médias (traçados em negrito no centro da caixa). Nas avaliações INT, INT+iPerf e INT+block, a dispersão interquartilica estabelece-se em 124, 43 e 18 para o modo *offload*, além de 72, 42 e 618 para o *kernel*, respectivamente. Ainda, o intervalo calculado (amplitude) compreende 204, 155 e 107 para o *offload* e 190, 624 e 731 para o *kernel*, significando uma variabilidade menor para o *offload*.

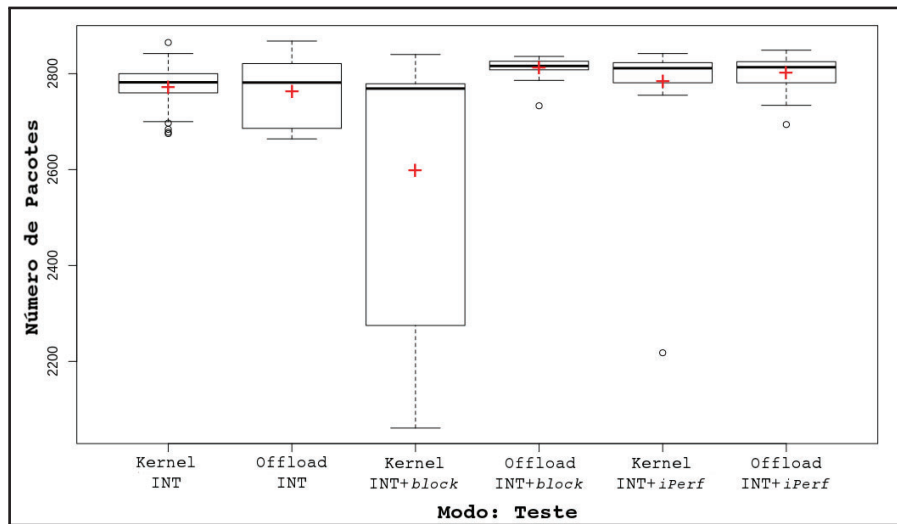


Figura 37. *Boxplot* estratificado entre as médias de pacotes *report* dos testes INT, INT+*iPerf* e INT+*block*. Fonte: Autor.

Ainda na Figura 37, nota-se um aumento significativo da caixa referente ao teste INT+*block* para o modo *kernel*, destoando dos demais experimentos. Isto deve-se à alta variabilidade apresentada pelo tratamento *kernel* frente à ativação da *flag* para *dropping* de pacotes (*XDP_DROP*) que, via de regra, tende a dobrar o processamento no nível XDP ²³. Como o modo *offload* realiza este procedimento na SmartNIC, averiguando o pacote imediatamente após sua análise condicional, o impacto desta computação não é sentida no *userspace*.

A Figura 38 elenca o desvio padrão para os pacotes INT processados. Nos testes, seguindo a ordem *kernel* e *offload*, constatou-se a presença de medianas idênticas no experimento INT (2.782) e aproximadas no INT+*iPerf* (2.812 e 2.814), ocorrendo uma quebra desta orientação no INT+*block* (2.767 e 2.816), com uma diferença de 2%. O desvio padrão também acompanha esse viés, ficando em 65, 35 e 20 e para o método *offload* e 47, 110 e 311 para o *kernel*, ou seja, uma diminuição média contínua de 44% para o primeiro modo e um acréscimo de 158% para o segundo. O mais díspar deles, a saber o teste INT+*block*, teve como desvio padrão quase o dobro do valor do teste anterior (INT+*iPerf*), extrapolando em cinco vezes o registro do primeiro teste (INT).

De acordo com a tendência demonstrada, as decisões tomadas a nível de *driver* do SO (técnica *kernel*) propendem para um maior consumo de recursos da máquina coleitora, enquanto as deliberações em hardware (método *offload*) dirigem-se à manutenção de seus patamares ao longo de seu funcionamento. A alta variabilidade para o modo *kernel* também incide para um comportamento instável, característica indesejável para os níveis de operação pretendidos.

²³Por ser uma instrução mais simples, *benchmarks* eBPF/XDP nativos atestam 20 Mpps para *dropping* e 10Mpps para *passing* [54].

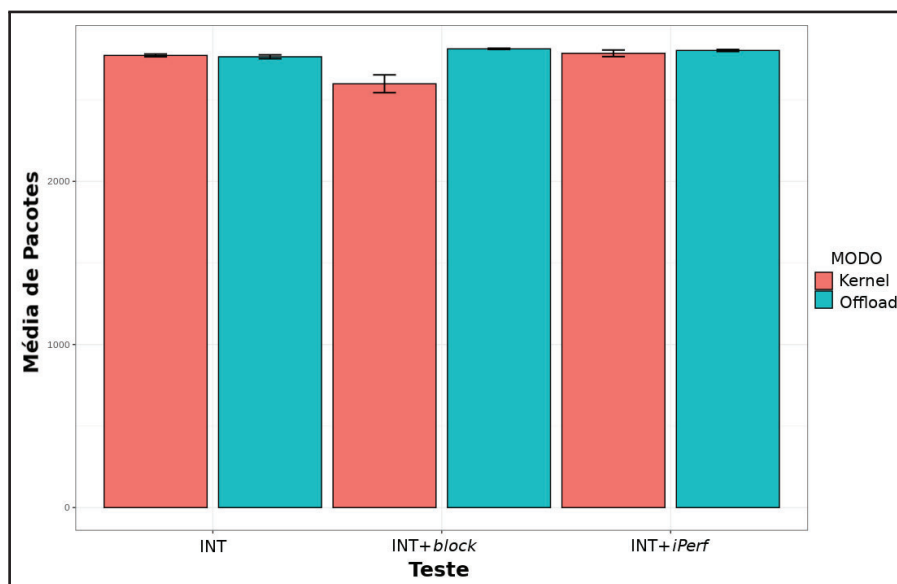


Figura 38. Média de pacotes *report* e desvio padrão entre os testes INT, INT+*iPerf* e INT+*block*. Fonte: Autor.

4.1.2 Testes com tráfego obtido do INT simulado

Explorando cenários diversificados, os testes seguintes empregam pacotes INT gerados artificialmente por meio da ferramenta Scapy [55]. Desenvolvido em Python, este utilitário permite a validação de estruturas por meio da construção artificial de pacotes de rede. Seguindo a ordem apresentada na Figura 39, um programa Python contendo as classes dos pacotes a ser gerados (neste caso, os cabeçalhos que compõem o datagrama a ser enviado) e bibliotecas Scapy é executado (1), produzindo, por sua vez, um arquivo PCAP²⁴ (2). Desta forma, por meio do programa de retransmissão TCPReplay [56], o arquivo PCAP contendo a simulacro dos pacotes é transmitido do Host 1 (3) ao coletor (4), transparecendo a este último como um tráfego advindo de uma rede de computadores real.

Em questão, os fluxos simulados contêm apenas as partes concernentes ao *report* INT, percebendo tráfego similar ao transmitido do *switch sink* ao coletor na rede P4 real. Assim, os pacotes gerados seguem a mesma estrutura dos pacotes INT originais, estando sujeitos a igual processamento por parte do coletor. Tal como nos testes com dados reais, os experimentos estruturam-se em duas formas de análises, a saber, o modo *kernel* e *offload*. As validações orientam-se principalmente pelo consumo de CPU e pela quantidade de métricas INT processadas. Disto, buscou-se determinar quanto (e como) o volume de fluxos de rede (seção 4.1.2.1), de *switches* no caminho (seção 4.1.2.2) e de metadados INT (seção 4.1.2.2) afetam o uso de CPU e, conseqüentemente, a capacidade de processamento do coletor do Visual-INT.

²⁴API comum a programas que lidam com captura de tráfego de redes.

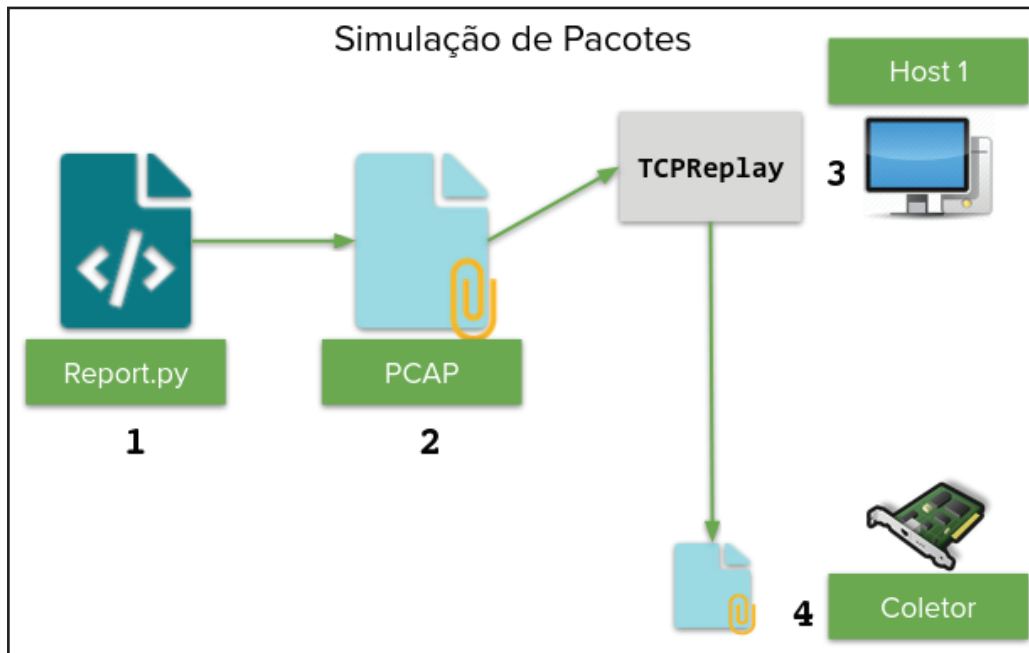


Figura 39. Fluxograma do teste com INT simulado. Fonte: Autor.

Os testes obedeceram a uma taxa contínua fixa de encaminhamento de *reports* de 7 Mpps (milhões de pacotes por segundo). Esta medida foi estabelecida ao considerar-se a taxa de encaminhamento, o número e a amplitude de cada porta (em *Gigabits*). Devido aos termos do próprio meio, pressupõe-se, para fins desta medição, a proporção de 10Gbps de tráfego entre as interfaces, que é a conectividade efetiva presente entre os *switches* e os *hosts* (conforme a Figura 11). Com isso, temos a composição da Equação 6:

$$Taxa \text{ em } Mpps = \frac{1.000.000.000bps \div 8bits}{178bytes} \times 1 \times 10 = 7Mpps \quad (6)$$

Onde 178 *bytes* correspondem ao valor fixado do cabeçalho *report*. Os números 1 e 10 descrevem a quantidade de portas utilizadas e o valor nominal passante (em Gbps).

4.1.2.1 Saturamento de fluxos de rede no Visual-INT

O fluxo de rede, entendido como a sequência de pacotes contendo a mesma origem e destino, é uma métrica importante para a análise de desempenho do coletor do Visual-INT, uma vez que esta constitui-se, dentro do modelo SDN, como a principal chave de valor a ser buscada no estabelecimento de regras pelas instâncias superiores (como o controlador SDN, por exemplo). Os testes anteriores, elaborados com tráfego originário dos *switches* P4, limitaram-se a apenas um único fluxo observável, sendo insuficientes para depreender o comportamento da aplicação em cenários com múltiplos fluxos.

Disto, elaborou-se testes com 10, 1.000 e 5.000 fluxos a uma taxa de 7Mpps, impondo-se a todos um caminho com seis *switches* passantes, valor máximo de empilha-

mento recomendado pelo modelo INT [11]. Destes processamentos, produziu-se, concernente ao consumo de CPU, o gráfico apresentado na Figura 40. Ao computar 10 fluxos, percebeu-se que a média, a mediana e o desvio padrão da carga de CPU para ambos os modos (*kernel*, na linha acinzentada e *offload* na amarelada) foi equivalente, fixando-se no valor de 1% para as duas primeiras métricas e 0,04 para a última.

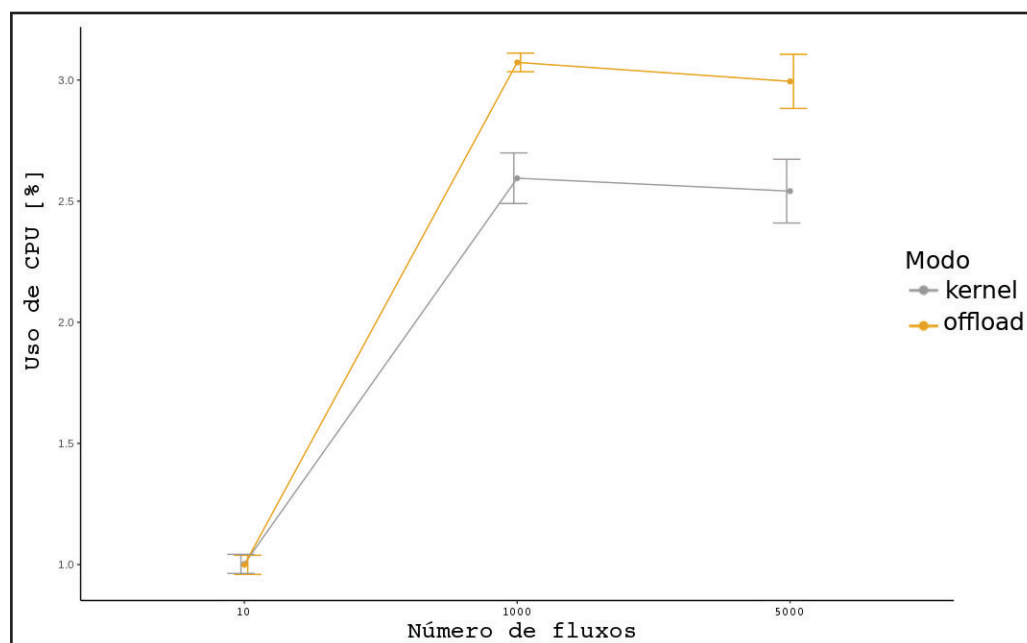


Figura 40. Uso de CPU mediante a quantidade de fluxos. Fonte: Autor.

Com 1.000 e 5.000 fluxos, porém, evidenciou-se um afastamento entre as duas métricas, chegando-se a uma diferença de 18% (de 2,56% a 3,07%) e 17% (de 2,54% a 2,99%) tanto para este quanto para aquele, respectivamente. Em compensação, o modo *offload* demonstrou um desvio padrão menor (0.04 e 0.11) em relação à técnica *kernel* (0.10 e 0.13).

Isolada, a alta no processamento do modo *offload* infere um desempenho piorado. No entanto, pela análise da Figura 41 no teste com 5.000 fluxos, constata-se que a quantidade de entradas persistidas no *userspace* por este supera em 34% (37.333 e 27.833) o método *kernel*, além de apresentar um desvio padrão 54% menor (9.166 a 14.184).

Disto, conclui-se uma eficiência maior do modo *offload* para quantidades mais significativas de fluxos. As baixas entradas no banco de dados realizadas pela técnica *kernel* mostram um descompasso no que tange à assertividade. Novamente, evidenciou-se uma inaptidão do *driver* do SO de executar, paralelamente, as funções de *parsing* de pacotes conjuntamente com as atribuições de verificação e persistência de dados no InfluxDB.

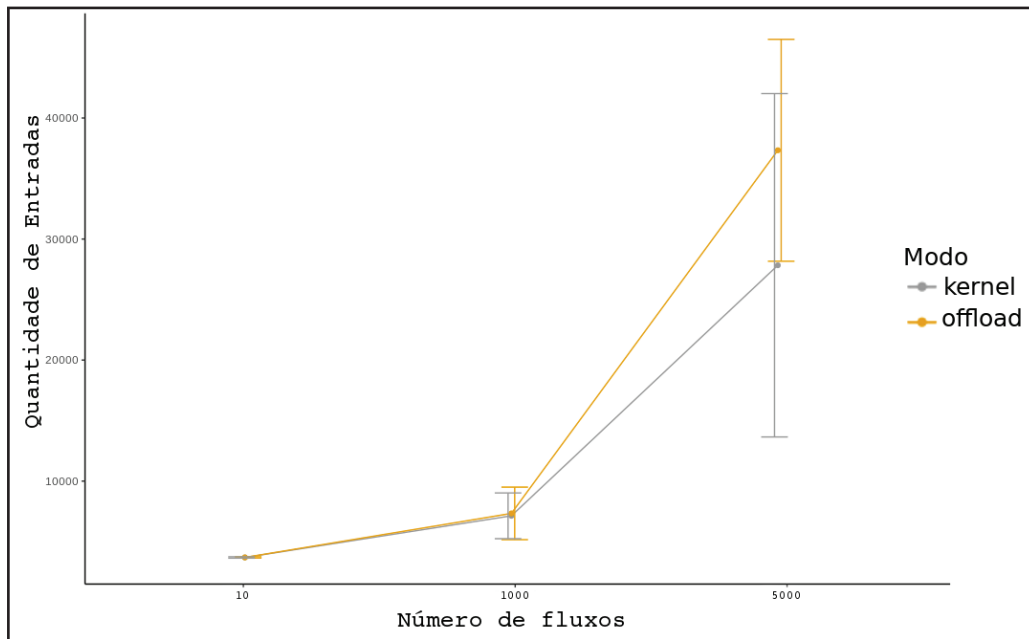


Figura 41. Entradas persistidas no *userspace* mediante a quantidade de fluxos. Fonte: Autor.

