UNIVERSIDADE FEDERAL DE SANTA MARIA
CENTRO DE TECNOLOGIA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Leonardo da Cruz Marcuzzo

# UMA ARQUITETURA PARA O OFFLOAD PARCIAL DE FUNÇÕES VIRTUALIZADAS DE REDE EM PLANO DE DADOS PROGRAMÁVEL

Santa Maria, RS
2020

**Leonardo da Cruz Marcuzzo**

# UMA ARQUITETURA PARA O OFFLOAD PARCIAL DE FUNÇÕES VIRTUALIZADAS DE REDE EM PLANO DE DADOS PROGRAMÁVEL

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação (PPGCC) da Universidade Federal de Santa Maria (UFSM, RS), como requisito parcial para obtenção do título de **Mestre em Ciência da Computação**.

Orientador: Prof. Dr. Carlos Raniery Paula dos Santos

Santa Maria, RS

2020

Marcuzzo, Leonardo da Cruz

Uma Arquitetura para o Offload Parcial de Funções Virtualizadas de Rede em Plano de Dados Programável / por Leonardo da Cruz Marcuzzo. – 2020.
76 f.: il.; 30 cm.

Orientador: Carlos Raniery Paula dos Santos
Dissertação (Mestrado) - Universidade Federal de Santa Maria, Centro de Tecnologia, Pós-Graduação em Ciência da Computação , RS, 2020.

1. NFV. 2. PDP. 3. P4. 4. Offloading. I. dos Santos, Carlos Raniery Paula. II.Uma Arquitetura para o Offload Parcial de Funções Virtualizadas de Rede em Plano de Dados Programável.

---

© 2020
Todos os direitos autorais reservados a Leonardo da Cruz Marcuzzo. A reprodução de partes ou do todo deste trabalho só poderá ser feita mediante a citação da fonte.
E-mail: lmarcuzzo@inf.ufsm.br

**Leonardo da Cruz Marcuzzo**

# UMA ARQUITETURA PARA O OFFLOAD PARCIAL DE FUNÇÕES VIRTUALIZADAS DE REDE EM PLANO DE DADOS PROGRAMÁVEL

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação (PPGCC) da Universidade Federal de Santa Maria (UFSM, RS), como requisito parcial para obtenção do título de **Mestre em Ciência da Computação**.

**Aprovado em 28 de Agosto de 2020:**

**Carlos Raniery Paula dos Santos, Dr. (UFSM)**
(Presidente/Orientador)

**Raul Ceretta Nunes, Dr. (UFSM)**

Prof. Elias P. Duarte Jr
Depto. de Informática
Mat.: 105210 - UFPR

**Elias Procópio Duarte Júnior, Dr. (UFPR)**

Santa Maria, RS

2020

# DEDICATÓRIA

*Dedico este trabalho aos meus pais, Ivalci e Vanilda, a minha irmã Isabella, e a todos que me ajudaram de alguma forma durante estes anos de mestrado.*

# AGRADECIMENTOS

Inicialmente agradeço aos meus pais, Ivalci João Marcuzzo e Vanilda da Cruz Marcuzzo, por sempre me incentivarem e apoiarem, principalmente nos momentos mais difíceis. Agradeço também a minha irmã Isabella da Cruz Marcuzzo pela paciência, pelas conversas, e também pelo apoio durante estes anos que moramos juntos.

Agradeço ao meu orientador, professor Carlos Raniery, pela paciência, pelo auxílio na resolução de problemas difíceis, pela disponibilidade para discussões, pelas novas ideias, e por sempre me incentivar a continuar fazendo um trabalho de qualidade.

Agradeço aos meus colegas do CPD, Alexandre, Fábio, Jéssica, Lucimara, e a toda divisão de suporte, por me proporcionarem um bom ambiente de trabalho onde me sinto confortável para desenvolver minhas atividades. Também agradeço aos meus colegas de trabalho do INPE, Lucas Powaczuk e Tiago Portilho, pelo companheirismo, discussões e ideias.

