UNIVERSIDADE DE PASSO FUNDO

Programa de Pós-Graduação em
Computação Aplicada

Dissertação de Mestrado

INT FLOW: APLICAÇÃO DE TELEMETRIA EM REDES DEFINIDAS POR SOFTWARE

PEDRO EDUARDO CAMERA

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

INT FLOW: APLICAÇÃO DE TELEMETRIA EM REDES DEFINIDAS POR SOFTWARE

Pedro Eduardo Camera

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

C182i Camera, Pedro Eduardo
Int flow : aplicação de telemetria em redes definidas por software / Pedro Eduardo Camera. – 2020.
94 f. ; 30 cm.

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

1. Software. 2. Telemetria. 3. Framework (Arquivo de computador). 4. Arquitetura de rede de computador. I. Trentin, Marco Antônio Sandini, orientador. II. Schmidt, Ricardo de Oliveira, coorientador. III. Título.

CDU: 004.72

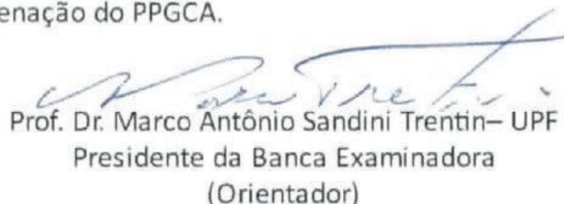
Catalogação: Bibliotecária Jucelei Rodrigues Domingues - CRB 10/1569

UPF | **PPGCA**
Programa de Pós-Graduação
em Computação Aplicada
Instituto de Ciências Exatas e Geociências | ICEG

ATA DE DEFESA DO TRABALHO DE CONCLUSÃO DE CURSO DO ACADÊMICO

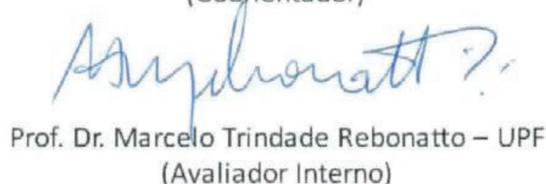
PEDRO EDUARDO CAMERA

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 "INT Flow: Aplicação de Telemetria em Redes Definidas por Software", de autoria de Pedro Eduardo Camera, 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 Lizandro 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

Meus agradecimentos vão primeiramente à minha família, por todo o apoio e incentivo para a realização desse mestrado.

Agradeço ao meu orientador Prof. Dr. Marco Antônio Sandini Trentin e também agradeço ao coorientador Prof. Dr. Ricardo de Oliveira Schmidt, por todos os conselhos e ensinamentos.

Também aos demais professores do PPGCA da UPF pelo conhecimento recebido.

Por fim, agradeço ao Me. Jeronimo Bezerra, pela oportunidade de trabalhar em um ambiente real de rede e também pelos conhecimentos recebidos.

INT FLOW: APLICAÇÃO DE TELEMETRIA EM REDES DEFINIDAS POR SOFTWARE

RESUMO

Desde do ano 2008, os conceitos de *Software Defined Networking* tem ganhado muita atenção entre engenheiros e pesquisadores na área de redes de computadores, principalmente devido à grande flexibilidade que os conceitos trazem para o gerenciamento de redes. Essa dissertação introduz uma nova metodologia de medições *in-band network telemetry*, um *framework* que coleta as informações dos *status* da rede. Para a programação dos dispositivos de rede, foi utilizado a linguagem *Programming Protocol-Independent Packet Processors*, uma linguagem utilizada para definir o comportamento do dispositivo de rede e implementar o método *in-band network telemetry*. A maioria dos aplicativos *in-band network telemetry* é restrita a ambientes de *data centers* ou simulados. O principal objetivo deste trabalho foi propor uma metodologia de monitoramento utilizando técnicas em *hardware* para monitorar uma rede *Metropolitan Area Network*. Esta pesquisa aplicou algumas métricas do *in-band network telemetry*, como registro de data e hora, caminho do fluxo e latência dos dispositivos. Os resultados obtidos demonstram que os pacotes enviados ao coletor de dados estão formatados de acordo com a solução programada no plano de dados, bem como que o modelo programado permite a captura de dados em um fluxo da rede. Em conclusão, o *in-band network telemetry flow*, a solução final apresentada, realizou o monitoramento em menor granularidade de tempo (milissegundos) do que outros protocolos como *simple network management protocol* (segundos), atualmente utilizados em empresas de telecomunicações.

Palavras-Chave: *Framework* INT, *metropolitan area network* e redes definidas por *software*.

INT FLOW: TELEMETRY APPLICATION IN SOFTWARE DEFINED NETWORKING

ABSTRACT

Since 2008, Software Defined Networking concepts have gained a lot of attention among computer network engineers and researchers, mainly due to the big flexibility that the concepts bring to network management. This dissertation introduces a new methodology of in-band network telemetry measurements, a framework that collects network status information. For programming the network devices, were used the language Programming Protocol-independent Packet Processors. It's a language used to define network device behavior and was used to implement the in-band network telemetry method. Most of the in-band network telemetry applications are restrained to data centers environments or simulated ones. The main objective of this work was to use these techniques in hardware to monitor the Metropolitan Area Network. This research applied some in-band network telemetry metrics, like timestamp, flow path, and hop latency. The results obtained demonstrate that the packets sent to the data collector are formatted according to the solution programmed in the data plane, another achievement was that the programmed model allows the capture of data in a network flow. In conclusion, in-band network telemetry flow, the presented final solution, performed monitoring in the lesser granularity of time (milliseconds) than other protocols like simple network management protocol (seconds), currently used in telecommunications companies.

Keywords: Framework INT, metropolitan area network and Software Defined Networking.

LISTA DE FIGURAS

Figura 1. Tráfego trocado entre diferentes ASs no Brasil na última década. . . 20
Figura 2. Representação de uma topologia onde o problema de pesquisa está inserido. 21
Figura 3. Comparação de SDN com *switch* tradicional. 23
Figura 4. Representação de uma topologia usando rede definida por software. 24
Figura 5. Comparação de um *switch* atual com um *switch* com SDN. 25
Figura 6. Representação da arquitetura do *openflow*. 26
Figura 7. Representação da arquitetura do controlador. 28
Figura 8. Interface web ONOS 29
Figura 9. Comparação do *openflow* e P4. 30
Figura 10. Exemplo de um *hearder* em P4 31
Figura 11. Etapa da máquina de estados do *parsers* 32
Figura 12. Exemplo de código P4. 34
Figura 13. Estrutura do código P4. 34
Figura 14. Arquitetura NFP com V1Model. 35
Figura 15. Diagrama do Compilador P4 36
Figura 16. Funcionamento do INT. 39
Figura 17. Placa smartNIC Netronome Agilio CX 40Gbps 44
Figura 18. Funcionamento da SmartNIC com P4 ativado. 45
Figura 19. Representação gráfica do analisador 46
Figura 20. Código e função *sandbox* C 46
Figura 21. Topologia de comunicação entre o PS com o RTE. 47
Figura 22. Demostração dos cinco servidores. 47
Figura 23. Switch 1 e switch 2 48
Figura 24. Topologia *switch* 1 e *switch* 2 49
Figura 25. Topologia completa 49
Figura 26. Ordem de execução das funções do código P4 51
Figura 27. Função *parser* do código IP/TCP e IP/UDP 52
Figura 28. Função *ingress* do código IP/TCP e IP/UDP 53
Figura 29. Plano de controle do *switch* 1 54
Figura 30. Plano de controle do switch 2 54

Figura 31. *Headers* para o INT 55
Figura 32. *Parser* com INT 57
Figura 33. *Ingress* com INT do *switch* 1 58
Figura 34. *Ingress* com INT do *switch* 2 59
Figura 35. Organização do pacote *transit* INT 59
Figura 36. Organização do pacote de *report* INT para o coletor 60
Figura 37. *Egress* com INT no *switch* 1 ou *switch* 2 61
Figura 38. Plano de controle com INT 62
Figura 39. Comando para modificação de *TimeStamp* 63
Figura 40. Cabeçalhos para armazenar a quantidade de *bytes* e de pacotes . 64
Figura 41. Interfaces que os dados de quantidade de *bytes* e pacote são coletados. 64
Figura 42. Representação da estrutura do projeto. 67
Figura 43. Estrutura em hexadecimal do *report* INT com o *template* 248. 68
Figura 44. Estrutura em hexadecimal do *transit* INT com o *template* 248. 69
Figura 45. Taxa de transferência do *transit* INT com o *template* 248 69
Figura 46. Taxa de transferência do *report* INT com o *template* 248. 69
Figura 47. *Timestamp* 64 bits 71
Figura 48. Informações do pacote INT. 72
Figura 49. Representação das informações do pacote INT no *switch*. 72
Figura 50. Cabeçalhos presente nos pacotes. 75
Figura 51. *Template* 128 e 192. 75
Figura 52. *Template* 224 e 240. 75
Figura 53. *Template* 248 e 160. 76
Figura 54. *Template* 184 e 88. 76
Figura 55. Hexadecimal com os dados de telemetria extra. 77

LISTA DE TABELAS

Tabela 1. Campos das versões do *openflow* 27

Tabela 2. Tabela *hardware* dos servidores 50

Tabela 3. *Templates* para coleta de dados 60

Tabela 4. Intervalo de tempo entre os pacotes INT. 70

Tabela 5. Tamanho dos pacotes *report* INT com o *template* 248 73

Tabela 6. Tamanho dos pacotes *transit* INT com o *template* 248 73

Tabela 7. *Templates* e tamanhos 74

LISTA DE SIGLAS

API – *Application Programming Interface*

ARM – *Advanced RISC Machine*

AS – *Autonomous System*

ASIC – *Application Specific Integrated Circuits*

DDR – *Double Data Rate*

GBPS – *Gigabit por Segundo*

ICMP – *Internet Control Message Protocol*

IDE – *Integrated Development Environment*

INT – *In-band Network Telemetry*

IOT – *Internet of Things*

IX – *Internet Exchange*

LPM – *Longest prefix match*

MAC – *Media Access Control Address*

MAN – *metropolitan area network*

MS – *Milissegundo*

MTU – *maximum transmission unit*

NFP – *Network Flow Processor*

NIC – *Network Interface Card*

NS – *Nanossegundo*

ONOS – *Open Network Operating System*

P4 – *Programming Protocol-Independent Packet Processors*

P4C – *P4 Compiler*

PING – *Packet Internet Network Grouper*

PS – *Programmer Studio*

PTP – *Precision Time Protocol*

QOS – *Quality Of Service*

RTE – *Run Time Environment*

SDK – *Software Development Kit*

SDN – *Software Defined Networking*

SNMP – *Simple Network Management Protocol*

TBPS – *Terabit por Segundo*

VXLAN – *Virtual Extensible Local Area Network*

SUMÁRIO

1	**INTRODUÇÃO**	**19**
2	**FUNDAMENTAÇÃO TEÓRICA**	**23**
2.1	REDES DEFINIDAS POR SOFTWARE	23
2.2	API *OPENFLOW*	26
2.3	CONTROLADORES	27
2.4	LINGUAGEM DE PROGRAMAÇÃO P4	30
2.5	ARQUITETURA NFP	34
2.6	P4C - COMPILADOR DA LINGUAGEM P4	36
2.7	MONITORAMENTO DE REDE	36
2.8	MÉTODO INT	37
2.9	MÉTRICAS DISPONÍVEIS	40
2.10	TRABALHOS RELACIONADOS	41
3	**PROCEDIMENTOS METODOLÓGICOS**	**43**
3.1	SMARTNIC NETRONOME	43
3.2	COMPILADOR AGILIO P4C SDK	45
3.3	AMBIENTE DE SIMULAÇÃO	47
3.3.1	**Ambiente Computacional**	50
3.4	ALGORITMO DE ENCAMINHAMENTO DE PACOTES IP/TCP E IP/UDP	50
3.4.1	**Plano de controle do processo TCP e UDP**	53
3.5	IMPLEMENTAÇÃO DO INT NO *PARSER SWITCH* 1 E *SWITCH* 2	55
3.6	IMPLEMENTAÇÃO DO INT NO *INGRESS SWITCH* 1 E *SWITCH* 2	57
3.6.1	***Templates* de coleta de valores INT**	60
3.7	IMPLEMENTAÇÃO DO INT NO *EGRESS SWITCH* 1 E *SWITCH* 2	60
3.7.1	**Plano de controle para o *Ingress* e *Egress* com INT**	61
3.8	IMPLEMENTAÇÃO DO *TIMESTAMP*	62
3.9	IMPLEMENTAÇÃO EXTRA DA CAPTURA DA QUANTIDADES DE *BYTES* E PACOTES	63
4	**APRESENTAÇÃO E DISCUSSÃO DOS RESULTADOS**	**67**
4.1	CABEÇALHOS NO PACOTE INT E RECURSOS	67

4.1.1 **Intervalo entre os pacotes INT** 70
4.2 INFORMAÇÕES DO PACOTE INT E AVALIAÇÃO *TIMESTAMP* 70
4.3 AUMENTO DO TAMANHO DO PACOTE COM INT (MTU) 72
4.4 ESTRUTURA DO PACOTE COM *TEMPLATE* 74
4.5 ANÁLISE DO PACOTE COM OS DADOS DE TELEMETRIA EXTRA 76

5 CONSIDERAÇÕES FINAIS **79**
5.1 RESSALVAS E LIMITAÇÕES 80
5.2 TRABALHOS FUTUROS 80

REFERÊNCIAS **83**

ANEXO A – Código *ingress switch* 1 . . . . . **89**

ANEXO B – Código *ingress switch* 2 . . . . . **91**

ANEXO C – Código *egress switch* 1 e *switch* 2 . . . . . **93**

1. INTRODUÇÃO

Provedores e operadoras de Internet estão em um processo elevado de crescimento de demanda de tráfego, processamento, uso de memória dos equipamentos e quantidade de ativos nas redes. Isso gera complexidade no gerenciamento das redes pertencentes ao mesmo domínio ou *autonomous system*[1].

O vigor do crescimento da Internet das coisas (IoT) e os avanços na área de computação em nuvem estão gerando um crescimento nos serviços *online*. Segundo Help Net Security [1], no ano de 2018, a quantidade de equipamentos conectados na Internet chegou a 22 bilhões e existe uma previsão de 38,6 bilhões até 2025. Todos esses dispositivos estarão compartilhando conteúdo e gerando tráfego na rede mundial de computadores.

Consequentemente, os provedores de acesso estão investindo em equipamentos que suportam maior capacidade de transmissão de dados. Com uma carga de pacotes muito alta passando nos dispositivos, os administradores de rede enfrentam muitos desafios para a supervisão da qualidade do tráfego que transita nos ativos de rede. Quanto mais os *switches*[2] ou roteadores processam altas quantidades de dados, mais é necessário um monitoramento minucioso dos *status* da qualidade destes fluxos de dados[3].

Com a finalidade de ilustrar o cenário atual da Internet, a Figura 1 apresenta o consumo do ponto de troca de tráfego[4] do Brasil denotado pela sigla PTT.br. Para se ter uma ideia do aumento do tráfego na Internet no Brasil, a Figura 2 apresenta a evolução do tráfego que circulou entre diferentes ASs no país, sendo que esse tráfego é a soma de consumo de banda de todos os PTTs instalados no território brasileiro, em um período de uma década.

Ainda, segundo a Figura 1, percebe-se que no ano de 2013, o tráfego era inferior a 25% de 1 Tbps. Atualmente, no ano de 2019, tem-se cerca de 4,37 Tbps. Ou seja, em sete anos ocorreu um aumento de cerca de 249,6% do tráfego. Cabe salientar que esses são dados de um ponto de troca de tráfego entre provedores de conteúdo e provedores de acesso no Brasil. Enfim, percebe-se um aumento significativo da utilização de meios de comunicação digital através da Internet [2].

[1] *Autonomous System* (AS) em português sistema autônomo, são as redes de um mesmo domínio administrativo, que são identificadas com um número (*Autonomous System Number*). O AS rede também possui os IPs públicos que foram destinados para essa rede e também representa uma estrutura de rede

[2] *Switches* é um equipamento que utiliza o endereço físico que está presente dentro do pacote, para determinar qual interface destinará o pacote. Em *switches* L3 utiliza outras informações para comutar o pacote para uma determinada interface.

[3] Fluxo de dados é uma sequência de pacotes com as mesmas características e com mesmo destino lógico.

[4] Ponto de Troca de Tráfego (PTT) denominada também como ponto de troca de tráfego, é um local que possui uma infraestrutura para compartilhar conteúdo e tráfego de Internet entre diferentes ASs, sem fins lucrativos, atualmente é administrado pelo NIC.br.

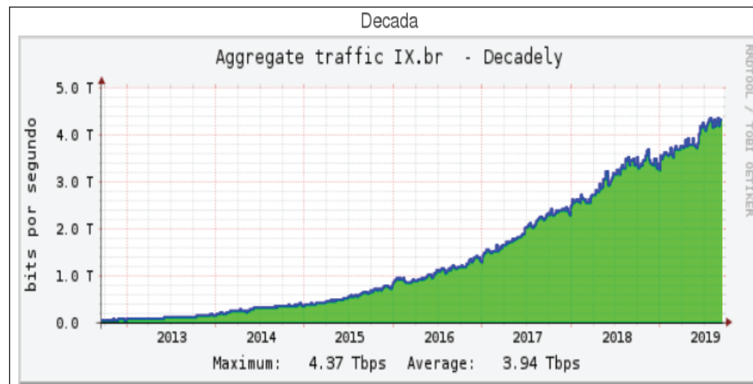


Figura 1. Tráfego trocado entre diferentes ASs no Brasil na última década. Fonte IX.br [2].

Atualmente, os sistemas monitoram as informações relevantes sobre as condições dos dispositivos de uma rede. Existem poucos *softwares* que supervisionam a qualidade do fluxo de dados que percorrem todos esses dispositivos da origem até o destino dentro da rede. A execução de um *troubleshooting*[5] do *hardware* e do fluxo de dados é necessário para avaliar a qualidade da rede. Assim, o administrador de rede tem informações do funcionamento dos *hardwares* na rede e também sobre o comportamento dos fluxos de dados.

Outro ponto que deve ser observado é que atualmente são utilizados protocolos específicos para realizar a supervisão dos dispositivos e que esses protocolos estão implementados em algoritmos fechados pelos fabricantes. Esse monitoramento coleta a informação de *status* em um intervalo de tempo predefinido no *firmware*[6] do equipamento de rede. Nesse caso, esse padrão de tempo não consegue capturar os dados em um intervalo de tempo inferior ao programado dentro do dispositivo a ser supervisionado. Desta maneira, algumas informações podem não estar sendo apuradas pelo *software* de monitoramento [3].

Na Figura 2 está sendo exposto um problema que pode passar despercebido por um administrador de rede. Nela pode ser observada a infraestrutura de um típico provedor de acesso, contendo os equipamentos desdo usuário final até o roteador de borda que se interliga com outros provedores. Nesse caminho existem vários *switches* interligados, que transmitem as informações do roteador de borda até o usuário final. Essa estrutura que possui os *switches*, normalmente é chamado de rede *metropolitan area network* (MAN), pois é a interligação de cidades ou localidades diferentes.

Em um certo momento o administrador da rede notou que a latência entre *switch* A e o *switch* D chegou a 300 ms. Nesse caso, o administrador deve verificar se o motivo desse aumento de latência está dentro da rede MAN, para após realizar a manutenção eficaz do problema nos *switches* ou no trajeto físico.

[5] *Troubleshooting* é uma abordagem sistemática à correção de problemas, utilizado para encontrar problemas em redes de computadores, *softwares* e aparelhos eletrônicos.
[6] *Firmware* são *softwares* programados diretamente no equipamento eletrônico.

Como se percebe na Figura 2, existem quatro *switches* que podem ser o motivo do aumento da latência. Também existe a possibilidade do gerente da rede não saber exatamente em quais portas e equipamentos, esse pacote passou para chegar ao destino final, pois durante o trajeto da origem até o destino existem vários caminhos e equipamentos por onde o pacote pode trafegar.

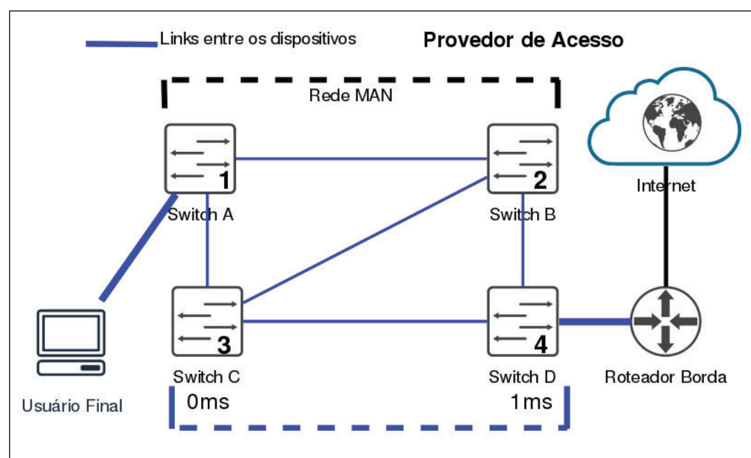


Figura 2. Representação de uma topologia onde o problema de pesquisa está inserido. Fonte Autor.

Se o gerente souber exatamente por quais *switches* e portas o pacote trafegou e também a latência em cada um dos *switches*, então será possível identificar onde ocorreu o problema.

Diante do exemplo acima, percebe-se que as tecnologias de monitoramento atuais não atendem mais a contento algumas situações impostas pelos novos serviços ou altas demandas presentes nas redes. Isso levou a comunidade científica a propor alternativas para esses sistemas de monitoramento. Uma das alternativas é a rede definida por *softwares*, pois ela permite o gerenciamento completo de um fluxo de dados, possibilitando administrar de apenas um ponto central e viabilizando programar a maneira que o tráfego irá circular pelos ativos de rede [4].

Nota-se no problema descrito acima que, para o administrador de rede, o monitoramento dos fluxos de dados da rede tem uma grande relevância para tomar decisões adequadas no comportamento do tráfego e ajuda a solucionar problemas de forma eficiente. Para isso, o gerente precisa de ferramentas que buscam informações sobre o comportamento do fluxo, dentro de uma rede com alto volume de tráfego, como uma rede MAN. Diante disto, o cenário apresentado até aqui levou ao seguinte problema de pesquisa: como é possível identificar, de forma pontual, rápida e eficaz, alterações de latência e o exato local do problema, em uma rede definida por *software* sobre uma rede MAN?

Desta forma, o presente trabalho tem como objetivo geral desenvolver uma solução que possa auxiliar na telemetria personalizada através de uma linguagem de programação para aplicação em redes, visando realizar a telemetria do fluxo no encaminhamento dos

pacotes. Essa telemetria estará utilizando os próprios pacotes, para rastrear o caminho percorrido pelos pacotes na rede. Em direção a esse objetivo, devem ser executados os objetivos específicos listados abaixo:

- Estudo da linguagem P4 para programação dos dispositivos de rede;
- Estudo de um método INT para monitoramento dentro da estrutura dos pacotes;
- Montagem de uma estrutura para a simulação da rede, configuração das máquinas com o compilador P4 e os *softwares* necessários;
- Realizar o desenvolvimento do método de telemetria dentro dos pacotes que transitam na rede;
- Realizar a simulação de rede e atestar o funcionamento do método e analisar os resultados coletados.

Cabe destacar que essa dissertação faz parte de um projeto maior, intitulado AmLight-INT, que tem como objetivo realizar uma supervisão dos fluxos de dados da rede SDN AmLight-ExP. A rede AmLight-ExP, por sua vez, é responsável por realizar a transmissão dos dados de um telescópio instalado no Chile, denominado *Large Survey Synoptic Telescop* (LSST), até os Estados Unidos, passando por diversos *switches*. A parte que compete a essa dissertação é o desenvolvimento de uma aplicação inicial para a coleta de dados através de um *framework* INT a fim de auxiliar na supervisão da qualidade dos fluxos de dados dessa rede.

Esta dissertação encontra-se assim organizada: No Capítulo 2, é apresentada uma revisão de literatura sobre os conceitos de redes definidas por *software* (Seção 2.1), a Seção 2.5 apresenta uma linguagem que busca uma programação do *chip* do dispositivo, e a Seção 2.8 mostra o conceito de *framework* de telemetria. Na Seção 3 será implementado em ambiente real de rede, buscando avaliar o funcionamento de uma implementação, utilizando conceito de redes definidas por *software* com uma linguagem de programação para redes.

2. FUNDAMENTAÇÃO TEÓRICA

Neste capítulo serão descritos os principais tópicos que envolvem as redes definidas por *software* presentes na literatura. Também será abordado o contexto de redes definidas por *software* com uma contextualização sobre *OpenFlow* e detalhes da linguagem P4.

2.1 REDES DEFINIDAS POR SOFTWARE

Rede definida por *software* (SDN) é um paradigma que apresenta o desacoplamento de um dispositivo de rede em plano de dados e plano de controle. Atualmente os dois planos são implementados dentro do mesmo dispositivo. Em SDN, é possível que a infraestrutura de rede seja gerenciada pelo plano de controle centralizado, onde o plano de controle apresenta uma abstração da infraestrutura física da rede [4].

Conforme Kreutz et al [5], SDN é um paradigma que realiza uma separação vertical, a qual realiza a divisão do dispositivo de rede em plano de controle (*control plane*) e plano de dados (*data plane*). Basicamente, é a separação do lógico (*software*), do físico (*hardware*) em dois planos com visões distintas, conforme ilustrado na Figura 3. Esse paradigma está em amplo crescimento, pois apresenta uma solução para as limitações das redes, como, por exemplo: a homogeneização dos dispositivos de rede, *software* fechado pelo fabricante, tomada de decisão de regras descentralizadas, protocolos implementados fixos nos circuitos dos dispositivos e topologias unificadas [6].

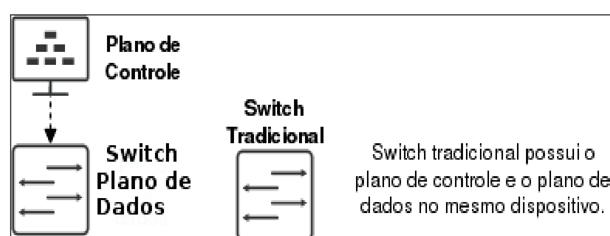


Figura 3. Comparação de SDN com *switch* tradicional. Fonte: autor.

A arquitetura teórica de SDN expõe algumas características que são desejáveis, como, por exemplo, o equipamento físico deve ser simples de configurar, com um baixo custo e deve operar com qualquer modelo ou marca de dispositivo de rede. Esse equipamento deve suportar os avanços da tecnologia dessa área, evitando atualizações constantes dos equipamentos. Os *softwares* devem ser dinâmicos e suportar uma estrutura com uma grande quantidade de funcionalidades (engenharia de tráfego, controle de acesso, tomadas de decisões automáticas, regras de QoS[7], entre outras). Deve ser modular, cada

[7] *Quality of Service* é a priorização de tráfego de acordo com o serviço.

um desses módulos têm configurações diferentes e deve permitir realizar a adição, edição e remoção dos módulos, conforme as necessidades do administrador. Os módulos contêm uma programação de serviços e tarefas separadas, que o *hardware* irá executar [7].

A Figura 4 ilustra uma topologia utilizando SDN, em que os *switches* são os dispositivos que estão operando como plano de dados. O plano de controle manipula e controla os *switches*. Ilustra também o funcionamento de uma rede onde o gerenciamento é centralizado. Neste caso, não será necessário realizar uma configuração dentro de cada dispositivo de rede. Basta realizar uma programação no plano de controle que o mesmo repassará para todos os dispositivos conectados nesse plano de controle [8].