4.1.2.2 Saturamento de *switches* no Visual-INT

Outra variável importante para a aferição da capacidade de coleta é a quantidade de *switches* no caminho do fluxo. Por meio desta, é possível precisar por quantos equipamentos o fluxo se moveu. Quanto maior o número de equipamentos, maior será a densidade de metadados INT. Conforme relatado no exemplo anterior, os testes progressivos com tráfego INT real perfizeram apenas dois equipamentos. Para descobrir o comportamento da aplicação com quantidades diversas de *switches*, ensaiou-se experimentos contendo de um a seis dispositivos. Nestes, fixou-se uma taxa constante de 7Mpps para a transmissão de *reports*, assim como a definição de 1.000 fluxos para cada escala mensurada.

Em relação ao consumo de CPU, a Figura 42 assinala as diferenças de desempenho entre os dois métodos (*kernel* e *offload*). Novamente, observou-se um uso de CPU inferior para o método *kernel*, variando de 2,43%, com um único *switch*, para 2,58%, com seis (um aumento de cerca de 6%). Já o modo *offload* sustentou intervalos menores, indo de 3,06% a 3,08% (acréscimo de 0,65%). Baixa também foi a variabilidade em modo *offload*, com média dos desvios padrões 26% inferior a do tratamento *kernel*.

Por tratar-se de uma taxa constante de 1.000 fluxos a 7Mpps, a quantidade de dados persistidos entre as duas modalidades tende a ser similar (característica observável na Figura 41, durante a vazão de 1.000 fluxos). Todavia, em 80% dos casos, as médias de entradas para o modo *offload* superaram o método *kernel*, destacando-se índices de 2% (7.700 e 7.500 para dois *switches*), 10% (8.300 e 7.400 para três *switches*), 24% (7.900 e 6.000 para quatro *switches*), 4% (6.900 e 6.600 para cinco *switches*) e 9% (7.200 e 6.500

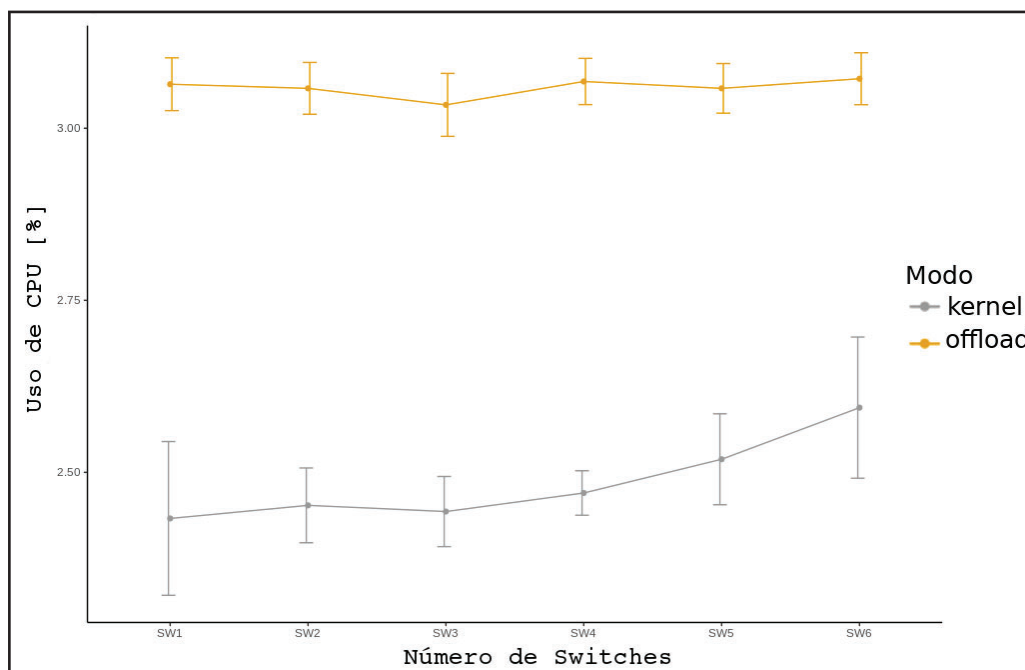


Figura 42. Uso de CPU mediante a quantidade de *switches*. Fonte: Autor.

para seis *switches*). Considerando-se a mediana, esta permaneceu inalterada, para o *offload*, em quatro das seis escalas, quedando-se em 7.500 entradas.

A Figura 43 elenca outro padrão observado nos testes, a saber, a quantidade de bytes processados no *userspace*. Com desvios padrões 96% menores, o tratamento *offload*, ao computar os pacotes a nível de hardware, poupou o *userspace* de uma carga de processamento superior à 910MB de dados nas seis escalas (116MB, 212MB, 251MB, 117MB, 61MB e 153MB, nesta ordem).

Com médias de entrada no banco de dados que, normalizadas, designam-se 9% maiores, além de um processamento no *userspace* 12,17% menor, infere-se que a alta de 21,06% no uso de CPU do método *offload* em relação ao *kernel* não exprime necessariamente um cômputo inferior, mas sim, uma entrega sensivelmente superior no que tange o processamento de pacotes e métricas INT, justificando-se, assim, a disparidade observada.

4.1.2.3 Saturamento de métricas INT no Visual-INT

O último teste foi realizado contemplando duas etapas. Na primeira delas, elaborou-se a combinação dos principais metadados INT apurados, que são a quantidade de *switches* (trabalhada anteriormente) e a latência do salto (*hop*). A informação de latência é crucial para a determinação de possíveis atrasos na rede ocasionados pelo comutador, uma vez que esta dispõe do tempo em que o pacote (contendo os metadados INT) permaneceu em cada equipamento. Por estar dentro da banda passante, o tratamento recebido pelo pacote INT reflete o que ocorreu com os demais datagramas.

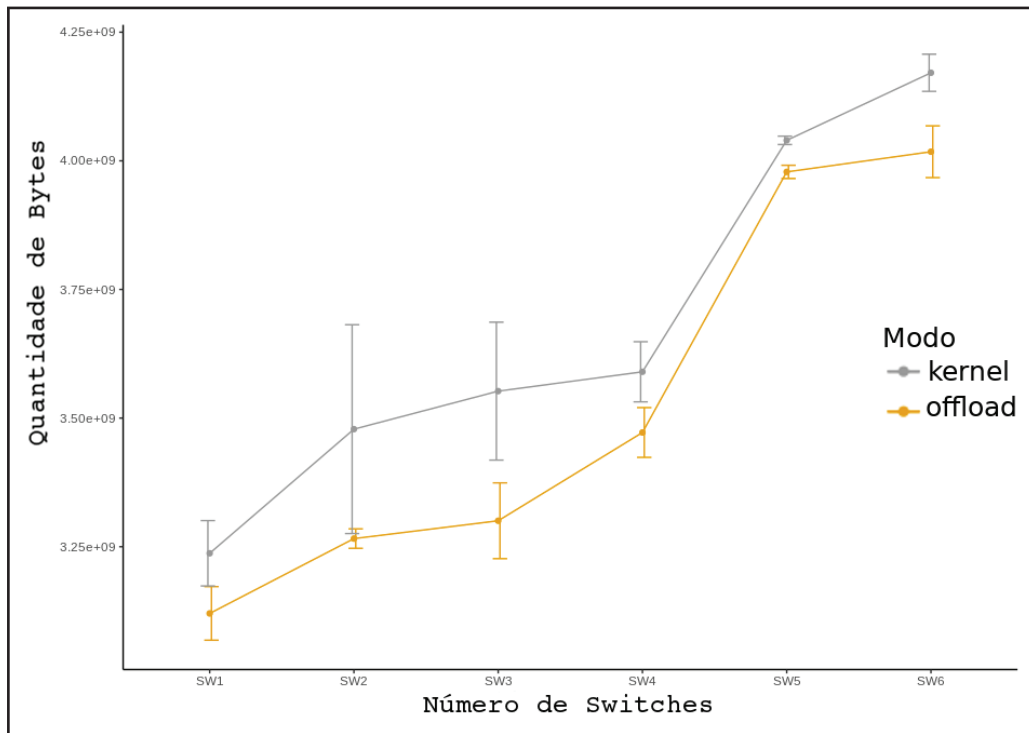


Figura 43. Bytes processados mediante a quantidade de *switches*. Fonte: Autor.

O segundo exercício associa todas as métricas INT estabelecidas para o coletor do Visual-INT, a saber, o ID do *switch*, os IDs das portas de entrada e saída, a latência do salto (*hop*), e os *timestamps* de porta de entrada e saída. Para uma melhor leitura, ambos os experimentos foram separados por testes contendo três e seis *switches*. Também utilizou-se da mesma convenção de 1.000 fluxos a uma taxa de encaminhamento de 7Mpps.

Com isso, tem-se o ilustrado na Figura 44, que apresenta o consumo de CPU durante a execução dos testes para os modos *kernel* e *offload*. Reiteradamente, o tratamento *kernel* apresenta cargas, no geral, 17% mais baixas. No entanto, o mesmo não é verdade em relação aos desvios padrões, em média 27% mais altos do que o modo *offload*. Ainda, acompanhando-se a tendência linear do crescimento do processamento, partindo-se, para isto, do teste mais simples (apenas três *switches* com latência) para o mais completo (seis *switches* e todas as métricas), a diferença entre as técnicas *kernel* e *offload* perfaz uma variabilidade de 4,92% (12,72% e 7,80%, nesta organização).

Seguindo o mesmo tipo de análise traçada previamente, a Figura 45 traz a quantidade de *bytes* passados para o *userspace*. Desta atividade, percebeu-se um decréscimo de 618MB entre o modo *offload* e a técnica *kernel* (sendo 54MB para o primeiro teste, seguido de 100 e 464MB para as últimas duas amostragens), confirmando-se a tendência anteriormente pautada. Todavia, pela primeira vez, constatou-se uma média de desvio padrão (para a quantidade de *bytes*) maior para o modo *offload* (cerca de 12%), desencadeada, majoritariamente, pela presença de *outliers* no último teste (seis *switches* e todas as métricas).

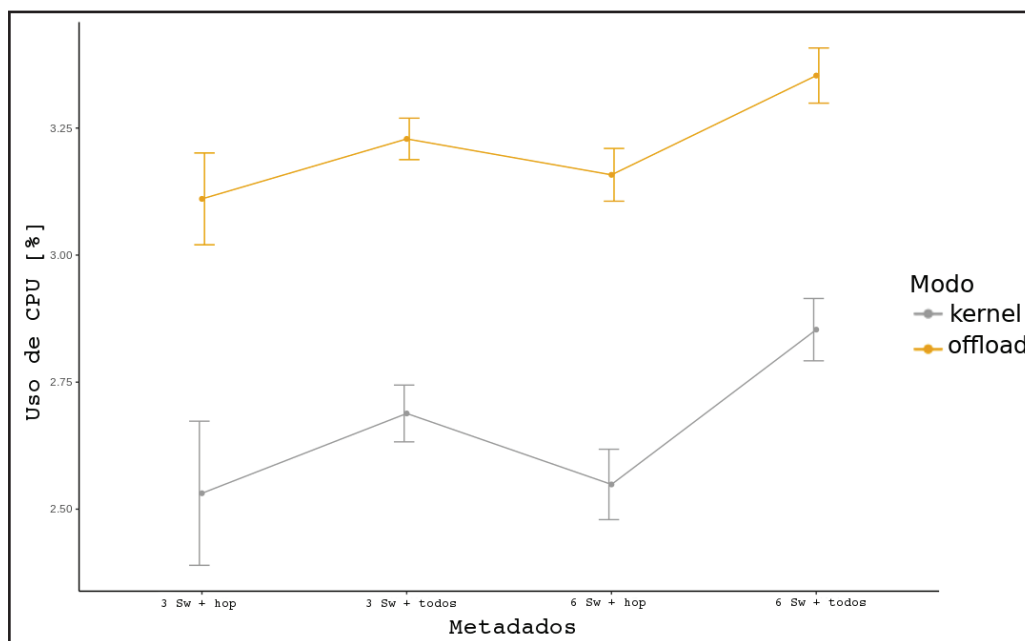


Figura 44. Uso de CPU mediante a quantidade de metadados INT. Fonte: Autor.

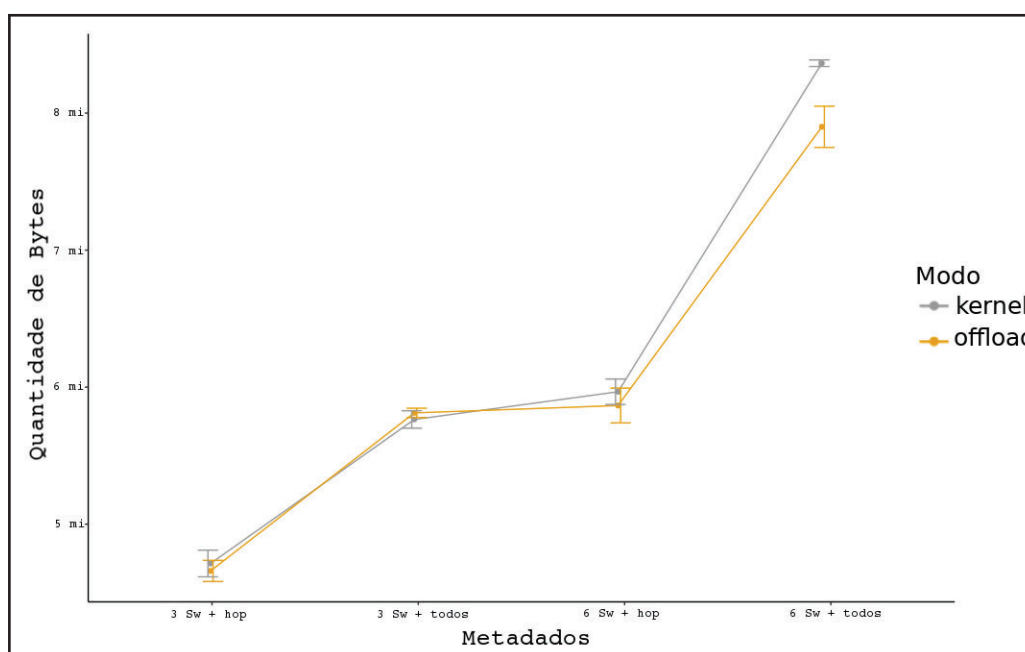


Figura 45. Bytes processados mediante a quantidade de metadados INT. Fonte: Autor.

Rematando, constatou-se, continuamente, a baixa variabilidade do uso de CPU no modo *offload* ante o emprego do método *kernel*. Tal como nos demais testes, o descarregamento do programa eBPF/XDP na placa reduziu a quantidade de bytes a serem analisados no *userspace*. Em fluxos maiores ou em ambientes com concorrência, estas características tornam-se desejáveis, principalmente na análise de cabeçalhos *report* de maior amplitude.

4.1.2.4 Avaliação dos testes com tráfego obtido do INT simulado

Para estimar médias condicionais e efeitos baseadas nos testes simulados realizados, a Figura 46 traz o modelo de regressão linear para o modo *kernel* em relação ao consumo de CPU e a quantidade de transmissão em Kpps (mil pacotes por segundo). O cálculo considera sua aplicação simples, designada pela equação $E(y) = \alpha + \beta \times X$, onde y é a variável dependente a ser explicada, E representa a esperança da média da variável dependente, sendo α a constante de interceptação da reta, X a variável independente e β como o coeficiente angular (ou de regressão), que define o efeito da variável independente sobre a dependente.

Percebeu-se, pelas características dos testes, um intervalo maior entre a primeira porção (pontos pretos no canto inferior esquerdo), referente ao teste com 10 fluxos e seis *switches*, que utiliza-se de pouca carga de CPU, e os demais experimentos com 1.000 e 5.000 fluxos (pontos pretos no canto superior direito), bem como os àqueles acerca do número de *switches* e métricas INT). Esta separação, todavia, não invalida sua significância (com ajuste do R^2 em 0.9458), bem como o efeito positivo (crescente) da quantidade de pacotes por segundo (pps) sobre o consumo de CPU (onde $\beta > 0$).