Agradeço aos colegas do GRECA, Vinícius, Tavares, Anderson, Nilton e Gabriel, pelas discussões e auxílio, e também aos colegas do GT-FENDE, Cassiano, Giovanni, José, Lucas e Muriel, por me ajudarem a crescer e aprender a trabalhar em equipe.

Agradeço aos meus colegas do ensino médio (e irmãos deles) Bernardo, Enrique, Fabiano, Ike, Luciano, Balboni e Zead, que até hoje continuam bons amigos e companheiros, mesmo seguindo caminhos diferentes. Também agradeço aos novos amigos que fui conhecendo nestes anos, Fernando, Freddy e todos aqueles que de alguma forma foram companheiros de jogos e diversão.

Por fim, agradeço aos professores da banca, professor Elias Duarte e professor Raul Ceretta, pelos comentários e críticas que contribuíram para que esta dissertação se tornasse ainda melhor. À Universidade Federal de Santa Maria (UFSM) por me proporcionar um ensino de qualidade desde a graduação, e a todos aqueles que contribuíram de alguma forma.

#### RESUMO

### UMA ARQUITETURA PARA O OFFLOAD PARCIAL DE FUNÇÕES VIRTUALIZADAS DE REDE EM PLANO DE DADOS PROGRAMÁVEL

AUTOR: LEONARDO DA CRUZ MARCUZZO
ORIENTADOR: CARLOS RANIERY PAULA DOS SANTOS

O aumento na quantidade de usuários e dispositivos conectados na Internet vem trazendo desafios cada vez maiores para os provedores de serviço. O paradigma de Virtualização de Funções de Rede, que têm por objetivo desacoplar as funcionalidades de rede de seu *hardware* proprietário, e executá-las em servidores de virtualização, apresenta-se como uma nova forma de projetar redes, permitindo maior flexibilidade e melhor uso dos recursos da infraestrutura. No entanto, existem aspectos relacionados a estes paradigmas que dificultam sua adoção, como o seu desempenho, que ainda não é comparável ao de *middleboxes*, e a segurança na execução das funções de rede. Assim, técnicas para mitigar estes problemas vem sendo estudadas. Uma das técnicas é o *offload* de funções de rede, que consiste em executar parte da função em outro dispositivo, como um *switch* programável, onde é realizado um pré-processamento dos pacotes enviados para a VNF. Além de trazer um melhor desempenho, o *offload* também traz benefícios como a flexibilidade na execução da função, que pode ser implantada em mais pontos da infraestrutura, e também maior segurança, podendo reduzir a superfície de ataque da função. Embora apresente claros benefícios, a implementação desta técnica é complexa, de modo que atualmente não existe uma arquitetura capaz de realizar o *offload* de elementos de uma função de rede virtualizada para dispositivos programáveis. Nesta dissertação é proposta uma arquitetura para o *offload* de funções virtualizadas de rede para o plano de dados programável. Esta arquitetura é composta por dois componentes, uma plataforma de funções de rede com suporte a *offload*, e um gerenciador que configura as infraestruturas para realizar o processo. Um protótipo da arquitetura proposta também foi implementado e avaliado, demonstrando o funcionamento da arquitetura e do processo de *offload* propostos.

**Palavras-chave:** NFV. PDP. P4. Offloading.

# ABSTRACT

## AN ARCHITECTURE FOR OFFLOADING VIRTUALIZED NETWORK FUNCTIONS INTO THE PROGRAMMABLE DATA PLANE

AUTHOR: LEONARDO DA CRUZ MARCUZZO
ADVISOR: CARLOS RANIERY PAULA DOS SANTOS

The increase of users and devices connected to the Internet has brought increasing challenges for service providers. The Network Functions Virtualization paradigm, whose objective is to decouple network functions from the underlying hardware and executing them on virtualization servers allows for greater flexibility and better use of infrastructure resources. However, there is few aspects which hinders the adoption of this new paradigm, such as performance, which is not yet comparable of middleboxes, as well as security concerns on the execution of the functions. Thus, techniques to mitigate this loss of performance are emerging. One of the techniques is the offloading of network functions where a part of the function runs on a programmable device before or after the main CPU, pre-processing the packets sent to the VNF. Besides the better performance, offload also brings benefits related to the flexibility on the execution of the function, which can run on more devices, as well as security, reducing the attack surface of the function. Although this represents clear benefits, its implementation is complex, so that currently there is no architecture capable of performing the offload of elements of a virtualized network function into programmable devices. In this dissertation an architecture is proposed for the offload of virtualized network functions into the programmable data plan. This architecture is composed of two components, a network function platform capable of supporting offload, and a manager that configures the infrastructures to carry out the process. A prototype of the proposed architecture was also implemented and evaluated, demonstrating the operation of the architecture and the offload proposal.