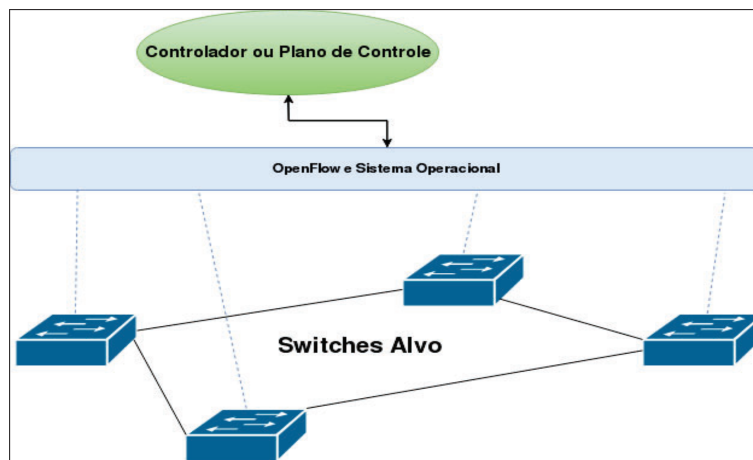


Figura 4. Representação de uma topologia usando SDN. Fonte: Bhaumik et al [8].

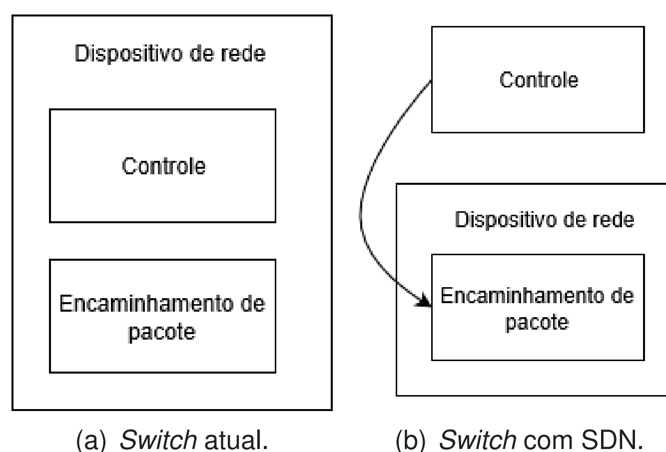
Uma parte da arquitetura do SDN é o plano de controle, que tem a função de controlar o encaminhamento de pacotes (plano de dados) dos dispositivos como, por exemplo, analisar e encaminhar o pacote para uma determinada porta de um *switch*, conforme as regras implementadas no plano de controle. O plano de controle pode executar operações de gerenciamento da rede. Esse plano possui algoritmos para a programação de tarefas de roteamento, *firewall*, entre outros, ou seja, possibilita a execução de algoritmos implementados pelo próprio administrador. Esses algoritmos executam diferentes funções para o controle dos equipamentos que são subordinados ao mesmo. O plano de controle pode executar os algoritmos implementados em um ou mais dispositivos que estão interagindo com as regras (*rules*) desse controlador.

A outra parte da arquitetura do SDN é o plano de dados, que tem a função de realizar o encaminhamento dos pacotes de maneira rápida, eficaz e sem perdas, seguindo as instruções encaminhadas pelo plano de controle. Também tem a função de encaminhar informações sobre os cabeçalhos dos pacotes e estatísticas para o plano de controle. No plano de dados, os pacotes passam pela *pipeline*[8] dos dispositivos, são analisados e depois encaminhados conforme a tabela de regras definida pelo plano de controle. O funci-

[8] *Pipeline* é segmentação de regras, onde uma instrução de processamento está dividida em etapas, e o pacote passa por cada etapa para passar pelo equipamento.

onamento da *pipeline* será detalhada na Seção 2.5. A passagem dos pacotes pelo plano de controle deve ser feita de maneira transparente e de maneira que o processamento destes pacotes não introduzam retardo significativo em seu trânsito. A arquitetura da *pipeline* está presente dentro dos dispositivos e é definida pelo fabricante.

As redes de computadores estão demonstrando algumas limitações como, por exemplo, a limitação de monitoramento completo do tráfego. As configurações, em muitos casos, não são centralizadas e possuem *hardwares* com consumo muito elevado de recursos, por ter protocolos desnecessários implementados pelos fabricantes. Atualmente os dispositivos de rede possuem a gerência e o encaminhamento dos pacotes dentro do *firmware* dos equipamentos, conforme ilustrado na Figura 5(a), forçando a utilização dos protocolos definidos pelos fabricantes dos produtos.



(a) *Switch* atual. (b) *Switch* com SDN.

Figura 5. Comparação de um *switch* atual com um *switch* com SDN.

Com o paradigma SDN nota-se alguns benefícios dessa tecnologia em crescimento. O primeiro é a capacidade de programabilidade do plano de dados, onde é possível programar o algoritmo de gerenciamento dos pacotes da maneira desejada pelo administrador da rede, além da possibilidade de criar funções nos dispositivos. O usuário pode ter exclusividade nos protocolos implementados, pois o programador pode programar protocolos exclusivos para sua rede. Com SDN os dispositivos tendem a ser mais eficientes, pois o administrador programa apenas as regras que são necessárias para o funcionamento da rede e consequentemente o *switch* vai consumir apenas os recursos para os protocolos necessários.

Outra vantagem desse paradigma é a confiabilidade, pois as funções dos dispositivos de rede não são mais implementadas por terceiros. Possui outra vantagem que é a possibilidade de criação de métodos de telemetria usando os campos dos cabeçalhos desses pacotes que trafegam na rede SDN. Portanto, o *software* de monitoramento pode fazer uso dessas informações que foram adicionadas aos pacotes que percorreram a rede.

2.2 API *OPENFLOW*

O *openflow* é uma API de comunicação entre controlador e os dispositivos de rede. As funções e serviços de decisão de alto nível são realizados pelo controlador. A troca de informações entre os dispositivos e o controlador ocorre através de pacotes, usando criptografia SSL (*Secure Socket Layer*). Esses pacotes são encaminhados até o *openflow* que está integrado aos dispositivos, conforme apresentado na Figura 6 [9].

Nota-se que na Figura 6 o controlador se comunica através do *openflow* em um canal seguro com o *switch*. Através dessa comunicação, o controlador atualiza as regras que estão presentes na tabela de fluxo de acordo com as informações encaminhadas pelo controlador. A tabela de fluxo controla os pacotes que trafegam por esse *switch*, onde são executadas as ações nos pacotes [10].

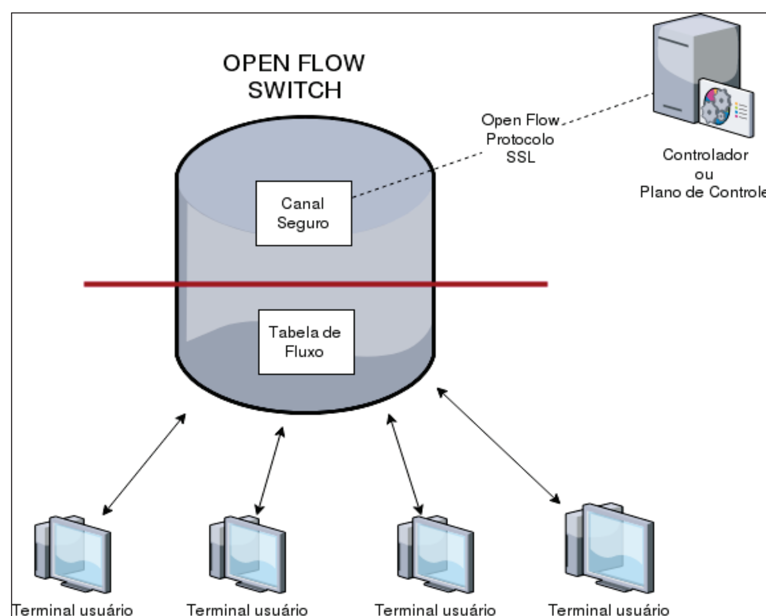


Figura 6. Representação da arquitetura do *openflow*. Fonte: Fernandez [10].

A implementação mais importante de SDN, atualmente, é a API[9] denominada *openflow* que é a comunicação entre o plano de controle e o plano de dados. O plano de controle pode definir o comportamento do dispositivo, pode fazer o dispositivo executar tarefa de *switch*, *firewall*, roteador entre outras aplicações de rede. Possibilita funcionar com o plano de controle remoto, pois é um servidor separado do dispositivo que tem o plano de dados sendo controlado.

Antes da adoção do SDN pelos fabricantes, havia no mercado muitos equipamentos de rede que eram conceituados de caixas pretas, por possuir o código totalmente fechado. A configuração desses equipamentos é realizada sobre um *framework* web ou termi-

[9] *Application Programming Interface*(API) são padrões ou regras de programação para realizar a comunicação entre dois *softwares* diferentes.

nal, ou um NMS[10] desenvolvido com código fechado pelo próprio fabricante. Atualmente os equipamentos estão vindos da fábrica com a API do *openflow* incorporada nos dispositivos, possibilitando utilizar controladores para gerenciar a rede.

O *openflow* consegue analisar as informações presentes nos campos dos cabeçalhos dos pacotes. Com o passar do tempo o protocolo *openflow* está aprimorando e ampliando sua capacidade em identificar os campos dos cabeçalhos. A quantidade de campos dos cabeçalhos passaram de 12 em 2009, para 41 campos em 2013, que foi a última implementação do *openflow*, conforme apresentado na Tabela 1, porém, essa quantidade de campos ainda não oferece flexibilidade suficiente para adicionar novos cabeçalhos e definir novas ações [12]. Os campos são dados armazenados dentro dos pacotes e são separados por um número predefinido de *bits*, onde armazenam informações, como o IP de origem, IP de destino, entre outras informações, para que o pacote transite pela rede e seja entregue para o destino.

Tabela 1. Campos das versões do *openflow*[12].

Versão	Data	Nº Campos do Cabeçalho
openflow 1.0	Dez 2009	12
openflow 1.1	Fev 2011	15
openflow 1.2	Dez 2011	36
openflow 1.3	Jun 2012	40
openflow 1.4	Out 2013	41

A API *openflow* ainda possui algumas limitações que a impede de ser uma API capaz de programar totalmente um plano de dados. Atualmente os *switches* e roteadores vêm com suporte ao *openflow*, porém, está habilitado no sistema operacional fechado pelo fabricante, no qual as regras internas do *openflow* não podem ser alteradas, deixando o administrador dependente da implementação do *software* do fabricante, onde não representa totalmente a proposta da arquitetura do SDN [13].

2.3 CONTROLADORES

O plano de controle possui um *software* chamado de controlador SDN sendo que, segundo Limoncelli [14], o controle executa ações e gerencia as tabelas de fluxos dos *switches* subordinados ao seu controle.

Esses controladores SDN ocultam o cenário da rede física ou virtual das camadas mais inferiores da estrutura de um *software* de virtualização, criando uma visão genérica da topologia da rede. Os controladores podem executar muitos serviços nos dispositivos como, por exemplo, exercer função de *switch*, roteador, entre outros *hosts* tradicionais de

[10] *Network Management System (NMS)* é um sistema de gerenciamento centralizado baseado na Web ou *software* de *desktop* usado para gerenciar e monitorar a condição atual de toda a infraestrutura de rede existente em um ambiente. [11]

uma rede. Ainda, pode viabilizar a execução de novos serviços, como atuar como serviço de conectividade fim-a-fim (*end-of-end*) passando por muitos domínios diferentes [15].

Os controladores possuem duas interfaces muito importantes para seu funcionamento. Uma interface é a *southbound*, a qual tem a função de realizar a comunicação entre controlador e as APIs de comunicação entre os *switches*. Também tem uma interface *northbound*, que tem a função de realizar a comunicação entre os aplicativos externos e o controlador, por exemplo, um aplicativo que busca informações da rede para visualização do usuário [16]. A Figura 7 representa as duas interfaces no controlador.

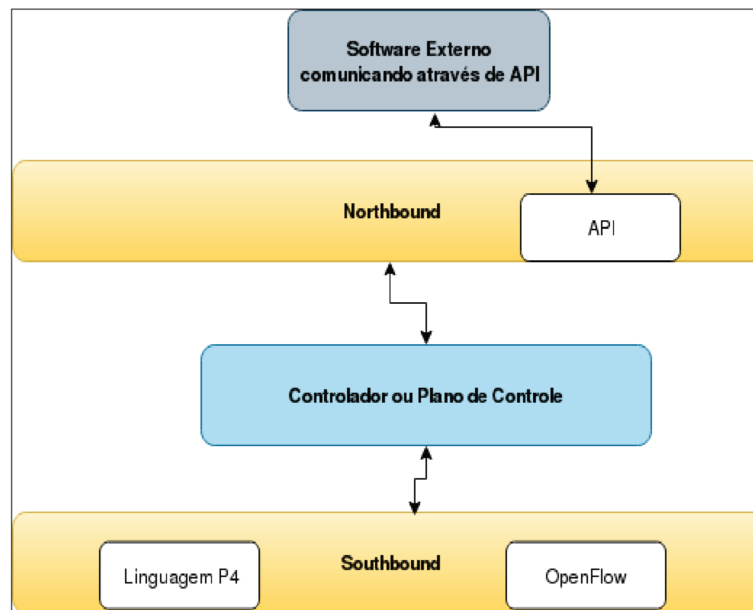


Figura 7. Representação da arquitetura do controlador. Fonte: Baswas et al. [17].

Segundo Stancu et al. [18] os controladores possuem várias funcionalidades. Através do *software* do plano de controle é possível monitorar a rede, verificar os *status* da utilização dos *links* e analisar o comportamento de toda a rede a partir dessas informações. O *software* também pode construir a topologia da rede baseada nos dados coletados dos *switches*. Também tem a função de definir ações para o plano de dados e gerenciar as regras que são encaminhadas para o *openflow* que executa nos dispositivos. Alguns dos *softwares* desenvolvidos para executar na camada do plano de controle são o NOX, POX e ONOS.

O autor Costa [19] aborda que o controlador de *openflow* denominado NOX tem o propósito de fornecer um desenvolvimento de aplicações de alto nível para um gerenciamento dos fluxos de dados. O NOX foi desenvolvido em linguagem C++ para o controle da API dos *switches* ou outros dispositivos de rede. Baseado no NOX surgiu o POX, que é um *software* de código aberto, com seu desenvolvimento em *Python*, porém, o POX é recomendado para fins didáticos ou de experiências, pois o POX tem um desempenho inferior ao NOX [20].

O controlador *open network operating system* (ONOS) é um sistema operacional SDN preparado para operar redes oferecendo alta disponibilidade e desempenho. A arquitetura do sistema é distribuída e funciona como um *cluster*, para ser um *software* escalável, mas preserva o conceito de SDN de ser um sistema com o plano de controle centralizado [16].

No ONOS as interfaces *southbound* permitem a interação com o *openflow* e com outras ferramentas como, por exemplo, a linguagem P4, que são executados nos dispositivos da rede. [21].

O ONOS possui uma interface que oferece uma visão geral da rede e dos dispositivos que o controlador está operando, facilitando a criação de aplicativos e serviços de rede. Possui uma interface *Web* que facilita a manutenção e visualização da rede, conforme apresentado na Figura 8. Este controlador possui um alto desempenho no *throughput*[11] dos equipamentos subordinados ao ONOS. [22]

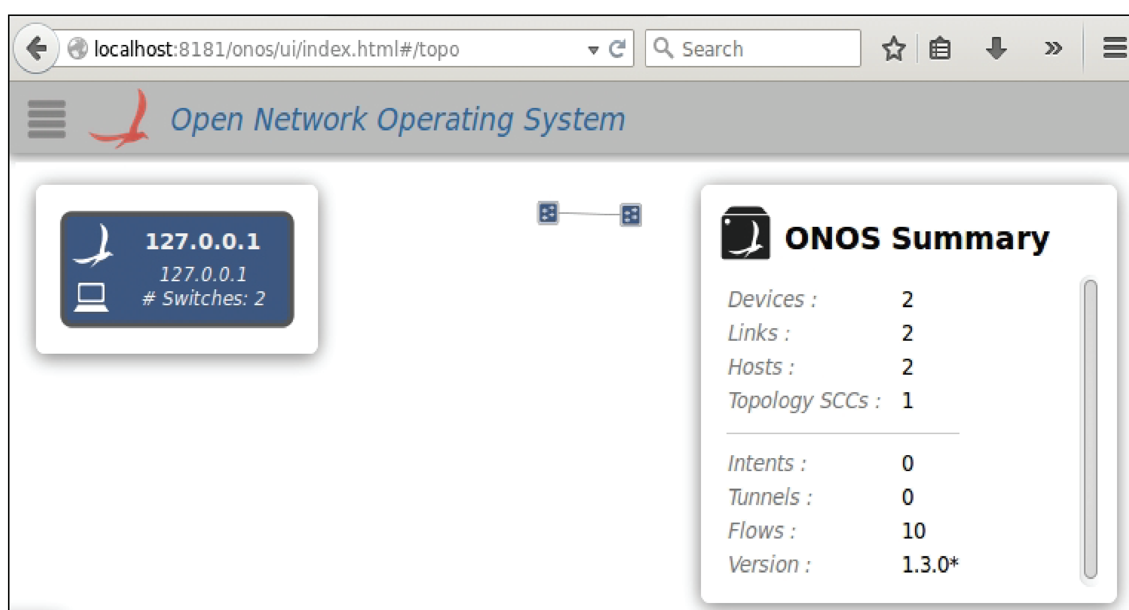


Figura 8. Interface web ONOS. Fonte: Subramanian et al. [23].

Da maneira que hoje está sendo disponibilizado o *openflow* pelos fabricantes ainda não é possível deixar uma rede totalmente programável. Os fabricantes implementam uma API fechada nos dispositivos, apenas utiliza um controlador para encaminhar comandos para os dispositivos alvos. Porém, não se tem a informação exata de como o equipamento vai interpretar os comandos enviados. Para tentar resolver esse problema da API com código fechado pelos fabricantes, foi desenvolvida uma linguagem de programação para o plano de dados ou plano de encaminhamento dos pacotes.

[11] *throughput* é uma taxa de transferência de um equipamento para outro.

2.4 LINGUAGEM DE PROGRAMAÇÃO P4

Conforme o conceito de SDN apresentado na Seção 2.2, foi projetada e desenvolvida uma linguagem de programação para possibilitar a programabilidade do plano de dados de um *host* de rede.

No ano de 2013 tiveram início as primeiras ideias, que foram apresentadas na publicação do Bosshart et al. [24] em 2014 no SIGCOMM, sobre a possibilidade de realizar a programação do plano de dados em uma linguagem de alto nível. A linguagem em questão é a P4, sigla referenciada ao nome *Programming Protocol-Independent Packet Processors* (programando processadores de pacotes independentes de protocolo). A primeira versão da linguagem P4 foi a 14 e em 2016 evoluiu para a versão 16, originada da cooperação de muitos desenvolvedores e com muitas experimentações realizadas.

A Figura 9 apresenta a comparação entre *openflow* com a linguagem P4. Nota-se que o *openflow* apenas traduz as regras encaminhadas pelo controlador. Essa tradução segue as instruções definidas pelo fabricante. Já, a linguagem P4 é compilada para funcionar no dispositivo alvo. Isso significa que o usuário pode realizar uma configuração mais personalizada e com isso ter mais liberdade para definir as suas próprias instruções. O lado esquerdo apresenta o esquema do paradigma P4, onde o controlador recebe as informações dos fluxos dos pacotes e executa as ações definidas na programação P4 e executa no alvo de rede (equipamento de rede), sendo que, neste contexto, o alvo é o dispositivo que receberá as instruções para serem executadas. Do lado direito da Figura 9 é apresentada uma implementação utilizando o protocolo *openflow* que é um protocolo de comunicação que traduz as regras para o alvo. Na Figura 9 o alvo está sendo apresentado com um *switch*.

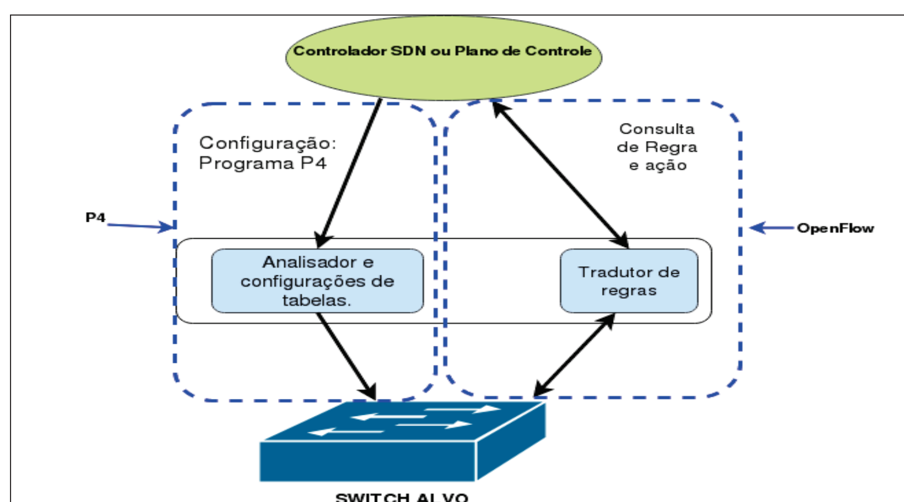


Figura 9. Comparação do *openflow* e P4. Fonte: Bosshart et al. [24].

Linguagem P4 estabelece como o pacote é processado pelo dispositivo no plano de encaminhamento dos pacotes. Inicialmente foi desenvolvida para funcionar em *switches*

programáveis para auxiliar nas definições de supervisão de uma rede como, por exemplo, medição de uma rede inteira usando uma estrutura de dados probabilística [25], balanceamento de carga em *datacenters* [26] e mecanismo de monitoramento proposto por Li et al [27], que foi programado usando P4 para armazenar fluxos existentes de tráfego com pouco uso de memória.

A linguagem P4 possui três objetivos principais, segundo Bosshart et al. [24]:

- **Reconfigurabilidade**: *Software* de gerenciamento pode ser configurado e reconfigurar os dispositivos alvos, de maneira transparente ao dispositivo final;
- **Independência de protocolo**: O dispositivo alvo não pode ser limitado aos protocolos que suporta. O controlador deve descrever os cabeçalhos para serem reconhecidos e também descrever as ações que serão executadas pelo dispositivo alvo;
- **Independência de hardware**: O *software* controlador não necessita saber as informações físicas do dispositivo alvo. O código P4 deve ser compilado para linguagem de máquina que será aplicado a qualquer alvo de rede.

Os elementos do P4 versão 16 podem ser divididos em 5 grupos principais, que permitem a programação da *pipeline*. O primeiro grupo contém os tipos de dados básicos e os complexos como, por exemplo, os *arrays* e os *headers*; o segundo são as expressões compostas e também as operações básicas e operadores; o terceiro são os elementos de controle dos fluxos, compostos de estruturas que representam o fluxo entre as tabelas e o plano de dados; o penúltimo grupo, o *parsers*, que é a máquina de estados para extração de componentes ou campos dos pacotes e o último grupo, o *externs* que é composto por bibliotecas com componentes adicionais.

O *header* é o mais importante tipo de dado composto, pois possui os campos que serão extraídos dos cabeçalhos dos pacotes que entram no *parser*. Os campos podem ser os tipos *bit*, *int*, *varbit*. No exemplo da Figura 10, que apresenta um cabeçalho *Ethernet* padrão muito utilizado nas redes dos dias atuais, verifica-se que esse *header* busca algumas informações, como o endereço de destino, o endereço de origem e o tipo do pacote. Ao lado consta quantos bits representa cada campo do pacote. Os *bits* representam a posição e o local que este campo está apresentado no cabeçalho do pacote [24].

```
1.  header ethernet_t {
2.      bit<48> dstAddr;
3.      bit<48> srcAddr;
4.      bit<16> etherType;
5.  }
```

Figura 10. Exemplo de um *hearder* em P4. Fonte: Bosshart et al. [24].

O *Parsers* tem a função de mapear os fluxos dos *bits* nos cabeçalhos, através de uma máquina de estados finita. Máquina de estados são as regras e os estados de como passar o pacote pelo analisador. Inicia-se o processo pelo estado de *start* e passa-se para os estados onde são extraídas as informações de acordo com o cabeçalho do pacote. Nesta etapa o estado pode ser *accept*, quando os pacotes são aceitos pelo *parsers*, ou pode ser *reject*, que são aqueles pacotes rejeitados pelo analisador. As etapas do *parsers* estão representadas na Figura 11 e são definidas pelo programador [28].

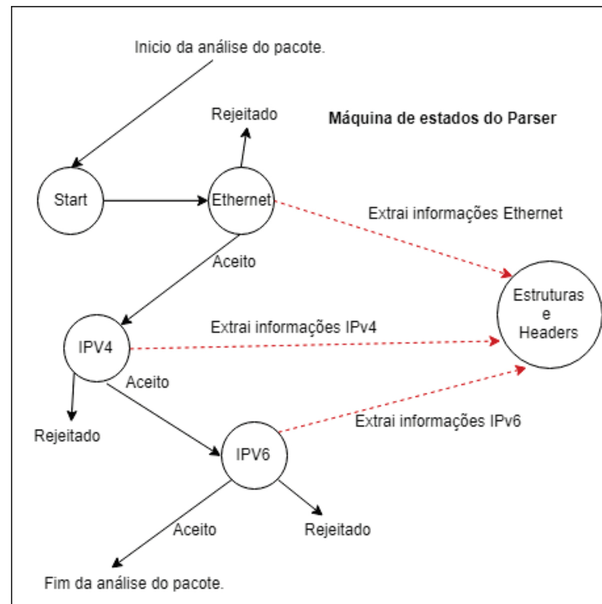


Figura 11. Etapa da máquina de estados do *parsers* . Fonte: Autor.

Os *match-action* são tabelas de correspondência e ação, que têm a função de verificar se o pacote possui os campos que correspondem às regras presentes na tabela. Após a verificação, será executada a ação determinada para o pacote.

Na operação do *match-action* ocorre a execução de três blocos: controle, tabela e ações. No controle existe um código onde ocorre a execução de forma sequencial, sem a presença de estrutura de *loop*. Pode implementar neste módulo, as funções como estruturas de decisão (*If*) e também possui sentenças, que se forem aceitas, com base nas regras implementadas em uma tabela do plano de controle do dispositivo de rede, é executada uma ação sobre os pacotes [28].

A Figura 12 mostra as etapas presentes no código P4 para modelo de *pipeline* que utiliza a arquitetura NFP (detalhada na Seção 2.5). Cada função foi enumerada na Figura 12. A descrição do que representa cada função está descrita abaixo [28] [24]:

- 1 : Bibliotecas: os algoritmos utilizando P4 possuem duas bibliotecas padrões desenvolvidas pela comunidade P4[12], denominadas "core.p4"e "v1model.p4".

[12]Link para comunidade P4: https://www.p4.org/.