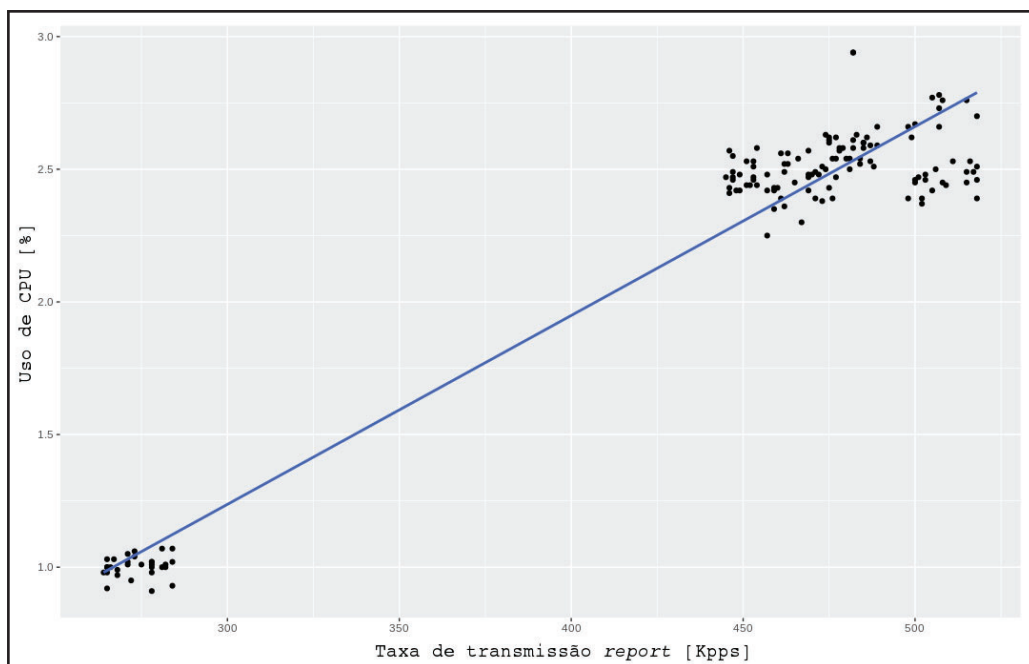


Figura 46. Modelo de regressão linear para o uso de CPU para o método *kernel* nos testes simulados. Fonte: Autor.

Da mesma forma, o método *offload* também apresentou distanciamento entre os testes com menor carga de metadados (pontos pretos no canto inferior esquerdo), conforme demonstrado na Figura 48. Assim como no caso anterior, a significância da medição foi alta (0.9395 de ajuste do R^2), apontando para um maior nível de confiança de que o efeito não é

nulo, além de um coeficiente de regressão (β) crescente, indicando que este mesmo efeito é positivo.

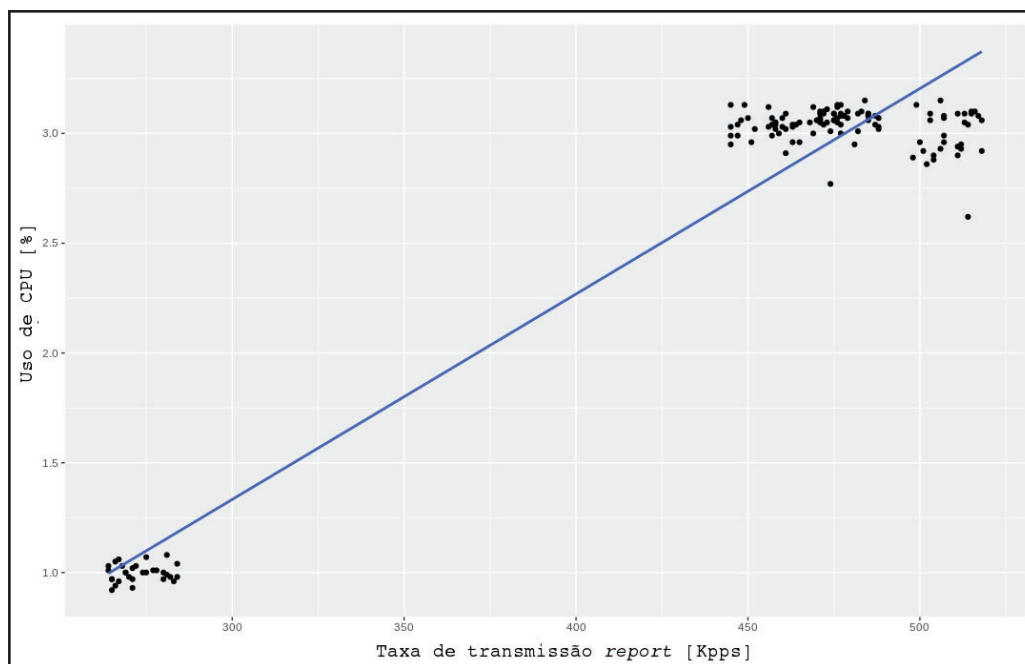


Figura 47. Modelo de regressão linear para o uso de CPU para o método *offload* nos testes simulados. Fonte: Autor.

Com isso, o modelo prevê um exaurimento total da capacidade de CPU (alcançe de 100%) para a aplicação do modo *kernel* aos 13Mpps ($142.87+132.8927 \times 100$), assumindo-se nesta faixa um *throughput* efetivo de 17Gbps (42,5% da capacidade nominal da porta da SmartNIC). Para o tratamento *offload* este nível é 23% mais baixo, uma vez que seu consumo de CPU tende a ser mais elevado, culminando aos 10Mpps ($174.23+100.44 \times 100$) com *throughput* efetivo de 13Gbps (32,5% das possibilidades de cada porta da placa).

Por este fato, na comparação direta com a carga de CPU, o modo *offload* sempre estará em desvantagem. Porém, ao utilizar-se da regressão linear para fins de comparação da quantidade de entradas no banco de dados para um mesmo valor hipotético de taxa de transmissão (neste caso 10Mpps para ambos os modos), a técnica *offload* apresenta cerca de 6% de séries a mais do que o tratamento *kernel*.

Disto, infere-se não um desempenho piorado do modo *offload* ante a aplicação de altos volumes de processamento de metadados INT, mas sim, uma frequência de entrega e disponibilidade maior. Com base nas características citadas ao longo dos testes (como desvio padrão, quantidade de *bytes* processados no *userspace* e número de entradas no banco de dados) o baixo consumo de CPU do método *kernel* não significa uma superioridade de desempenho, mas um descompasso entre as funções de extração, análise e persistência dos dados auferidos.

4.2 VALIDAÇÃO DO ARMAZENAMENTO DOS DADOS EM SÉRIES TEMPORAIS

O envio de dados processados no *userspace* para o InfluxDB se dá por meio de uma API em C, fornecida por terceiros, denominada de `influxdb.h` [57] que, para adequar-se aos experimentos com *timestamps* de 64 bits, precisou ser modificada. Este uso deve-se à ausência de APIs oficiais do InfluxDB que permitam a comunicação entre aplicações formatadas em C puro e o banco. Os dados são enviados via o método POST do *http*, seguindo o protocolo *line*, isto é, um formato baseado em texto para gravar pontos no InfluxDB.

| | |
|-------------|---|
| Measurement | "flow_path, 10.1.1.5:255->10.1.1.5:32766, proto=11" |
| Time | 1571264132969769085 |
| Fields | path="1:2" |

Figura 48. Modelo de *measurement* no InfluxDB contendo o caminho de fluxo. Fonte: Autor.

A Figura 48 denota a forma hierárquica como os dados são persistidos no banco de dados, onde a chave principal é uma *string* definida pelo usuário, denominada *measurement*. O *measurement* nada mais é do que a sequência de entrada observada de mesmo tipo. Por tratar-se de um banco de dados de séries temporais, a chave primária de cada registro é o próprio *timestamp* que, se não definida antes do encaminhamento, assume, na chegada, o tempo corrente da máquina física. Por fim, tem-se a última estrutura de chave e valor, onde é possível possuir apenas uma entrada para cada tipo, ou seja, uma única chave e um único valor.

4.2.1 Abordagem ingênua

No decorrer dos experimentos, foi utilizada a abordagem ingênua (*naive approach*), que em análise de séries temporais, diz-se de um modelo de previsão de dados inaugural que desconsidera a sazonalidade. Disto, nos testes de validação anteriores, todos as novas entradas encaminhadas pelo Analisador do Coletor, considerando-se as atualizações de *measurements* antigos, foram persistidos dentro do InfluxDB. O principal motivo desta prática foi estabelecer a primeira relação de entradas máximas possíveis dentro do banco de dados.

Para os testes com tráfego real (seção 4.1.1), por exemplo, estabeleceu-se o *TE* (Total de Entradas) pela Equação 7, onde o *M* seria o produto da multiplicação dos conjuntos de métricas *m* constantes em cada *report* (que totalizam 14, sendo 1 para a caminho

do fluxo, 1 para a latência de fluxo e 2 medições para cada *switch* para latência de salto, timestamps de entrada e saída) sobre a quantidade de fluxos (neste caso apenas um), E a média da quantidade de entradas por segundo, T e t sendo o tempo de medição e $push$ de envio em segundos, nesta ordem.

$$TE = (E + \frac{M}{t}) \times t \quad (7)$$

$$TE = (45 + \frac{14}{60}) \times 60 = 2.714 \quad (8)$$

Com número muito próximo ao estabelecido para os testes com fluxo real (cuja a média total foi de 2.753), percebeu-se a correlação da abordagem pelo resultado da aplicação da mesma na Equação 8. Para os testes com tráfego simulado, emprega-se a Equação 9, que utilizam-se de taxa de envio a em Kpps (T_x), da quantidade de parâmetros habilitadas dentro dos cabeçalhos *report* (M) e da própria duração da medição (T). Com isso, tem-se a Equação 9, que apresenta o cálculo TE , e como exemplo de aplicabilidade desta, a Equação 10, que emprega a primeira para o teste com 5.000 fluxos e seis *switches* (seção 4.1.2.1), cuja média total de entradas recebidas foi de 30.800.000:

$$TE = T_x \times T \times M \quad (9)$$

$$TE = 300.000 \times 60 \times 2 = 30.000.000 \quad (10)$$

Para os testes com 10 e 1.000 fluxos, respectivamente, as entradas representaram 4.040 (com médias de entrada de 3.964) e 33.600.000 (com 32.7000.000 de média), aproximando-se do método exposto. Com base nestas comparações, é possível inferir-se a quantidade total de entradas percebidas pelo InfluxDB. Dentro destas margens, como esperado, entende-se que todos os pacotes puderam ser processados e, conseqüentemente, terem acedido ao banco de dados.

4.2.2 Detecção de Eventos

A significação dos dados INT durante a análise de resultados demonstrou que nem todas as entradas representam uma mudança expressiva do estado da rede. A celeridade do hardware culmina em custos muito elevados no que tange o armazenamento. Visando a redução de entradas no banco de dados (denotada pela *naive approach*) e, conseqüentemente, somente a persistência de dados realmente significativos, elaborou-se uma rotina de periodização da entrada de análises provenientes do *userspace* para dentro do InfluxDB.

Este subprograma consiste na consulta, pelo Analisador do Coletor (localizado no *userspace*), do mapa descarregado para dentro da placa (no caso do modo *offload*) ou do *driver* do SO (para o modo *kernel*) dos valores de latência do salto anteriores. Como elemento coletado mais suscetível a mudanças, determinou-se que apenas variações desde

a última atualização que obedecessem certo *threshold* (100 nanossegundos), tanto para mais quanto para menos, deveriam ser armazenadas.

Utilizando-se de testes com tráfego real, cujas configurações igualam-se a do primeiro experimento (seção 4.1.1.1), isto é, um constante fluxo de *reports* à 60Kbps, verificou-se, conforme a Figura 49, uma diminuição em relação aos números obtidos naquele primeiro teste (segundo a média) de 71% para ambos os modo *offload* (2.764 para 800) e *kernel* (2.772 para 797), respectivamente. Com mediana e moda idênticas nos dois casos (799), notou-se uma menor dispersão para o modo *offload* (53 contra 59), corroborando o comportamento identificado nas demais validações.



Figura 49. Médias de quantidade de entradas no banco de dados e erro padrão (modos *kernel* e *offload*) para detecção de eventos. Fonte: Autor.

Com IC de 95%, demonstrado na Tabela 10, constatou-se diferenças mínimas entre as médias obtidas no uso de CPU. Considerando-se o teste progresso (seção 4.1.1.1), onde todos os pacotes foram aceitos e transformados em entradas no banco de dados, o tratamento *offload* permaneceu inalterado, enquanto o modo *kernel* aumentou em 20% a abrangência de seu intervalo.

Tabela 10. Médias obtidas com IC de 95% para detecção de eventos. Fonte: Autor.

| Modo | Uso de CPU | Nro pacotes INT |
|---------|-------------|-----------------|
| Kernel | 0.51 ± 0.46 | 801 ± 791 |
| Offload | 0.52 ± 0.47 | 805 ± 794 |

No que concerne o consumo de CPU, novamente, o método *kernel* apresentou uma pequena vantagem em relação ao *offload* (0.50 para 0.49). Todavia, este comparado ao primeiro apresentou erro padrão mais bem distribuído.

Disto, conclui-se que o dispositivo de *threshold*, além de reduzir o custo de armazenamento, não alterou significativamente o comportamento da aplicação nos dois modos

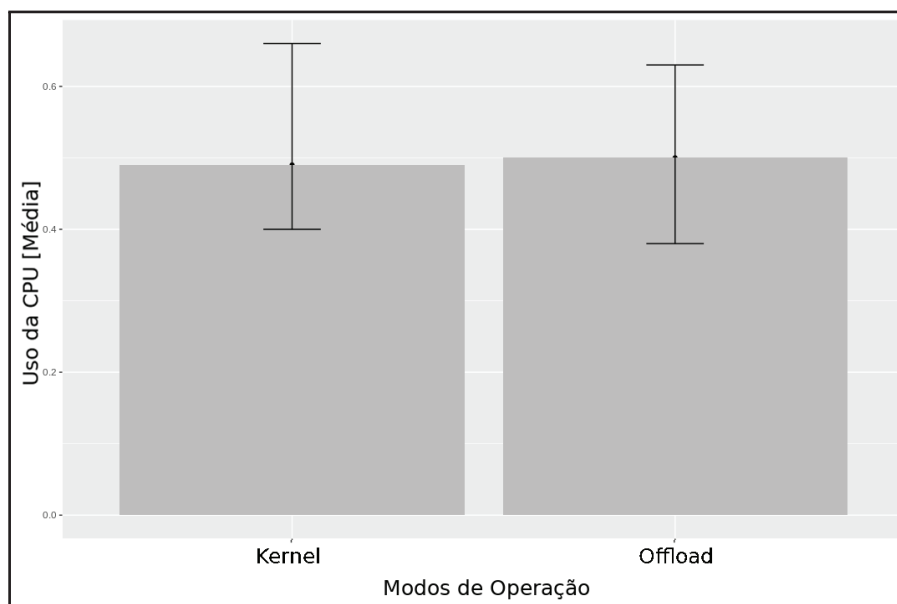


Figura 50. Comparativo entre médias de CPU e erro padrão (modos *kernel* e *offload*) para detecção de eventos. Fonte: Autor.

(para testes com tráfego real). Para o tráfego simulado, todavia, não é possível realizar este tipo de aferição diretamente, uma vez que as simulações possuem caráter estático e a latência de salto, regra pela qual o *threshold* é estabelecido, é essencialmente dinâmica.

Porém, ao estender o comportamento apresentado pelo tráfego real para o simulado, com base na abordagem ingênua anteriormente traçada (Equação 9 / Equação 7), têm-se reduções hipotéticas de 3 (de 3.964 para 1.101), 6.083 (33.600.000 para 5.030) e 5.548 vezes (30.000.000 para 5.548) para 10, 1.000 e 5.000 fluxos.

4.3 VALIDAÇÃO DO UTILITÁRIO DE VISUALIZAÇÃO

No Visual-INT, a visualização dos metadados computados dá-se de duas maneiras. A primeira e mais imediata é efetuada pelo terminal do SO, por onde são listados continuamente os dados aferidos durante o tempo de execução. Esta demonstração também serve como uma espécie de *debugger*, uma vez que, por meio de seu *output*, é possível determinar a correte funcional²⁵ do algoritmo aplicado. O segundo modo de observar os metadados é pelo utilitário de visualização hospedado em um servidor NodeJS.

A Figura 51 abrange uma parte resumida da visualização via terminal, estando sua versão estendida disponível no ANEXO A. Nela, percebe-se a diferenciação de cada parte do hexadecimal de 64 *bits* (1 e 2), originalmente extraída da primeira parte do cabeçalho *report* (*timestamp* do *switch sink*). Este mesmo hexadecimal é apresentado na notação decimal (3) e, para fácil leitura, denotado em formato de data (4). Há também a presença

²⁵Diz-se, em lógica computacional, da produção de uma saída condizente com a entrada por determinado algoritmo.

discricionária dos principais metadados coletados, como os IDs dos *switches* (5) e das portas de entrada e saída (6 e 7), os dados de latência de salto (8) e os *timestamps* de *ingress* e *egress* (9 e 10), estes três últimos seguindo igual modelo de formatação. Por fim, é mostrada a elaboração do *measurement* enviado ao banco de dados (11), contendo uma *string* com o resumo das informações processadas (como o *timestamp* do *sink* e o caminho - *path* - percorrido).

| | |
|-----|--|
| 1. | 32 bits hexa: 5DA79684 |
| 2. | 64 bits hexa: 39cd807d |
| 3. | Flow Sink Time ns: 1571264132969769085 |
| 4. | Flow Sink Time = Wed Oct 16 22:15:32 2019 |
| 5. | Switch IDs[0] = 2 |
| 6. | Ingress Port IDs[0] = 0 |
| 7. | Egress Port IDs[0] = 4 |
| 8. | 32 bits hexa: A258697B |
| | 64 bits hexa: 39cd807d |
| | Hop Latency[0] ns: 2723703163969769085 Hop Latency[0] ms: 0.000969 |
| 9. | 32 bits hexa: 5DA79684 |
| | 64 bits hexa: 39cd807d |
| | Ingress Timestamp[0] ns: 1571264132969769085 Ingress TimeStamps[0] = Wed Oct 16 22:15:32 2019 |
| 10. | 32 bits hexa: 5DA79684 |
| | 64 bits hexa: 39cd807d |
| | Egress Timestamp[0] ns: 1571264132969769085 Egress TimeStamps[0] = Wed Oct 16 22:15:32 2019 |
| 11. | influxdb-c::post_http: iv[1] = 'flow_path\,\ 10.1.1.1:255->10.1.1.5:32766\,\ proto=11 path="1:2" 1571264132969769085' |

Figura 51. Bytes processados mediante a quantidade de metadados INT. Fonte: Autor.