**Keywords:** NFV. PDP. P4. Offloading.

# LISTA DE FIGURAS

Figura 1 – Arquitetura de alto nível do paradigma NFV. . . . . . . 18
Figura 2 – Arquitetura SDN . . . . . . . 22
Figura 3 – Configuração de um dispositivo programável. . . . . . . 27
Figura 4 – Visão interna de uma VNF . . . . . . . 31
Figura 5 – Exemplos de funções de rede. . . . . . . 32
Figura 6 – Possíveis locais onde elementos de uma função podem ser executados em cenários de *offload*. . . . . . . 35
Figura 7 – Arquitetura da VNF proposta . . . . . . . 45
Figura 8 – Arquitetura do gerenciador de Offload . . . . . . . 49
Figura 9 – Configuração de *Offload* passada ao agente. . . . . . . 57
Figura 10 – Exemplo de um classificador TCP implementado em P4. . . . . . . 59
Figura 11 – Tabela adicionada ao *pipeline* P4. . . . . . . 60
Figura 12 – Cenário de teste de desempenho . . . . . . . 66
Figura 13 – Configuração Click antes e após o processo de *offload*. . . . . . . 67
Figura 14 – Testes de vazão para diferentes tamanhos de pacotes. . . . . . . 68
Figura 15 – Latência para protocolos VoIP e Telnet . . . . . . . 68

# LISTA DE TABELAS

Tabela 1 – Características principais dos trabalhos relacionados. . . . . . . . . . . . . . . . . . . . . . . 42

# LISTA DE ABREVIATURAS E SIGLAS

| | |
|---|---|
| API | *Application Programming Interface* |
| ARP | *Address Resolution Protocol* |
| ASIC | *Application-Specific Integrated Circuit* |
| CAM | *Content-Addressable Memory* |
| DPDK | *Data Plane Development Kit* |
| DNS | *Domain Name System* |
| DMA | *Direct Memory Access* |
| eBPF | *Extended Berkeley Packet Filter* |
| EMS | *Element Management System* |
| ETSI | *European Telecommunications Standards Institute* |
| FPGA | *Field-Programmable Gate Array* |
| GPU | *Graphics Processing Unit* |
| HLS | *High-Level Synthesis* |
| HTTP | *HyperText Transfer Protocol* |
| HTTPS | *HyperText Transfer Protocol Secure* |
| ICMP | *Internet Control Message Protocol* |
| ISG NFV | *Industry Specification Group for NFV* |
| IoT | *Internet of Things* |
| IOMMU | *Input–Output Memory Management Unit* |
| IP | *Internet Protocol* |
| MANO | *Management and Orchestration* |
| MTU | *Maximum Transmission Unit* |
| NAT | *Network Address Translation* |
| NF | *Network Function* |
| NFV | *Network Function Virtualization* |
| NFVI | *Network Function Virtualization Infrastructure* |
| NFVO | *Network Function Virtualization Orchestrator* |
| NGFW | *Next-generation Firewall* |
| NSH | *Network Service Header* |
| ONF | *Open Network Foundation* |
| PDP | *Programmable Data Plane* |
| PNF | *Physical Network Function* |

| | |
|---|---|
| RMT | *Reconfigurable Match Tables* |
| REST | *Representational State Transfer* |
| SDN | *Software Defined Network* |
| SDK | *Software Development Kit* |
| SFC | *Service Function Chain* |
| SNMP | *Simple Network Management Protocol* |
| SR-IOV | *Single-Root Input/Output Virtualization* |
| SWA | *Software Architecture* |
| TCAM | *Ternary Content-Addressable Memory* |
| TCP | *Transmission Control Protocol* |
| UDP | *User Datagram Protocol* |
| VIM | *Virtualized Infrastructure Manager* |
| VNF | *Virtualized Network Function* |
| VNFC | *Virtualized Network Function Component* |
| VNFM | *Virtualized Network Function Manager* |
| VPN | *Virtual Private Network* |
| XML | *Extensible Markup Language* |