- 2 : *Headers* e estruturas: estrutura para armazenamento de informações dos cabeçalhos dos pacotes e de metadados coletados no *parser*.
- 3 : *Parser* ou analisador: essa função analisa todos os pacotes que entram no equipamento e depois extrai as informações para as estruturas ou *headers*.
- 4 : *Verify Checksum*: essa função realiza a verificação do *checksum* dos pacotes quando entram no alvo P4.
- 5 : *Ingress* ou processamento de entrada: nesse estágio inicia o processamento dos pacotes que estão adentrando no alvo P4. Nessa etapa é realizado o *match-action*, onde o alvo de rede busca algumas informações do cabeçalho do pacote e verifica se correspondem com as informações cadastradas em suas tabelas. Se corresponder executa a ação que foi programada e encaminha o pacote para a porta de saída.
- 6 : *Egress* ou processamento de saída: nesse estágio o pacote já está sendo encaminhado à interface de saída do equipamento. O alvo de rede novamente vai buscar as informações dos cabeçalhos e verificar em suas tabelas. Se alguma informação for aceita pela tabela, é executada a ação e depois é destinado o pacote para a próxima função.
- 7 : *Compute Checksum* : essa função realiza a atualização do *checksum* do pacote.
- 8 : *Deparser* : nesse percurso é realizada a montagem dos *headers* novamente nos pacotes de dados. Libera as informações armazenadas nas estruturas e destina para o pacote.
- 9 : *Main* : a função *main* é a etapa que chama todas as outras funções do código P4 de forma organizada, para ser destinada para um alvo de rede programável.

A Figura 13 apresenta a estrutura de funcionamento do código P4, onde os pacotes entram no dispositivo. Após, vão passar pela função *parser* onde as informações do pacote serão extraídas e encaminhadas para as estruturas de *headers* onde armazenam as informações enquanto o pacote passa por outras funções. A próxima função vai verificar o *checksum* dos cabeçalhos exportados. Na função seguinte encontra-se dentro o *match-action*, onde se dá o processamento de entrada (*ingress*), que irá analisar se o *header* armazenado corresponde a uma chave específica. Se corresponder vai aplicar a ação no *header*. A próxima etapa é o processamento de saída (*egress*), o qual que também possui uma chave[13] que, caso corresponder, será aplicada uma determinada ação no *header*.

Na próxima função, os *headers* são verificados novamente e se os cabeçalhos estiverem corretos, os pacotes vão para a última etapa, o *deparser*. Por fim, os cabeçalhos são remontados no pacote e destinados para a interface de saída do dispositivo alvo.

[13]Chave é o nome dado pelo P4 para os campos das estruturas ou *headers* que são encaminhados para o plano de controle, para verificar se corresponde com as informações que estão na tabela do plano de controle.

```
#include <core.p4>
#include <v1model.p4>

header ethernet_t {
    macAddr_t destinationAddress;
    macAddr_t sourceAddress;
    bit<16>   etherType;
}

parser MyParser(packet_in packet,out headers hdr,inout metadata meta,inout standard_metadata_t standard_metadata) {
    state start {}
}
control MyVerifyChecksum(inout headers hdr, inout metadata meta) {
    apply {}
}
control MyIngress(inout headers hdr,inout metadata meta,inout standard_metadata_t standard_metadata) {

    action ipv4_forward() { }
    table ipv4_lpm {}
    apply {}
}

control MyEgress(inout headers hdr,inout metadata meta,inout standard_metadata_t standard_metadata) {

    action add_meta() { }
    table add {}
    apply { }
}
control MyComputeChecksum(inout headers hdr, inout metadata meta) {
     apply { update_checksum() }
}

control MyDeparser(packet_out packet, in headers hdr) {
    apply {}
}

V1Switch( MyParser(), MyVerifyChecksum(), MyIngress(), MyEgress(), MyComputeChecksum(), MyDeparser() ) main;
```

Figura 12. Exemplo de código P4. Fonte: Autor.

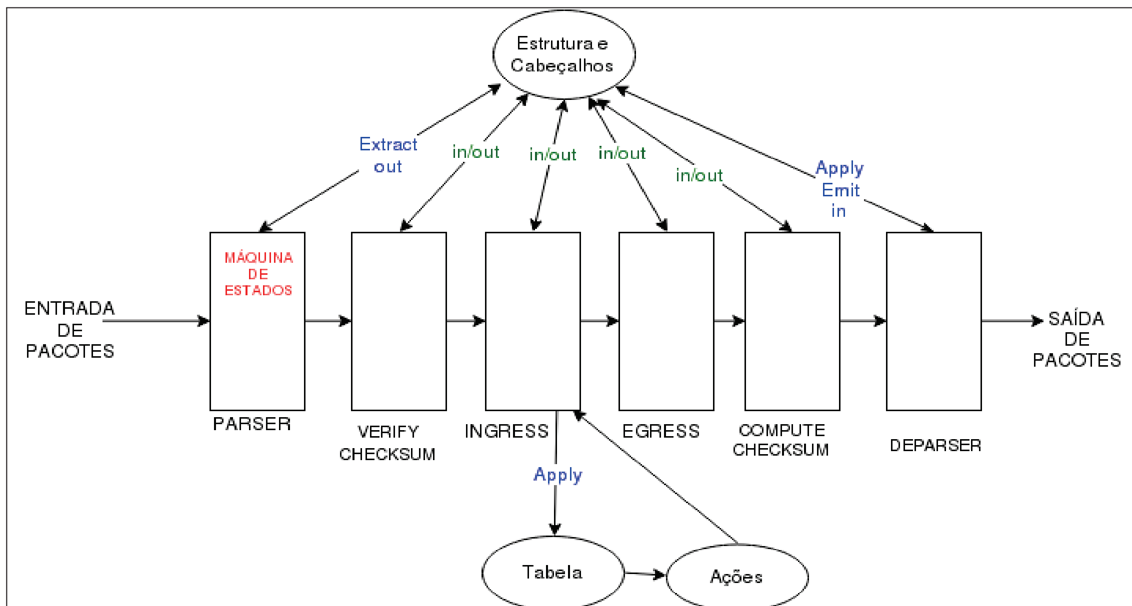


Figura 13. Estrutura do código P4. Fonte: Autor.

2.5 ARQUITETURA NFP

A arquitetura chamada de *Network Flow Processor*(NFP) realiza o processamento dos pacotes de dados em uma SmartNIC. Essas NFPs são designadas para serem utilizadas em servidores, *hosts* de redes, ambientes de virtualização e *cloud*. A arquitetura NFP está presente nas smartNICs da Netronome e fornece um alto desempenho e uma ampla capacidade de programação [29]. Essa seção descreve o funcionamento da arquitetura NFP.

Primeiramente é importante saber que uma NFP possui quarenta *microengines*, sendo que essas são os núcleos de processamentos de uma placa NFP. Cada núcleo suporta o processamento de oito *threads*. Cada *microengine* possui espaço para armazenar instruções de controle e essa *microengine* pode habilitar o modo de compartilhamento e assim compartilham os recursos entre outros núcleos. Desta forma, aumenta a capacidade de armazenamento de instrução de controle. A frequência de comunicação é de 1,4 GHz. Cada núcleo possui 256 registros que são utilizados para programação e também para questões de informações lógicas e físicas. Por exemplo, os registradores são utilizados para o controle do *TimeStamp* da placa. Também possui registros que são *buffer* para armazenamento temporário de informações durante as transferências de dados [30] [31] [29]. Essas informações são importantes para entender o mecanismo de funcionamento da arquitetura da placa Netronome.

A Figura 14 representa o funcionamento da arquitetura NFP. Os pacotes entram na smartNIC e são identificados e armazenados nos *buffers* e memórias. Enquanto os pacotes estão dentro da placa eles são processados pelas *microengines*, onde cada uma delas executa o processamento de uma tarefa independentemente [31]. Essas tarefas são as etapas do código que foi compilado nesta placa.

Para a programação do P4 na versão 16 foi utilizado o *V1model*. O *V1model* representa a organização das etapas de cada função ou operação que são executadas pelo dispositivo de rede. Toda essa estrutura está sendo dividida e sendo executada em cada uma das *microengines*. Depois que o pacote foi processado pelas *microengines*, ele será reordenado para sair do dispositivo de rede.

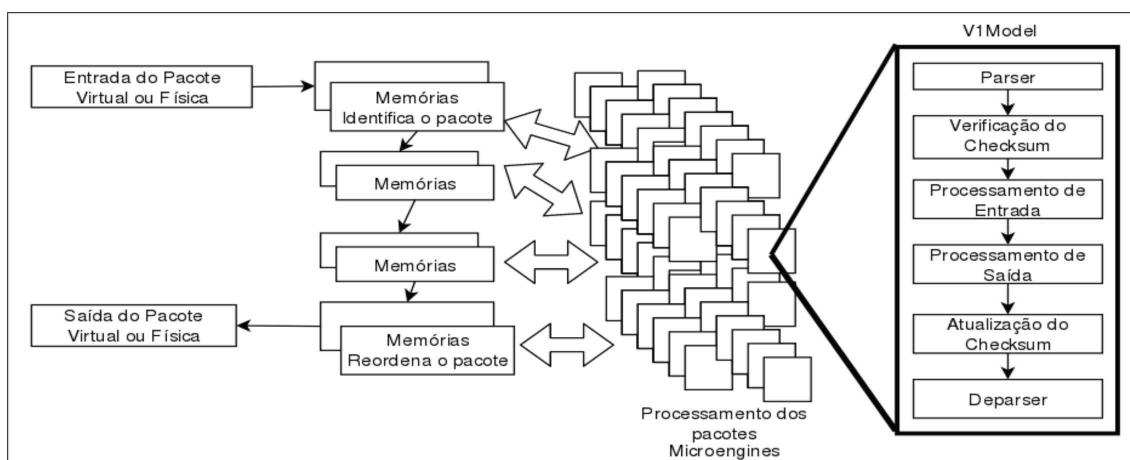


Figura 14. Arquitetura NFP com V1Model. Fonte: Autor.

Na próxima seção será descrito o funcionamento do compilador do código P4. Esse compilador transformará o código em um *firmware* suportado pela placa da Netronome, respeitando as regras da arquitetura NFP.

2.6 P4C - COMPILADOR DA LINGUAGEM P4

O P4C é o compilador do código P4. Ele compila o código para qualquer dispositivo que suporta a linguagem P4. Esse é o compilador padrão que se encontra no site oficial[14] da linguagem P4, porém, cada fabricante de *hardware* pode modificar esse compilador para os seus dispositivos. Para a execução do presente trabalho será utilizado o compilador P4C SDK 6 da Netronome(Seção 3.2).

A Figura 15 mostra o fluxo do compilador. A entrada de um fluxo no compilador P4 inicia com um código escrito com a extensão “*.p4*” e com um *log* para analisar as falhas da compilação. A etapa inicial é converter o código em um formato intermediário único (IR), na etapa de *frontend*. A próxima etapa é a transformação do código intermediário genérico em um código de baixo nível que é suportado pela plataforma à qual o código será encaminhado. Essa etapa recebe o nome de estágio *backend*. Nessa etapa o compilador deve suportar uma grande quantidade de equipamentos diferentes, de marcas e arquiteturas diferentes, aumentando o ecossistema de dispositivo alvo suportado pelo compilador [32].

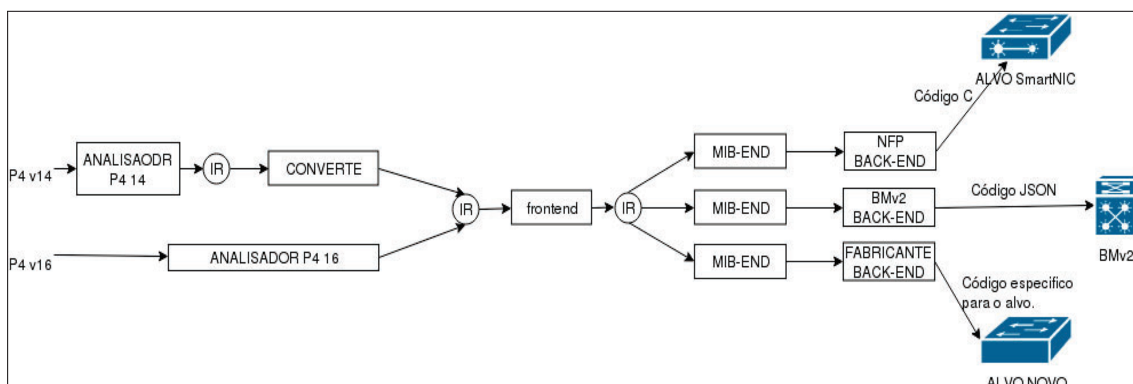


Figura 15. Diagrama do Compilador P4. Fonte: KIM [33].

Cada fabricante implementa uma etapa diferente no *backend*, pois nesta etapa o fabricante adiciona as especificações a serem utilizadas pelo equipamento físico.

2.7 MONITORAMENTO DE REDE

Monitoramento de rede são medições que visam auxiliar no conhecimento do comportamento de uma rede e também podem vir a auxiliar na solução dos problemas. Os sistemas de monitoramento mostram informações das verificações dos *status* dos equipamentos ou do tráfego passante na infraestrutura da rede. Com o monitoramento de rede é possível monitorar o *status* da CPU, memória, tráfego das interfaces, monitoramento de

[14] Site do compilador: https://github.com/p4lang/p4c.

fluxo, perdas e o tempo de resposta. Possibilita, também, criar relatórios e mensagens para alertar, antecipando problemas ou falhas da estrutura de rede.

Nos últimos anos, o monitoramento do tráfego se tornou necessário para todos os administradores de rede. Pois com esses *softwares*, gera-se a possibilidade de realizar decisões para melhorias na rede ou identificar falhas. O administrador de rede atual precisa de um *software* de supervisão de rede que seja o mais preciso possível. Para um monitoramento mais preciso é necessário que o *software* capture informações sobre a qualidade do fluxo de dados. Este tipo de análise dos fluxos é essencial para identificação de um ataque de grande ou pequeno porte na rede, que possa estar afetando o desempenho da rede [34].

A supervisão dos fluxos de dados que trafegam na estrutura que partem da mesma origem e são entregues ao mesmo destino chama-se *flow measurement* (medição de fluxo). A interface de monitoramento coleta dados e gera estatísticas sobre o fluxo das camadas de transporte ou enlace e encaminha para um servidor que armazena as informações coletadas. Essas aquisições de dados sobre o fluxo são realizadas utilizando um protocolo de entrega como, por exemplo, o Netflow [35], Ipfix [36] e Sflow [37].

Como o paradigma de SDN possibilita a programabilidade de uma estrutura de rede e permite programar o plano de dados, é possível desenvolver um método de supervisão que busca coletar dados mais específicos de um fluxo de pacotes dentro do próprio plano de encaminhamento, possibilitando alguns monitoramentos mais personalizados [38]. Desta maneira, na Seção 2.8 será demonstrado um método de monitoramento que utiliza a linguagem P4 e os conceitos de SDN, e na Seção 2.9 algumas métricas que podem ser supervisionadas.

2.8 MÉTODO INT

O método *In-band Network Telemetry*(INT) é um mecanismo que pode realizar medições dos dispositivos percorrendo o plano de dados de uma rede. Pode realizar análise pelos pacotes individualmente, sem afetar o desempenho, de maneira simplificada. O INT é um *framework* programado dentro do plano de dados dos dispositivos de rede [39].

O INT armazena as informações do estado da rede dentro do pacote conforme passa pelos dispositivos que possui o INT programado. No último dispositivo, antes do pacote ser entregue, o plano de dados desse equipamento analisa o conteúdo do pacote e reconhece todas as informações de telemetria dos equipamentos que foram encaminhados. Essas informações de telemetria podem ser analisadas dentro do plano de dados e só após serem encaminhadas para o plano de controle [32], para depois ser encaminhado para um *software* que irá demonstrar essas informações em uma interface gráfica.

Com a difusão do conceito de máquinas virtuais e redes virtuais, surgiram soluções para roteadores, *firewalls* e *switches* virtualizados, atendendo à demanda de comunicação

entre os *hosts* virtualizados, surgindo então a necessidade de um sistema de monitoramento eficaz para esses equipamentos virtualizados. Para suprir a necessidade de um monitoramento *point-to-point* completo, onde inclui a supervisão da parte física (*hardware*) com a parte virtual (*hardware* virtual) numa interface gráfica, surgiu o método INT. Esse método busca coletar dados para um monitoramento mais completo programado diretamente no dispositivo ou plano de encaminhamento virtual, ou físico [40], para realizar procedimentos de manutenção de forma rápida contra problemas e falhas de topologia de rede.

Atualmente notou-se que o INT tem um potencial muito grande no monitoramento de redes virtuais ou físicas, pois possibilita um monitoramento mais dinâmico e gerenciado pelo administrador. Agora esse método está sendo utilizado também nas redes físicas, facilitando o gerenciamento e o monitoramento das redes.

Segundo Hira et al. [40], o método INT pode mostrar informações que hoje não são possíveis de visualizar com as ferramentas atuais de monitoramento, conforme alguns exemplos a seguir:

- Utilizando a ferramenta de *traceroute* para avaliar a rota que o pacote está trafegando, porém essa ferramenta suporta apenas um caminho, com o INT possibilita analisar múltiplos caminhos que o pacote pode percorrer até atingir o destino;

- Monitoramento da utilização de banda de múltiplos *links* ao longo do caminho do pacote, capturando informações de todo o caminho do pacote, do ponto de origem até o ponto de chegada do pacote;

- O monitoramento da latência de todos os possíveis caminhos que o pacote pode ser transmitido pela rede.

Percebe-se que, com os exemplos acima descritos, é possível ter subsídios para criar um roteamento mais dinâmico e inteligente através de informações coletadas pelo INT. Com um sistema de controle de roteamento com os dados capturados do INT é possível criar um *software* que pode realizar troca de rota do tráfego, buscando a melhor qualidade e menor tempo de encaminhamento fim-a-fim (*end-to-end*). Pois, com esse método pode-se constatar um congestionamento em alguma etapa do caminho que o pacote percorreu e também saber a latência de outros possíveis caminhos. Desta forma, o roteador sabe qual é a melhor trajetória para esse fluxo de dados.

O INT pode ser executado em múltiplos dispositivos de rede programados, sendo capaz de coletar dados sobre o fluxo e direcionar essas informações para um *software* capaz de identificar vários problemas de rede de forma intuitiva para o usuário, essas informações são coletadas e transportadas em cada pacote que trafega pelos dispositivos de rede. [41].

O INT exporta as informações diretamente do plano de dados, por este motivo não sobrecarrega o processador ou ocorre alguma perda de desempenho, porque não precisa

interagir com o plano de controle a todo instante. Hoje utilizam-se para o monitoramento os protocolos ICMP[15] ou SNMP[16] que podem ser considerados ineficientes, porque o sistema de supervisão remoto busca o estado da rede periodicamente em tempos predefinidos pelo fabricante. O dispositivo retorna as informações e o *software* de monitoramento armazena e apresenta essas informações. Por ser realizada uma análise periodicamente, pode não ser possível detectar alterações na rede que ocorre em milissegundos, problema esse que o gerente pode não conseguir perceber pelo monitoramento atual [40].

Para a programação do método INT, é fundamental a possibilidade de programação do plano de dados. Uma boa opção de linguagem a ser utilizada atualmente é a linguagem P4 que se mostra muito eficiente para a aplicação desse *framework* INT. Pois é possível criar cabeçalhos e criar campos de dados para adicionar informações de telemetria dentro do pacote de dados.

A Figura 16 ilustra o cabeçalho de um pacote quando passa por um dispositivo que está realizando a telemetria por INT. Por exemplo, se o INT está analisando a latência dos pacotes, o método vai inserir no cabeçalho um campo INT 0 com o valor da latência quando passa por um dispositivo. Ao passar por outro equipamento, serão adicionadas as informações de telemetria em mais um campo dentro do pacote. Quando chega no final, poderá verificar todos os valores de latência armazenados no INT *N* de cada dispositivo por onde o pacote passou. No final do encaminhamento do pacote, será possível avaliar a qualidade do fluxo de dados que transitou por esses *switches*.

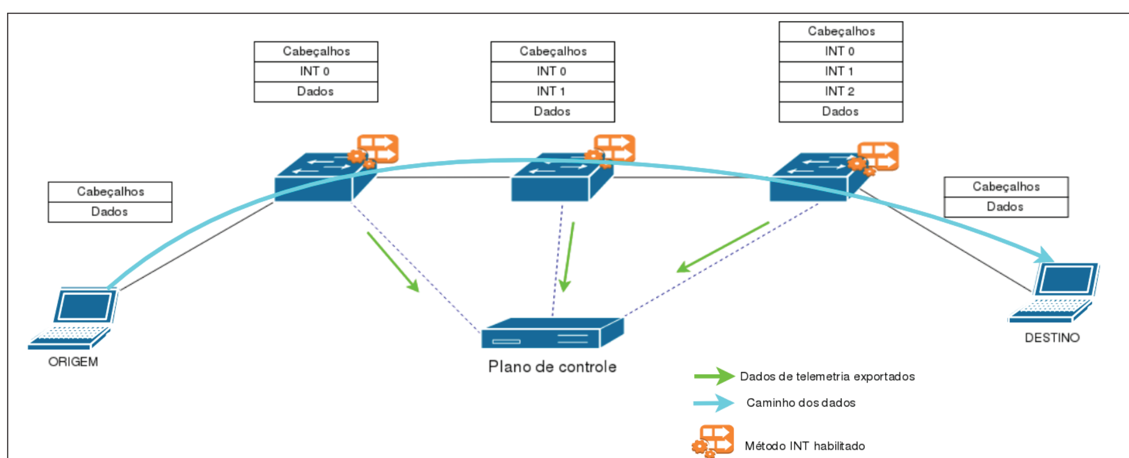


Figura 16. Funcionamento do INT. Fonte: Autor.

Esses dados são avaliados em cada pacote que transita pelo plano de dados e, caso necessário, serão exportados para o plano de controle para a visualização do admi-

[15]ICMP(*Internet Control Message Protocol*) é um protocolo utilizado para fornecer relatório de erros para o *host* origem.

[16]SNMP(*Simple Network Management Protocol*) é um protocolo utilizado para monitoramento de *hardware*, pois os *softwares* de monitoramento encaminham pacotes SNMP que requisitam as informações para os *hosts* supervisionados, e o mesmo retorna a informação que foi requisitada pelo sistema monitor.

nistrador da rede ou para execução de algum processo, visando melhorar o desempenho da rede.

Atualmente a capacidade de poder monitorar a parte física e lógica de uma rede, diretamente no plano de dados e após, informar ao usuário de maneira legível e ágil, está se tornando muito importante para garantir o funcionamento da rede mundial de computadores. Pois, conforme ocorre o aumento da quantidade de usuários na Internet, cresce a complexidade de supervisionar e gerenciar as redes. O INT traz algumas métricas que podem auxiliar na resolução do problema.

2.9 MÉTRICAS DISPONÍVEIS

As possíveis métricas utilizadas em alvos programados em P4 que estão utilizando INT, serão apresentadas nesta seção. Essas métricas serão encaminhadas para um coletor, por dentro dos pacotes de dados, para que um sistema coletor de informações INT extraia as métricas. Depois serão encaminhadas para um banco de dados que armazenará as informações de cada pacote de dados. Um *software* busca as informações do banco em intervalos de tempos. Depois esse *software* define os fluxos de dados e destina os dados para uma interface de visualização, a fim de auxiliar o usuário na compreensão desses dados coletados.

A Seguir são apresentados os dados que podem ser coletados pelo INT, conforme apresentado pelo grupo P4 [42]:

- *Switch* ID: valor de identificação do *switch*, que identifica por quais *switches* os pacotes transitaram. Esse valor deve ser único para cada *switch* do mesmo domínio;
- Versão do plano de controle: esse valor traz as informações de alteração dos estados do plano de controle. Com essa informação, sabe-se que o pacote passou por determinada versão do plano de controle.

Outros dados são coletados no *ingress* e no *egress*. Como já mencionado na Seção 2.6, o *ingress* é a etapa em que a *pipeline* processa todos os pacotes recebidos por uma interface (RX). Nesta etapa pode-se coletar as seguintes métricas:

- *Ingress port ID*: identificação por onde o pacote foi recebido;
- *Ingress timestamp*: o horário que o pacote entrou no *ingress*;
- *Ingress port RX pkt count*: total de pacotes recebidos no dispositivo;
- *Ingress port RX byte count*: total de bytes recebidos no dispositivo;
- *Ingress port RX drop count*: total de pacotes descartados no *ingress* do dispositivo;

- *Ingress port RX utilization*: percentual da utilização da interface de entrada do pacote.

No *egress*, como já mencionado na Seção 2.6, é a etapa em que a *pipeline* processa todos os pacotes que são transferidos de uma interface (TX) para um próximo equipamento. Nesta etapa pode-se coletar as seguintes métricas:

- *Egress port* ID: identificação para onde o pacote foi transferido;
- *Egress timestamp*: o horário que o pacote saiu no *egress*;
- *Egress port RX pkt count*: total de pacotes transferidos no dispositivo;
- *Egress port RX byte count*: total de bytes transferidos no dispositivo;
- *Egress port RX drop count*: total de pacotes descartados no *egress* do dispositivo;
- *Egress port RX utilization*: percentual da utilização da interface de saída do pacote.

Também pode armazenar métricas sobre as filas dos alvos de rede. Essas métricas são apresentadas abaixo:

- *Queue occupancy*: porcentagem de ocupação das filas das interfaces;
- *Queue congestion status*: informa se a fila está congestionada.

2.10 TRABALHOS RELACIONADOS

O artigo de Jonghwan Hyun et al. [43] tem muita relação com a presente dissertação. A topologia apresentada no artigo foi desenvolvida para realizar testes em um coletor de dados para o ONOS [44]. Ele apresenta um projeto com uma implementação de monitoramento de dispositivos de rede através do INT programado em linguagem P4. Neste trabalho foi desenvolvida uma máquina virtual com o *software* de controle ONOS instalado para realizar o gerenciamento dos fluxos. Também foi instalado o *software* Grafana para demonstrar os dados coletados pelo coletor INT. Neste trabalho foi utilizado um ambiente de emulação chamado de Mininet, onde foi desenvolvida uma rede na qual conseguiram realizar uma interação com o compilado P4C e com o BMv2[17] para simular um *switch* P4.

O ambiente criado pelo Jonghwan Hyun et al. [43] foi totalmente virtualizado. Nele são relatados problemas de desempenho. Muitos desses problemas referem-se a elevada utilização de CPU e a utilização de uma grande quantidade de memória. Esses problemas ocorreram porque o coletor precisa analisar todos os dados que chegam do INT. Um problema muito importante relatado no trabalho dele foi o do custo da utilização de banda, pois

[17] *Behavioral Model* Versão 2 é *software* que simula o comportamento de *switch* através de fórmulas matemáticas.

durante os testes percebeu-se que diminuiu 16% na taxa de transferência quando o método INT está ativo em todos os fluxos de dados e também em fluxo muito grande de dados. Ele menciona em realizar possíveis métodos para serem utilizados a fim de reduzir esse consumo de dados. Um dos métodos apresentado foi o desenvolvimento de uma telemetria por amostragem dos fluxos de dados.

Já no trabalho de Jan-Olof Andersson [45], foi realizada a implementação do método INT e adicionou-se um algoritmo detector de evento sobre o plano de dados dos *switches* com o objetivo de reduzir o consumo de recurso no coletor de dados INT. Conforme mencionado pelo autor, o objetivo dele foi alcançado parcialmente e possui ainda muitas possibilidades a serem estudadas.

A presente dissertação utilizou um ambiente de programação diretamente em placas que suporta a programação do P4, neste caso utilizando placas SmartNIC da Netronome. O presente trabalho conserva alguns conceitos implementados no trabalho de Jan-Olof Andersson [45], sendo o principal conceito, foi a utilização das placas da Netronome para realizar a programação sobre o P4. A implementação INT está sendo projetada para trabalhar sobre fluxos específicos de rede, com o objetivo de ter pacotes INT com tamanho menor e reduzir a quantidade de pacotes encaminhados para o coletor de dados INT.