Por último está a exibição de dados por meio do utilitário de visualização, codificado no ANEXO B. Neste, o usuário pode selecionar o tipo de medição a ser demonstrada, que será exibida ou em um gráfico estilo Sankey, para a latência e caminho do fluxo, ou através do *plotly.js*, para acompanhamentos da latência por equipamento. Estas medições são carregadas automaticamente ao iniciar-se o programa eBPF/XDP na SmartNIC. Por meio do servidor NodeJS, via Telegraf (programa de conectividade com o InfluxDB), são recuperadas informações utilizando-se da mesma *string* armazenada anteriormente pelo Analisador do Coletor no banco de dados.

Por exemplo, ao requisitar a representação do caminho de fluxo, o usuário obtém um resultado similar ao da Figura 52, através da escolha de *measurement* (explicitada pelo tipo *flow_path*) e submissão pelo botão Get Data (1). O usuário também pode, a qualquer tempo, parar e reiniciar a busca dos dados (2) bem como selecionar um intervalo maior (3) para a demonstração dos mesmos, que pode variar de 5 a 60 segundos. As informações são centralizadas na tela (4) e, mediante a sobreposição do cursor do *mouse*, apresentam-se detalhes acerca do fluxo (latência em nanossegundos e nós de origem e

destino). Customizado, entende-se esta como a primeira adaptação de uso dinâmico para o gráfico Sankey.

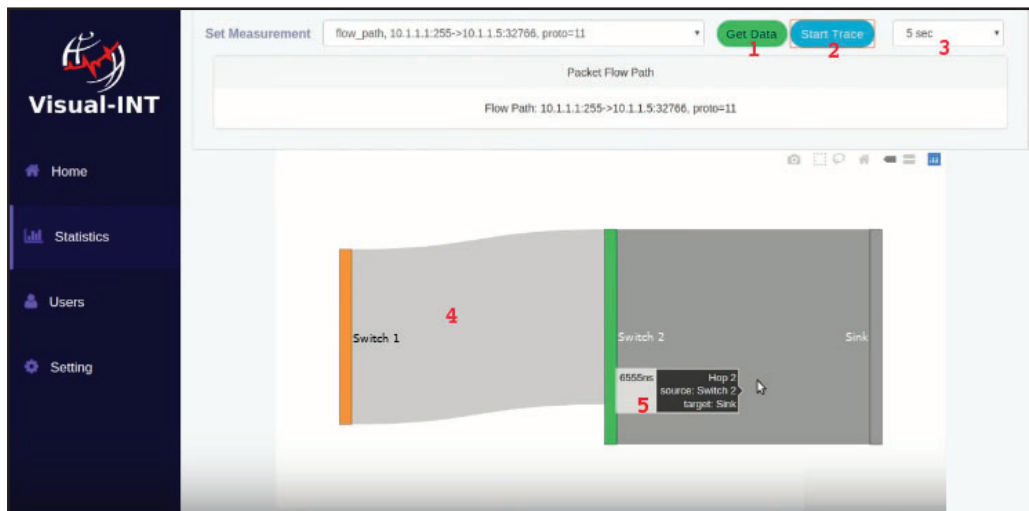


Figura 52. Interface do gráfico Sankey no utilitário de visualização. Fonte: Autor.

O gráfico *plotly.js*, que trata das visualizações de latência por dispositivo, tem como resultado a representação constante na Figura 53. O campo de identificação da interface dispõe do ID do *switch* investigado (1) seguido, logo abaixo, da imagem centralizada do gráfico de linha. O eixo *x* do diagrama faz jus ao *timestamp* em nanossegundos da latência do salto e o *y* o retorno da *query* contendo a latência registrada em cada intervalo de tempo.

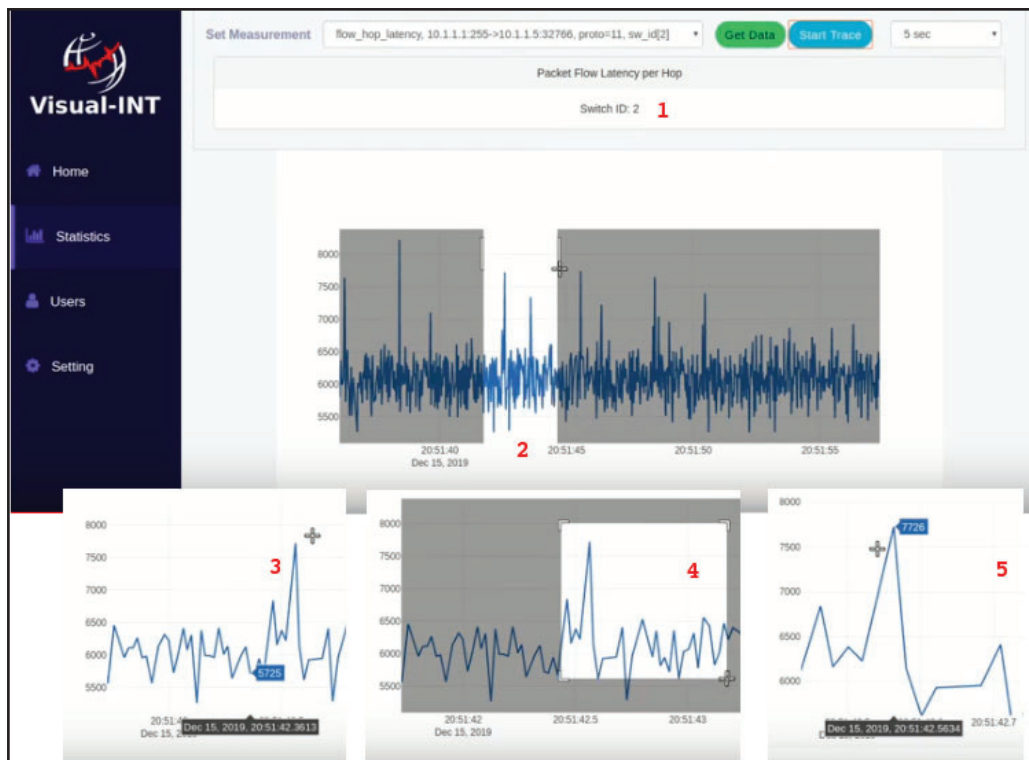


Figura 53. Interface do gráfico *plotly.js* no utilitário de visualização. Fonte: Autor.

Ainda na Figura 53, é possível perceber a funcionalidade de seleção de granularidade (2), onde cada iteração do usuário provoca a aproximação do conteúdo do gráfico até se alcance um nível necessário de observação (3, 4 e 5). A diferença entre os dois diagramas consiste, por exemplo, na amplitude de visualização. Enquanto os gráficos de latência são específicos para um determinado dispositivo, o esquema Sankey demonstra a latência em todo o caminho, sendo assim, este último trabalha mais diretamente com a *string* do caminho do fluxo, de onde é retirada, por funções em JavaScript, as posições e tempos de cada equipamento.

4.4 DISCUSSÃO DOS RESULTADOS

Buscou-se por uma metodologia de validação bimodal (*kernel* e *offload*), corroborar a aplicabilidade, dentro do hardware, de um programa eBPF/XDP para a extração de metadados INT. Optou-se por esta abordagem pela ausência de paralelos para a referida temática, onde assomam-se exemplos aplicados a ambientes virtualizados. Desta feita, por intermédio de testes com tráfego real e simulado, constatou-se um melhor desempenho geral para o modo *offload*.

Com base nas validações precedentes, constata-se o alcance dos principais propósitos dirigidos ao Visual-INT, que são a determinação do caminho de fluxo e a precisão da latência dos equipamentos em tempo real. Mecanismos extras, como a definição de *thresholds*, garantiram custos menores de armazenamento sem que com isso houvesse impactos severos no desempenho da aplicação.

Por tratar-se de uma aplicação inédita em hardware, é necessário ponderar sobre alguns aspectos que diferenciam o coletor do Visual-INT dos demais trabalhos relacionados à extração de metadados de telemetria (elaborados na seção 2.4). Traçando um paralelo com o INTCollector, iniciativa mais próxima da presente proposta, vê-se que a implementação do sistema de coleta eBPF/XDP baseia-se no BCC (*BPF Compiler Collection*) [58]. Desenvolvida pelo projeto IOVisor, esta tecnologia abstrai a complexidade da instrumentação do *kernel* em C ao permitir que a programação eBPF suceda-se por linguagens de programação interpretadas, como o Python e Lua, por meio de um *wrapper* à volta do LLVM.

De um modo geral, esta abordagem facilita a criação de programas de análises e controle de tráfego para o *kernel* e, de igual maneira, dificulta a integração destes ao *driver* do hardware da SmartNIC (que trabalha em C99). O coletor do Visual-INT, por sua vez, comunica-se diretamente com o hardware e o com *kernel*, possuindo menos abstrações técnicas. Neste sentido, a independência de bibliotecas externas garante maior rapidez na execução do código. Ao passo em que o INTCollector consegue processar 154.8Kbps de métricas INT com 1% de CPU, o Visual-INT computa 274Kbps com a mesma carga de CPU, uma eficiência de 76%.

Incorporado ao Mininet, a rede do INTCollector alcança (em testes práticos) *throughputs* entre interfaces de aproximadamente de 1Mpps (milhões de pacotes por segundo), insuficientes para inferir seu comportamento em bandas de alta velocidade. Como visto nos testes práticos, o Visual-INT, pela sua aplicabilidade ao hardware e ambiente real, supera este número em sete vezes. Além disso, as medições obtidas num ambiente inteiramente emulado, tal qual o do INTCollector, traduzem-se em prognósticos que desconsideram as circuncidantes e complexidades inerentes a um ambiente físico que, em contrapartida, estão amplamente refletidas no Visual-INT.

Outro ponto importante é a aplicação dos *timestamps*. O presente trabalho empreende *timestamps* em nanossegundos (64 *bits*) ao passo que o INTCollector os dispõe somente em segundos (32 *bits*). Nisto, o último possui granularidade de tempo similar a ferramentas tradicionais de monitoramento, como o SNMP. Ademais, devido à simulação, não pôde ser significado o *ingress timestamp* na ferramenta, ao contrário do que ocorre no Visual-INT.

Embora o INTCollector colete três métricas a mais que o Visual-INT (a saber os IDs de fila e ocupação de fila, dados de ocupação e congestionamento de fila e uso de banda), estas constituem-se como parcialmente implementados na rede figurada do INTCollector, uma vez que seus *switches* P4 simulados operam sobre recursos (metadados internos) disponibilizados pelo emulador Mininet. Desta forma, os dados designados não correspondem estritamente ao comportamento da rede analisada. Outrossim, a validação do experimento é majoritariamente realizada por intermédio da simulação de tráfego, o que permite a efetuação de coletas não oriundas de uma rede previamente existente, seja ela física ou virtual. Mesmo que perceba menos métricas que o anterior, o Visual-INT detém cerca de 32 *bytes* de informação de telemetria por equipamento, totalizando 4 *bytes* a mais para cada *switch* descrito no *report* do INTCollector.

5. CONCLUSÃO

O projeto AmLight-INT, visando a detecção e resolução de falhas na rede SDN de pesquisa acadêmica Am-Light-ExP, busca implantar o monitoramento INT. Por tratar-se de uma rede WAN, surgiu a necessidade de desenvolver protótipos próprios para a coleta de dados de telemetria, haja vista o caráter local das aplicações de telemetria até então. Apoiada pela Universidade Internacional da Flórida (FIU), o presente trabalho consta como um dos empreendimentos de coleta de metadados INT presentes na iniciativa AmLight-INT.

5.1 CONSIDERAÇÕES FINAIS

No que refere-se às tecnologias empregadas, o Visual-INT desponta na vanguarda ao servir-se da SmartNIC Netronome Agilio CX 2x10GbE, modelo que atualmente figura como o único hardware do mercado com suporte ao *offloading* eBPF/XDP [59]. A maior contribuição do método exposto no Visual-INT é o aprimoramento das técnicas de coleta INT, que passam a ser executadas no hardware, diferindo-se das demais propostas relatadas na literatura acadêmica que são, majoritariamente, manipuladas dentro de ambientes emulados/simulados.

Conforme discutido na seção 4.4, os ganhos de desempenho nesta abordagem superam os resultados de trabalhos anteriores, além de possibilitar a validação da solução INT em hardware para aplicações de alta demanda. Similarmente, o exercício de um instrumento próprio de visualização oportuniza o uso de diferentes medições, possibilitando uma visão fracionada dos elementos, o que torna a representação dos dados muito mais próxima da realidade observada.

Desta maneira, o presente estudo propõe a solução de problemas reais (precisar o caminho do fluxo e a latência dos dispositivos) enfrentados pelo projeto parceiro AmLight-INT. Proposto com tecnologias equivalentes, os resultados gerados servirão de base para novas investigações dentro da temática INT na referida iniciativa.

5.2 RESSALVAS E LIMITAÇÕES

Por tratar-se de uma área de investigação pioneira em redes de computadores, a presente proposta defrontou-se com algumas insuficiências no decorrer de sua implantação, a maioria partindo de condicionantes externas que amoldaram as expectativas de alcance do trabalho. Dentre as limitações concernentes ao hardware, estão a não implementação dos IDs de fila e ocupação de fila, conseqüentemente, das próprias métricas de ocupação e congestionamento de fila, não suportados pelas SmartNICs Netronome atu-

almente. Combinadas, as duas métricas permitem monitorar os gargalos e usos dessas estruturas por intermédio de pequenas frações coletadas ao longo da passagem, inferindo-se assim quanto do espaço máximo do *buffer* disponível para a fila fora utilizado.

Da mesma forma, a utilização de banda da interface de saída (*egress*), denominada TX (*Transmission*), carece do amparo de mecanismos de rastreamento de taxa atual, tais quais os algoritmos *bin bucketing* e *moving average*. Uma vez que o *framework* INT não estipula nenhuma outra mecânica de aferição de transmissão, deixando essas particularidades a cargo dos fornecedores de dispositivos, tal parâmetro não pode ser empregue na rede P4 associada ao Visual-INT e tão pouco ser mensurado pelo último.

No que se refere à aplicabilidade do coletor em hardware, o uso de alguns *helpers* eBPF ²⁶ (habituais de versionamentos em software) ainda encontram-se faltantes, sendo comum a parcialidade de funcionamento e a espera de *releases* complementares. Por exemplo, para as atualizações de mapas do coletor terem efeito (*bpf_map_update_elem*), utilizou-se de pacotes de sistema diretamente providos pela equipe da Netronome (*upstream*). Todavia, a solução aplicada tem como característica a degradação de desempenho, visto que os *cores* da placa, em *threads* paralelas, concorrem entre si para atualizar o mapa, necessitando assim de *spinlocks* (estruturas que paralisam os demais núcleos) para provisionar que apenas uma ME (*Micro-Engine*) acesse às chaves e modifique os valores.

O tamanho do registro de cada mapa também é limitado (64 *bytes*), impedindo que uma série de informações fiquem armazenadas no hardware (como as *flags* de controle), sendo apenas identificadas no baixo nível os dados sobre o fluxo. Devido a isto, a seleção de envio ou não de metadados para o InfluxDB (através da detecção de eventos) é realizada no Analisador do Coletor, orquestrado no caminho normal.

5.3 TRABALHOS FUTUROS

Até então, a interface gráfica não abarca concepções que consideram a experiência do usuário (como o *design UX*) no entendimento das interações entre o sistema e seus clientes, sendo estas potenciais perspectivas a serem exploradas para futuras versões da aplicação. Outro tópico importante é a superação das limitações de hardware das SmartNICs Netronome, o que pode desencadear novas funcionalidades e resoluções relacionadas às tecnologias P4, eBPF e XDP.

O armazenamento de dados em séries temporais também propicia o surgimento de novas vertentes de pesquisa, uma vez que a análise histórica de informações pode, ao basear-se em modelos que ponderam sobre *forecasting*, modelar o comportamento do Visual-INT mediante o seu próprio uso. A automatização de ações seguindo determinados

²⁶Funções auxiliaadoras no trato com o *kernel*.

thresholds de eventos predeterminados pelo usuário facilitam a manutenibilidade da rede e atestam um caráter ativo na correção de falhas.