# SUMÁRIO

**1 INTRODUÇÃO** ........ 13
**2 REVISÃO DE LITERATURA** ........ 17
2.1 VIRTUALIZAÇÃO DE FUNÇÕES DE REDE ........ 17
2.2 REDES DEFINIDAS POR SOFTWARE ........ 21
2.3 PLANO DE DADOS PROGRAMÁVEIS ........ 24
2.4 DISCUSSÃO ........ 28
**3 OFFLOADING DE FUNÇÕES VIRTUALIZADAS DE REDE** ........ 30
3.1 ARQUITETURA INTERNA DE VNFS ........ 30
3.2 OFFLOADING ........ 33
3.3 TRABALHOS RELACIONADOS ........ 37
**3.3.1 Plataformas de VNFs com suporte a *offload*** ........ 37
**3.3.2 Orquestração de *offload* em NFV** ........ 39
3.4 DISCUSSÃO ........ 41
**4 ARQUITETURA DE *OFFLOAD* PARCIAL DE VNFS** ........ 44
4.1 ARQUITETURA DA VNF COM SUPORTE A OFFLOAD ........ 44
**4.1.1 *Offload Agent*** ........ 46
4.2 ARQUITETURA DO GERENCIADOR DE OFFLOAD ........ 48
4.3 DISCUSSÃO E LIMITAÇÕES ........ 51
**5 DESENVOLVIMENTO DO PROTÓTIPO** ........ 53
5.1 PLATAFORMA DE VNF ........ 53
**5.1.1 Offload Agent** ........ 54
5.2 *FRAMEWORK* DE *OFFLOAD* ........ 56
**5.2.1 Módulo de Usuário** ........ 56
**5.2.2 Módulo de Tradução e Banco de Dados de Elementos** ........ 58
**5.2.3 Módulos de comunicação** ........ 60
**5.2.4 Módulo de gerenciamento de *offload*** ........ 62
**6 AVALIAÇÃO E VALIDAÇÃO** ........ 63
6.1 MÉTRICAS DE AVALIAÇÃO ........ 63
6.2 FERRAMENTAS E AMBIENTE DE TESTES ........ 64
**6.2.1 FOP4** ........ 64
**6.2.2 ONOS** ........ 65
**6.2.3 Ferramentas de medição e monitoramento** ........ 65
6.3 CENÁRIO DE AVALIAÇÃO E METODOLOGIA ........ 65
6.4 RESULTADOS ........ 67
6.5 DISCUSSÃO ........ 70
**7 CONCLUSÃO** ........ 71
**REFERÊNCIAS** ........ 73

# 1 INTRODUÇÃO

O crescente número de usuários e dispositivos com os mais variados perfis de uso, em conjunto com o desenvolvimento e implantação de novas tecnologias requerem que redes de computadores sejam capazes de suportar cenários e demandas cada vez mais complexos. Funções de Rede (*Network Functions* - NF) desempenham um papel fundamental nestas infraestruturas, sendo responsáveis por executar diversas tarefas especializadas, desde as mais simples, como *firewalls* e roteamento, até mais avançadas, como *Virtual Private Networks - VPN* e *Next-Generation Firewalls - NGFW*. Tradicionalmente, estas funções de rede são implementadas através de um conjunto de *hardware* e *software* altamente especializado e muitas vezes proprietário, conhecidos como *middleboxes*, instalados fisicamente em pontos específicos da infraestrutura (WALFISH et al., 2004).