3. PROCEDIMENTOS METODOLÓGICOS

Após estudar o funcionamento das tecnologias que envolvem o conceito de SDN e a conectividade das redes atuais, foi desenvolvido um projeto utilizando smartNICs para possibilitar as análises do funcionamento da programação da linguagem P4 com a implementação da telemetria diretamente em um processador programável. Esse projeto irá auxiliar no processo de análise dos pacotes INT customizados para uma implementação sobre plano de dados. A infraestrutura que foi montada para a criação desse projeto está descrito na Seção 3.3, e o funcionamento das smartNICs está descrito na Seção 3.1.

Neste presente trabalho busca-se desenvolver uma ferramenta capaz de coletar informações para fins de telemetria de um ou mais *switches* com uma precisão superior às ferramentas existentes atualmente para monitoramento de redes. Desta maneira, será necessário a programação do plano de dados do *switch* e não depender de um *firmware* do fabricante.

No capítulo apresenta-se o projeto que foi desenvolvido para a criação de um ambiente para análise de rede utilizando programação diretamente na interface. Implementou-se todo o ambiente sem a utilização de interfaces virtuais, sendo usadas apenas interfaces físicas da empresa Netronome. Após a implementação de um ambiente de simulação iniciou-se o desenvolvimento de um código P4 para operar as placas, a ser detalhado nas Seções 3.4, 3.5, 3.6, 3.7 e 3.8.

3.1 SMARTNIC NETRONOME

Atualmente, a empresa Netronome está desenvolvendo placas de rede que possuem uma arquitetura que permite a programação através da linguagem P4, chamadas de smartNICs (NIC: *Network Interface Card* ou Placa de Interface de Rede), as quais são projetadas para suportar o alto crescimento das demandas das redes. O uso dessas NICs são cada vez mais necessárias para servidores de dados em nuvem, justamente pelo fato de encaminhar o processamento dos dados de entrada(*input*) e saída(*output*) de rede, para uma central de processamento presente nas NICs. A possibilidade de programação de tecnologias SDN sobre as placas está impulsionando o uso dessas NICs [29].

A smartNIC é conhecida também como adaptador inteligente de servidor. Um dos principais motivos do aumento da utilização das placas foi a complexidade atual das operações executadas no plano de dados. Pois, com tantos protocolos novos surgindo hoje em dia, como, por exemplo, o VXLAN[18] ou outros protocolos de encapsulamento e também

[18] VXLAN (virtual extensível LAN) é um protocolo utilizado para encapsulamento de camada 2 em camada 3, com a VXLAN possibilita a virtualização de 16 milhões de rede em camada 2.

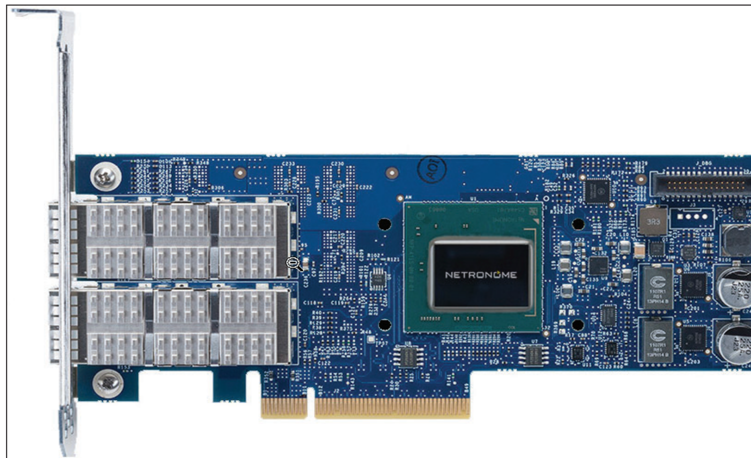


Figura 17. Placa smartNIC Netronome Agilio CX 40 Gbps. Fonte: Netronome [46].

com as elevadas larguras de banda passantes nas interfaces está ocorrendo uma carga excessiva para processamento dessa demanda no plano de controle. Portanto, o plano de dados possibilitará que essa demanda seja processada em mais baixo nível, ou seja, dentro do processador da smartNIC [29].

O processador utilizado nessas smartNICs é o NFP-4000 da própria Netronome da 6º geração, ele possui um processamento *multi-thread*. O NFP-4000 é direcionado para o processamento dos pacotes no plano de dados.

Por permitir a programação P4, as smartNICs contemplam uma arquitetura de correspondência e ação. Significa que, de acordo com os dados coletados dos cabeçalhos dos pacotes, o *software* P4 vai verificar se os dados encaminhados pelo P4 correspondem com as informações nas tabelas do plano de controle. Após, aplica-se uma determinada ação nos pacotes. Com essa arquitetura, essas placas se adaptam facilmente e rapidamente aos novos recursos [46].

As placas da Netronome Agilio CX possuem modelos com largura de banda de 10 Gbps, 25 Gbps e 40 Gbps. Esses modelos contêm 2 GB de memória DDR (Double Data Rate) e utilizam um processador ARM (Advanced Risc Machine), processadores denominados NFP (*Network Flow Processor* ou processador de fluxo de rede). Eles têm a possibilidade do processamento *in-of-band*[19] ou mecanismos *out-of-band*[20], proporcionando a programação do plano de dados e, se for necessário, gerenciá-lo pelo plano de controle [29].

As placas da Netronome utilizada nesse projeto possui duas portas de 40 Gbps. Quando ativado o *software* de programação P4 nos *hosts* que possui a smartNIC instalada. A placa passa a funcionar com todo o processamento e o encaminhamento de pacote dentro da placa sem passar para o *host* hospedeiro. Na Figura 18 nota-se que às duas portas de 40 Gbps físicas (na cor laranja) não aparecem no comando "ip link", utilizado no linux

[19] *in-of-band* é um mecanismo de gerenciamento sobre o tráfego de rede.
[20] *out-of-band* é um mecanismo de gerenciamento fora do tráfego de rede.

para visualizar as interfaces de rede no computador. Aparece apenas as interfaces virtuais (na cor azul), desta forma para um pacote entrar nesse computador ele deve chegar pela interface física e ser processador pelo algoritmo em P4 para então se encaminhado para uma interface virtual. Caso o pacote for encaminhado para a outra interface física, esse pacote é apenas processado pelo processador da SmartnNIC.

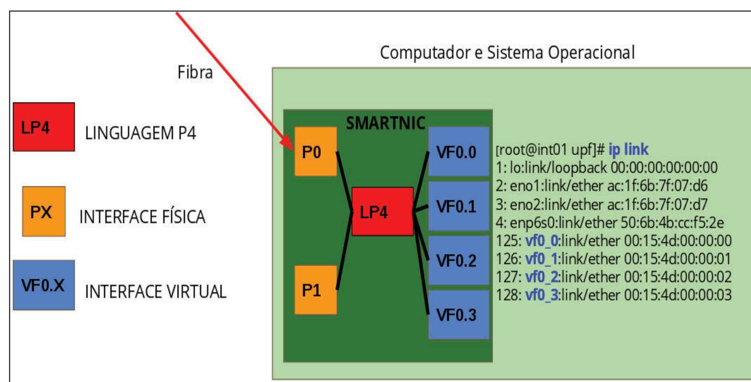


Figura 18. Funcionamento da SmartNIC com P4 ativado. Fonte: Autor.

Atualmente já existe um IDE (*Integrated Development Environment* ou ambiente de desenvolvimento integrado) para programação e aplicações de *debugs* para criação de um código em linguagem P4 e execução diretamente na smartNIC, sendo que maiores detalhes serão apresentados nas próximas seções.

3.2 COMPILADOR AGILIO P4C SDK

O Agilio P4C SDK é um *kit* de desenvolvimento de *software* ou *Software Development Kit* que oferece um conjunto de ferramentas e recursos que permitem o desenvolvimento completo de um *firmware* para a placa da Netronome.

A Agilio P4C SDK possui uma interface gráfica, com vários recursos para a programação do P4 e um *debugger*, que busca coletar dados da compilação e do comportamento do *software* desenvolvido, além de uma ferramenta gráfica para mostrar as etapas do *parser*, *ingress* e *egress*. Conforme o exemplo apresentado na Figura 19, a interface gráfica é um *software* chamado de *Programmer Studio*(PS), atualmente na versão 6.

O PS foi desenvolvido para os sistemas operacionais *Windows* e MAC OS. Para a utilização em sistemas *Linux* é necessária a instalação do *software* "*Wine*" para emular a plataforma *Windows* e poder executar o compilador em *Linux*.

Esse SDK tem suporte a sintaxe da linguagem P4, disponibilizado na comunidade P4.org. O compilador se adapta para tratar a linguagem P4 de modo a transformar o código para as especificações de cada NFP. O SDK gera um arquivo no formato ".nffw" que em seguida é encaminhado para a smartNIC.

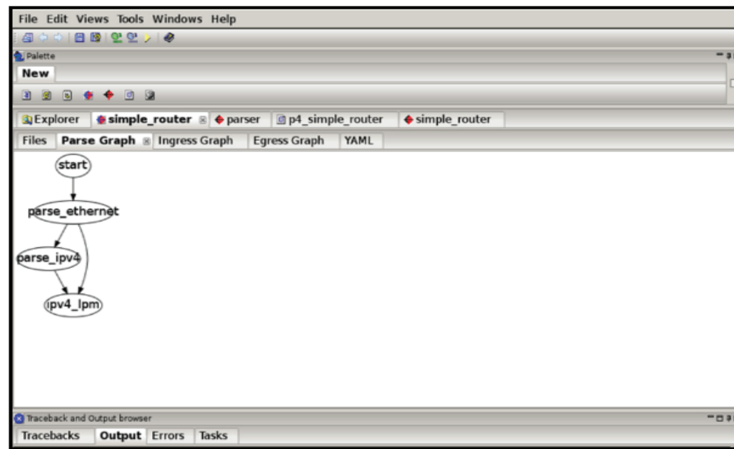


Figura 19. Representação gráfica do analisador. Fonte: Netronome [46].

Durante a compilação do código P4, o compilador utiliza um código intermediário que gera um arquivo com a extensão em “*yaml*”. O arquivo “*yaml*” é analisado e transformado em código final, que após construirá o *firmware* no formato ".nffw".

No final da compilação é gerado um *firmware* usando linguagem C. Esse *firmware* é conduzido para ser armazenado na memória de processamento, conhecido como núcleo de fluxo (*microengines*) da NFP. Cada um destes núcleos pode executar independentemente os pacotes.

A linguagem P4 suporta a inclusão de algoritmos em linguagem C que fornece supervisão da arquitetura e desenvolvimento de novos recursos. Essa função implementada em C é chamada de *sandbox* C. Para a execução dessa função deve ser adicionada uma ação de “*primitive action*” dentro do código P4, conforme demonstrado na Figura 20.

```
action encap_act(prt, tag) {
  filter_func();
  modify_field(standard_metadata.egress_spec, prt);
  xlan_encap(tag);
}
```

```
primitive_action filter_func();
```

Figura 20. Código e função *sandbox* C. Fonte: Netronome [47].

No projeto o programa PS não está no mesmo dispositivo em que as placas da Netronome estão instaladas. A Figura 21 ilustra que o PS está se conectando remotamente utilizando os protocolos IP e TCP para gerenciar o *Run Time Environment*(RTE). O RTE tem a função de receber as informações encaminhadas pelo PS e aplicar sobre a smartNIC. Desta forma, percebe-se que o RTE fica instalado dentro do *host* que possui a placa. O PS está instalado em um *laptop* fora do ambiente do projeto, e realiza a comunicação através da Internet.

Basicamente, o PS tem a função de gerenciar o RTE controlando as operações executadas dentro da placa e também é a interface gráfica que possibilita a programação da

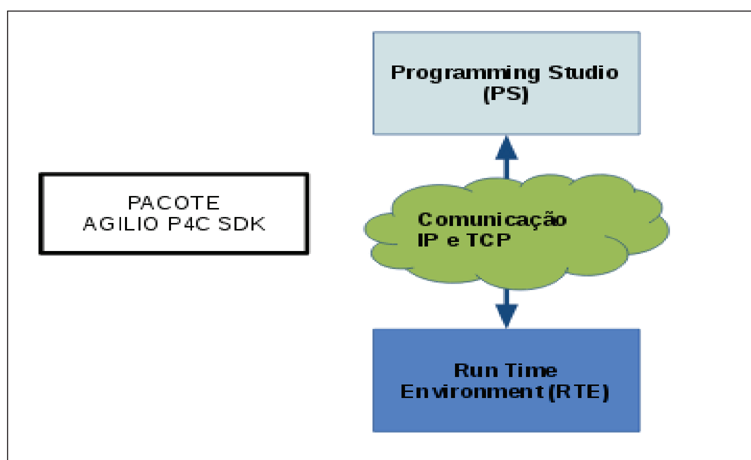


Figura 21. Topologia de comunicação entre o PS com o RTE. Fonte: Autor.

linguagem P4 ou C. O PS também recebe informações do comportamento da placa que são encaminhadas pelo RTE. São informações referentes ao *debugger* e ao comportamento da placa. O *host* RTE também realiza a adição do *firmware* gerado no PS.

3.3 AMBIENTE DE SIMULAÇÃO

O projeto necessitou da instalação de cinco servidores físicos com três smartNICs da Netronome (Figura 22) e após sucedeu-se a instalação do sistema CentOS e o *software* RTE necessário para controlar a placa de rede da Netronome. O sistema operacional CentOS foi escolhido principalmente por ter uma maior compatibilidade com o programa de compilação do *firmware* da Netronome e também por ser considerado sólido, por ser um sistema preparado para funcionar em ambientes corporativos e seguros.

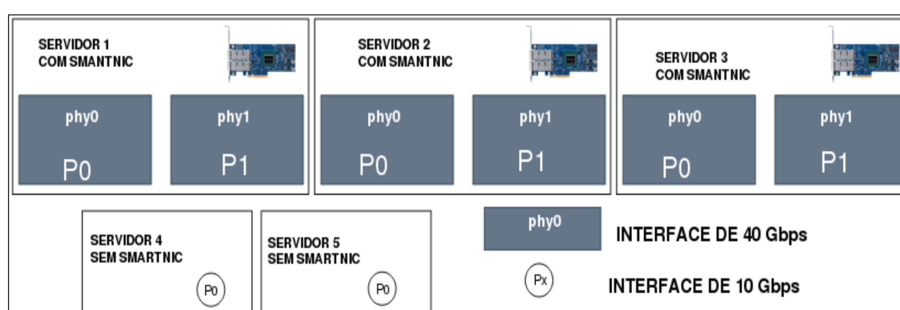


Figura 22. Demostração dos cinco servidores. Fonte: Autor.

O ambiente de experimentação possui dois servidores com smartNICs e que serão descritos como *switch* 1 e *switch* 2. Neles está instalado o sistema operacional CentOS e também o Agilio RTE e seus pacotes requisitados. Esses servidores exercem a função de um alvo de rede que comuta os pacotes conforme as informações dos cabeçalhos.

A Figura 23 está ilustrando dois computadores, que a partir desse momento serão denominados de *switch* 1 e *switch* 2 onde cada um possui uma smartNIC instalada. Essa

placa é composta por duas interfaces de 40 Gbps. Para a montagem da topologia projetada, necessitou a divisão da interface de 40 Gbps em 4 interfaces de 10 Gbps. Para realizar essa divisão da interface foi necessário a configuração do modo *breakout*. Esse modo desempenha a divisão da largura de banda da interface. Esse provisionamento possibilita a utilização de uma cabo *breakout cable* que realiza a separação física do sinal óptico da fibra. Então percebe-se que no *switch* 1, na porta *phy0* (40 Gbps) e *phy1* (40 Gbps) está ativo o modo *breakout*. O modo *breakout* viabilizou a criação de 8 interfaces de 10 Gbps no total (as interfaces de 10 Gbps são: P0,P1,P2,P3,P4,P5,P6,P7).

No *switch* 2 (Figura 23) é possível verificar apenas uma interface de 40 Gbps com a segmentação de portas ativas. A segunda interface de 40 Gbps terá uma interligação com outro *host* que não vai possuir a configuração do modo *breakout* habilitado, porque o *host* não permite o modo *breakout* ativo.

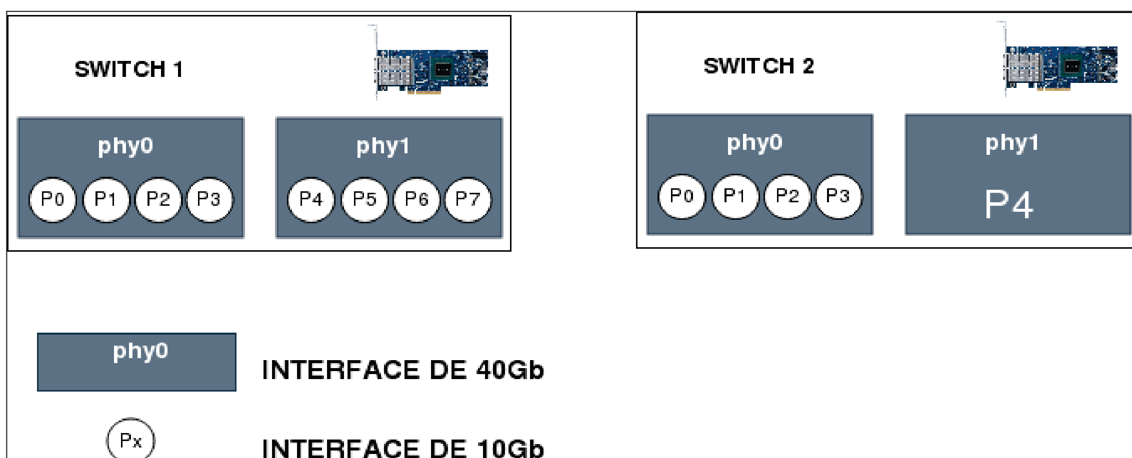


Figura 23. Switch 1 e switch 2. Fonte: Autor.

Nesta etapa, ocorreu a instalação do Agilio P4 SDK, que demandou a execução do processo de *downgrade*[21]. Após a realização do procedimento de *downgrade* do sistema operacional, foi possível a instalação do *software* para placa da Netronome. Também foi habilitado na BIOS do computador o SR-IOV que permite a virtualização das entradas e saídas de interfaces, sendo fundamental para a execução do *software* Agilio que necessita de pelo menos 4 interfaces virtuais com comunicação para o *host* hospedeiro. No presente projeto foi habilitado o SR-IOV apenas pela obrigatoriedade de instalação do *software*, pois não haverá tráfego de pacotes nas interfaces virtuais.

A Figura 24 ilustra a ligação física dos dispositivos. Observa-se que o *switch* 1 na porta *P0* possui uma interligação com o *switch* 2 pela interface *p0*. Esse enlace tem uma velocidade de 10 Gbps. Também possui dois equipamentos ligados nos *switches*, os equipamentos denominados *host* 1 e *host* 2, que estão conectados ao *switch* 1 e *switch* 2

[21]Necessitou a realização do *downgrade* do *kernel* e do *release* do CentOS, pois atualmente o CentOS está com a versão estável 7.6 e a aplicação da Agilio requisita a versão 7.5 com o *kernel* do *Linux* na versão 3.10.0-862.

respectivamente, por uma interface de 10 Gbps. Estes *hosts* têm a função de gerar pacotes e tráfego para avaliar o funcionamento do ambiente.

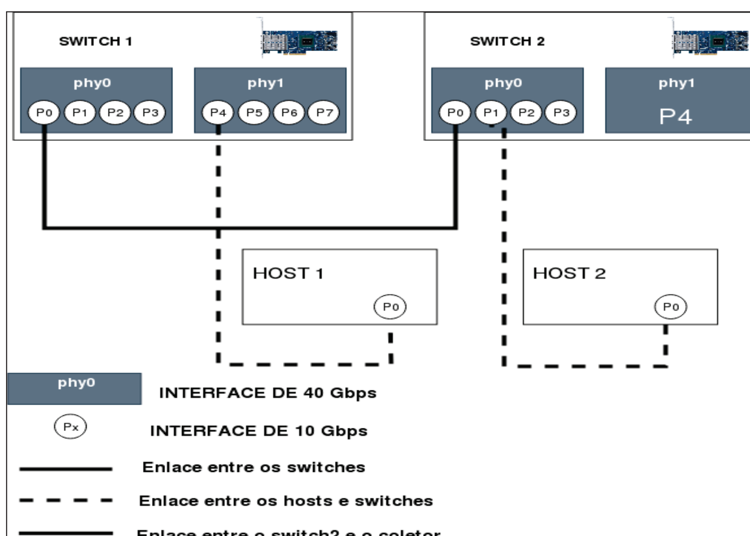


Figura 24. Topologia *switch* 1 e *switch* 2. Fonte: Autor.

Como pode ser percebido na Figura 25, identifica-se a topologia completa que é utilizada no experimento. Nela é possível identificar que a segunda porta (*P4*) de 40 Gbps do *switch* 2 está conectada diretamente na outra NIC na interface *P1* em um equipamento denominado de coletor. Essa interface vai encaminhar o INT *Report* para o coletor. A interface *P0* do coletor não terá cabo conectado. Esses dados encaminhados por essa interface vão ter as informações coletadas dos *switches*, e no coletor ocorrerá o processamento desses dados e o armazenamento.

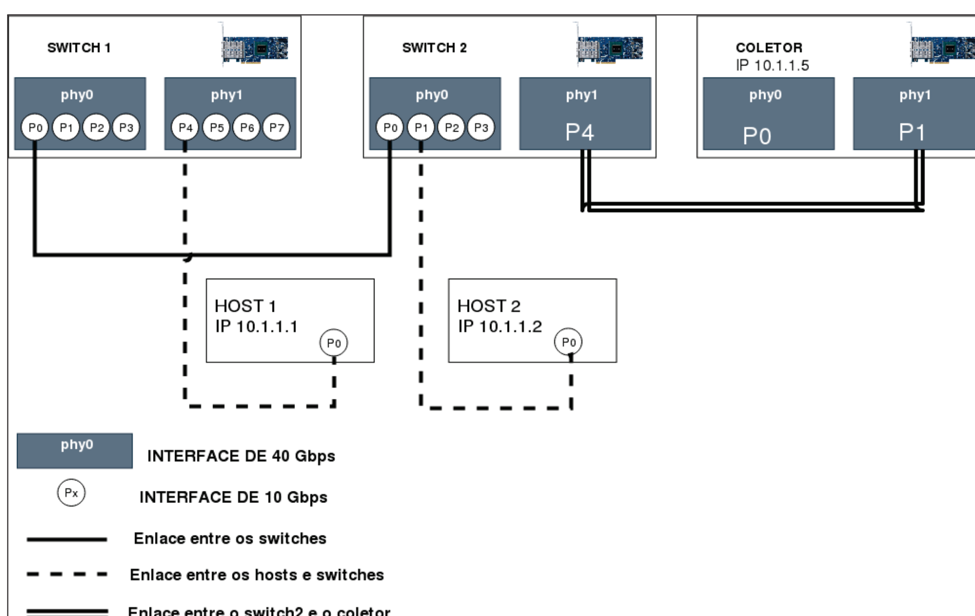


Figura 25. Topologia completa. Fonte: Autor.

O projeto de desenvolvimento de um sistema coletor de dados INT e com processamento desses dados está sendo desenvolvido em outra dissertação. Ainda na Figura 25

estão representados os IPs que estão configurados em cada computador. Nota-se que apenas as máquinas *host* 1, *host* 2 e o coletor possuem IPv4 aplicado em suas interfaces. Como os pacotes de dados não terão acesso aos servidores que hospedam as placas da Netronome, os *switches* foram projetados para não possuir IPv4 configurado.

3.3.1 Ambiente Computacional

Nesta seção serão apresentados detalhes dos *hardwares* que foram utilizados neste projeto. Essas configurações estão na Tabela 2, e percebe-se que os computadores têm uma alta capacidade de processamento e foram projetados para suportar uma grande quantidade de tráfego simultâneo.

Tabela 2. Tabela *hardware* dos servidores

Dispositivo	Processador	RAM	Placa Mãe	Placa de rede	SO
Switch 1	AMD EPYC 7451	128 G	H11SSL-C	Agilio CX 40Gbps	CentOS 7.5
Switch 2	AMD EPYC 7451	128 G	H11SSL-C	Agilio CX 40Gbps	CentOS 7.5
host 1	AMD Opteron 6128	32 G	H8SGL	Myricom 10G	CentOS 7.6
host 2	AMD Opteron 6128	32 G	H8SGL	Myricom 10G	CentOS 7.6
Coletor	AMD EPYC 7451	128 G	H11SSL-C	Agilio CX 40 Gbps	Ubuntu 18.04

Em conformidade com a topologia apresentada na Seção 3.3 que foi preparada para executar experimentos de uma rede e também com as informações detalhadas nesta seção, repara-se que os computadores estão preparados para receber o código P4 e na próxima Seção 2.5, apresenta-se a versão mais simples do código.

3.4 ALGORITMO DE ENCAMINHAMENTO DE PACOTES IP/TCP E IP/UDP

De acordo com o que foi apresentado na Seção 2.5, a linguagem de programação P4 possibilita programar o comportamento do plano de dados de um dispositivo alvo. No presente projeto os dispositivos alvos estão denominados como *switch* 1 e *switch* 2. Esta seção tem como objetivo descrever o algoritmo que foi desenvolvido para possibilitar o tráfego IPv4 por dentro dos *switches*.

O *software* desenvolvido em linguagem P4 para esses *switches* seguem uma ordem de operações para que o pacote seja processado no plano de dados da placa da Netronome. A Figura 26[22] demonstra a ordem de operações que devem ser seguidas pelo programador. Neste caso, deve iniciar com o analisador dos pacotes (*parser*), depois segue para ser verificado o *checksum*, após inicia o processamento de entrada (*ingress*), em

[22] Padrão é o caminho padrão para um pacote com INT seguir no algoritmo. Alternativo é o caminho alternativo possibilita a passagem de outros *headers* ou opções dentro do algoritmo. Falso é quando o pacote não é validado pelas outras opções.

seguida ocorre o processamento de saída (*egress*). A penúltima função é realizar a atualização do *checksum* e por último montar os cabeçalhos novamente no pacote(*deparser*) e encaminhar para sua porta de destino. É muito importante seguir está ordem, pois caso a ordem seja alterada o *switch* não irá funcionar.

Por exemplo, se for realizado antes o *ingress* e subsequente o *parser*, quando for realizado o processo de *ingress* o *switch* não terá os dados dos cabeçalhos do pacote, então o pacote será descartado. Caso ocorrer a definição incorreta da ordem das funções no programa P4, o próprio PS já informa para o usuário que esse tipo de código está errado e deve ser ajustado.

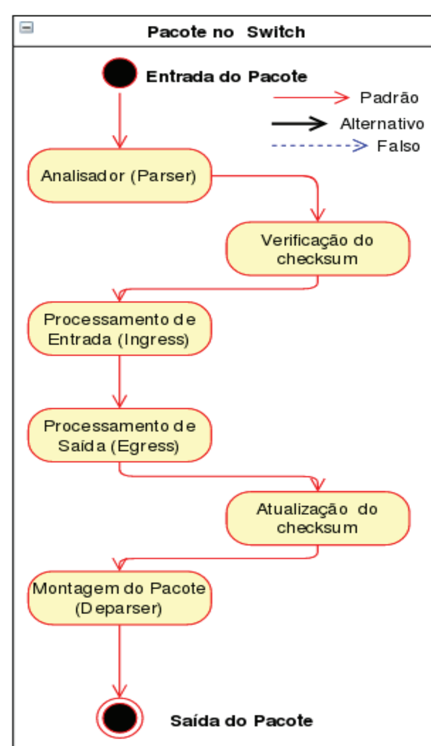


Figura 26. Ordem de execução das funções do código P4. Fonte: Autor.