REFERÊNCIAS BIBLIOGRÁFICAS

- [1] CISCO. *Cisco Visual Networking Index: Forecast and Methodology, 2016-2021, White Paper*. 2018. Disponível em: <<https://www.cisco.com/c/en/us/solutions/collateral/service-provider/global-cloud-index-gci/white-paper-c11-738085.html>>. Acessado: 20-01-19.
- [2] XIA, W. et al. A survey on software-defined networking. *IEEE Communications Surveys Tutorials*, IEEE, New Jersey, USA, v. 17, n. 1, p. 27–51, 2015. ISSN 1553-877X. Disponível em: <<https://doi.org/10.1109/COMST.2014.2330903>>.
- [3] SINGH, S.; JHA, R. K. A survey on software defined networking: Architecture for next generation network. *Journal of Network and Systems Management*, Amsterdam, Netherlands, v. 25, n. 2, p. 321–374, 2017. ISSN 1573-7705. Disponível em: <<https://doi.org/10.1007/s10922-016-9393-9>>.
- [4] KREUTZ, D. et al. Software-defined networking: A comprehensive survey. In: *Proceedings of the IEEE*. Singapore: IEEE Computer Society, 2015. v. 105, n. 1, p. 14–76. ISSN 1558-2256. Disponível em: <<https://doi.org/10.1109/JPROC.2014.2371999>>.
- [5] HAAS, C. W.; SALERNO, D. C.; SHEINBEIN, D. Stored program controlled network: 800 service using spc network capability, network implementation and administrative functions. *The Bell System Technical Journal*, New Jersey, USA, v. 61, n. 7, p. 1745–1757, 1982. ISSN 0005-8580. Disponível em: <<https://doi.org/10.1002/j.1538-7305.1982.tb04371.x>>.
- [6] KIM, C. et al. *In-band Network Telemetry Via Programmable Dataplanes*. 2015. In: ACM SIGCOMM. London, United Kingdom: ACM. Presented as industrial demo. Disponível em <<https://nkatta.github.io/papers/int-demo.pdf>>. Acessado: 20-01-19.
- [7] BEZERRA, J. et al. International networking in support of extremely large astronomical data-centric operations. In: *Astronomical Data Analysis Software and Systems (ADASS XXVII) conference*. Alexandria, VA, USA: National Science Foundation, 2017. ISSN 1080-7926. Disponível em: <<http://par.nsf.gov/biblio/10056971>>.
- [8] LSST. *(LSST) Large Synoptic Survey Telescope*. 2019. Disponível em: <<https://www.lsst.org/>>. Acessado: 20-01-19.
- [9] MATHIS, M. et al. The macroscopic behavior of the tcp congestion avoidance algorithm. *SIGCOMM Comput. Commun. Rev.*, ACM, New York, NY, USA, v. 27, n. 3, p. 67–82, 1997. ISSN 0146-4833. Disponível em: <<http://doi.acm.org/10.1145/263932.264023>>.

- [10] TSANG, Y.; COATES, M.; NOWAK, R. D. Network delay tomography. *IEEE Transactions on Signal Processing*, New Jersey, USA, v. 51, n. 8, p. 2125–2136, 2003. ISSN 1053-587X. Disponível em: <<https://doi.org/10.1109/TSP.2003.814520>>.
- [11] KIM, C. et al. *In-band Network Telemetry (INT)*. 2016. 28 páginas. P4 ORG. Palo Alto, CA, USA. Disponível em: <<https://p4.org/assets/INT-current-spec.pdf>>. Acessado: 20-01-2019.
- [12] CAMPBELL, A. T. et al. A survey of programmable networks. *SIGCOMM Comput. Commun. Rev.*, ACM, New York, NY, USA, v. 29, n. 2, p. 7–23, 1999. ISSN 0146-4833. Disponível em: <<http://doi.acm.org/10.1145/505733.505735>>.
- [13] FRANCA, A. Ambiente de experimentação fibre. 2018. 31 slides. Apresentado dentro da disciplina de Arquitetura de redes, da UFPE (Universidade Federal de Pernambuco). Disponível em: <http://www.cin.ufpe.br/~suruagy/cursos/AR/2018-1-Ambiente_de_experimentacao_FIBRE.pdf>. Acessado: 20-01-19.
- [14] MCKEOWN, N. *Make SDNs Work*. 2012. Disponível em: <<https://youtu.be/W734gLC9-dw>>. Acessado: 16-06-2018.
- [15] SOOD, K.; XIANG, Y. The controller placement problem or the controller selection problem? *Journal of Communications and Information Networks*, Posts & Telecom Press, Singapore, v. 2, n. 3, p. 1–9, 2017. ISSN 2509-3312. Disponível em: <<https://doi.org/10.1007/s41650-017-0030-x>>.
- [16] TENNENHOUSE, D. L. et al. A survey of active network research. *IEEE Communications Magazine*, IEEE, New Jersey, USA, v. 35, n. 1, p. 80–86, 1997. ISSN 0163-6804. Disponível em: <<https://doi.org/10.1109/35.568214>>.
- [17] CASADO, M. et al. Sane: A protection architecture for enterprise networks. In: *Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15*. Berkeley, CA, USA: USENIX Association, 2006. (USENIX-SS'06). ISSN 2150-3761. Disponível em: <<http://dl.acm.org/citation.cfm?id=1267336.1267346>>.
- [18] CASADO, M. et al. Ethane: Taking control of the enterprise. In: *Proceedings of the 2007 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*. New York, NY, USA: ACM, 2007. (SIGCOMM '07), p. 1–12. ISBN 978-1-59593-713-1. Disponível em: <<http://doi.acm.org/10.1145/1282380.1282382>>.
- [19] AKYILDIZ, I. F. et al. A roadmap for traffic engineering in sdn-openflow networks. *Computer Networks*, Computer Networks, Amsterdam, Netherlands, v. 71, p. 1–30, 2014. ISSN 1389-1286. Disponível em: <<https://doi.org/10.1016/j.comnet.2014.06.002>>.

- [20] HUANG, T.; YU, F. R.; LIU, Y.-j. Special issue on future network: Software-defined networking. *Frontiers of Information Technology & Electronic Engineering*, Frontiers of Information Technology & Electronic Engineering, Beijing, China, v. 17, n. 7, p. 603–605, 2016. ISSN 2095-9230. Disponível em: <<https://doi.org/10.1631/FITEE.SDN2016>>.
- [21] MCKEOWN, N. et al. Openflow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, ACM, New York, NY, USA, v. 38, n. 2, p. 69–74, mar. 2008. ISSN 0146-4833. Disponível em: <<http://doi.acm.org/10.1145/1355734.1355746>>.
- [22] SUZUKI, K. et al. A survey on openflow technologies. *IEICE Transactions on Communications*, IEICE, Londres, United Kingdom, E97.B, n. 2, p. 375–386, 2014. ISSN 1745-1345. Disponível em: <<https://doi.org/10.1587/transcom.E97.B.375>>.
- [23] Floodlight OpenFlow Controller Website. 2018. Disponível em: <<http://www.projectfloodlight.org/floodlight/>>. Acessado: 16-06-2019.
- [24] Trema OpenFlow Controller Website. 2018. Disponível em: <<http://trema.github.io/trema/>>. Acessado: 16-06-2019.
- [25] GONG, Y. et al. A survey on software defined networking and its applications. *Frontiers of Computer Science*, Springer and Higher Education Press, Beijing, China, v. 9, n. 6, p. 827–845, 2015. ISSN 2095-2236. Disponível em: <<https://doi.org/10.1007/s11704-015-3448-z>>.
- [26] BOSSHART, P. et al. P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, ACM, New York, NY, USA, v. 44, n. 3, p. 87–95, 2014. ISSN 0146-4833. Disponível em: <<http://doi.acm.org/10.1145/2656877.2656890>>.
- [27] SILVA, J. Santiago da; BOYER, F.-R.; LANGLOIS, J. P. P4-compatible high-level synthesis of low latency 100 gb/s streaming packet parsers in fpgas. In: *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. New York, NY, USA: ACM, 2018. (FPGA '18), p. 147–152. ISBN 978-1-4503-5614-5. Disponível em: <<http://doi.acm.org/10.1145/3174243.3174270>>.
- [28] HANCOCK, D.; MERWE, J. van der. Hyper4: Using p4 to virtualize the programmable data plane. In: *Proceedings of the 12th International Conference on Emerging Networking Experiments and Technologies*. New York, NY, USA: ACM, 2016. (CoNEXT '16), p. 35–49. ISBN 978-1-4503-4292-6. Disponível em: <<http://doi.acm.org/10.1145/2999572.2999607>>.
- [29] TU, N. V. *INTCollector: A High-performance Collector for In-band Network Telemetry*. Dissertação (Mestrado) — Department of Computer Science and Engineering – Pohang University of Science and Technology, Pohang, Korea, 2018. Disponível em: <http://dpm.postech.ac.kr/thesis/18/Tu_thesis.pdf>.

- [30] RISSO, F. *Toward Flexible and Efficient In-Kernel Network Function Chaining with IOVisor*. 2018. In: IEEE International Conference on High Performance Switching and Routing (HPSR), Bucharest, Romania. 50 slides. Disponível em: <<http://hpsr2018.ieee-hpsr.org/files/2018/06/18-06-18-IOVisor-HPSR.pdf>>. Acessado: 20-01-19.
- [31] SIRACUSANO, G.; BIFULCO, R. Is it a smartnic or a key-value store?: Both! In: *Proceedings of the SIGCOMM Posters and Demos*. New York, NY, USA: ACM, 2017. (SIGCOMM Posters and Demos '17), p. 138–140. ISBN 978-1-4503-5057-0. Disponível em: <<http://doi.acm.org/10.1145/3123878.3132014>>.
- [32] BORKMAN, D. Making the kernel's networking data path programmable with bpf and xdp. 2017. In: Open Source Summit North America '17. 17 slides. Disponível em: <https://sched.ws/hosted_files/ossna2017/da/BPFandXDP.pdf>. Acessado: 20-01-19.
- [33] VILJOEN, N. *BPF, eBPF, XDP and Bpfilter... What are These Things and What do They Mean for the Enterprise?* 2018. Disponível em: <<https://www.netronome.com/blog/bpf-ebpf-xdp-and-bpfilter-what-are-these-things-and-what-do-they-mean-enterprise/>>. Acessado: 20-01-19.
- [34] CLAISE, B. *Cisco systems NetFlow services export version 9*. 2004. In: Informational, RFC 3954, Internet Engineering Task Force. Disponível em: <<https://tools.ietf.org/html/rfc3954>>. Acessado: 20-01-19.
- [35] WANG, M.; LI, B.; LI, Z. sflow: Towards resource-efficient and agile service federation in service overlay networks. In: IEEE. *24th International Conference on Distributed Computing Systems*. Hachioji, Tokyo, Japan, Japan: IEEE, 2004. p. 628–635. ISBN 0-7695-2087-1. Disponível em: <<https://doi.org/10.1109/ICDCSW.2004.1283986>>.
- [36] HAAG, P. Netflow tools nfsen and nfdump. In: FIRST. *18th Annual FIRST Conference*. Baltimore, USA: FIRST, 2006. p. 1–31. Disponível em: <<https://www.first.org/resources/papers/conference2006/haag-peter-papers.pdf>>.
- [37] van Adrichem, N. L. M.; Doerr, C.; Kuipers, F. A. Opennetmon: Network monitoring in openflow software-defined networks. In: *2014 IEEE Network Operations and Management Symposium (NOMS)*. New Jersey, USA: IEEE, 2014. p. 1–8. ISSN 1542-1201. Disponível em: <<https://doi.org/10.1109/NOMS.2014.6838228>>.
- [38] Suh, J. et al. Opensample: A low-latency, sampling-based measurement platform for commodity sdn. In: *2014 IEEE 34th International Conference on Distributed Computing Systems*. New Jersey, USA: IEEE, 2014. p. 228–237. ISSN 1063-6927. Disponível em: <<https://doi.org/10.1109/ICDCS.2014.31>>.
- [39] YU, C. et al. Flowsense: Monitoring network utilization with zero measurement cost. In: ROUGHAN, M.; CHANG, R. (Ed.). *Passive and Active Measurement*. Berlin, Heidelberg:

Springer Berlin Heidelberg, 2013. p. 31–41. ISBN 978-3-642-36516-4. Disponível em: <https://doi.org/10.1007/978-3-642-36516-4_4>.

- [40] SERKANT. *Prometheus INT exporter*. 2017. Disponível em: <https://github.com/serkantul/prometheus_int_exporter>. Acessado: 20-01-19.
- [41] Van Tu, N.; Hyun, J.; Hong, J. W. Towards onos-based sdn monitoring using in-band network telemetry. In: *2017 19th Asia-Pacific Network Operations and Management Symposium (APNOMS)*. Seoul, South Korea: IEEE, 2017. p. 76–81. ISBN 978-1-5386-1101-2. Disponível em: <<https://doi.org/10.1109/APNOMS.2017.8094182>>.
- [42] AMPATH. *(AMPATH) America's Path*. 2019. Disponível em: <<https://ampath.net/>>. Acessado: 20-01-19.
- [43] NETRONOME. *Agilio CX 2x40GbE SmartNIC: SmartNIC for High-Performance Cloud, SDN and NFV Networking*. 2017. Disponível em: <https://www.netronome.com/m/documents/PB_Agilio_CX_2x40GbE.pdf>. Acessado: 20-01-19.
- [44] NETRONOME. *Agilio eBPF for Fast, Versatile Security: High Performance, Flexible Protection For Cloud Data Center and Edge Computing Applications*. 2019. Disponível em: <https://www.netronome.com/m/documents/PB_Agilio-eBPF.pdf>. Acessado: 20-01-19.
- [45] KICINSKI, J.; VILJOEN, N. *eBPF Hardware Offload to Smart-NICs: cls_bpf and XDP*. 2018. White Paper. 6 páginas. Disponível em: <https://www.netronome.com/m/documents/eBPF_HW_OFFLOAD_HNiMne8_2_.pdf>. Acessado: 20-01-19.
- [46] INFLUXDATA. *Real-time Visibility Into Stacks, Sensors and Systems (InfluxDB is The Open Source Time Series Database)*. 2019. Disponível em: <<https://www.influxdata.com/>>. Acessado: 20-01-19.
- [47] Guimarães, V. T. et al. A survey on information visualization for network and service management. *IEEE Communications Surveys Tutorials*, IEEE, New Jersey, USA, v. 18, n. 1, p. 285–323, 2016. ISSN 2373-745X.
- [48] CARD, S.; MACKINLAY, J.; SHNEIDERMAN, B. *Readings in Information Visualization: Using Vision To Think*. Burlington, Massachusetts, USA: Morgan Kaufmann, 1999. ISBN 978-1-55860-533-6.
- [49] PLOTLY. *Plotly JavaScript Open Source Graphing Library*. 2019. Disponível em: <<https://plot.ly/javascript/>>. Acessado: 20-01-19.

- [50] NETRONOME. *Stat Watch*. 2018. Repositório GitHub. Disponível em: <https://github.com/Netronome/nfp-drv-kmods/blob/master/tools/stat_watch.py>. Acessado: 20-01-19.
- [51] SISSEL, J. *Stat Watch*. 2016. Repositório GitHub. Disponível em: <<https://github.com/jordansissel/xdotool>>. Acessado: 20-01-19.
- [52] JAIN, R. *The art of computer systems performance analysis - techniques for experimental design, measurement, simulation, and modeling*. Wiley, 1991. XIII, 212-225 p. (Wiley professional computing). ISBN 978-0-471-50336-1. Disponível em: <http://www.eletrica.ufpr.br/pedroso/2014/TE816/Art_Of_Computer_Systems_Performance_Analysis_Techniques_For_Experimental_Measurements_Simulation_And_Modeling-Raj_Jain.pdf>.
- [53] IPERF. *iPerf - The Ultimate Speed Test Tool for TCP, UDP and SCTP*. 2019. Disponível em: <<https://iperf.fr/>>. Acessado: 20-01-19.
- [54] BRENDEN, B. *eXpress Data Path: Getting Linux to 20 Mpps*. 2016. 8 slides. Apresentado no Linux Meetup, em Santa Clara, Califórnia. Disponível em: <<https://www.slideshare.net/IOvisor/express-data-path-linux-meetup-santa-clara-july-2016>>. Acessado: 20-01-19.
- [55] SCAPY. *Scapy: Packet Crafting for Python2 and Python3*. 2019. Disponível em: <<https://scapy.net/>>. Acessado: 20-01-19.
- [56] APPNETA. *Tcpreplay - Pcap editing and replaying utilities*. 2019. Disponível em: <<https://tcpreplay.appneta.com/>>. Acessado: 20-01-19.
- [57] ZHANG, O. *XDOtool*. 2019. Repositório GitHub. Disponível em: <<https://github.com/orca-zhang/influxdb-c/blob/master/influxdb.h>>. Acessado: 20-01-19.
- [58] PROJECT, I. V. *BPF Compiler Collection*. 2019. Repositório GitHub. Disponível em: <<https://github.com/iovisor/bcc>>. Acessado: 20-01-19.
- [59] VIERA, M. A. M. et al. Processamento rápido de pacotes com ebpf e xdp. In: SBRC. *XXXVII Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos*. Gramado, Rio Grande do Sul, Brasil: SBC, 2019. p. 92–141. ISSN 2177-9384. Disponível em: <https://www.winet.dcc.ufmg.br/ebpf/processamento_rapido_de_pacotes_com_ebpf_e_xdp.pdf>.

ANEXO A – SAÍDA DO TERMINAL PARA O CAMINHO DE FLUXO

```

32 bits hexa: 5DA79684
64 bits hexa: 39cd807d
Flow Sink Time ns (decimal): 1571264132969769085
Flow Sink Time saving on DB: 1571264132969769085
Flow Sink Time = Wed Oct 16 22:15:32 2019
-----
influxdb-c::post\_http: iv[0] =
,
POST /write?db=INTdatabase\&u=root\&p= HTTP/1.1
Host: 127.0.0.1
Content-Length: 85
,

influxdb-c::post\_http: iv[1] =
,
flow\_path\, \ 10.1.1.1:255->10.1.1.5:32766\, \ proto=11 path="1:2" 1571264132969769085
,
-----
IP Origem 10.1.1.1:255 > IP Destino 10.1.1.5:32766 - Proto 11
-----
Metadados INT:

Switch IDs[0] = 2
Switch IDs[1] = 1
-----
Ingress Port ID[0] = 0
Ingress Port ID[1] = 4
-----
Egress Port ID[0] = 4
Egress Port ID[1] = 0
-----
32 bits hexa: A258697B
64 bits hexa: 39cd807d
Hop Latency[0] ns (decimal): 2723703163969769085
Hop Latency[0] ms (decimal): 0.000969

32 bits hexa: A2586988
64 bits hexa: 250c830b
Hop Latency[1] ns (decimal): 2723703176621576971
Hop Latency[1] ms (decimal): 0.000621
-----
32 bits hexa: 5DA79684
64 bits hexa: 39cd807d
Ingress Timestamp[0] ns (decimal): 1571264132969769085
Ingress TimeStamp[0] = Wed Oct 16 22:15:32 2019

32 bits hexa: 5DA79677
64 bits hexa: 250c830b
Ingress Timestamp[1] ns (decimal): 1571264119621576971
Ingress TimeStamp[1] = Wed Oct 16 22:15:19 2019
-----
32 bits hexa: 5DA79684
64 bits hexa: 39cd807d
Egress Timestamp[0] ns (decimal): 1571264132969769085
Egress TimeStamp[0] = Wed Oct 16 22:15:32 2019

32 bits hexa: 5DA79677
64 bits hexa: 250c830b
Egress Timestamp[1] ns (decimal): 1571264119621576971
Egress TimeStamp[1] = Wed Oct 16 22:15:19 2019
=====

```


ANEXO B – UTILITÁRIO DE VISUALIZAÇÃO

```

const loadData = () => {

  //Dropdown menu
  let dropdown = document.getElementById('flow-dropdown');
  dropdown.length = 0;

  let defaultOption = document.createElement('option');
  defaultOption.text = 'Choose Measurement/Flow';

  dropdown.add(defaultOption);
  dropdown.selectedIndex = 0;

  //Get the measurements
  fetch('/api/v1/meas')
    .then(
      function(response) {
        if (response.status !== 200) {
          console.warn('Looks like there was a problem. Status Code: ' +
            response.status);
          return;
        }

        // Examine the text in the response
        response.json().then(function(data) {
          let option;

          for (let i = 0; i < data.length; i++) {
            option = document.createElement('option');
            option.text = data[i].name;
            dropdown.add(option);
          }
        });
      }
    )
    .catch(function(err) {
      console.error('Fetch Error -', err);
    });

  var current_option;
  var defaultInterval;
  var newInterval;

  function killRefresh() {
    //for (var i = 1; i < defaultInterval; i++)
    clearInterval(defaultInterval);
    //clearInterval(defaultInterval);
    console.log("Killed refresh");
  }

  $("#flow-dropdown").change(function() {
    current_option = $(this).val();
    killRefresh();
    console.log(current_option);
  });

  function changeStatus() {
    if ($("#mute-button").hasClass("MuteOff")) {
      $("#mute-button").removeClass("MuteOff").addClass("MuteOn");
      $("#mute-button").removeClass("btn-info").addClass("btn-danger");
      document.getElementById("buttonUp").innerHTML = 'Stop Trace';
    }
  }

  //Connect with DB
  $(document).ready(function() {
    function send() {
      //Get available button send
      document.getElementById('buttonUp').removeAttribute('disabled');
      //Get available dropdown refreshInterval
      document.getElementById('refresh-dropdown').removeAttribute('disabled');
      $.getJSON('http://67.17.206.218/db_connection.php?value=' + current_option,

```

```

function(data) {
    var str = current_option;
    var substring1 = "flow_hop_latency";
    var substring2 = "flow_path";

    if (str.includes(substring1) === true) { //start-if for flow_hop_latency
        //determine ID switch
        var id = str.search("id");
        var id_pos = id + 3;
        document.getElementById("result-header").innerHTML = "Packet Flow Latency per Hop" + "<br>";
        document.getElementById("result").innerHTML = "Switch ID: " + str.charAt(id_pos);

        let trace1 = {
            x: [],
            y: [],
            mode: "lines"
        };
        data.results[0].series[0].values.forEach(function(val) {
            trace1.x.push(val[0]);
            trace1.y.push(val[1]);
        });
        Plotly.newPlot('graphs-container2', [trace1]);
    }

    if (str.includes(substring2) === true) { //start-if for flow_path
        console.log("you did");
        var flowPath1 = str.substr(0, str.indexOf(','));
        var flowPathMinus = str.replace(flowPath1 + ",", '');
        console.log("FlowPath1: " + flowPath1);
        console.log("FlowPathMinus: " + flowPathMinus);
        document.getElementById("result-header").innerHTML = "Packet Flow Path" + "<br>";
        document.getElementById("result").innerHTML = "Flow Path: " + flowPathMinus;
        var query = current_option;
        console.log("Query: " + query);

        function getJSONPath(query) {
            var result = null;
            var scriptUrl = query;
            $.ajax({
                url: 'http://67.17.206.218/db_connection.php?value=' + scriptUrl + '&path=path',
                type: 'get',
                dataType: 'JSON',
                async: false,
                success: function(data) {
                    result = data;
                }
            });
            return result;
        }

        var getJSONQuery = getJSONPath(query);
        var data = getJSONQuery.results[0].series[0].values[0][1];
        var dataString = JSON.stringify(data);
        console.log("dataString: " + dataString);

        dataString = dataString.replace(/"/g, ''); //remove double quotation marks
        dataString = dataString.replace(/\[\|\]/g, ''); //remove brackets from string
        console.log("Depois do regex replace: " + dataString);
        console.log(typeof dataString);

        var parsed = parse(dataString) //parse string function
        console.log("Substituido o ':' por ',' : " + parsed);
        console.log(typeof parsed);

        function parse(input) {
            string = input.split(':'); //put string in an array, replacing ':'
            return string;
        }

        var array = JSON.parse("[ " + parsed + " ]") //compose a new array with string inside
        //Array [0,1,2]
        console.log("Assumindo formato de array '[1,2,3]': " + array);

        var sourcePop = array.slice(0); //clone an array
        console.log("SourcePop: " + sourcePop);

        var targetShift = array.slice(0);
        console.log("TargetShif: " + sourcePop);

        sourceArray = sourcePop;
        targetShift.shift(); //delete the first element of an array
    }
}

```



```

targetArray = targetShift;
targetArray.push(7); //insert "Sink"
console.log("TargetArray: " + sourcePop);

var arr = array.slice(0);
console.log("arr: " + arr);

//function to build up the array value
var valueArray = getArrayValue(arr);
console.log("ValueArray: " + valueArray);

function getArrayValue(array) {
    originalArray = array.slice(0);
    console.log("OriginalArray: " + Object.keys(originalArray).length);
    var valueArray = [];

    for (i = 0; i < Object.keys(originalArray).length; i++) {
        function url(array) {
            var array = array;
            var query = 'flow_hop_latency,' + flowPathMinus + ', sw_id[' + array[i] + ']';
            return query;
        }

        function getJSON(urlquery) {
            var result = null;
            var scriptUrl = urlquery;
            $.ajax({
                url: 'http://67.17.206.218/db_connection.php?value=' + scriptUrl + '&type=sankey',
                type: 'get',
                dataType: 'JSON',
                async: false,
                success: function(data) {
                    result = data;
                }
            });

            return result;
        }

        var urlquery = url(originalArray);
        var getJSONquery = getJSON(urlquery);
        var data = getJSONquery.results[0].series[0].values[0][1];

        valueArray.push(data);
    }
    return valueArray;
}

var labelArray = [];

switch (array.length) {
    case 1:
        labelArray = ["Hop 1"];
        break;
    case 2:
        labelArray = ["Hop 1", "Hop 2"];
        break;
    case 3:
        labelArray = ["Hop 1", "Hop 2", "Hop 3"];
        break;
    case 4:
        labelArray = ["Hop 1", "Hop 2", "Hop 3", "Hop 4"];
        break;
    case 5:
        labelArray = ["Hop 1", "Hop 2", "Hop 3", "Hop 4", "Hop 5"];
        break;
    case 6:
        labelArray = ["Hop 1", "Hop 2", "Hop 3", "Hop 4", "Hop 5", "Hop 6"];
        break;
    default:
        labelArray = [];
}

console.log("Source str: " + JSON.stringify(sourceArray));
console.log("Target str: " + JSON.stringify(targetArray));
console.log("Value str: " + JSON.stringify(valueArray));
console.log("Label str: " + JSON.stringify(labelArray));
console.log(sourceArray);
console.log(targetArray);
console.log(valueArray);
console.log(labelArray);

```

```

Plotly.d3.json('../JSON/banco.json', function(fig) {
  var data = {
    type: "sankey",
    domain: {
      x: [0, 1],
      y: [0, 1]
    },
    orientation: "h",
    valueformat: ".0f",
    valuesuffix: "ns",
    node: {
      pad: 15,
      thickness: 15,
      line: {
        color: "black",
        width: 0.5
      },
      label: [
        "0",
        "Switch 1",
        "Switch 2",
        "Switch 3",
        "Switch 4",
        "Switch 5",
        "Switch 6",
        "Sink"
      ],
      color: fig.data[0].node.color
    },
    link: {
      source: sourceArray,
      target: targetArray,
      value: valueArray,
      label: labelArray
    }
  }
  Plotly.react('graphs-container2', [data])
});

});

$("#send").click(killRefresh);
$("#send").click(send);
$("#send").click(changeStatus);
$("#send").click(setDefaultInterval);

//if send was clicked, set default interval
function setDefaultInterval() {
  defaultInterval = setInterval(send, 5000);
  console.log("Default value set " + defaultInterval);
}

//refresh Interval
$("#refreshInterval").change(function() {
  newInterval = parseInt($(this).val()) * 1000;
  clearInterval(defaultInterval);
  console.log("Old Interval cleared: " + defaultInterval);
  console.log("New Interval: " + newInterval);
  //refresh only if MuteOn was available
  if ($("#mute-button").hasClass("MuteOn")) {
    if (newInterval > 0) {
      //defaultInterval = setInterval(send, newInterval)
      console.log("Settled data");
    } else {
      killRefresh();
      console.log("Has muteOn");
    }
  }
  if ($("#mute-button").hasClass("MuteOff")) {
    killRefresh();
    console.log("Has muteOff");
  }
})

//start-stop tracing function
$('#mute-button').on('click tap touch', function() {
  var $this = $(this);

```

```

if ($this.hasClass("MuteOn")) {
    $this.removeClass("MuteOn").addClass("MuteOff");
    $this.removeClass("btn-danger").addClass("btn-info");
    document.getElementById("buttonUp").innerHTML = 'Start Trace';
    killRefresh();
} else {
    $this.removeClass("MuteOff").addClass("MuteOn");
    $this.removeClass("btn-info").addClass("btn-danger");
    document.getElementById("buttonUp").innerHTML = 'Stop Trace';
    killRefresh();
    if (newInterval > 0) {
        console.log("Interval: " + newInterval);
    }
    if (newInterval <= 0) {
        killRefresh();
        $("#alert-zero span").text("Please set an interval different from 'Off'");
        $("#alert-zero").css('display', '');
    } else {
        var e = document.getElementById("refresh-dropdown");
        var strUser = e.options[e.selectedIndex].value;
        console.log("strUser: " + strUser);
        withoutInterval = strUser * 1000;
        console.log("withoutInterval: " + withoutInterval);
        defaultInterval = setInterval(send, withoutInterval);
    }
}
});
});
}

$(window).on('load', loadData);

```


ANEXO C – ANALISADOR DO COLETOR (EBPF_PROG_USER.C)

```

#include <assert.h>
#include <libgen.h>
#include <perf-sys.h>
#include <poll.h>
#include <signal.h>
#include <stdlib.h>
#include <string.h>
#include <arpa/inet.h>
#include <bpf/bpf.h>
#include <bpf/libbpf.h>
#include <linux/if_ether.h>
#include <linux/if_link.h>
#include <net/if.h>
#include <sys/ioctl.h>
#include <sys/mman.h>
#include <sys/sysinfo.h>
#include <time.h>
#include <stdio.h>
#include <math.h>
#include <string.h>
#include "influxdb.h"
#include "xdpdump_common.h"
#include <stdint.h>
#include <math.h>
#include "ebpf_collector.h"

#define NS_IN_SEC 1000000000
#define PAGE_CNT 8
#define MAX_INT_HOP 6

struct perf_event_sample {
    struct perf_event_header header;
    __u32 size;
    char data[];
};

static __u32 xdp_flags;
static int ifindex;
bool dump_payload;

//convert hex to 64 bits decimal
__u64 hexToDecimal64 (char hex[17]) {
    __u64 decimal, place;
    int i = 0, val, len;
    decimal = 0;
    place = 1;
    len = strlen(hex);
    len--;
    for(i=0; hex[i]!='\0'; i++) {
        if(hex[i]>='0' && hex[i]<='9') {
            val = hex[i] - 48;
        } else if(hex[i]>='a' && hex[i]<='f') {
            val = hex[i] - 97 + 10;
        } else if(hex[i]>='A' && hex[i]<='F') {
            val = hex[i] - 65 + 10;
        }
        decimal += val * pow(16, len);
        len--;
    }
    return decimal;
}

//change hexadecimal case
void upper_string(char s[]) {
    int c = 0;
    while (s[c] != '\0') {
        if (s[c] >= 'a' && s[c] <= 'z') {
            s[c] = s[c] - 32;
        }
        c++;
    }
}

//Convert hex to decimal

```

```

__u32 hexToDecimal(char hexVal[]) {
    upper_string(hexVal);
    int len = strlen(hexVal);

    // Initializing base value to 1, i.e 16^0
    int base = 1;
    double dec_val = 0;

    // Extracting characters as digits from last character
    for (int i = len - 1; i >= 0; i--) {
        // if in '0'-'9', converts by subtracting 48 from ASCII value
        if (hexVal[i] >= '0' && hexVal[i] <= '9') {
            dec_val += (hexVal[i] - 48) * base;
            // incrementing base by power
            base = base * 16;
        }
        // if in 'A'-'F', converts by subtracting 55 from ASCII value
        else if (hexVal[i] >= 'A' && hexVal[i] <= 'F') {
            dec_val += (hexVal[i] - 55) * base;
            // incrementing base by power
            base = base * 16;
        }
    }
    return dec_val;
}

// variable to save file descriptor (out of scope)
int map;

//function to save file descriptor
static int map_stats (int offload_map_fd) {
    map = offload_map_fd;
}

static void unload_prog (int sig) {
    bpf_set_link_xdp_fd (ifindex, -1, xdp_flags);
    printf ("unloading xdp program...\n");
    exit (0);
}

void meta_print (void *data, int size) {
    //void meta_print(void * data, __u64 timestamp) {
    struct {
        __u32 src_ip;
        __u32 dst_ip;
        __u16 src_port;
        __u16 dst_port;
        __u16 ip_proto;

        __u8 num_INT_hop;

        __u32 switch_id[MAX_INT_HOP];
        __u16 ingress_port_id[MAX_INT_HOP];
        __u16 egress_port_id[MAX_INT_HOP];
        __u64 hop_latency[MAX_INT_HOP];
        __u64 ingress_timestamp[MAX_INT_HOP];
        __u64 egress_timestamp[MAX_INT_HOP];

        __u32 cnt_packets[MAX_INT_HOP];
        __u32 cnt_bytes[MAX_INT_HOP];

        __u64 flow_sink_timestamp;
        __u64 flow_latency;

        __u8 new_flow;
        __u8 flow_alive;
    } *e = data;

    //key for map lookup/update
    struct quintuple_t key_map;
    key_map.source_ip = e->src_ip;
    key_map.destiny_ip = e->dst_ip;
    key_map.source_port = e->src_port;
    key_map.destiny_port = e->dst_port;
    key_map.ip_protocol = e->ip_proto;

    //value for map lookup/update
    struct map_t value_map;

    //recovering offload_map_fd value by map variable