A primeira função a ser executada em um programa P4 é a *parser*, que também é chamada de analisador. Nesta etapa são extraídos os dados dos cabeçalhos dos pacotes para as estruturas de *headers* que vão armazenar essas informações enquanto o pacote passa pelo *switch*. Para o desenvolvimento desse algoritmo, foi necessária a criação de uma estrutura que extrai as informações dos cabeçalhos *Ethernet*, IPv4, ARP, TCP e UDP.

Como o analisador é uma máquina de estados, foi implementada uma estrutura que inicia quando o pacote entra no *parser* e depois entra no estado de *start* e é encaminhado para a extração do *header ethernet* e depois avaliação se é ARP (0x806) ou IPV4 (0x800), se for ARP extrai os dados e verifica o próximo cabeçalho se é IPV4, se for, é destinado para o estado de IPV4. Quando o pacote vai no estado de IPV4 é extraído os dados e depois verifica o protocolo de comunicação se é TCP (0x06) ou UDP (0x11), extrai-se os dados de acordo com o protocolo e depois é destinado para a próxima etapa. Caso os pa-

cotes tiverem de outros cabeçalhos eles são destinados direto para aproxima etapa, exceto caso o pacote não tiver *header ethernet*, neste caso, o pacote é descartado. A Figura 27[23] ilustra as etapas e a sequência de extração de dados que é executada até o pacote ser destinado para o *ingress*.

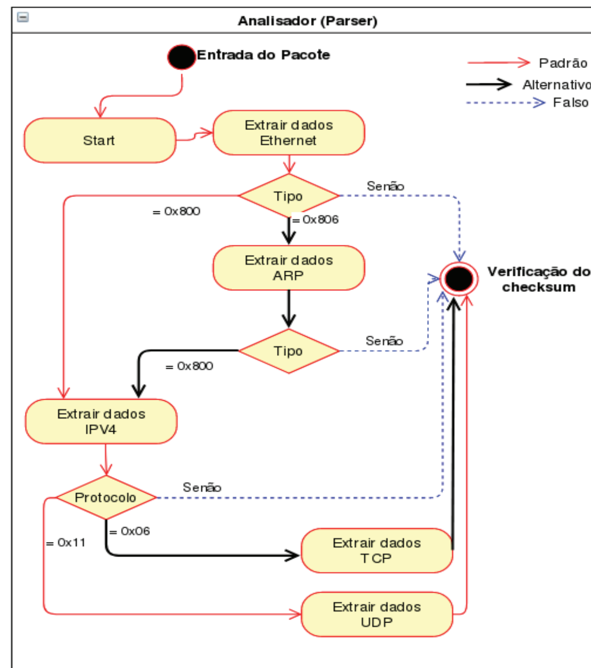


Figura 27. Função *parser* do código IP/TCP e IP/UDP. Fonte: Autor.

Pode-se notar que no *parser* foi adicionado o *header* ARP, que possibilita o tráfego de pacotes ARP por dentro dos *switches*. Esse cabeçalho é importante, pois a primeira comunicação de um *host* com a rede é através da ARP. Por exemplo, quando é utilizado o comando *ping* em um computador, o primeiro pacote que é encaminhado pelo comando *ping* é o pacote ARP em modo *broadcast*[24], com o intuito de buscar o MAC do equipamento de destino, para atualizar a tabela ARP do *host* de origem.

No *ingress* são executadas as operações para quando o pacote está entrando no *switch*. No projeto o pacote é recebido do *parser*, então o *software* P4 vai verificar se o cabeçalho IPv4 está presente no pacote (válido), se estiver, vai encaminhar os dados do pacote para o controle e depois espera até que o plano de controle processe as informações recebidas e retorna as ações que devem ser executadas pelo *ingress* e os dados necessários para essas ações. Após o pacote pode seguir para a próxima etapa. Porém, se o *header* não for IPv4 válido, seguirá para a próxima etapa e avalia se o cabeçalho ARP está dentro do pacote, se estiver presente esse *header* então vai encaminhar os dados do ARP para o controle e aguardar o retorno das ações e das informações do controle. Porém,

[23]Padrão é o caminho padrão para um pacote com INT seguir no algoritmo. Alternativo é o caminho alternativo possibilita a passagem de outros *headers* ou opções dentro do algoritmo. Falso é quando o pacote não é validado pelas outras opções.
[24]*Broadcast* é a transmissão de um pacote, que todos os *host* de uma rede local vão receber.

se não tiver um *header* IPv4 ou ARP válido dentro do pacote, o *switch* vai descartar esse pacote dentro do *ingress*. Todas estas etapas encontram-se ilustradas na Figura 28[25].

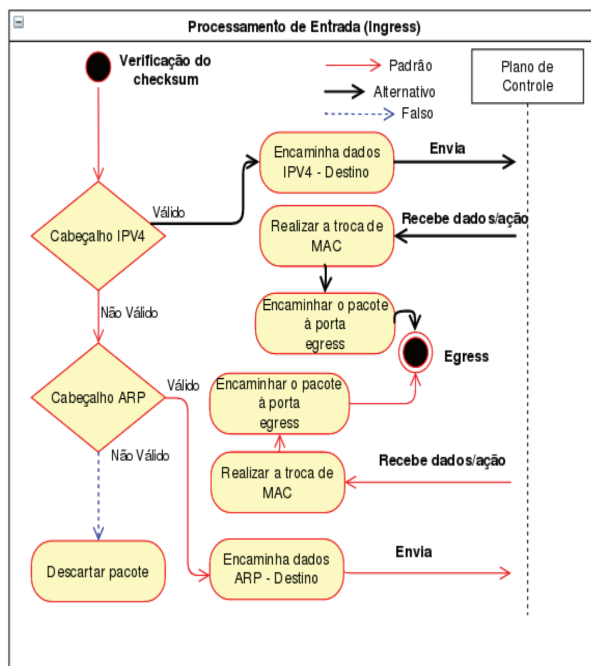


Figura 28. Função *ingress* do código IP/TCP e IP/UDP. Fonte: Autor.

Nesta seção foi descrito o funcionamento do plano de dados dos *switches*. Como relatado, algumas informações estão sendo tratadas sobre o plano de controle desses *switches*, então na próxima seção serão explanadas as atividades executadas no controle.

3.4.1 Plano de controle do processo TCP e UDP

Conforme descrito na Seção 2.1, as decisões são tomadas no plano de controle do equipamento e o plano de dados apenas executa as ações no pacote. As instruções do plano de controle das placas da Netronome estão presentes no arquivo chamado "*User Config File*"que gerencia o programa P4 presente no *firmware* da placa da Netronome. O arquivo "*User Config File*" contém as configurações passadas pelo administrador de rede para que o *switch* possa controlar as modificações realizadas nos pacotes que estão passando através dele, ou seja, são as configurações lógicas do *switch*.

Para a gestão das duas NICs foram desenvolvidas regras para o plano de controle. Essas regras foram adicionadas em dois arquivos diferentes, um arquivo para cada *switch*. Esses arquivos contêm as regras e dados para os dois possíveis *headers* (IPv4 e ARP) que foram encaminhados pelo *ingress*. O controle vai receber a informação de endereço de

[25]Padrão é o caminho padrão para um pacote com INT seguir no algoritmo. Alternativo é o caminho alternativo possibilita a passagem de outros *headers* ou opções dentro do algoritmo. Falso é quando o pacote não é validado pelas outras opções.

destino do *header* IPv4 ou ARP. A seguir verifica em uma tabela de rotas qual destino seguir e quais alterações no cabeçalho *Ethernet* devem ser realizadas. A Figura 29 apresenta o controle do *switch* 1 e na Figura 30 o controle do *switch* 2, sendo que nestas tabelas são apresentadas as informações correspondentes aos dados encaminhados pelo plano de dados e também as ações e informações destinadas para o *ingress* executar.

SWITCH 1			
Operação Controle	Dados do Ingress	Correspondência	Ação
IPV4	Endereço de destino IPV4	10.1.1.1/32	to_host1
		10.1.1.2/32	to_host2
		10.1.1.5/32	to_coletor
Operação Controle	Dados do Ingress	Correspondência	Ação
ARP	Endereço de destino ARP	10.1.1.1/32	to_host1
		10.1.1.2/32	to_host2
		10.1.1.5/32	to_coletor
Ações	Informações encaminhadas para o Ingress		
to_host1	dst=00:60:dd:45:47:ff, src=00:15:4d:13:20:d2, porta=p4		
to_host2	dst=00:15:4d:13:16:0c, src=00:15:4d:13:20:ce, porta=p0		
to_coletor	dst=00:15:4d:13:16:0c, src=00:15:4d:13:20:ce, porta=p0		

Figura 29. Plano de controle do *switch* 1. Fonte: Autor.

SWITCH 2			
Operação Controle	Dado Ingress	Correspondência	Ação
IPV4	Endereço de destino IPV4	10.1.1.1	to_host1
		10.1.1.2	to_host2
		10.1.1.5	to_coletor
Operação Controle	Dado Ingress	Correspondência	Ação
ARP	Endereço de destino ARP	10.1.1.1	to_host1
		10.1.1.2	to_host2
		10.1.1.5	to_coletor
Ações	Informações encaminhadas para o Ingress		
to_host1	dst=00:15:4d:13:20:ce, src=00:15:4d:13:16:0c, porta=p0		
to_host2	dst=00:60:dd:45:47:f6, src=00:15:4d:13:16:0d, porta=p1		
to_coletor	dst=00:15:4d:13:0e:b7, src=00:15:4d:13:16:10, porta=p4		

Figura 30. Plano de controle do *switch* 2. Fonte: Autor.

O plano de controle do *switch* vai aumentando a quantidade de regras em sua tabela de acordo com o surgimento de novas requisições feitas pelo plano de dados.

3.5 IMPLEMENTAÇÃO DO INT NO *PARSER SWITCH* 1 E *SWITCH* 2

Nesta seção será apresentado o funcionamento do *parser* com a adição dos cabeçalhos INT nas estruturas de *headers*. Porém, antes de iniciar o *parser*, serão detalhados os *headers* criados para conduzir as informações dos pacotes INT.

A Figura 31 ilustra os nove cabeçalhos INT criados sobre o código P4 para realizar o transporte dos dados INT através dos *switches* da rede MAN. Na lista abaixo da Figura 31, explica-se o funcionamento de cada um dos *headers*:

INTshim_t	
Bits	*Campos*
8	shim_type
8	shim_reserved1
8	tail_port
8	shim_rsvd2

telemetryreport_t	
Bits	*Campos*
8	f_versio
8	f_next_proto
1	f_drop
1	f_queue
1	f_flow
5	f_rsvd
8	f_hw_id
32	f_seq_num
64	f_ingress_ts

INThopByhopHeader_t	
Bits	*Campos*
8	int_version
8	int_replication
1	int_copy
1	int_exceeded
8	int_rsvd_1
8	int_ins_cnt
8	int_max_hops
8	int_total_hops
8	int_instruction_bit
6	int_rsvd_instructions

INTtail_t	
Bits	*Campos*
32	tail_header
8	tail_proto
16	tail_port
8	tail_dscp

int_switch_id_t	
Bits	*Campos*
32	int_switch_id

int_ingress_egress_ports_t	
Bits	*Campos*
16	int_ingress_id
16	int_egress_id

int_hop_latency_t	
Bits	*Campos*
64	int_hop_latency

int_ingressTimestamp_t	
Bits	*Campos*
64	int_ingressTimestamp

int_egressTimestamp_t	
Bits	*Campos*
64	int_egressTimestamp

Figura 31. *Headers* para o INT. Fonte: Autor.

- Cabeçalho *INTshim_t*: identifica o início dos cabeçalhos INT. Neste cabeçalho são adicionados as informações da versão e também da porta lógica utilizada;
- Cabeçalho *INThopByhopHeader_t*: tem a função de definir limites para o pacote INT. Nele é armazenado a quantidade de saltos que o pacote trafegou, quantidade máxima de saltos que esse pacote pode coletar, versão do INT e principalmente definir quais dados que INT deve coletar dos *switches*;
- Cabeçalho *INTtail_t*: define o final dos cabeçalhos INT. Nele estão as informações do cabeçalho original da rede do pacote sem o INT;
- Cabeçalho *telemetryreport_t*: esse cabeçalho é utilizado apenas no último *switch*. Ele define o início de um pacote que será reportado para o coletor;

- Cabeçalho *int_switch_id_t* : armazena a informação do ID do *switch*;
- Cabeçalho *int_ingress_egress_ports_t*: armazena a informação do ID das interfaces de entrada e saída do *switch*;
- Cabeçalho *int_hop_latency_t*: armazena a informação de quanto tempo (nanossegundo) o pacote ficou no *switch*;
- Cabeçalho *int_ingressTimestamp_t*: armazena a informação do tempo que o pacote entrou no *ingress*;
- Cabeçalho *int_egressTimestamp_t*: armazena a informação do tempo que o pacote saiu do *egress*.

Os *parsers* do *switch* 1 e do *switch* 2 foram projetados para funcionar da mesma maneira. Foi utilizado o *parser* implementado anteriormente na Seção 3.4 e apresentado na Figura 27. Para que o *switch* tenha as informações dos *headers* INT, foram adicionados os cabeçalhos *INTshim_t* e *INThopByhopHeader_t* no *parser*. Pois, com esses *headers* o *software* P4 tem as informações necessárias para coletar os dados do *switch* 1 e *switch* 2.

A Figura 32[26] apresenta um diagrama que representa o *parser* com o INT implementado. O *parser* recebe o pacote no *start* e destina para a extração do *ethernet* e depois avalia o tipo de pacote, se nele contém *header* ARP ou IPV4. Se tiver ARP extrai os dados da ARP e destina para verificar se é um IPV4. Se o pacote chegar no estado de IPV4 é extraído o IPV4. Se o pacote for outro cabeçalho é destinado para a próxima etapa. Depois do IPV4 avalia-se qual é tipo de protocolo de transporte tem nesse pacote. Se é cabeçalho TCP ou UDP, sabendo qual o é protocolo, vai ser extraído os dados do protocolo correspondente. Percebe-se que após realizar a extração dos dados de TCP ou de UDP, o algoritmo verifica se o campo *typeServiceDiffServ*[27] do cabeçalho IPv4 possui o valor em hexadecimal 0x17. Se contiver essa informação, o algoritmo vai realizar a extração dos cabeçalhos *INTshim_t* e *INThopByhopHeader_t*, e somente depois ele vai seguir para a verificação do *checksum*. Caso o valor do campo *typeServiceDiffServ* não for 0x17 o pacote é considerado parte de um tráfego normal de rede com *header* UDP ou TCP. Neste caso o pacote é encaminhado diretamente para a verificação do *checksum*.

Como percebido na explicação acima, o *parser* não analisa os *headers* dos dados coletados de dentro dos *switches* e nem do cabeçalho *INTTail_t*, desta maneira a memória do *switch* é utilizada apenas para armazenar informações necessárias para que sejam utilizados as funções do *framework* INT.

[26] Padrão é o caminho padrão para um pacote com INT seguir no algoritmo. Alternativo é o caminho alternativo possibilita a passagem de outros *headers* ou opções dentro do algoritmo. Falso é quando o pacote não é validado pelas outras opções.

[27] *typeServiceDiffServ* destina-se a permitir discriminação de serviço escalável na Internet. RFC2474 [48].

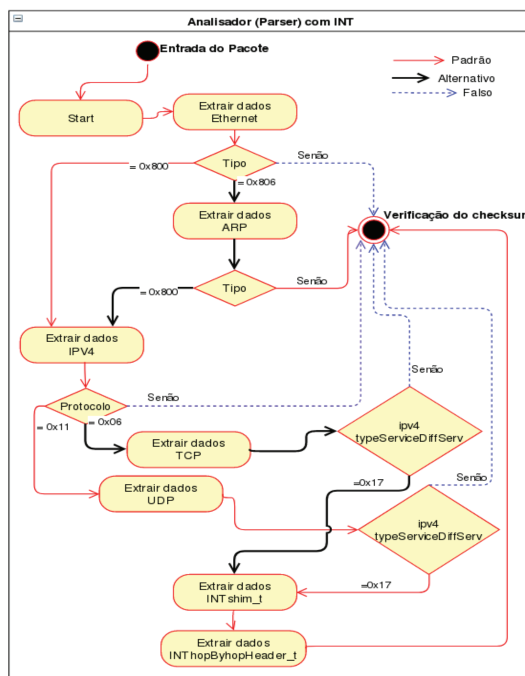


Figura 32. *Parser* com INT. Fonte: Autor.

Todo o pacote de dados que incluir o valor 0x17 dentro do campo *typeServiceDiffServ* é considerado um pacote INT e com isso vai executar todas as operações destinadas ao INT. O pacote deve sair de uma *probe*, sendo que nesta implementação esse pacote é gerado dentro do *host* 1. A seguir é destinado a ir até o coletor. Este pacote vai atuar como se fosse uma "sonda", pois passará em cada *switch* coletando os dados sobre o comportamento dos equipamentos para avaliar o fluxo de dados.

3.6 IMPLEMENTAÇÃO DO INT NO *INGRESS SWITCH* 1 E *SWITCH* 2

Nesta seção será apresentada a implementação do processamento de entrada do pacote (*ingress*). Nesta etapa é implementada a função de encaminhamento de pacote e também de adições ou alterações dos *headers*. Na implementação realizada para esse projeto foram criados dois *ingress*, um para ser executado no *switch* 1, que executa a formação dos cabeçalhos INT no pacote e o outro para o *switch* 2, onde é elaborado o *header report* e a reorganização do pacote, para ser encaminhado para o coletor. A Figura 33[28] ilustra as operações executadas no *switch* 1 na etapa do *ingress*.

Primeira avaliação, verifica se pacote tem *header* ARP, se tiver ARP é realizado a interação com o plano de controle para buscar as ações e dados para encaminhar um pacote ARP. Caso não for ARP o pacote é avaliado para verificar se o pacote não tem

[28]Padrão é o caminho padrão para um pacote com INT seguir no algoritmo. Alternativo é o caminho alternativo possibilita a passagem de outros *headers* ou opções dentro do algoritmo. Falso é quando o pacote não é validado pelas outras opções.

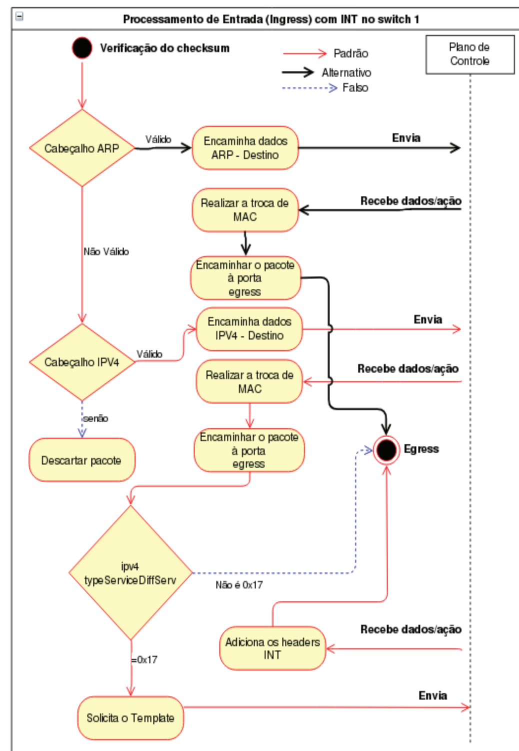


Figura 33. *Ingress* com INT do *switch* 1. Fonte: Autor.

header IPV4, se não tiver IPV4 ou ARP o pacote é descartado. Se for IPV4 é executado as ações e operações do IPV4, conforme definido no plano de controle.

A única diferença destas operações do *ingress* TCP ou UDP com o INT em relação a anterior do *ingress* TCP ou UDP sem o INT, foi que após a avaliação do IPv4, o algoritmo analisa o campo *typeServiceDiffServ*. Se este campo possuir o valor 0x17, esse valor representa o serviço INT ativo. Neste caso, é solicitado o valor do *template* para o plano de controle. Depois que o plano de controle retornar o valor do *template* é criado os cabeçalhos *INTshim_t* e *INThopByhopHeader_t* e é adicionado o valor do *template* (Seção 3.4) no campo *int_instruction_bit*, que representa os valores que serão coletados pelo INT. O *switch* 1 vai receber o nome de INT *source*, pois é nesse dispositivo que o pacote INT é criado e anexado nos cabeçalhos dos pacotes.

Salienta-se que a implementação realizada nessa dissertação está trabalhando com o tráfego com o campo *typeServiceDiffServ* já editado com o valor 0x17. Caso for necessário trabalhar com o tráfego sem marcação inicialmente, o INT *source* (*switch* 1) deverá possuir regras para identificar qual o tráfego deve ser monitorado pelo INT. Essas regras podem ser criadas de acordo com a informação de IP ou da porta de destino do pacote. Desta forma, toda vez que um pacote tiver a informação de um determinado destino, será aplicado o monitoramento através do INT.

A Figura 34[29] demonstra as tarefas que são destinadas para o *switch* 2. Nessa etapa percebe-se que novamente o algoritmo valida o ARP e depois valida o IPv4 se está presente no pacote, após executa as funções destinadas para esses dois *headers*. Após isso verifica-se o pacote é um INT. Se for, busca as informações no plano de controle, informações estas que são o IP, porta lógica e MAC para que seja criado um pacote *report*.

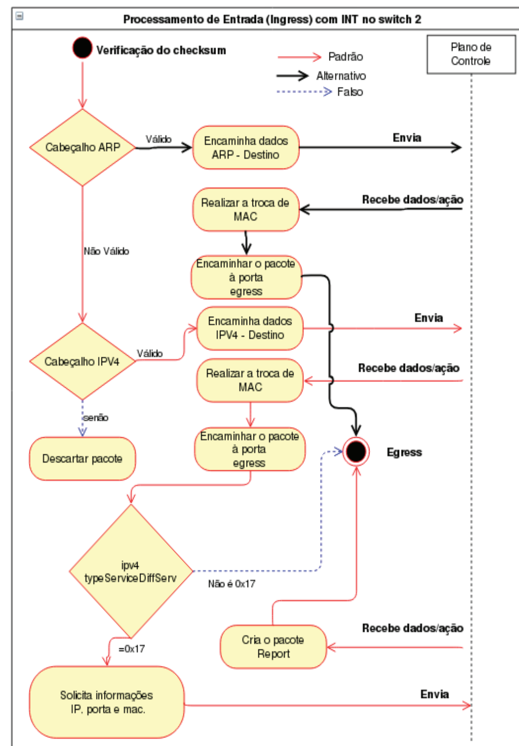


Figura 34. *Ingress* com INT do *switch* 2. Fonte: Autor.

O pacote transmitido entre o *switch* 1 e o *switch* 2 é denominado de *transit* INT. A Figura 35 ilustra a organização dos cabeçalhos desse pacote. Percebe-se que, quando aparece a informação INT *data*, irá representar os seguintes cabeçalhos, que realiza a coleta INT de dados de um *switch*: *int_switch_id_t* , *int_ingress_egress_ports_t*, *int_hop_latency_t*, *int_ingressTimestamp_t* e *int_egressTimestamp_t*.

ethernet_t -> ipv4_t -> udp_t -> INTshim_t -> INThopByhopHeader_t -> INT data switch 1 ->INTtail_t

Figura 35. Organização do pacote de *transit* INT. Fonte: Autor.

O pacote *report* tem uma organização única, para que o coletor consiga interpretar o pacote e coletar os dados. A organização do pacote para reportar ao coletor está organizada conforme sequência apresentada na Figura 36, sendo este pacote denominado de *report* INT.

[29]Padrão é o caminho padrão para um pacote com INT seguir no algoritmo. Alternativo é o caminho alternativo possibilita a passagem de outros *headers* ou opções dentro do algoritmo. Falso é quando o pacote não é validado pelas outras opções.

ethernet_t -> ipv4_t -> udp_t -> telemetryreport_t -> ethernet_t -> ipv4_t -> udp_t -> INT data switch 2 -> INT data switch 1 ->INTtail_t

Figura 36. Organização do pacote de *report* INT para o coletor. Fonte: Autor.

O Anexo A e B apresentam o código do *ingress* no *switch* 1 e no *switch* 2, respectivamente. A Seção 3.6.1 irá demostrar os possíveis *templates* que foram desenvolvidos e que o administrador pode escolher e aplicar no programa P4 para esse projeto.

3.6.1 *Templates* de coleta de valores INT

Para deixar o trabalho mais dinâmico, foi implementado um sistema de *template*, em que o usuário pode escolher quais informações dos *switches* ele quer coletar. Atualmente foram implementados oito tipos de *templates*, listados na Tabela 3. Os *templates* representam os bits ativos no campo *int_instruction_bit* dentro do *header INThopByhopHeader_t*. Dessa forma cada bit ativo representa um cabeçalho de dados que será coletado no *switch* no qual o pacote INT percorre. Percebe-se na Tabela 3 que ainda possui três *bits* que ainda não foram ativos. Esses *bits* podem ser ativos quando tiver mais metadados para coletar dados dos *switches*.

Tabela 3. *Templates* para coleta de dados

Valor	bits	Informações coletadas ou *Headers* INT
128	10000000	*switch* ID
192	11000000	*switch* ID e interfaces IDs
224	11100000	*switch* ID, interfaces IDs e *hop latency*
240	11110000	*switch* ID, interfaces IDs, *hop latency* e *TimeStamp ingress*
248	11111000	*switch* ID, interfaces IDs, *hop latency*, *TimeStamp ingress* e do *egress*
160	10100000	*switch* ID e *hop latency*
184	10111000	*switch* ID, *hop latency*, *TimeStamp ingress* e do *egress*
88	01011000	interfaces IDs, *TimeStamp ingress* e do *egress*

Sabendo quais são as possibilidades de coletar informações de *status* no *framework* INT que foi implementado para esse projeto, a próxima seção apresenta como os dados são coletados e organizados no *egress*.

3.7 IMPLEMENTAÇÃO DO INT NO *EGRESS SWITCH* 1 E *SWITCH* 2

Foi implementado o processamento de saída (*egress*) para exportar as informações coletadas nos *switches*. Neste projeto o *egress* busca capturar as informações de acordo com o valor do *template* que está na Tabela 3. Essas informações são processadas e organizadas dentro de um pacote INT, para serem transmitidas para outro *switch* ou serem encaminhadas para o coletor de dados.

A Figura 37[30] apresenta o diagrama do *egress*. Novamente quando os dados do pacote chegam no *egress*, verifica se é um pacote INT, analisando a informação do cabeçalho IPv4. Se for pacote INT o algoritmo vai encaminhar o valor do campo *int_instruction_bit* para o plano de controle, que vai retornar qual a determinada ação será executada. Essa ação irá realizar a coleta dos dados do *switch*. Caso o pacote não for considerado parte de um fluxo INT esse pacote não entra nas regras de coleta de dados e o pacote é encaminhado para a interface de saída.

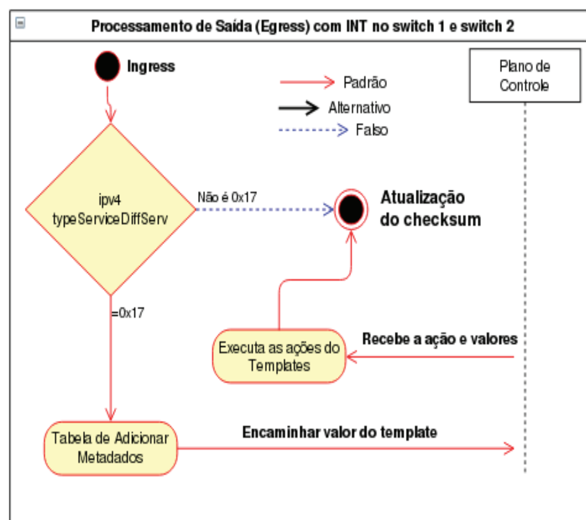


Figura 37. *Egress* com INT no *switch* 1 ou *switch* 2. Fonte: Autor.