```

```

int map_fd = map;

//Show if map was not found
if (bpf_map_lookup_elem (map_fd, &key_map, &value_map) < 0) {
    printf ("Err: lookup does not find map \n");
}

//Recovers map value if it returns 0 (It's ok)
if (bpf_map_lookup_elem (map, &key_map, &value_map) == 0) {
    printf ("Map was found \n");
}

//older
__u64 older = htobe64 (value_map.flow_latency);
printf ("Flow latency older: %lld \n", older);

//older to string
char olderString[64];
sprintf (olderString, "%llx", older);
__u64 older_converted = hexToDecimal (olderString);

//newer
__u64 newer = htobe64 (e->flow_latency);
printf ("Flow latency newer: %lld \n", newer);

//newer to string
char newerString[64];
sprintf (newerString, "%llx", newer);
__u64 newer_converted = hexToDecimal (newerString);

//get fist two digits
int first_result = (older_converted/2)/ 100;
int second_result = (newer_converted/2) / 100;
printf ("Flow latencies older: %d and new %d \n", first_result, second_result);

if (first_result != second_result) {
    char src_str[INET6_ADDRSTRLEN];
    char dst_str[INET6_ADDRSTRLEN];
    char src_port_str[16];
    char dst_port_str[16];
    char proto_str[16];
    char meas[250];
    char path_str[32];
    char str_path[50] = "";

    //IP string
    inet_ntop (AF_INET, &e->src_ip, src_str, INET_ADDRSTRLEN);
    inet_ntop (AF_INET, &e->dst_ip, dst_str, INET_ADDRSTRLEN);

    //IP string
    inet_ntop (AF_INET, &e->src_ip, src_str, INET_ADDRSTRLEN);
    inet_ntop (AF_INET, &e->dst_ip, dst_str, INET_ADDRSTRLEN);

    //"casting" deci/hexa to string
    sprintf (src_port_str, "%d", ntohs (e->src_port));
    sprintf (dst_port_str, "%d", ntohs (e->dst_port));
    sprintf (proto_str, "%x", e->ip_proto);

    //influx connection
    influx_client_t c;
    c.host = strdup ("127.0.0.1");
    c.port = 8086;
    c.db = strdup ("INTdatabase");
    c.usr = strdup ("root");
    c.pwd = strdup ("");

    //measurement set
    sprintf (meas, "flow_path, %s:%s->%s:%s, proto=%s", src_str,
            src_port_str, dst_str, dst_port_str, proto_str);

    __u8 j = 0;

    //make out string path
    for (int i = e->num_INT_hop - 1; i >= 0; i--) {
        char sw_id_str[32];
        sprintf (sw_id_str, "%x", ntohl (e->switch_id[i]));
        if (i != 0) {
            sprintf (path_str, "%s:", sw_id_str);
        } else {
            sprintf (path_str, "%s", sw_id_str);
        }
    }
}

```

```

        strcat (str_path, path_str);
    }

    //determine timestamp as flow_sink_timestamp
    __u64 val_32 = htobe64 (e->flow_sink_timestamp << 32);
    char time_32[64];
    sprintf (time_32, "%llx", val_32);
    __u64 time_path_read = hexToDecimal (time_32);

    //Showing in nanoseconds
    __u32 val_64 = htobe64 (e->flow_sink_timestamp);
    char time_64[32];
    sprintf (time_64, "%x", val_64);
    printf ("First 32 bits hexa: %s \n", time_32);
    printf ("Last 64 bits hexa: %s \n", time_64);
    printf ("Flow Sink Time ns: %lld \n", (val_32 * NS_IN_SEC) + val_64);

    long long sink_ts_db = (val_32 * NS_IN_SEC) + val_64;
    printf ("Sink TS going to DB: %lld \n", sink_ts_db);

    //make a string from sink_ts_db
    char time_path_db[32];
    sprintf (time_path_db, "%lld", sink_ts_db);

    //conversion in human readable time
    time_t tm = time_path_read;
    printf ("Flow Sink Time = %s \n", ctime (&tm));

    //for Sankey's graph
    char label_str1[10] = "Switch 1";
    char label_str2[10] = "Switch 2";
    char label_str3[10] = "Switch 3";
    char label_str4[10] = "Switch 4";
    char label_str5[10] = "Switch 5";
    char label_str6[10] = "Switch 6";

    printf ("\n ----- \n");

    //sending info to InfluxDB
    post_http (&c,
              INFLUX_MEAS (&meas),
              INFLUX_F_STR ("path", &str_path),
              INFLUX_TS (&time_path_db),
              INFLUX_END);

    printf ("\n ----- \n");

    //Hop_latency
    for (int i = 0; i < e->num_INT_hop; i++) {

        //hexa to decimal 32 bits
        __u64 val_32 = htobe64 (e->hop_latency[i] << 32);
        char time_32[64];
        sprintf (time_32, "%llx", val_32);

        //hexa to decimal 64 bits
        __u32 val_64 = htobe64 (e->hop_latency[i]);
        char time_64[32];
        sprintf (time_64, "%x", val_64);

        long long hop_ts_db = (val_32 * NS_IN_SEC) + val_64;

        //make a string from hop_ts_db
        char time_hop_db[32];
        sprintf (time_hop_db, "%lld", hop_ts_db);
        printf ("Hop TS going to DB: %lld \n", hop_ts_db);

        //string switch ID
        char sw_id_str[32];
        sprintf (sw_id_str, "%x", ntohs (e->switch_id[i]));

        //measurement set
        sprintf (meas, "flow_hop_latency, %s:%s->%s:%s, proto=%s, sw_id[%s]",
                src_str, src_port_str, dst_str, dst_port_str, proto_str, sw_id_str);

        //determine timestamp as egress_sink_timestamp
        __u64 egr_val_32 = htobe64 (e->egress_timestamp[i] << 32);
        char time_egr_val_32[64];
        sprintf (time_egr_val_32, "%llx", egr_val_32);

        //Showing in nanoseconds

```



```

__u32 egr_val_64 = htobe64 (e->egress_timestamp[i]);
char time_egr_val_64[32];
sprintf (time_egr_val_64, "%x", egr_val_64);

long long egr_ts_db = (egr_val_32 * NS_IN_SEC) + egr_val_64;
printf ("Egress TS going to DB: %lld \n", egr_ts_db);

//make a string from egr_ts_db
char time_egr_db[32];
sprintf (time_egr_db, "%lld", egr_ts_db);

printf("\n ----- \n");

printf("Hop latency para switch ID[%x] \n", ntohl (e->switch_id[i]));

//sending info to InfluxDB
post_http (&c,
           INFLUX_MEAS (&meas),
           INFLUX_F_STR ("hop_lat", &time_hop_db),
           INFLUX_TS (&time_egr_db),
           INFLUX_END
);

printf("\n ----- \n");
}

printf("IP Origen %s:%d > IP Destino %s:%d - Proto %x\n",
       src_str, ntohs (e->src_port),
       dst_str, ntohs (e->dst_port), e->ip_proto);

//Save into map
struct map_t value_map_actual;
value_map_actual.flow_sink_timestamp = e->flow_sink_timestamp;
value_map_actual.flow_latency = e->flow_latency;

//Show if map was updated
if (bpf_map_update_elem (map_fd, &key_map, &value_map_actual, BPF_ANY) <0) {
    printf("Err: Map does not updated \n");
}
}

printf("\n ----- \n");

//Printing Metadata
printf("Metadados INT: \n \n");

for (int i = 0; i < e->num_INT_hop; i++) {
    printf ("Switch ID[%d] = %x \n", i, ntohl (e->switch_id[i]));
}

printf("\n ----- \n");

for (int i = 0; i < e->num_INT_hop; i++) {
    printf ("Ingress Port ID[%d] = %d \n", i,
           ntohs (e->ingress_port_id[i]));
}

printf("\n ----- \n");

for (int i = 0; i < e->num_INT_hop; i++) {
    printf ("Egress Port ID[%d] = %x \n", i, ntohs (e->egress_port_id[i]));
}

printf("\n ----- \n");

for (int i = 0; i < e->num_INT_hop; i++) {

    //hexa to decimal 32 bits
    __u64 val_32 = htobe64 (e->hop_latency[i] << 32);
    char time_32[64];
    sprintf (time_32, "%llx", val_32);

    //Showing in nanoseconds
    __u32 val_64 = htobe64 (e->hop_latency[i]);
    char time_64[32];
    sprintf (time_64, "%x", val_64);
    printf ("First 32 bits hexa: %s \n", time_32);
    printf ("Last 64 bits hexa: %s \n", time_64);
    char meas[250];
    char path_str[32];
    char str_path[50] = "";
    char proto_str[16];

```

```

//IP string
inet_ntop (AF_INET, &e->src_ip, src_str, INET_ADDRSTRLEN);
inet_ntop (AF_INET, &e->dst_ip, dst_str, INET_ADDRSTRLEN);

//IP string
inet_ntop (AF_INET, &e->src_ip, src_str, INET_ADDRSTRLEN);
inet_ntop (AF_INET, &e->dst_ip, dst_str, INET_ADDRSTRLEN);

//"casting" deci/hexa to string
sprintf (src_port_str, "%d", ntohs (e->src_port));
sprintf (dst_port_str, "%d", ntohs (e->dst_port));
sprintf (proto_str, "%x", e->ip_proto);

//influx connection
influx_client_t c;
c.host = strdup ("127.0.0.1");
c.port = 8086;
c.db = strdup ("INTdatabase");
c.usr = strdup ("root");
c.pwd = strdup ("");

//measurement set
sprintf (meas, "flow_path, %s:%s->%s:%s, proto=%s", src_str,
        src_port_str, dst_str, dst_port_str, proto_str);

__u8 j = 0;

//make out string path
for (int i = e->num_INT_hop - 1; i >= 0; i--) {
    char sw_id_str[32];
    sprintf (sw_id_str, "%x", ntohs (e->switch_id[i]));
    if (i != 0) {
        sprintf (path_str, "%s:", sw_id_str);
    } else {
        sprintf (path_str, "%s", sw_id_str);
    }
    strcat (str_path, path_str);
}

//determine timestamp as flow_sink_timestamp
__u64 val_32 = htobe64 (e->flow_sink_timestamp << 32);
char time_32[64];
sprintf (time_32, "%llx", val_32);
__u64 time_path_read = hexToDecimal (time_32);

//Showing in nanoseconds
__u32 val_64 = htobe64 (e->flow_sink_timestamp);
char time_64[32];
sprintf (time_64, "%x", val_64);
printf ("First 32 bits hexa: %s \n", time_32);
printf ("Last 64 bits hexa: %s \n", time_64);
printf ("Flow Sink Time ns: %lld \n", (val_32 * NS_IN_SEC) + val_64);

long long sink_ts_db = (val_32 * NS_IN_SEC) + val_64;
printf ("Sink TS going to DB: %lld \n", sink_ts_db);

//make a string from sink_ts_db
char time_path_db[32];
sprintf (time_path_db, "%lld", sink_ts_db);

//conversion in human readable time
time_t tm = time_path_read;
printf ("Flow Sink Time = %s \n", ctime (&tm));

//for Sankey's graph
char label_str1[10] = "Switch 1";
char label_str2[10] = "Switch 2";
char label_str3[10] = "Switch 3";
char label_str4[10] = "Switch 4";
char label_str5[10] = "Switch 5";
char label_str6[10] = "Switch 6";

printf ("\n ----- \n");

//sending info to InfluxDB
post_http (&c,
        INFLUX_MEAS (&meas),
        INFLUX_F_STR ("path", &str_path),
        INFLUX_TS (&time_path_db),
        INFLUX_END);

```

```

printf ("\n ----- \n");

//Hop_latency
for (int i = 0; i < e->num_INT_hop; i++) {

    //hexa to decimal 32 bits
    __u64 val_32 = htobe64 (e->hop_latency[i] << 32);
    char time_32[64];
    sprintf (time_32, "%llx", val_32);

    //hexa to decimal 64 bits
    __u32 val_64 = htobe64 (e->hop_latency[i]);
    char time_64[32];
    sprintf (time_64, "%x", val_64);

    long long hop_ts_db = (val_32 * NS_IN_SEC) + val_64;

    //make a string from hop_ts_db
    char time_hop_db[32];
    sprintf (time_hop_db, "%lld", hop_ts_db);
    printf ("Hop TS going to DB: %lld \n", hop_ts_db);

    //string switch ID
    char sw_id_str[32];
    sprintf (sw_id_str, "%x", ntohl (e->switch_id[i]));

    //measurement set
    sprintf (meas, "flow_hop_latency, %s:%s->%s:%s, proto=%s, sw_id[%s]",
            src_str, src_port_str, dst_str, dst_port_str, proto_str, sw_id_str);

    //determine timestamp as egress_sink_timestamp
    __u64 egr_val_32 = htobe64 (e->egress_timestamp[i] << 32);
    char time_egr_val_32[64];
    sprintf (time_egr_val_32, "%llx", egr_val_32);

    //Showing in nanoseconds
    __u32 egr_val_64 = htobe64 (e->egress_timestamp[i]);
    char time_egr_val_64[32];
    sprintf (time_egr_val_64, "%x", egr_val_64);

    long long egr_ts_db = (egr_val_32 * NS_IN_SEC) + egr_val_64;
    printf ("Egress TS going to DB: %lld \n", egr_ts_db);

    //make a string from egr_ts_db
    char time_egr_db[32];
    sprintf (time_egr_db, "%lld", egr_ts_db);

    printf ("\n ----- \n");

    printf ("Hop latency para switch ID[%x] \n", ntohl (e->switch_id[i]));

    //sending info to InfluxDB
    post_http (&c,
              INFLUX_MEAS (&meas),
              INFLUX_F_STR ("hop_lat", &time_hop_db),
              INFLUX_TS (&time_egr_db),
              INFLUX_END
    );

    printf ("\n ----- \n");
}

printf ("IP Origen %s:%d > IP Destino %s:%d - Proto %x\n",
        src_str, ntohs (e->src_port),
        dst_str, ntohs (e->dst_port), e->ip_proto);

//Save into map
struct map_t value_map_actual;
value_map_actual.flow_sink_timestamp = e->flow_sink_timestamp;
value_map_actual.flow_latency = e->flow_latency;

//Show if map was updated
if (bpf_map_update_elem (map_fd, &key_map, &value_map_actual, BPF_ANY) < 0) {
    printf ("Err: Map does not updated \n");
}
}

printf ("\n ----- \n");

```

```

//Printing Metadata
printf("Metadados INT: \n \n");

for (int i = 0; i < e->num_INT_hop; i++) {
    printf ("Switch ID[%d] = %x \n", i, ntohl (e->switch_id[i]));
}

printf("\n ----- \n");

for (int i = 0; i < e->num_INT_hop; i++){
    printf ("Ingress Port ID[%d] = %d \n", i,
        ntohs (e->ingress_port_id[i]));
}

printf("\n ----- \n");

for (int i = 0; i < e->num_INT_hop; i++) {
    printf ("Egress Port ID[%d] = %x \n", i, ntohs (e->egress_port_id[i]));
}

printf("\n ----- \n");

for (int i = 0; i < e->num_INT_hop; i++) {

    //hexa to decimal 32 bits
    __u64 val_32 = htobe64 (e->hop_latency[i] << 32);
    char time_32[64];
    sprintf (time_32, "%llx", val_32);

    //Showing in nanoseconds
    __u32 val_64 = htobe64 (e->hop_latency[i]);
    char time_64[32];
    sprintf (time_64, "%x", val_64);
    printf ("First 32 bits hexa: %s \n", time_32);
    printf ("Last 64 bits hexa: %s \n", time_64);

    printf ("First 32 bits hexa: %s \n", time_32);
    printf ("Last 64 bits hexa: %s \n", time_64);
    printf ("Hop Latency[%d] ns (decimal): %lld \n \n", i, (val_32 * NS_IN_SEC) + val_64);
    printf ("Hop Latency[%d] ms (decimal): 0.%06d \n", i, val_64);
    printf ("Hop Latency[%d]: %d \n \n", i, val_64);

}

printf("\n ----- \n");

for (int i = 0; i < e->num_INT_hop; i++) {

    //hexa to decimal
    __u64 val_32 = htobe64 (e->ingress_timestamp[i] << 32);
    char time_32[64];
    sprintf (time_32, "%llx", val_32);
    __u64 ingr_time_read = hexToDecimal (time_32);

    //Showing in nanoseconds
    __u32 val_64 = htobe64 (e->ingress_timestamp[i]);
    char time_64[32];
    sprintf (time_64, "%x", val_64);
    printf ("First 32 bits hexa: %s \n", time_32);
    printf ("Last 64 bits hexa: %s \n", time_64);
    printf ("Ingress Timestamp[%d] ns (decimal): %lld \n", i, (val_32 * NS_IN_SEC) + val_64);

    //conversion in human readable time
    time_t tm = ingr_time_read;
    printf ("Ingress TimeStamp[%d] = %s \n \n", i, ctime (&tm));
}

printf("\n ----- \n");

for (int i = 0; i < e->num_INT_hop; i++) {

    //hexa to decimal
    __u64 val_32 = htobe64 (e->egress_timestamp[i] << 32);
    char time_32[64];
    sprintf (time_32, "%llx", val_32);
    __u64 egr_time_read = hexToDecimal (time_32);

    //Showing in nanoseconds
    __u32 val_64 = htobe64 (e->egress_timestamp[i]);
    char time_64[32];