O Anexo C possui o código P4 implementado no *egress* do projeto para essa dissertação.

3.7.1 Plano de controle para o *Ingress* e *Egress* com INT

No plano de controle também foram adicionados as informações e ações para adicionar os *headers* INT no *switch* 1, que está representado na Figura 38(a). Nota-se, também, que as informações de ações para coletar dados do *switch* 1 foram inseridos na tabela do plano de controle. No *switch* 2 que está ilustrado na Figura 38(b) adicionou-se dados e ações para criar o pacote de *report* para destinar ao coletor de dados. É Importante perceber que o plano de controle do *switch* 1 define qual é o *template* utilizado para coletar os dados e o plano do *switch* 2 define como os dados vão ser reportados para o coletor.

Após a implementação do algoritmo para a coleta e organizar os dados do *framework* INT, deve-se ajustar o relógio das placas da Netronome. Na próxima seção será aplicado uma configuração para manter o *clock* da smartNIC ajustado.

[30] Padrão é o caminho padrão para um pacote com INT seguir no algoritmo. Alternativo é o caminho alternativo possibilita a passagem de outros *headers* ou opções dentro do algoritmo. Falso é quando o pacote não é validado pelas outras opções.

SWITCH 1 com INT			
Operação Controle	Dados do Ingress	Correspondência	Ação
IPV4	Endereço de destino IPV4	10.1.1.1/32	to_host1
		10.1.1.2/32	to_host2
		10.1.1.5/32	to_coletor
Operação Controle	Dados do Ingress	Correspondência	Ação
ARP	Endereço de destino ARP	10.1.1.1/32	to_host1
		10.1.1.2/32	to_host2
		10.1.1.5/32	to_coletor
Operação Controle	Dados do Ingress	Correspondência	Ação
INT Source	default	default	add_source_INT
Operação Controle	Dados do Egress	Correspondência	Ação
add_metadata	instruction	128	action_eq128
	instruction	192	action_eq192
	instruction	224	action_eq224
	instruction	240	action_eq240
	instruction	248	action_eq248
	instruction	160	action_eq160
	instruction	184	action_eq184
	instruction	88	action_eq88
Ações	Informações encaminhadas para o Ingress		
to_host1	dst=00:60:dd:45:47:ff, src=00:15:4d:13:20:d2, porta=p4		
to_host2	dst=00:15:4d:13:16:0c, src=00:15:4d:13:20:ce, porta=p0		
to_coletor	dst=00:15:4d:13:16:0c, src=00:15:4d:13:20:ce, porta=p0		
add_source_INT	instruction=248		

(a) Plano de Controle com INT *switch* 1

SWITCH 2 com INT			
Operação Controle	Dado Ingress	Correspondência	Ação
IPV4	Endereço de destino IPV4	10.1.1.1	to_host1
		10.1.1.2	to_host2
		10.1.1.5	to_coletor
Operação Controle	Dado Ingress	Correspondência	Ação
ARP	Endereço de destino ARP	10.1.1.1	to_host1
		10.1.1.2	to_host2
		10.1.1.5	to_coletor
Operação Controle	Dado Ingress	Correspondência	Ação
int_report	default	default	report
Operação Controle	Dados do Egress	Correspondência	Ação
add_metadata	instruction	128	action_eq128
	instruction	192	action_eq192
	instruction	224	action_eq224
	instruction	240	action_eq240
	instruction	248	action_eq248
	instruction	160	action_eq160
	instruction	184	action_eq184
	instruction	88	action_eq88
Ações	Informações encaminhadas para o Ingress		
to_host1	dst=00:15:4d:13:20:ce, src=00:15:4d:13:16:0c, porta=p0		
to_host2	dst=00:60:dd:45:47:f6, src=00:15:4d:13:16:0d, porta=p1		
to_coletor	dst=00:15:4d:13:0e:b7, src=00:15:4d:13:16:10, porta=p4		
report	port=32766,mac=00:15:4d:13:0e:b7, ip=10.1.1.5		

(b) Plano de Controle com INT *switch* 2

Figura 38. Plano de controle com INT

3.8 IMPLEMENTAÇÃO DO *TIMESTAMP*

Durante a coleta de dados do *TimeStamp*, percebeu-se que o tempo era reiniciado para um valor padrão toda vez que a placa da Netronome era reiniciada. A placa possui um mecanismo de contagem de nanossegundos, que fica armazenado nos registros da placa. Desta forma, percebe-se que o *TimeStamp* fica armazenado dentro da memória da placa da Netronome. Para conseguir alterar o registro de *clock* foi utilizada uma ferramenta chamada de "NFP-REG". Essa ferramenta está presente dentro do RTE e tem a função de alterar e visualizar os valores dos registros da smartNIC, sendo possível alterar os valores de tempo.

Então foi criado um *script* para sincronizar o *TimeStamp* do computador hospedeiro com a placa da Netonome. Portanto, é coletado o tempo dos computadores em hexadecimal e adicionado ao *script* que transfere o valor coletado diretamente para a placa, utilizando uma sequência de comados para acessar o registro e a incrementação do *TimeStamp* da SmartNIC. Na Figura 39 são apresentados os comandos que foram utilizados para realizar esse procedimento.

O *script* é executado em intervalos de trinta minutos. Nesse *script* é realizado a sincronização do NTP entre os *hosts* hospedeiro das placas e depois é executado o procedimento para atualizar o registro do *TimeStamp* da placa da Netronome.

1	nfp-reg xpb:Nbi0IsldXpbMap.NbiTopXpbMap.MacGlbAdrMap.MacCsr.MacTimeStampSetSec.MacTimeStampSetSec=0xd000
2	nfp-reg xpb:Nbi0IsldXpbMap.NbiTopXpbMap.MacGlbAdrMap.MacCsr.MacTimeStampSetNSec.MacTimeStampSetNsec=0x5678
3	nfp-reg xpb:Nbi0IsldXpbMap.NbiTopXpbMap.MacGlbAdrMap.MacCsr.MacSysSupCtrl.TimeStampEn=0
4	nfp-reg xpb:Nbi0IsldXpbMap.NbiTopXpbMap.MacGlbAdrMap.MacCsr.MacSysSupCtrl.TimeStampSet=1
5	nfp-xpb 0x8045000 0x90
6	nfp-xpb 0x8045000 0x91
7	nfp-reg xpb:Nbi0IsldXpbMap.NbiTopXpbMap.MacGlbAdrMap.MacCsr.MacSysSupCtrl.TimeStampEn=1
8	nfp-reg xpb:Nbi0IsldXpbMap.NbiTopXpbMap.MacGlbAdrMap.MacCsr.MacSysSupCtrl.TimeStampSet=0

Figura 39. Comando para modificação de *TimeStamp*. Fonte: Autor.

Nos itens abaixo é detalhado o funcionamento dos oito comandos apresentado na Figura 39.

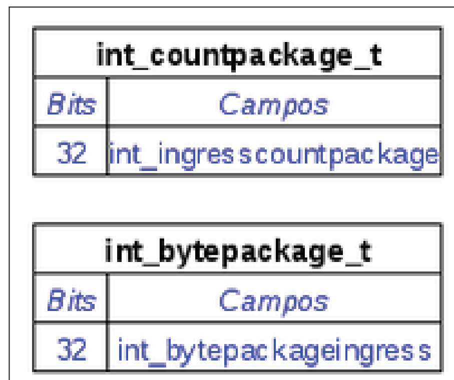
- Comando 1 : tem a função de alterar os valores em segundos do *TimeStamp*;
- Comando 2 : realiza a alteração dos valores em nanossegundos;
- Comando 3 : desativa o *TimeStamp* da placa;
- Comando 4 : habilita a opção de incluir informações de *TimeStamp*;
- Comando 5 : para o contador do *TimeStamp*;
- Comando 6 : iniciará o incrementador de *TimeStamp* com o valor novo que foi alterado;
- Comando 7 : habilita o *TimeStamp* da placa novamente;
- Comando 8 : desativa a opção de alterar o *TimeStamp* da placa da Netronome.

Como já explicado nesta seção, a implementação de sincronização foi entre o dispositivo e a placa da Netronome, não sendo sincronizado o *TimeStamp* entre os dois *switches*. Durante a implementação foi realizado um teste usando o PTP (*Precision Time Protocol*) para sincronizar os *switches*, porém, quando foi aplicada a configuração sobre as interfaces criadas pelo SR-IOV para a comunicação do *host* hospedeiro com a smartNIC, verificou-se que a placa não aceita a configuração do protocolo PTP de *TimeStamp* ao nível de *hardware* para a sincronização.

3.9 IMPLEMENTAÇÃO EXTRA DA CAPTURA DA QUANTIDADES DE *BYTES* E PACOTES

Durante o desenvolvimento do código em P4 com o INT percebeu a possibilidade de realizar a telemetria de mais dois dados dos *switches*, esses dados são a quantidade de

pacotes e a quantidade de *bytes* na interface. Foi criado um algoritmo que realiza a coleta das informações dos *bytes* e dos pacotes passantes na porta de entrada dos *switches*. Para transportar esses dados novos que está sendo coletado dos *switches*, desenvolveu-se dois *headers* que estão representadas na Figura 40, o cabeçalho na parte superior, denominado de "int_ingresscountpackage" armazena a quantidade de pacotes que passou na interface de *ingress* do *switch*. Já o cabeçalho denominado de "int_bytepackage_t" tem a função de guardar os dados referentes a quantidade de *bytes* da interface de entrada.



int_countpackage_t	
Bits	*Campos*
32	int_ingresscountpackage

int_bytepackage_t	
Bits	*Campos*
32	int_bytepackageingress

Figura 40. Cabeçalhos para armazenar a quantidade de *bytes* e de pacotes. Fonte: Autor.

Para adicionar esses *headers* nos pacotes, foi criado dois *templates*, um que é o valor 252 que possui os *bits* ativos "11111100", o sexto *bit* foi ativado para o representar o cabeçalho "int_ingresscountpackage" o outro é o valor 254 que contempla os *bits* "11111110", o sétimo *bit* representar o cabeçalho "int_bytepackage_t". O valor 254 captura todos os dados de telemetria possíveis nesse projeto. Esses dados são coletados toda vez que o pacote passa pela interface de entrada do *switch* então conforme a Figura 41, os dados são coletados da interface P4 do *switch* 1 e depois coletado da porta P0 *switch* 2.

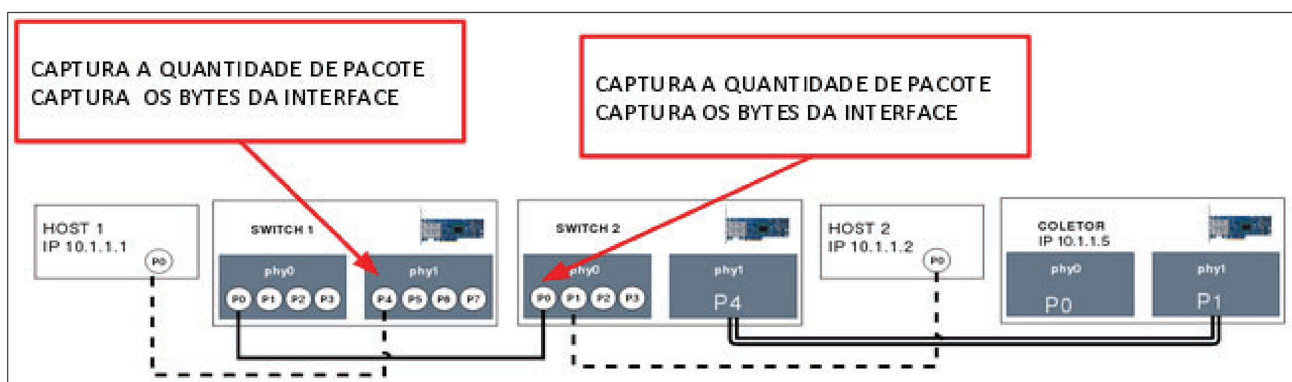


Figura 41. Interfaces que os dados de quantidade de *bytes* e pacote são coletados. Fonte: Autor.

O algoritmo realiza a coleta dessas duas informações sempre que um pacote INT *flow* trafegar pelo *switch* e depois de capturar os dados o algoritmo zera o registro que armazena essas informações no *switch* e inicia novamente a contagem. Desta forma, os

valores capturados são de um intervalo de tempo entre os pacotes INT *flow* e conforme o *TimeStamp* capturado no mesmo pacote, é possível ter a informação correta de quantos bits e pacotes trafegou na interface em um determinado tempo.

4. APRESENTAÇÃO E DISCUSSÃO DOS RESULTADOS

Nesse capítulo serão apresentados os resultados da solução programada em P4 dentro do plano de dados de uma placa da Netronome. A estrutura desenvolvida durante o trabalho está sendo representada na Figura 42. Utilizou-se três smartNIC da Netronome para criar os dois *switches* e um coletor de dados. Os *switches* foram programados em P4 para executar as tarefas do INT *flow*. O propósito dessa dissertação foi o desenvolvimento de uma aplicação que realiza a telemetria dos *switches* e reporta esses dados para o coletor de dados INT.

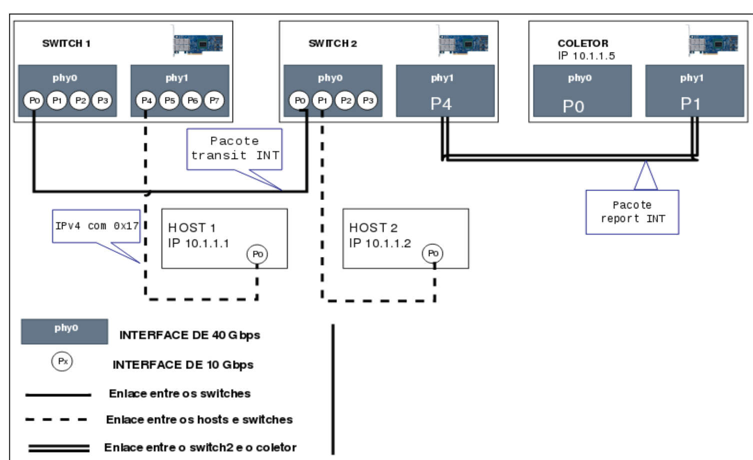


Figura 42. Representação da estrutura do projeto. Fonte: Autor.

A Figura 42 mostra dois pacotes INT. O *transit* INT que é o pacote que trafega entre os *switches* com os metadados de telemetria e o *report* INT que armazena em seu pacote os metadados para ser entregue para o coletor.

4.1 CABEÇALHOS NO PACOTE INT E RECURSOS

Para realizar a avaliação do método INT *flow*, foi necessário validar se o pacote *report* INT está sendo entregue para o coletor com os cabeçalhos organizados conforme foi programado em P4. Para capturar os pacotes, em hexadecimal, com o intuito de avaliar se as informações estão corretas, foi utilizado o *TCPdump*[31].

A Figura 43 ilustra o pacote *report* INT. No lado esquerdo da imagem apresenta-se o conteúdo do pacote em hexadecimal. Nele foram aplicadas diferentes colorações para representar cada cabeçalho utilizado. Nota-se que no pacote repetem-se os cabeçalhos

[31] *TCPdump* é um *software* gratuito em linha de comando, utilizado para capturar pacotes de uma interface de rede.

Ethernet, IPv4 e UDP. Os cabeçalhos que estão antes do INT *report*[32] contêm as informações de rede para que os pacotes trafeguem pelos *switches* e sejam aceitos pelos *hosts* e pelo coletor. Os cabeçalhos *Ethernet*, IPv4 e UDP que estão após o cabeçalho INT *report* representam as informações de rede que o coletor necessita capturar, para poder salvar os dados no banco. Esses cabeçalhos foram denominados "internos", pois ficam dentro dos cabeçalhos do *report* INT. Após os cabeçalhos internos vêm as informações coletadas do *switch* 2 e depois as informações do *switch* 1 e por último o cabeçalho *tail* que representa o fim dos dados INT.

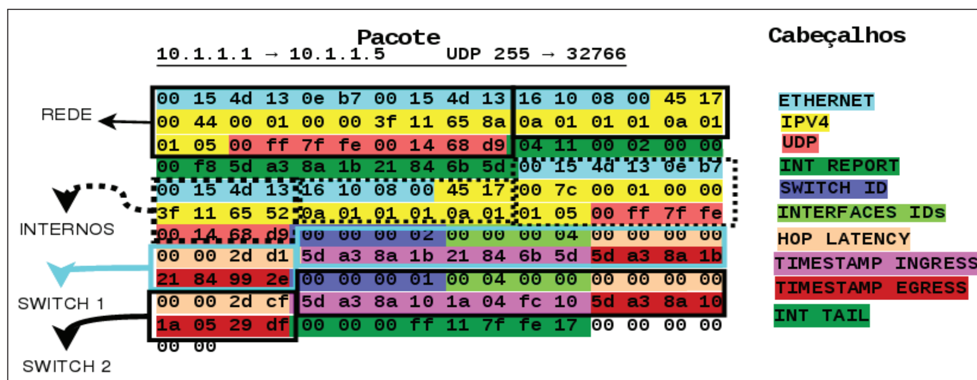


Figura 43. Estrutura em hexadecimal do pacote *report* INT com o *template* 248. Fonte: Autor.

Nota-se que no pacote capturado no *TCPdump*, apresentado na Figura 43, utilizou o *template* 248[33]. Quando é selecionado outro *template* de dados apenas são retirados os cabeçalhos que não serão coletados pelo *framework* INT. Dessa forma o pacote diminui o tamanho e o consumo de recursos.

Quando o pacote INT trafega entre os *switches* denomina-se *transit* INT, pois tem a função de levar a informação entre os dispositivos. Nesses pacotes a principal diferença é sua estrutura organizacional. Nesse caso o pacote não vai possuir o *header* INT *report*, porém em seu lugar vai receber os cabeçalhos *INTshim* e *INThopByhopHeader* e também não possui os cabeçalhos internos. Na Figura 44 está sendo demonstrada a estrutura do pacote *transit* INT. Percebe-se que os pacotes chegaram a seus destinos de maneira correta, conforme as instruções dentro da placa da Netronome.

Outro teste realizado teve como objetivo capturar os pacotes *transit* INT e *report* INT em intervalo de tempo de uma hora a fim de verificar e consumo de tráfego desses pacotes INT. A Figura 45 representa o consumo de tráfego de um pacote *transit* INT, nela percebe-se que o consumo ficou em média 49,72 kbps para o pacote *transit* INT e em torno de 66,2 kbps para o pacote *report* INT, o consumo do *report* INT apresenta-se na

[32] O INT *report* representa o *telemetryreport* que é o cabeçalho inicial do INT, após esse *headers* todos os dados serão analisados pelo coletor.

[33] *Template* 248 coleta as informações de ID do *switch*, IDs das interfaces, *hop latency*, *timestamp* do *ingress* e o *timestamp* do *egress*.

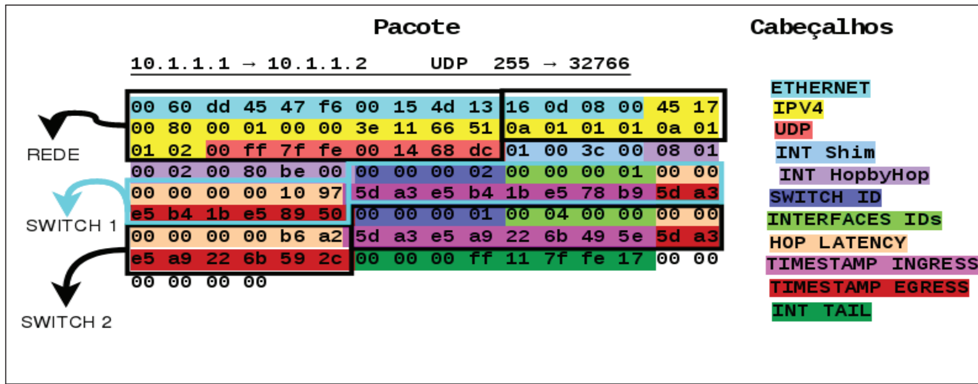


Figura 44. Estrutura em hexadecimal do *transit* INT com o *template* 248. Fonte: Autor.

Figura 46. Como pode ser percebido no consumo de tráfego dos pacotes *transit* INT e *report* INT geram pouco tráfego na rede (Figuras 45 e 46).

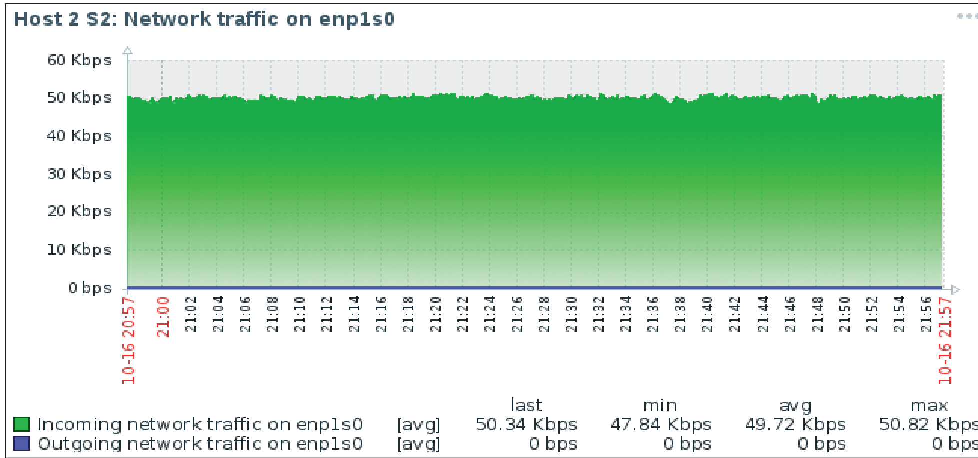


Figura 45. Taxa de transferência do *transit* INT com o *template* 248. Fonte: Autor.

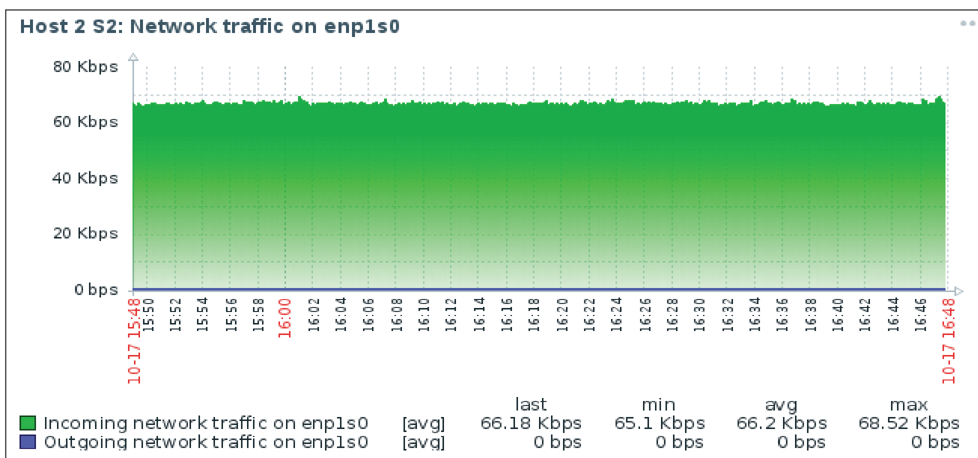


Figura 46. Taxa de transferência do *report* INT com o *template* 248. Fonte: Autor.

Utilizando os pacotes capturados no teste anterior, verificou-se que a estrutura dos *headers* e os campos não se alteraram. Os pacotes capturados em hexadecimal do teste anterior estão armazenados no GitHub [34].

4.1.1 Intervalo entre os pacotes INT

Nessa seção buscou-se verificar a frequência da coleta de dados dos *switches* com P4 utilizando o método INT *flow*. Desta forma foi realizado a coleta de pacotes INT, coletado a informação do *egress timestamp* e foi realizado a subtração do *egress timestamp* entre o valor do pacote atual com o anterior. O pacote INT *flow* está realizando a coleta de dados em média a cada 23 milissegundos, conforme ilustrado na Tabela 4. Esse resultado demonstra que o INT *flow* é mais eficiente que muitos protocolos que estão sendo utilizados nos dias atuais, por exemplo, o SNMP.

Intervalo entre os pacotes INT				
Pacotes	Egress Hexadecimal	Egress Timestamp (ns)	Intervalo (ns)	Intervalo (ms)
0	5dc868e8040b8bf7	6757766586578996215		
1	5dc868e8058903e9	6757766586603996137	24999936	24.999936
2	5dc868e806e84c29	6757766586627017769	23019520	23.01952
3	5dc868e8082854e7	6757766586647991527	20980736	20.980736
4	5dc868e8099696e5	6757766586671994597	23999488	23.999488
5	5dc868e80ab8f653	6757766586691024467	19030016	19.030016
6	5dc868e80c35f92a	6757766586715994410	24970240	24.97024
7	5dc868e80db3e240	6757766586741023296	25029632	25.029632
8	5dc868e80f4f78bf	6757766586767997119	26970112	26.970112
9	5dc868e810eb33fd	6757766586794980349	26990592	26.990592
10	5dc868e812693fe8	6757766586820018152	25029632	25.029632
11	5dc868e8138a9343	6757766586838979395	18960384	18.960384
12	5dc868e815269513	6757766586865980691	27010048	27.010048
13	5dc868e816c2ac4f	6757766586892987471	26999808	26.999808
14	5dc868e8183199af	6757766586917034415	24049664	24.049664
15	5dc868e819c0fe20	6757766586943208992	26170368	26.170368
16	5dc868e81b785c3e	6757766586972003390	28800000	28.8
17	5dc868e81ce6643a	6757766586995991610	23989248	23.989248
18	5dc868e81e6445d6	6757766587021018582	25020416	25.020416
19	5dc868e81fffd676	6757766587047990902	26980352	26.980352
20	5dc868e8219bc376	6757766587074986870	26989568	26.989568
21	5dc868e822cccef9	6757766587094978297	19990528	19.990528
22	5dc868e8243aff7e	6757766587118976894	23999488	23.999488
23	5dc868e825e6679c	6757766587146987420	28010496	28.010496
24	5dc868e8278242f6	6757766587173978870	26989568	26.989568
25	5dc868e828f061fb	6757766587197972987	24000512	24.000512
			Média	**23.62000772**

Tabela 4. Intervalo de tempo entre os pacotes INT. Fonte: Autor.

4.2 INFORMAÇÕES DO PACOTE INT E AVALIAÇÃO *TIMESTAMP*

Nessa seção será realizada a avaliação do *timestamp*. Primeiramente, na Figura 47 ilustra um *timestamp* de 64 bits coletado de um pacote INT *flow* e demonstra os 32 bits

[34] Link para a avaliação: https://github.com/PedroEduardo68/Projeto-INT-Flow-P4/tree/master/validation/one_hour/

iniciais que são a representação da unidade mínima de tempo em segundo. Mostra também os 32 bits finais que representam a unidade mínima de tempo em nanossegundos.

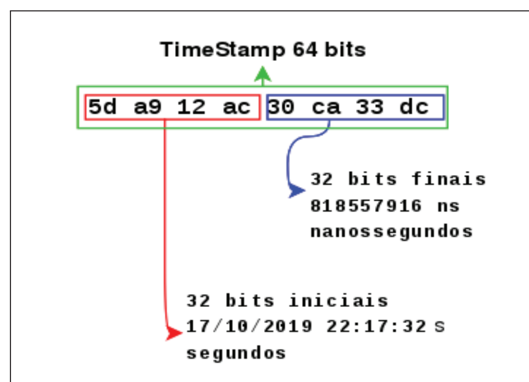


Figura 47. *Timestamp* 64 bits . Fonte: Autor.

A avaliação do *timestamp* dos dados que estão sendo coletados de dentro dos *switches* ocorreu da seguinte forma: foi realizada a coleta de um pacote com o *template* 248 e buscou-se o *timestamp* do *ingress* e *egress* e após transformou os primeiros 32 bits do *timestamp* de hexadecimal em data e hora, para validar se estavam de acordo com a data e hora do momento que o pacote foi coletado. O pacote foi coletado no dia 17 de outubro de 2019 às 22:17:32. Conforme ilustra a Figura 48, o *timestamp* apresentando dentro do pacote é igual ao horário da coleta desse pacote, demonstrando que os equipamentos estão sincronizados nos 32 bits iniciais do *timestamp*. O *switch* 1 e o *switch* 2 mostraram a mesma hora e a mesma data, certificando que as placas estão sincronizadas com seus *hosts* hospedeiros. Isto significa que o tempo dos equipamentos está sendo atualizado de acordo com o que foi descrito na Seção 3.8. Nota-se que os valores 818.557.916, 818.562.252, 641.480.156 e 641.484.077 em nanossegundos (32 bits finais do *timestamp*) estão diferentes, verificando que o pacote buscou a informação no exato momento que chegou no *ingress* ou saiu do *egress*.

A Figura 49 ilustra os dados coletados dos *switches* que está apresentado no hexadecimal da Figura 48. Na Figura 49 mostra o que cada informação representa nos *switches*. As informações de *timestamp* representam a entrada e saída do pacote INT dentro do *switch*, essas informações são importantes para avaliar se o *switch* está com o processamento de pacote funcionando normalmente sem alterações na latência. A identificação do *switch* e as identificações das interfaces de entrada e saída são importantes para saber o exato trajeto do fluxo de dados, para saber em qual ponto ocorreu o aumento de latência. A latência do *switch* (*hop latency*) representa o tempo, em nanossegundos, que o pacote ficou em processamento dentro do *switch*.

Conforme visualizado nas Figuras 49 e 48 foi possível verificar que os pacotes estão vindos com os campos e os dados coletados pelos INT de forma correta, agora deve ser analisado o tamanho dos pacotes. Na próxima seção será demonstrado o crescimento do MTU de um pacote INT.

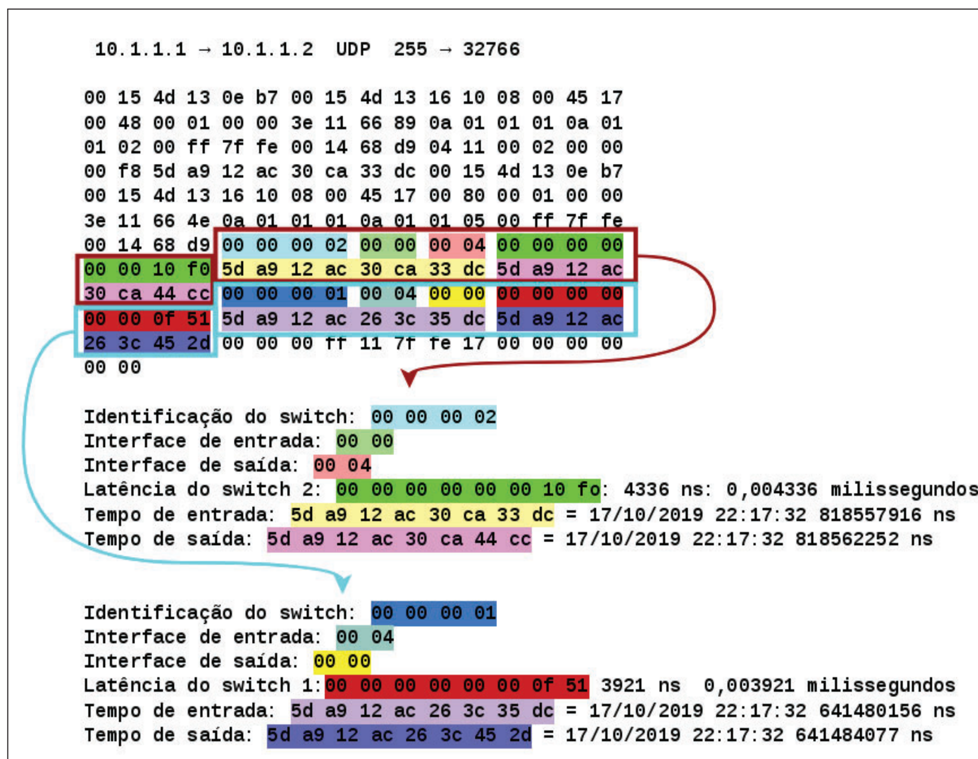


Figura 48. Informações do pacote INT . Fonte: Autor.

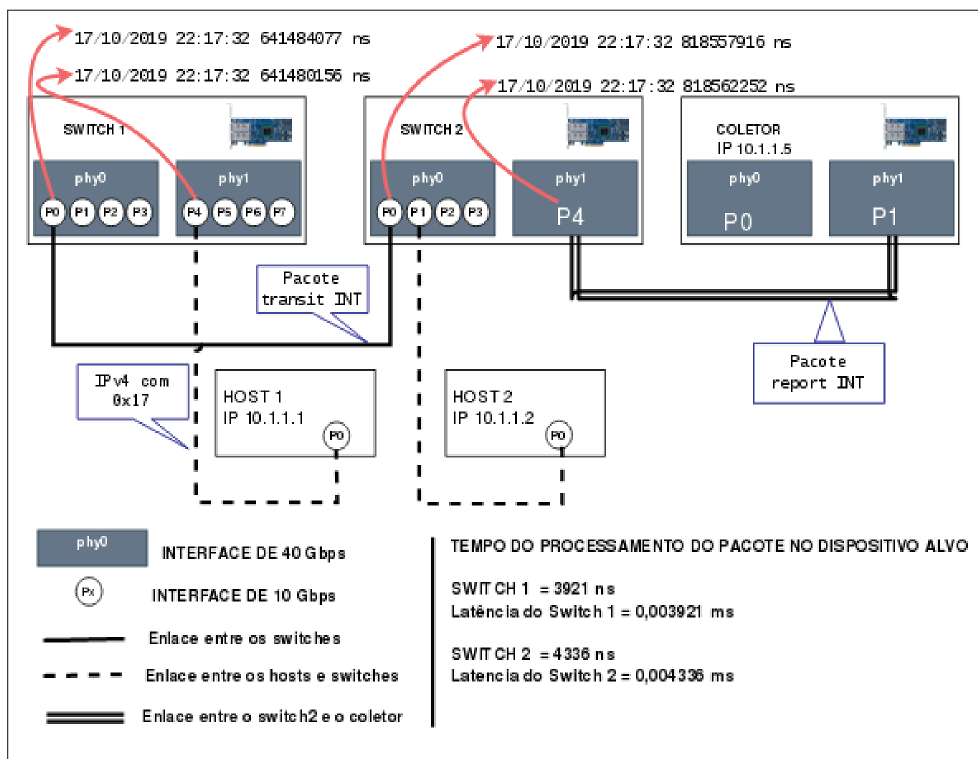


Figura 49. Representação das informações do pacote INT no *switch* . Fonte: Autor.

4.3 AUMENTO DO TAMANHO DO PACOTE COM INT (MTU)

Um pacote de *report* INT tem um acréscimo no tamanho, de acordo com a quantidade de *switches* por onde ele passa e *template* de informações que realiza a coleta de dados do *switch*.

Nessa avaliação foi capturado os pacotes com dados de 0 até 2 *switches* que fazem parte da topologia física desse trabalho, foi avaliado a diferença de tamanho entre os pacotes, desta maneira foi possível realizar a simulação do tamanho do pacote com mais *switches*.

Um pacote *report* INT sem os dados coletados dos *switches* possui um tamanho de 108 Bytes. Os dados coletados em cada *switch* demandaram 32 bytes do tamanho do pacote usando o *template* 248. Então pode-se projetar a demanda conforme a quantidade de *switch* usando o mesmo *template*. O pacote vai possuir o tamanho de 428 bytes usando o *template* 248 com 10 *switches*. A Tabela 5 apresenta a demanda de MTU do pacote *report* INT usando o *template* 248.

Tabela 5. Tamanho dos pacotes *report* INT para o coletor com o *template* 248

Telemetria em switches	Tamanho em Bytes
0 *switch*	108 Bytes
1 *switch*	140 Bytes
2 *switch*	172 Bytes
3 *switch*	204 Bytes
4 *switch*	236 Bytes
5 *switch*	268 Bytes
6 *switch*	300 Bytes
7 *switch*	332 Bytes
8 *switch*	364 Bytes
9 *switch*	396 Bytes
10 *switch*	428 Bytes

O pacote *transit* INT sem os dados coletados dos *switches* possui um tamanho de 62 bytes. A Tabela 6 mostra o crescimento do pacote conforme vai passando pelos *switches* usando o *template* 248.

Tabela 6. Tamanho dos pacotes INT entre *switches* com o *template* 248

Telemetria em switches	Tamanho em Bytes
0 *switch*	62 Bytes
1 *switch*	94 Bytes
2 *switch*	126 Bytes
3 *switch*	158 Bytes
4 *switch*	190 Bytes
5 *switch*	222 Bytes
6 *switch*	254 Bytes
7 *switch*	286 Bytes
8 *switch*	318 Bytes
9 *switch*	350 Bytes
10 *switch*	382 Bytes

Cada *template* possui um tamanho diferente, pois cada *template* realiza a coleta de um conjunto de informação de telemetria. Desta forma, cada conjunto de informações

fica com um tamanho diferente. A Tabela 7 ilustra o tamanho de informações coletadas em cada *template*.

Tabela 7. *Templates* e tamanhos

Templates	bits	Tamanho
128	10000000	4 Bytes
192	11000000	8 Bytes
224	11100000	16 Bytes
240	11110000	24 Bytes
248	11111000	32 Bytes
160	10100000	12 Bytes
184	10111000	28 Bytes
88	01011000	20 Bytes

De acordo com o que foi demonstrado nesta seção percebe-se que apresentou o crescimento de um pacote *transit* INT e de um pacote *report* INT conforme a quantidade de *switches* que realizou a telemetria. Na próxima seção apresenta os outros possíveis *templates* que pode ser utilizado nesse projeto.

4.4 ESTRUTURA DO PACOTE COM *TEMPLATE*

Nesta seção será analisada a estrutura dos pacotes pertencentes ao INT *flow*. A estrutura do pacote irá variar de acordo com o *template* que coleta os dados. Serão demonstrados os oito *templates* que foram desenvolvidos para essa aplicação. O GitHub [35] contém os pacotes com os diferentes *templates* em hexadecimal salvo em oito arquivos, cada arquivo representa os pacotes com um tipo de *template*.

O primeiro pacote está usando o *template* de número 128 que consegue capturar o ID do *switch*. Assim, o pacote tem a informação de quais *switches* o pacote passou. A Figura 51(a) demonstra que o pacote passou pelo *switch* 1 e depois pelo *switch* 2. O segundo é o *template* de número 192 que coletou as informações de ID do *switches* e também das interfaces por onde esse pacote passou, conforme demonstrado na Figura 51(b). No terceiro *template* que é o valor 224 (Figura 52(a)) são extraídas as informações e ID do *switch*, IDs das interfaces e também o *hop latency*. A Figura 52(b) ilustra a estrutura de um pacote com o *template* 240. Com esse *template* coleta-se as informações de ID do *switch*, IDs das interfaces, *hop latency* e o *timestamp* do *ingress*. O *timestamp* do *ingress* representa o momento que o pacote entrou no processamento de entrada do plano de dados.

A Figura 50 representa a legenda que identifica os cabeçalhos dos pacotes que são ilustrados nas Figuras 51, 52, 53 e 54.

[35] Link que contém os arquivos com a captura dos pacotes de diferentes *templates*: https://github.com/PedroEduardo68/Projeto-INT-Flow-P4/tree/master/validation/capture_of_package_template/

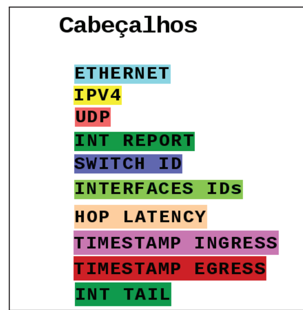


Figura 50. Cabeçalhos presente nos pacotes. Fonte: Autor.

Pacote template 128

10.1.1.1 → 10.1.1.5   UDP 255 → 32766

```
00 15 4d 13 0e b7 00 15 4d 13 16 10 08 00 45 17
00 2c 00 01 00 00 3e 11 66 a2 0a 01 01 01 0a 01
01 05 00 ff 7f fe 00 14 68 d9 04 11 00 02 00 00
00 80 5d b4 f9 7e 08 f7 58 1f 00 15 4d 13 0e b7
00 15 4d 13 16 10 08 00 45 17 00 34 00 01 00 00
3e 11 66 9a 0a 01 01 01 0a 01 01 05 00 ff 7f fe
00 14 68 d9 00 00 00 02 00 00 00 01 00 00 00 ff
11 7f fe 17 00 00 00 00 00 00
```

(a) Representação do pacote com *template* 128. Fonte: Autor.

Pacote template 192

10.1.1.1 → 10.1.1.5   UDP 255 → 32766

```
00 15 4d 13 0e b7 00 15 4d 13 16 10 08 00 45 17
00 30 00 01 00 00 3e 11 66 9e 0a 01 01 01 0a 01
01 05 00 ff 7f fe 00 14 68 d9 04 11 00 02 00 00
00 c0 5d b4 fb 99 0b ba 02 0c 00 15 4d 13 0e b7
00 15 4d 13 16 10 08 00 45 17 00 40 00 01 00 00
3e 11 66 8e 0a 01 01 01 0a 01 01 05 00 ff 7f fe
00 14 68 d9 00 00 00 02 00 00 00 04 00 00 00 01
00 04 00 00 00 00 00 ff 11 7f fe 17 00 00 00 00
00 00
```

(b) Representação do pacote com *template* 192. Fonte: Autor.

Figura 51. *Template* 128 e 192.

Pacote template 224

10.1.1.1 → 10.1.1.5   UDP 255 → 32766

```
00 15 4d 13 0e b7 00 15 4d 13 16 10 08 00 45 17
00 38 00 01 00 00 3e 11 66 96 0a 01 01 01 0a 01
01 05 00 ff 7f fe 00 14 68 d9 04 11 00 02 00 00
00 e0 5d b4 fe df 24 f8 82 be 00 15 4d 13 0e b7
00 15 4d 13 16 10 08 00 45 17 00 58 00 01 00 00
3e 11 66 76 0a 01 01 01 0a 01 01 05 00 ff 7f fe
00 14 68 d9 00 00 00 02 00 00 00 04 00 00 00 00
00 00 0f 8d 00 00 00 01 00 04 00 00 00 00 00 00
00 00 0f a1 00 00 00 ff 11 7f fe 17 00 00 00 00
00 00
```

(a) Representação do pacote com *template* 224. Fonte: Autor.

Pacote template 240

10.1.1.1 → 10.1.1.5   UDP 255 → 32766

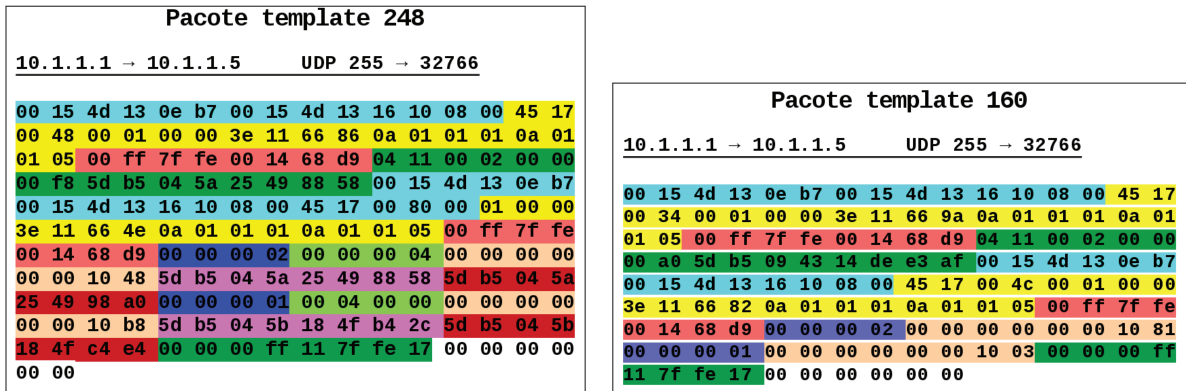
```
00 15 4d 13 0e b7 00 15 4d 13 16 10 08 00 45 17
00 40 00 01 00 00 3e 11 66 8e 0a 01 01 01 0a 01
01 05 00 ff 7f fe 00 14 68 d9 04 11 00 02 00 00
00 f0 5d b5 01 ee 13 2d cf 74 00 15 4d 13 0e b7
00 15 4d 13 16 10 08 00 45 17 00 6c 00 01 00 00
3e 11 66 62 0a 01 01 01 0a 01 01 05 00 ff 7f fe
00 14 68 d9 00 00 00 02 00 00 00 04 00 00 00 00
00 00 10 22 5d b5 01 ee 13 2d cf 74 00 00 00 01
00 04 00 00 00 00 00 00 00 00 10 03 5d b5 01 e6
05 d1 5d 9c 00 00 00 ff 11 7f fe 17 00 00 00 00
00 00
```

(b) Representação do pacote com *template* 240. Fonte: Autor.

Figura 52. *Template* 224 e 240.

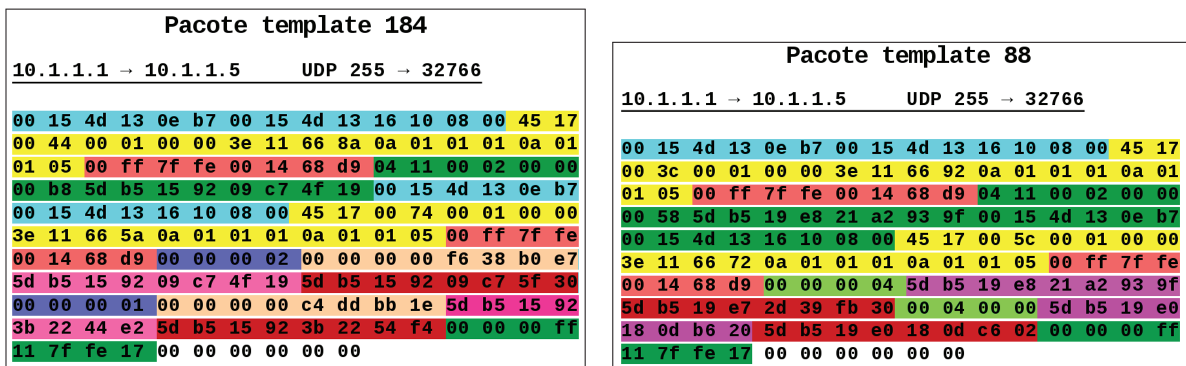
A Figura 53(a) mostra um pacote com o *template* 248, com esse *template* coleta-se as informações de IDs dos *switches*, IDs das interfaces, *hop latency*, *timestamp* do *ingress* e o *timestamp* do *egress*. O destaque desse *template* é que ele pode extrair a informação de tempo em que o pacote saiu do processamento de saída. O *template* 160 (Figura 53(b)) tem apenas as informações de ID do *switch* e o *hop latency*. O *template* 184, que captura os dados de ID do *switch*, *hop latency*, *timestamp* do *ingress* e o *timestamp* do *egress*, conforme ilustrado na Figura 54(a). A Figura 54(b) irá coletar os dados de IDs das

interfaces e também o *timestamp* do *ingress* e após o *timestamp* do *egress*, esse representa o *template* 88.



(a) Representação do pacote com *template* 248. Fonte: Autor.
(b) Representação do pacote com *template* 160. Fonte: Autor.

Figura 53. *Template* 248 e 160.



(a) Representação do pacote com *template* 184. Fonte: Autor.
(b) Representação do pacote com *template* 88. Fonte: Autor.

Figura 54. *Template* 184 e 88.

Nessa seção foram apresentados os pacotes de *report* INT gerados com os oito *templates* possíveis. Como se percebe nas Figuras 51, 52, 53 e 54 os campos e os cabeçalhos desenvolvidos nesse projeto, estão chegando ao coletor conforme as definições programadas dentro do plano de dados. Assim, o administrador de rede pode escolher o *template* que atende as suas necessidades da rede.

4.5 ANÁLISE DO PACOTE COM OS DADOS DE TELEMETRIA EXTRA

Com a finalidade de avaliar os últimos campos adicionados no *software* desenvolvido nesse trabalho. Foi capturado um pacote em hexadecimal com o *template* de número 254 que tem todos os dados do *template* 248 e também contem os dados extras que são

a quantidade de bytes e a quantidade de pacotes que são extraídos do *switch* 1 e 2. Na Figura 55, apresenta os 4 campos dados novos, que cada campo possui 32 bits, então são adicionados nos pacotes de dados mais 128 bits de dados.

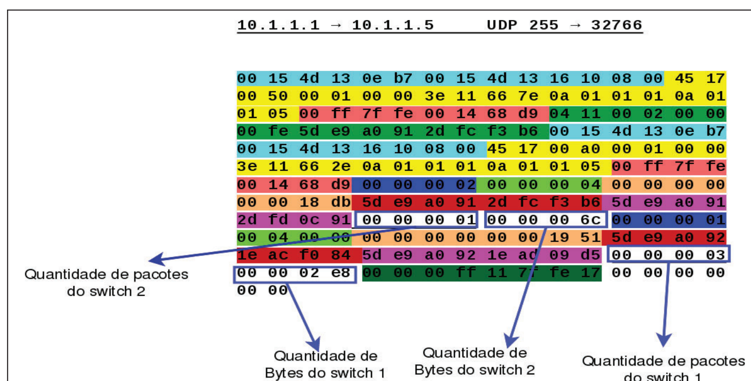


Figura 55. Hexadecimal com os dados de telemetria extra. Fonte: Autor.

Desta forma, percebe-se que os 128 bits adicionados no pacote de dados estão organizados corretamente dentro da estrutura dos dados de telemetria, conforme definido dentro do algoritmo aplicado no plano de dados.

5. CONSIDERAÇÕES FINAIS

Este trabalho descreveu os conceitos de SDN (redes definidas por *software*) e realizou a programação do plano de dados da placa da Netronome. Para a construção do trabalho foi implementado uma rede com cinco computadores, sendo que dois desses computadores foi programado para funcionar como um *switch*. Desta forma, foi possível simular os *switches* de uma rede MAN.

Após a avaliação dos dados coletados do INT *flow*, constatou que a solução desenvolvida nesse trabalho está realizando a telemetria dos dados de maneira correta. Observou-se que a solução programada em P4 dentro das placas da Netronome está realizando a organização e os ajustes dentro dos pacotes de maneira correta e eficiente, assim deixando a estrutura do pacote pronta para ser coletada pelo coletor INT. Desta forma, percebeu-se que a organização dos *headers* realizada na placa da Netronome está funcionando conforme as regras definidas em seu plano de dados.

Porém, percebeu-se que as regras de plano de controle local definido dentro do arquivo "*user config file*" são fixas dentro do *firmware* da placa da Netronome. Então qualquer alteração que é realizada no "*user config file*" deve realizar a compilação de um *firmware* novo para a placa, necessitando o reinício da smartNIC e causando uma interrupção no funcionamento da placa durante a aplicação do *firmware* novo.

Como constatou-se nos resultados o INT *flow* consegue capturar os dados de telemetria em um fluxo constante que utiliza 0,0006% da capacidade de uma porta física de 10 Gbps. Conclui-se que o INT *flow* não interfere nos outros fluxos de dados e também não sobrecarrega o coletor de dados conforme relatado em outros trabalhos. Mesmo assim, o INT *flow* é mais eficiente que a coleta realizada utilizando o protocolo SNMP que é basicamente realizada a cada segundo ou em tempo superior. O INT *flow* pode realizar coleta a cada 23 milissegundos, e entregar os dados de forma organizada para o coletor de dados.

As contribuições na área profissional foi o desenvolvido um *software* em P4, denominado INT *flow*, que consome 0,0006% da banda de um interface de 10 Gbps e consegue capturar informações do *TimeStamp*[36] em nanossegundos (64 bits), com isso pode identificar algumas falhas de rede. Esse código está disponível no Github[37] para ser utilizado, inicialmente foi programado para funcionar sobre equipamentos da Netronome, mas pode facilmente ser alterado para funcionar em outros equipamentos e outras arquiteturas. Ao focar no objetivo do trabalho, foram obtidas algumas contribuições na área acadêmica, as principais contribuições foram o estudo da placa da Netronome e também o de detalha-

[36] Durante a realização desse projeto foi interagido cinco vezes com o suporte da Netronome. A interação que mais se destaca, é sobre o assunto do *TimeStamp*, na qual a interação conjunta produziu uma maneira de atualizar o *TimeStamp* da placa, que até o momento não tinha sido realizada.

[37] Site para o código: https://github.com/PedroEduardo68/Projeto-INT-Flow-P4

mento do funcionamento da linguagem P4, os detalhamentos do funcionamento da Agilio P4C SDK e principalmente incentivar o estudo dessa nova tecnologia.

A dificuldade de realização desse trabalho ocorreu principalmente na montagem da topologia. Inicialmente o trabalho estava sendo projetado para funcionar em um ambiente virtualizado. Porém, foram encontrados vários desafios de desempenho e *softwares* para montagem do ambiente virtualizado. Após isso iniciou-se a montagem de um ambiente totalmente físico com as placas da Netronome. Depois dessa etapa de montagem estar finalizada, iniciou a programação do algoritmo P4, sendo que nesta parte encontraram-se vários desafios, como por exemplo, a limitação de memória, registros e regras da SmartNIC, que devem ser respeitadas para um bom funcionamento da linguagem P4.

5.1 RESSALVAS E LIMITAÇÕES

A seguir serão apresentados alguns detalhes sobre as limitações encontradas na placa de rede Netronome, modelo agilio CX 40 Gbps, utilizada nesse trabalho. Essas limitações e ressalvas não ocorrem por causa do *software* desenvolvido, mas pelo *hardware* utilizado.

Cabe salientar que os dados das estruturas de *hearders* e de *metadata*, quando utilizados pelo P4, ficam armazenados na mesma estrutura de memória da placa, ou seja, ocorre uma concorrência entre as duas estruturas de dados. Desta forma, nesse trabalho não foram capturados os campos de metadados do INT, apenas capturados os *headers* de controle do INT, buscando utilizar menos recurso de memória no armazenamento das estruturas.

Quando está trabalhando com pilhas de cabeçalhos no pacote, em linguagem P4, a estrutura da placa da Netronome vai suportar apenas 16 *headers* em formato de pilha. Portanto, nota-se que a placa da Netronome tem um limite máximo de *switches* que podem ser empilhados os dados de telemetria.

5.2 TRABALHOS FUTUROS

No decorrer da implementação desse trabalho encontrou algumas questões que precisam serem estudadas e implementadas futuramente. Em seguida, apresentam-se essas questões para serem realizadas em trabalhos futuros:

- Aperfeiçoar a sincronização do *TimeStamp* em 64 bits da placa da Netronome, de maneira que possa sincronizar o *TimeStamp* diretamente nas placas da Netronome, sem interação com o *host* hospedeiro da placa.
- Projetar uma criptografia para os cabeçalhos de telemetria.

- Implementar um plano de controle independente do *firmware* da smartNIC, para que as alterações do plano de controle não necessite o reinício da placa da Netronome.

REFERÊNCIAS BIBLIOGRÁFICAS

[1] SECURITY, H. N. *Number of connected devices reached 22 billion, where is the revenue?* 2019. Disponível em: <https://www.helpnetsecurity.com/2019/05/23/connected-devices-growth/>. Acesso em: Aug. 2019.