```

```

    sprintf (time_64, "%x", val_64);
    printf ("First 32 bits hexa:  %s \n", time_32);
    printf ("Last 64 bits hexa:  %s \n", time_64);
    printf ("Egress Timestamp[%d] ns (decimal):  %lld \n", i, (val_32 * NS_IN_SEC) + val_64);

    //conversion in human readable time
    time_t tm = egr_time_read;
    printf ("Egress TimeStamp[%d] = %s \n \n", i, ctime (&tm));
}

printf("\n ----- \n");

for (int i = 0; i < e->num_INT_hop; i++) {
    //Counting packets
    __u32 cnt_packets = htobe64(e->cnt_packets[i]);
    printf("Counting packets:  %d \n", cnt_packets);
}

printf("\n ----- \n");

for (int i = 0; i < e->num_INT_hop; i++) {
    //Counting bytes
    __u32 cnt_bytes = htobe64(e->cnt_bytes[i]);
    printf("Counting packets:  %d \n", cnt_bytes);
}

printf("\n ===== \n");
}

int event_printer (struct perf_event_sample *sample) {
    int i;
    meta_print (sample->data, sample->size);
    return LIBBPF_PERF_EVENT_CONT;
}

static enum bpf_perf_event_ret event_received (void *event, void *printfn) {
    int (*print_fn) (struct perf_event_sample *) = printfn;
    struct perf_event_sample *sample = event;

    if (sample->header.type == PERF_RECORD_SAMPLE)
        return print_fn (sample);
    else
        return LIBBPF_PERF_EVENT_CONT;
}

int event_poller (struct perf_event_mmap_page **mem_buf, int *sys_fds, int cpu_total) {
    struct pollfd poll_fds[MAX_CPU];
    void *buf = NULL;
    size_t len = 0;
    int total_size;
    int pagesize;
    int res;
    int n;

    /* Create pollfd struct to contain poller info */
    for (n = 0; n < cpu_total; n++) {
        poll_fds[n].fd = sys_fds[n];
        poll_fds[n].events = POLLIN;
    }

    pagesize = getpagesize ();
    total_size = PAGE_CNT * pagesize;
    for (;;) {
        /* Poll fds for events, 250ms timeout */
        poll (poll_fds, cpu_total, 250);

        for (n = 0; n < cpu_total; n++) {
            if (poll_fds[n].revents) {
                /* events found */
                res = bpf_perf_event_read_simple (mem_buf[n],
                                                  total_size,
                                                  pagesize, &buf, &len,
                                                  event_received,
                                                  event_printer);

                if (res != LIBBPF_PERF_EVENT_CONT)
                    break;
            }
        }
    }
}

```

```

    free (buf);
}

int setup_perf_poller (int perf_map_fd, int *sys_fds, int cpu_total, struct perf_event_mmap_page **mem_buf) {
    struct perf_event_attr attr = {
        .sample_type = PERF_SAMPLE_RAW,
        .type = PERF_TYPE_SOFTWARE,
        .config = PERF_COUNT_SW_BPF_OUTPUT,
        .wakeup_events = 1,
    };
    int mmap_size;
    int pmu;
    int n;

    mmap_size = getpagesize () * (PAGE_CNT + 1);

    for (n = 0; n < cpu_total; n++) {
        /* create perf fd for each thread */
        pmu = sys_perf_event_open (&attr, -1, n, -1, 0);
        if (pmu < 0) {
            printf ("error setting up perf fd\n");
            return 1;
        }
        /* enable PERF events on the fd */
        ioctl (pmu, PERF_EVENT_IOC_ENABLE, 0);

        /* give fd a memory buf to write to */
        mem_buf[n] = mmap (NULL, mmap_size, PROT_READ | PROT_WRITE,
            MAP_SHARED, pmu, 0);
        if (mem_buf[n] == MAP_FAILED) {
            printf ("error creating mmap\n");
            return 1;
        }
        /* point eBPF map entries to fd */
        assert (!bpf_map_update_elem (perf_map_fd, &n, &pmu, BPF_ANY));
        sys_fds[n] = pmu;
    }
    return 0;
}

static void usage (const char *prog) {
    fprintf (stderr,
        "%s -i interface [OPTS]\n\n"
        "OPTS:\n"
        "  -h      help\n"
        "  -H      Hardware Mode (XDPOFFLOAD)\n"
        "  -N      Native Mode (XDPDRV)\n"
        "  -S      SKB Mode (XDPGENERIC)\n"
        "  -x      Show packet payload\n", prog);
}

int main (int argc, char **argv){
    static struct perf_event_mmap_page *mem_buf[MAX_CPU];
    struct bpf_prog_load_attr prog_load_attr = {
        .prog_type = BPF_PROG_TYPE_XDP,
        .file = "ebpf_prog_kern.o",
    };
    struct bpf_map *perf_map;
    struct bpf_map *offload_map;
    struct bpf_object *obj;
    int sys_fds[MAX_CPU];
    int perf_map_fd;
    int offload_map_fd;
    int prog_fd;
    int n_cpus;
    int opt;

    xdp_flags = XDP_FLAGS_DRV_MODE;          /* default to DRV */
    n_cpus = get_nprocs ();
    dump_payload = 0;

    if (optind == argc) {
        usage (basename (argv[0]));
        return -1;
    }

    while ((opt = getopt(argc, argv, "hHi:NSx")) != -1) {
        switch (opt) {
            while ((opt = getopt(argc, argv, "hHi:NSx")) != -1) {
                switch (opt) {
                    case 'h':

```

```

        usage(basename(argv[0]));
        return 0;
    case 'H':
        xdp_flags = XDP_FLAGS_HW_MODE;
        prog_load_attr.ifindex = ifindex; /* set HW ifindex */
        break;
    case 'i':
        ifindex = if_nametoindex(optarg);
        break;
    case 'N':
        xdp_flags = XDP_FLAGS_DRV_MODE;
        break;
    case 'S':
        xdp_flags = XDP_FLAGS_SKB_MODE;
        break;
    case 'x':
        dump_payload = 1;
        break;
    default:
        printf("incorrect usage\n");
        usage(basename(argv[0]));
        return -1;
}
}

if (ifindex == 0) {
    printf ("error, invalid interface\n");
    return -1;
}

/* use libbpf to load program */
if (bpf_prog_load_xattr (&prog_load_attr, &obj, &prog_fd)) {
    printf ("error with loading file\n");
    return -1;
}

if (prog_fd < 1){
    printf ("error creating prog_fd\n");
    return -1;
}

signal (SIGINT, unload_prog);
signal (SIGTERM, unload_prog);

/* use libbpf to link program to interface with corresponding flags */
if (bpf_set_link_xdp_fd (ifindex, prog_fd, xdp_flags) < 0) {
    printf ("error setting fd onto xdp\n");
    return -1;
}

perf_map = bpf_object__find_map_by_name (obj, "perf_map");
perf_map_fd = bpf_map__fd (perf_map);

offload_map = bpf_object__find_map_by_name (obj, "offload_map");
offload_map_fd = bpf_map__fd (offload_map);

if (perf_map_fd < 0) {
    printf ("error cannot find map\n");
    return -1;
}

if (offload_map_fd < 0) {
    printf ("error cannot find map\n");
    return -1;
}

/* Initialize perf rings */
if (setup_perf_poller (perf_map_fd, sys_fds, n_cpus, &mem_buf[0]))
    return -1;

//Send map file descriptor
map_stats (offload_map_fd);
event_poller (mem_buf, sys_fds, n_cpus);

return 0;

```


ANEXO D – COLETOR (EBPF_PROG_KERN.C)

```

#include <uapi/linux/bpf.h>
#include <linux/if_ether.h>
#include <linux/if_packet.h>
#include <linux/if_vlan.h>
#include <linux/ip.h>
#include <linux/ipv6.h>
#include <linux/udp.h>
#include <linux/tcp.h>
#include <stdbool.h>
#include <stddef.h>
#include <linux/string.h>
#include "bpf_helpers.h"
#include "bpf_endian.h"
#include "xdpdump_common.h"
#include "ebpf_collector.h"
#include <arpa/inet.h>
#include <string.h>
#include <stdio.h>

//Some sizes
#define MAX_HOP 6

//-----

//Map eBPF
struct bpf_map_def
SEC ("maps")
offload_map = {
    .type = BPF_MAP_TYPE_HASH,
    .key_size = sizeof (struct quintuple_t),
    // .key_size = sizeof (__u32),
    .value_size = sizeof (struct map_t),
    .max_entries = 1024
};

//Map perf_event
struct bpf_map_def
SEC ("maps")
perf_map = {
    .type = BPF_MAP_TYPE_PERF_EVENT_ARRAY,
    .key_size = sizeof (__u32),
    .value_size = sizeof (__u32),
    .max_entries = MAX_CPU,
};

//-----

int
bpf_collector (struct xdp_md *ctx){
    void *data_end = (void *) (long) ctx->data_end;
    void *cursor = (void *) (long) ctx->data;

    //First Parser: Ethernet -> IP -> UDP -> Report

    //Ethernet
    struct ethernet_t *eth = cursor;
    __u32 offset = sizeof (*eth);
    cursor = cursor + offset;
    if (cursor > data_end) {
        return XDP_DROP;
    }

    //test 1
    if (bpf_htons (eth->type) != 0x0800)
        //return XDP_PASS;
        return XDP_DROP; //drop

    //IPv4
    struct iphdr *ip = cursor;
    cursor = cursor + sizeof (*ip);
    if (cursor + 1 > data_end) {
        return XDP_DROP;
    }
    offset = offset + sizeof (*ip);

```

```

//test 2
if (ip->protocol != IPPROTO_UDP)
    //return XDP_PASS;
    return XDP_DROP; //drop

//UDP
struct udphdr *udp = cursor;
cursor = cursor + sizeof (*udp);
if (cursor + 1 > data_end) {
    return XDP_DROP;
}
offset = offset + sizeof (*udp);

//test 3
if (bpf_htons (udp->dest) != 32766)
    //return XDP_PASS;
    return XDP_DROP; //drop

//Report
struct report_t *report = cursor;
cursor = cursor + sizeof (*report);
if (cursor + 1 > data_end) {
    return XDP_DROP;
}
offset = offset + sizeof (*report);

struct get_info_t int_metadados = {
    .flow_sink_timestamp = report->ingress_ts
};

//Second Parser: Ethernet -> IP -> UDP/TCP -> INT

//Inner Ethernet
struct ethernet_t *in_eth = cursor;
cursor = cursor + sizeof (*in_eth);
if (cursor + 1 > data_end) {
    return XDP_DROP;
}
offset = offset + sizeof (*in_eth);

//Inner IPv4
struct iphdr *in_ip = cursor;
cursor = cursor + sizeof (*in_ip);
if (cursor + 1 > data_end) {
    return XDP_DROP;
}
offset = offset + sizeof (*in_ip);

int_metadados.source_ip = in_ip->saddr;
int_metadados.destiny_ip = in_ip->daddr;
int_metadados.ip_protocol = in_ip->protocol;

struct ports_t port = { };

//Inner UDP
if (in_ip->protocol == IPPROTO_UDP) {
    struct udphdr *in_udp = cursor;
    cursor = cursor + sizeof (*in_udp);
    if (cursor + 1 > data_end) {
        return XDP_DROP;
    }
    offset = offset + sizeof (struct udphdr);
    port.source = in_udp->source;
    port.dest = in_udp->dest;
}

//Inner TCP
if (in_ip->protocol == IPPROTO_TCP) {
    struct tcphdr *in_tcp = cursor;
    cursor = cursor + sizeof (*in_tcp);
    if (cursor + 1 > data_end) {
        return XDP_DROP;
    }
    offset = offset + sizeof (struct tcphdr);
    port.source = in_tcp->source;
    port.dest = in_tcp->dest;
}

int_metadados.source_port = port.source;
int_metadados.destiny_port = port.dest;

```

```

// INT SHIM
// INT header

//Third Parse: INT metadata

__u8 num_INT_hop = 2;

int_metadados.INT_total_hops = num_INT_hop;
struct INT_data_16_t *INT_data_16;
struct INT_data_32_t *INT_data_32;
struct INT_data_64_t *INT_data_64;

/*
(SWITCH2)
||||'--int_switch_id
|||'---int_ingress_egress_ports
||'----int_hop_latency
|'-----int_ingressTimestamp
'-----int_egressTimestamp

(SWITCH1)
||||'--int_switch_id
|||'---int_ingress_egress_ports
|||'--int_switch_id
|||'---int_ingress_egress_ports
||'----int_hop_latency
|'-----int_ingressTimestamp
'-----int_egressTimestamp
*/

__u8 i = 0;
#pragma unroll
while (i < MAX_HOP) {
    //Switch ID
    INT_data_32 = cursor;
    cursor = cursor + sizeof (*INT_data_32);
    if (cursor + 1 > data_end) {
        return XDP_DROP;
    }
    offset = offset + sizeof (*INT_data_32);
    int_metadados.switch_id[i] = INT_data_32->data;

    //Ingress e Egress ID
    //ingress
    INT_data_16 = cursor;
    cursor = cursor + sizeof (*INT_data_16);
    if (cursor + 1 > data_end) {
        return XDP_DROP;
    }
    offset = offset + sizeof (*INT_data_16);
    int_metadados.ingress_port_id[i] = INT_data_16->data;
    //egress
    INT_data_16 = cursor;
    cursor = cursor + sizeof (*INT_data_16);
    if (cursor + 1 > data_end) {
        return XDP_DROP;
    }
    offset = offset + sizeof (*INT_data_16);
    int_metadados.egress_port_id[i] = INT_data_16->data;

    //Hop latency
    INT_data_64 = cursor;
    cursor = cursor + sizeof (*INT_data_64);
    if (cursor + 1 > data_end) {
        return XDP_DROP;
    }
    offset = offset + sizeof (*INT_data_64);
    int_metadados.hop_latency[i] = INT_data_64->data;
    int_metadados.flow_latency += int_metadados.hop_latency[i];

    /*
    //QUEUE ID
    //QUEUE OCCUPANCY
    */

    //Ingress Timestamp
    INT_data_64 = cursor;
    cursor = cursor + sizeof (*INT_data_64);
    if (cursor + 1 > data_end) {
        return XDP_DROP;
    }
}

```

```

    }
    offset = offset + sizeof (*INT_data_64);
    int_metadados.ingress_timestamp[i] = INT_data_64->data;

    //Egress Timestamp
    INT_data_64 = cursor;
    cursor = cursor + sizeof (*INT_data_64);
    if (cursor + 1 > data_end) {
        return XDP_DROP;
    }
    offset = offset + sizeof (*INT_data_64);
    int_metadados.egress_timestamp[i] = INT_data_64->data;

    //INT Counting Packets
    INT_data_32 = cursor;
    cursor = cursor + sizeof (*INT_data_32);
    if (cursor + 1 > data_end) {
        return XDP_DROP;
    }
    offset = offset + sizeof (*INT_data_32);
    int_metadados.cnt_packets[i] = INT_data_32->data;

    //INT Counting Bytes
    INT_data_32 = cursor;
    cursor = cursor + sizeof (*INT_data_32);
    if (cursor + 1 > data_end) {
        return XDP_DROP;
    }
    offset = offset + sizeof (*INT_data_32);
    int_metadados.cnt_bytes[i] = INT_data_32->data;

    i++;
    num_INT_hop--;
    if (num_INT_hop <= 0)
        break;
}

//INT_tail

//Defining key map
struct quintuple_t key_map = { };
key_map.source_ip = int_metadados.source_ip;
key_map.destiny_ip = int_metadados.destiny_ip;
key_map.source_port = int_metadados.source_port;
key_map.destiny_port = int_metadados.destiny_port;
key_map.ip_protocol = int_metadados.ip_protocol;

//Defining value map
struct map_t value_map = { };
value_map.flow_sink_timestamp = int_metadados.flow_sink_timestamp;
value_map.flow_latency = int_metadados.flow_latency;

struct map_t *previous_value_map = bpf_map_lookup_elem (&offload_map, &key_map);
if (!previous_value_map) {
    bpf_map_update_elem (&offload_map, &key_map, &value_map, BPF_ANY);
    int_metadados.new_flow = 1;
} else {
    int_metadados.flow_alive = 1;
}

bpf_perf_event_output (ctx, &perf_map, BPF_F_CURRENT_CPU, &int_metadados,
    sizeof (int_metadados));

return XDP_DROP;
}

char _license[] SEC ("license") = "Dual BSD/GPL";

```




UPF

UNIVERSIDADE
DE PASSO FUNDO

UPF Campus I - BR 285, São José
Passo Fundo - RS - CEP: 99052-900
(54) 3316 7000 - www.upf.br