[2] TOTAL de Banda (Todos IX.br). 2019. Disponível em: <https://ix.br/agregado/>. Acesso em: Set. 15, 2019.

[3] HENDRIKS, L. et al. Assessing the quality of flow measurements from openflow devices. In: 8TH INTERNATIONAL WORKSHOP ON TRAFFIC MONITORING AND ANALYSIS (TMA). Louvain La Neuve, Belgium: Universidade de Twente, 2016. 1–8.

[4] Kaur, G.; Gupta, P. Hybrid approach for detecting ddos attacks in software defined networks. In: *2019 Twelfth International Conference on Contemporary Computing (IC3)*. Noida, India: IEEE, 2019. p. 1–6.

[5] KREUTZ, D. et al. Software-defined networking: A comprehensive survey. *Proceedings of the IEEE*, v. 103, n. 1, p. 14–76, 2014.

[6] BATISTA, D. M. et al. Perspectives on software-defined networks: interviews with five leading scientists from the networking community. *Journal of Internet Services and Applications*, v. 6, n. 1, p. 2–10, Oct. 2015.

[7] CASADO, M. et al. Fabric: A retrospective on evolving sdn. In: *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*. New York, NY, USA: ACM, 2012. (HotSDN '12), p. 85–90. ISBN 978-1-4503-1477-0. Disponível em: <http://doi.acm.org/10.1145/2342441.2342459>.

[8] BHAUMIK, P. et al. Software-defined optical networks (sdons): a survey. *Photonic Network Communications*, v. 28, n. 1, p. 4–18, Aug. 2014.

[9] MARCONDES, C. *Projeto de Desenvolvimento em OpenFlow*. Espírito Santo, 2011. 18 p.

[10] FERNANDEZ, M. P. Comparing openflow controller paradigms scalability: Reactive and proactive. In: *2013 IEEE 27th International Conference on Advanced Information Networking and Applications (AINA)*. Barcelona, Spain: IEEE, 2013. p. 1009–1016.

[11] OTHMAN, W. M. et al. Network management system (nms) using local collector mediation devices. In: *2016 10th International Conference on Telecommunication Systems Services and Applications (TSSA)*. Indonesia: IEEE, 2016. p. 1–4.

[12] MCKEOWN, N. et al. Openflow: Enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, v. 38, n. 6, p. 69–74, Abr. 2008.

[13] SPALLA, E. S. et al. Resilient strategies to sdn: An approach focused on actively replicated controllers. In: *2015 XXXIII Brazilian Symposium on Computer Networks and Distributed Systems*. Vitória, Brazil: SBRC, 2015. p. 246–259.

[14] LIMONCELLI, T. A. Openflow: A radical new idea in networking. *ACM Queue*, v. 10, n. 6, p. 40–46, Jum. 2012.

[15] BERNARDOS, C. J. et al. 5g exchange (5gex) – multi-domain orchestration for software defined infrastructures. In: EUCNC 2015 - EUROPEAN CONFERENCE ON NETWORKS AND COMMUNICATIONS. Paris,France: IEEE Xplore, 2015. 6 p.

[16] KIM, W. et al. Ofmon: Openflow monitoring system in onos controllers. In: *2016 IEEE NetSoft Conference and Workshops (NetSoft)*. Seoul, South Korea: IEEE, 2016. p. 397–402.

[17] BISWAS, M. I. et al. A dynamic approach to mib polling for software defined monitoring. *Journal of Computer and Communications*, v. 5, n. 5, p. 24–41, Jan. 2017.

[18] STANCU, A. L. et al. A comparison between several software defined networking controllers. In: *2015 12th International Conference on Telecommunication in Modern Satellite, Cable and Broadcasting Services (TELSIKS)*. Serbia, Niš: IEEE, 2015. p. 223–226.

[19] COSTA, L. R. Openflow e o paradigma de redes definidas por software. *Universidade de Brasília*, v. 1, n. 143, p. 1–143, Aug. 2013.

[20] OTHMAN, W. M. et al. Implementation and performance analysis of sdn firewall on pox controller. In: *2017 IEEE 9th International Conference on Communication Software and Networks (ICCSN)*. GuangZhou,China: IEEE, 2017. p. 1461–1466.

[21] YANG, Y. et al. Design and implementation of sdn-based lightweight ipv6 mobility manager in onos. In: *Proceedings of the 11th International Conference on Ubiquitous Information Management and Communication*. New York, NY, USA: ACM, 2017. (IMCOM '17), p. 107:1–107:5. ISBN 978-1-4503-4888-1. Disponível em: <http://doi.acm.org/10.1145/3022227.3022333>.

[22] YAMEI, F.; QING, L.; QI, H. Research and comparative analysis of performance test on sdn controller. In: *2016 First IEEE International Conference on Computer Communication and the Internet (ICCCI)*. Wuhan,China: IEEE, 2016. p. 207–210.

[23] SUBRAMANIAN, S.; VORUGANTI, S. *Software-Defined Networking (SDN) with OpenStack*. Brighton, UK: Packt Publishing, 2016. 216 p.

[24] BOSSHART, P. et al. P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, v. 44, n. 9, p. 87–95, Jul. 2014.

[25] LIU, Z. et al. One sketch to rule them all: Rethinking network flow monitoring with univmon. In: *Proceedings of the 2016 ACM SIGCOMM Conference*. New York, NY, USA: ACM, 2016. (SIGCOMM '16), p. 101–114. ISBN 978-1-4503-4193-6. Disponível em: <http://doi.acm.org/10.1145/2934872.2934906>.

[26] OLTEANU, V. et al. Stateless datacenter load-balancing with beamer. In: *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. Renton, WA, USA: ACM, 2018. p. 125–139.

[27] LI, Y. et al. Flowradar: A better netflow for data centers. In: *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. Santa Clara, CA: ACM, 2016. p. 311–324.

[28] CONSORTIUM, P. L. *P416 Language Specification Version 1.0.0*. USA, 2017. 129 p.

[29] TAUSANOVITCH, N. *What Makes a NIC a SmartNIC, and Why is it Needed?* 2016. Disponível em: <https://www.netronome.com/blog/what-makes-a-nic-a-smartnic-and-why-is-it-needed>. Acesso em: Jan. 20, 2019.

[30] Cao, P. et al. Trying linear network coding on a network flow processor. In: *2011 Fifth International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing*. Korean Bible University (KBU), Seoul, Ko: IEEE Computer Society, 2011. v. 5, p. 156–163.

[31] Pennefather, S.; Bradshaw, K.; Irwin, B. Exploration and design of a synchronous message passing framework for a cpu-npu heterogeneous architecture. In: *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. Vancouver,BC , Canada: IEEE, 2018. v. 1, p. 46–56.

[32] TECHNOLOGIES, N. *100G In-band Network Telemetry With Netcope P4*. 2018. Disponível em: <https://www.netcope.com/getattachment/670aabd2-89f6-4ecf-8620-9b437a256f24/100G-In-band-Network-Telemetry-With-NP4.asp>. Acesso em: Aug. 8, 2018.

[33] KIM, C. *Programming the network dataplane*. 2016. Disponível em: <http://netseminar.stanford.edu/seminars/033116.pdf>. Acesso em: Dez. 13, 2018.

[34] KEKELY, L. et al. Software defined monitoring of application protocols. *IEEE Transactions on Computers*, v. 65, n. 2, p. 615–626, Feb. 2016.

[35] CLAISE, B. *Cisco Systems NetFlow Services Export Version 9*. 2004. Disponível em: <https://rfc-editor.org/rfc/rfc3954.txt>. Acesso em: Aug. 20, 2018.

[36] AITKEN, P.; CLAISE, B.; TRAMMELL, B. *Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of Flow Information*. 2013. Disponível em: <https://rfc-editor.org/rfc/rfc7011.txt>. Acesso em: Aug. 16, 2018.

[37] PHAAL, P.; PANCHEN, S.; MCKEE, N. *InMon Corporation's sFlow: A Method for Monitoring Traffic in Switched and Routed Networks*. 2001. Disponível em: <https://www.rfc-editor.org/rfc/rfc3176.txt>. Acesso em: Aug. 20, 2018.

[38] SEZER, S. et al. Are we ready for sdn? implementation challenges for software-defined networks. *IEEE Communications Magazine*, v. 51, n. 7, p. 36–43, Jul. 2013.

[39] LIANG, J. et al. In-band network function telemetry. In: *Proceedings of the ACM SIGCOMM 2018 Conference on Posters and Demos*. New York, NY, USA: ACM, 2018. (SIGCOMM '18), p. 42–44. ISBN 978-1-4503-5915-3. Disponível em: <http://doi.acm.org/10.1145/3234200.3234236>.

[40] HIRA, M.; WOBKER, L. *Improving Network Monitoring and Management with Programmable Data Planes*. 2018. Disponível em: <https://p4.org/p4/inband-network-telemetry/>. Acesso em: Nov. 11, 2018.

[41] KIM, C. et al. *In-band Network Telemetry via Programmable Dataplanes*. 2015. Disponível em: <https://cs.nyu.edu/ anirudh/int-demo.pdf>. Acesso em: Aug. 8, 2018.

[42] KIM, C. et al. *In-band Network Telemetry (INT)*. Santa Clara, CA, USA, 2016. 32 p.

[43] Hyun, J.; Van Tu, N.; Hong, J. W. Towards knowledge-defined networking using in-band network telemetry. In: *NOMS 2018 - 2018 IEEE/IFIP Network Operations and Management Symposium*. Taipei, Taiwan: IEEE, 2018. v. 1, p. 1–7.

[44] 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://dpnm.postech.ac.kr/thesis/18/Tu_thesis.pdf>.

[45] ANDERSSON, J.-O. *Offloading INTCollector Events with P4*. Dissertação (Mestrado) — Faculty of Health, Science and Technology, Karlstad, Sweden, 2019. Disponível em: <http://www.diva-portal.org/smash/get/diva2:1345544/FULLTEXT01.pdf>.

[46] AGILIO, N. *Netronome Agilio CX Dual-Port 40 Gigabit Ethernet SmartNIC*. 2016. Disponível em: <https://www.netronome.com/m/documents/PB_Agilio_CX_2x40GbE.pdf>. Acesso em: Jan. 29, 2019.

[47] NETRONOME. *Programming NFP with P4 and C*. 2018. Disponível em: <https://www.netronome.com/m/documents/WP_Programming_with_P4_and_C.pdf>. Acesso em: Fev. 25, 2019.

[48] NICHOLS, K. et al. *Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers*. 1998. Disponível em: <https://tools.ietf.org/html/rfc2474>. Acesso em: Set. 16, 2019.

ANEXO A – CÓDIGO *INGRESS SWITCH* 1

```
/************************************************************************
**************  I N G R E S S   S W I T C H 1  *************************
************************************************************************/

control MyIngress(inout headers hdr, inout metadata meta, inout standard_metadata_t standard_metadata) {
            register<bit<32>>(8) ingresscount;
            register<bit<32>>(8) bytecountingress;
    action drop() {
        mark_to_drop();
    }
    action port_forward(){
        meta.switch_local.port_out = standard_metadata.egress_spec;
        meta.switch_local.port_in = standard_metadata.ingress_port;
        meta.switch_local.switch_id = HW_ID;
    }
// **********  Table and Action of header ARP ********
    action arp_forward(macAddr_t dstmac, macAddr_t srcmac, egressSpec_t espec) {
        standard_metadata.egress_spec = espec;
        hdr.ethernet.sourceAddress = srcmac;
        hdr.ethernet.destinationAddress = dstmac;
    }
    table arp_lpm {
        key = {
            hdr.arp.targetProtocolAddress: exact;
        }
        actions = {
            arp_forward;
            drop;
            NoAction;
        }
        default_action = drop();
    }
// **********  Table and Action of header IPV4 ********
    action ipv4_forward(macAddr_t dstmac, macAddr_t srcmac, egressSpec_t espec) {
        standard_metadata.egress_spec = espec;
        hdr.ethernet.sourceAddress = srcmac;
        hdr.ethernet.destinationAddress = dstmac;
        hdr.ipv4.timeToLive = hdr.ipv4.timeToLive - 1;
    }
    table ipv4_lpm {
        key = {
            hdr.ipv4.destinationAddress: exact;
        }
        actions = {
            ipv4_forward;
            drop;
            NoAction;
        }
        default_action = drop();
    }
// **********  Table and Action of  INT header souce ********
    action set_ipv4_service() {
        hdr.ipv4.typeServiceDiffServ = DSCP_INT;
    }
    action add_int_source(bit<8> instruction) {
        port_forward();
        // insert INT shim header
        hdr.shimINT.setValid();
        hdr.shimINT.shim_type = 1;                    // int_type: Hop-by-hop type (1) , destination type (2)
        hdr.shimINT.shim_reserved1 = 0;
        hdr.shimINT.shim_length = 0;
        hdr.shimINT.shim_rsvd2 = 0;
        hdr.hopINT.setValid();
        hdr.hopINT.int_version = 8;
        hdr.hopINT.int_replication = 1;
        hdr.hopINT.int_copy = 0;
        hdr.hopINT.int_exceeded = 0;
        hdr.hopINT.int_rsvd_1 = 0;
        hdr.hopINT.int_ins_cnt = 8;
        hdr.hopINT.int_max_hops = 2;
        hdr.hopINT.int_total_hops = 0;
        hdr.hopINT.int_instruction_bit = instruction;
        hdr.hopINT.int_rsvd_instructions = 0;
        // add the header len (3 words) to total len
        hdr.ipv4.totalLength = hdr.ipv4.totalLength + INT_HEADER_SIZE + INT_SHIM_HEADER_SIZE;
        hdr.udp.lengthUDP = hdr.udp.lengthUDP + INT_HEADER_SIZE + INT_SHIM_HEADER_SIZE;
```

```
            set_ipv4_service();
        }
        table int_source {
            actions = {
                add_int_source;
            }
        }
    apply {
            meta.switch_local.ingresststamp = meta.intrinsic_metadata.ingress_global_tstamp;
            if(hdr.ethernet.isValid()){
                //Count package
                ingresscount.read(meta.switch_local.ingresspackage, (bit<32>)meta.switch_local.port_in);
                meta.switch_local.ingresspackage = meta.switch_local.ingresspackage + 1;
                ingresscount.write((bit<32>)meta.switch_local.port_in, meta.switch_local.ingresspackage);
                //Count bytes
                bytecountingress.read(meta.switch_local.ingressbyte, (bit<32>)meta.switch_local.port_in);
                meta.switch_local.ingressbyte = meta.switch_local.ingressbyte + (bit<32>)standard_metadata.packet_length;
                bytecountingress.write((bit<32>)meta.switch_local.port_in, meta.switch_local.ingressbyte);
            }
            if(hdr.ipv4.typeServiceDiffServ == DSCP_INT){
                  ingresscount.write((bit<32>)meta.switch_local.port_in, 0);
                  bytecountingress.write((bit<32>)meta.switch_local.port_in, 0);
            }
            if(hdr.arp.isValid()) {
                arp_lpm.apply();
            }
            else if (hdr.ipv4.isValid()) {
                ipv4_lpm.apply();
                if(hdr.ipv4.typeServiceDiffServ == DSCP_INT){
                  int_source.apply();
                }
            }
        }
}
```

ANEXO B – CÓDIGO *INGRESS SWITCH* 2

```
/*************************************************************************
************** I N G R E S S   S W I T C H 2   *******************
*************************************************************************/

control MyIngress(inout headers hdr, inout metadata meta, inout standard_metadata_t standard_metadata) {
   action drop() {
        mark_to_drop();
    }
    action port_forward(){
        meta.switch_local.port_out = standard_metadata.egress_spec;
        meta.switch_local.port_in = standard_metadata.ingress_port;
        meta.switch_local.switch_id = HW_ID;
    }
// ********** Table and Action of header ARP ********
    action arp_forward(macAddr_t dstmac, macAddr_t srcmac, egressSpec_t espec) {
        standard_metadata.egress_spec = espec;
        hdr.ethernet.sourceAddress = srcmac;
        hdr.ethernet.destinationAddress = dstmac;
    }
    table arp_lpm {
        key = {
            hdr.arp.targetProtocolAddress: exact;
        }
        actions = {
            arp_forward;
            drop;
            NoAction;
        }
        size = 128;
        default_action = drop();
    }
// ********** Table and Action of header IPV4 ********
    action ipv4_forward(macAddr_t dstmac, macAddr_t srcmac, egressSpec_t espec) {
        standard_metadata.egress_spec = espec;
        hdr.ethernet.sourceAddress = srcmac;
        hdr.ethernet.destinationAddress = dstmac;
        hdr.ipv4.timeToLive = hdr.ipv4.timeToLive - 1;
    }
    table ipv4_lpm {
        key = {
            hdr.ipv4.destinationAddress: exact;
        }
        actions = {
            ipv4_forward;
            drop;
            NoAction;
        }
        size = 128;
        default_action = drop();
    }
// ********** Table and Action of  INT header souce ********
  action send_telemetry_report(macAddr_t monitor_mac, ip4Addr_t monitor_ip, egressSpec_t monitor_port) {
        port_forward();
        meta.switch_local.instruction = hdr.hopINT.int_instruction_bit;
        hdr.shimINT.setInvalid();
        hdr.hopINT.setInvalid();
        //Report Ethernet Header
        hdr.report_ethernet.setValid();
        hdr.report_ethernet.destinationAddress = monitor_mac;
        hdr.report_ethernet.sourceAddress =  hdr.ethernet.sourceAddress;
        hdr.report_ethernet.etherType = ETHERTYPE_IPV4;
        //Report IPV4 Header
        hdr.report_ipv4.setValid();
        hdr.report_ipv4.version = hdr.ipv4.version ;
        hdr.report_ipv4.headerLength = hdr.ipv4.headerLength;
        hdr.report_ipv4.typeServiceDiffServ = DSCP_INT;
        /* Total Len is report_ipv4_len + report_udp_len + report_fixed_hdr_len + ethernet_len + ipv4_totalLen */
        hdr.report_ipv4.totalLength =  hdr.ipv4.totalLength;
        /* Dont Fragment bit should be set */
        hdr.report_ipv4.identification = hdr.ipv4.identification;
        hdr.report_ipv4.fragmentOffset = hdr.ipv4.fragmentOffset;
        hdr.report_ipv4.timeToLive = hdr.ipv4.timeToLive;
        hdr.report_ipv4.protocol = PROTOCOLO_UDP;
        hdr.report_ipv4.sourceAddress = hdr.ipv4.sourceAddress;
        hdr.report_ipv4.destinationAddress = monitor_ip;
```

```
        //Report UDP Header
        hdr.report_udp.setValid();
        hdr.report_udp.sourcePort = hdr.udp.sourcePort;
        hdr.report_udp.destinationPort = monitor_port;
        hdr.report_udp.lengthUDP =  hdr.udp.lengthUDP;
        hdr.report_udp.checksum = hdr.udp.checksum;
        hdr.report.setValid();
        hdr.report.f_version = 4;
        hdr.report.f_next_proto = hdr.ipv4.protocol;
        hdr.report.f_drop = 0;
        hdr.report.f_queue = 0 ;
        hdr.report.f_flow = 0 ;
        hdr.report.f_rsvd = 0 ;
        hdr.report.f_hw_id = (bit<8>)HW_ID ;
        hdr.report.f_seq_num = (bit<32>) meta.switch_local.instruction;
        hdr.report.f_ingress_ts = meta.switch_local.ingresststamp;
    }
    table process_int_report {
        actions = {
            send_telemetry_report;
        }
    }
            register<bit<32>>(8) ingresscount;
            register<bit<32>>(8) bytecountingress;
   apply {
        meta.switch_local.ingresststamp = meta.intrinsic_metadata.ingress_global_tstamp;
        if(hdr.arp.isValid()) {
            arp_lpm.apply();
        }
        else if (hdr.ipv4.isValid()) {
            ipv4_lpm.apply();
            if(hdr.ipv4.typeServiceDiffServ == DSCP_INT){
                process_int_report.apply();
            }
        }
       if(hdr.ethernet.isValid()){
            //Count package
            ingresscount.read(meta.switch_local.ingresspackage, (bit<32>)meta.switch_local.port_in);
            meta.switch_local.ingresspackage = meta.switch_local.ingresspackage + 1;
            ingresscount.write((bit<32>)meta.switch_local.port_in, meta.switch_local.ingresspackage);
            //Count bytes
            bytecountingress.read(meta.switch_local.ingressbyte, (bit<32>)meta.switch_local.port_in);
            meta.switch_local.ingressbyte = meta.switch_local.ingressbyte + (bit<32>)standard_metadata.packet_length;
            bytecountingress.write((bit<32>)meta.switch_local.port_in, meta.switch_local.ingressbyte);
        }
        if(hdr.ipv4.typeServiceDiffServ == DSCP_INT){
                ingresscount.write((bit<32>)meta.switch_local.port_in, 0);
                bytecountingress.write((bit<32>)meta.switch_local.port_in, 0);
        }
    }
}
```

ANEXO C – CÓDIGO *EGRESS SWITCH* 1 E *SWITCH* 2

```
/*************************************************************************
****************  E G R E S S     **************************************
*************************************************************************/
control MyEgress(inout headers hdr, inout metadata meta, inout standard_metadata_t standard_metadata) {
    action action_switch_id(){
        hdr.switch_id.setValid();
        hdr.switch_id.int_switch_id = (bit<32>)meta.switch_local.switch_id;
        hdr.shimINT.shim_length = hdr.shimINT.shim_length + (bit<8>)INT_SWITCH_SIZE;
    } //bit 0
    action action_ingress_egress_ports(){
        hdr.int_ingress_egress_ports.setValid();
        hdr.int_ingress_egress_ports.int_ingress_id = meta.switch_local.port_in;
        hdr.int_ingress_egress_ports.int_egress_id = meta.switch_local.port_out;
        hdr.shimINT.shim_length = hdr.shimINT.shim_length + (bit<8>)INT_INGRESS_EGRESS;
    }  //bit 1
    action action_hop_latency(){
        meta.switch_local.egresststamp = meta.intrinsic_metadata.current_global_tstamp;
        hdr.hop_latency.setValid();
        hdr.hop_latency.int_hop_latency = ((meta.switch_local.egresststamp) - (meta.switch_local.ingresststamp));
        hdr.shimINT.shim_length = hdr.shimINT.shim_length + (bit<8>)INT_HOP_SIZE;
    }  //bit 2
    action action_ingress_tstamp(){
        hdr.ingressTimestamp.setValid();
        hdr.ingressTimestamp.int_ingressTimestamp = meta.switch_local.ingresststamp;
        hdr.shimINT.shim_length = hdr.shimINT.shim_length + (bit<8>)INT_IN_TIMESTAMP_SIZE;
    }  //bit 3
    action action_egress_tstamp(){
        hdr.egressTimestamp.setValid();
        hdr.egressTimestamp.int_egressTimestamp = meta.switch_local.egresststamp;
        hdr.shimINT.shim_length = hdr.shimINT.shim_length + (bit<8>) INT_OUT_TIMESTAMP_SIZE;
    }  //bit 4
    action action_countpackage(){
        hdr.countpackage.setValid();
        hdr.countpackage.int_ingresscountpackage = meta.switch_local.ingresspackage;
        hdr.shimINT.shim_length = hdr.shimINT.shim_length + (bit<8>) INT_COUNT_PACKAGE_SIZE;
    }  //bit 5
    action action_bytepackage(){
        hdr.bytepackage.setValid();
        hdr.bytepackage.int_bytepackageingress = meta.switch_local.ingressbyte;
        hdr.shimINT.shim_length = hdr.shimINT.shim_length + (bit<8>) INT_BYTE_PACKAGE_SIZE;
    } //bit 6
    action action_eq128(){
        action_switch_id();
        hdr.hopINT.int_total_hops = hdr.hopINT.int_total_hops + 1;
        hdr.ipv4.totalLength = hdr.ipv4.totalLength + (bit<16>)hdr.shimINT.shim_length;
    }
    action action_eq192(){
        action_switch_id();
        action_ingress_egress_ports();
        hdr.hopINT.int_total_hops = hdr.hopINT.int_total_hops + 1;
        hdr.ipv4.totalLength = hdr.ipv4.totalLength + (bit<16>)hdr.shimINT.shim_length;
    }
    action action_eq224(){
        action_switch_id();
        action_ingress_egress_ports();
        action_hop_latency();
        hdr.hopINT.int_total_hops = hdr.hopINT.int_total_hops + 1;
        hdr.ipv4.totalLength = hdr.ipv4.totalLength + (bit<16>)hdr.shimINT.shim_length;
    }
    action action_eq240(){
        action_switch_id();
        action_ingress_egress_ports();
        action_hop_latency();
        action_ingress_tstamp();
        hdr.hopINT.int_total_hops = hdr.hopINT.int_total_hops + 1;
        hdr.ipv4.totalLength = hdr.ipv4.totalLength + (bit<16>)hdr.shimINT.shim_length;
    }
    action action_eq248(){
        action_switch_id();
        action_ingress_egress_ports();
        action_hop_latency();
        action_ingress_tstamp();
        action_egress_tstamp();
        hdr.hopINT.int_total_hops = hdr.hopINT.int_total_hops + 1;
        hdr.ipv4.totalLength = hdr.ipv4.totalLength + (bit<16>)hdr.shimINT.shim_length;
```

```
    }
    action action_eq252(){
        action_switch_id();
        action_ingress_egress_ports();
        action_hop_latency();
        action_ingress_tstamp();
        action_egress_tstamp();
        action_countpackage();
        hdr.hopINT.int_total_hops = hdr.hopINT.int_total_hops + 1;
        hdr.ipv4.totalLength = hdr.ipv4.totalLength + (bit<16>)hdr.shimINT.shim_length;
    }
    action action_eq254(){
        action_switch_id();
        action_ingress_egress_ports();
        action_hop_latency();
        action_ingress_tstamp();
        action_egress_tstamp();
        action_countpackage();
        action_bytepackage();
        hdr.hopINT.int_total_hops = hdr.hopINT.int_total_hops + 1;
        hdr.ipv4.totalLength = hdr.ipv4.totalLength + (bit<16>)hdr.shimINT.shim_length;
    }
    action action_eq160(){
        action_switch_id();
        action_hop_latency();
        hdr.hopINT.int_total_hops = hdr.hopINT.int_total_hops + 1;
        hdr.ipv4.totalLength = hdr.ipv4.totalLength + (bit<16>)hdr.shimINT.shim_length;
    }
    action action_eq184(){
        action_switch_id();
        action_hop_latency();
        action_ingress_tstamp();
        action_egress_tstamp();
        hdr.hopINT.int_total_hops = hdr.hopINT.int_total_hops + 1;
        hdr.ipv4.totalLength = hdr.ipv4.totalLength + (bit<16>)hdr.shimINT.shim_length;
    }
    action action_eq88(){
        action_ingress_egress_ports();
        action_ingress_tstamp();
        action_egress_tstamp();
        hdr.hopINT.int_total_hops = hdr.hopINT.int_total_hops + 1;
        hdr.ipv4.totalLength = hdr.ipv4.totalLength + (bit<16>)hdr.shimINT.shim_length;
    }
    table add_metadata {
        key = {
            meta.switch_local.instruction: exact;
        }
        actions = {
            action_eq128;
            action_eq192;
            action_eq224;
            action_eq240;
            action_eq248;
            action_eq254;
            action_eq252;
            action_eq160;
            action_eq184;
            action_eq88;
        }
    }
    apply {
        if(hdr.report_ipv4.typeServiceDiffServ == DSCP_INT){
            add_metadata.apply();
        }
    }
}
```

UPF

UNIVERSIDADE
DE PASSO FUNDO

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