*Middleboxes* compõem uma grande parte das redes atuais, e embora apresentem benefícios como alto desempenho e segurança, sua utilização torna as redes inflexíveis, devido à dificuldade na implantação de novos protocolos ou tecnologias causado pelo seu alto *time-to-market*, a necessidade de ferramentas proprietárias para gerenciamento e configuração, assim como sua escalabilidade limitada, resultando em um alto custo operacional e de capital (SHERRY et al., 2012). Novas tecnologias e paradigmas, como redes 5G, computação em nuvem e *Internet of Things - IoT*, requerem infraestruturas cada vez mais flexíveis, escaláveis e programáveis, e diversas pesquisas vem sendo desenvolvidas com o objetivo de atender a estes requisitos (Akpakwu et al., 2018).

O paradigma de Virtualização de Função de Redes (*Network Function Virtualization* - NFV) (ETSI, 2012) busca desacoplar as funções de rede do seu *hardware* proprietário e executá-las utilizando técnicas de virtualização já existentes, através de servidores *off-the-shelf*. Funções de rede virtualizadas (*Virtual Network Function* - VNF) apresentam diversas vantagens com relação a *middleboxes*, como a interoperabilidade entre funções de rede de fabricantes distintos, a agilidade no suporte e implantação de novos protocolos, escalabilidade horizontal e vertical, e um melhor uso dos recursos computacionais (Hawilo et al., 2014), já que várias funções podem ser implantadas em um mesmo servidor de virtualização, reduzindo custos operacionais e de capital. VNFs também permitem uma implementação de forma modular, permitindo sua reutilização entre diferentes funções.

Já o paradigma de Redes Definidas por Software (*Software Defined Network* - SDN)

foca-se nos outros dispositivos que compõe a infraestrutura de rede, como *switches* e roteadores, desacoplando o plano de controle do plano de dados, de modo que os dispositivos apenas encaminham os pacotes de acordo com instruções recebidas de um controlador onde a lógica da rede é centralizada, através de um protocolo de comunicação (*e.g.*, OpenFlow) (Kreutz et al., 2015). A centralização da lógica da rede fornece um ponto único de gerenciamento e configuração, que pode ser utilizado tanto por operadores de rede como ferramentas de orquestração, facilitando a implantação de regras e políticas de forma global na rede.

Uma extensão do paradigma SDN que surgiu recentemente é o Plano de Dados Programáveis (*Programmable Data Planes* - PDP), que tem como objetivo permitir a reprogramabilidade do plano de dados de dispositivos de rede. Tradicionalmente, este plano de dados utiliza uma abstração chamada de *match+action*, que define um *pipeline* de comparações e ações estáticas a serem tomadas para os pacotes processados pelo dispositivo. Em dispositivos programáveis, estas ações, além de regras de *parsing* e *deparsing* de pacotes podem ser reconfiguradas através de uma linguagem de programação de alto nível (*e.g.,* P4 (BOSSHART et al., 2014)), permitindo o suporte a novos protocolos e ações a serem tomadas, de modo que o plano de dados do dispositivo pode se adequar as necessidades específicas da infraestrutura (BIFULCO; RÉTVÁRI, 2018).

Apesar destes paradigmas apresentarem diversos benefícios e se mostrarem necessários em redes de nova geração, sua adoção ainda é limitada, pois existem diversos problemas em aberto relacionados a questões como desempenho, orquestração, e sua utilização em conjunto de forma efetiva. Um dos principais desafios é relacionado ao desempenho das funções de rede virtualizadas, pois tecnologias existentes de virtualização não foram desenvolvidas com foco em alto desempenho na entrada e saída de dados, de modo que técnicas auxiliares se fazem necessárias para mitigar a perda de desempenho ocorrida (Mijumbi et al., 2016). Estas técnicas incluem implementações em *software*, como otimizações de código ou melhor uso de processadores genéricos e otimizações auxiliadas por *hardware*, como o *offload* parcial da função.

O *offloading* parcial da função consiste em delegar a execução de parte da função de rede para um outro processador especializado, como, por exemplo, interfaces de rede programáveis (*i.e,* SmartNICs) ou *switches* programáveis, reduzindo o consumo de recursos do processador principal da função, e aproveitando o alto desempenho dos dispositivos programáveis para executar tarefas tradicionalmente associadas ao processamento de pacotes, deixando tarefas mais

complexas (com alto uso de memória ou disco) para serem executadas no processador principal. Antes do tráfego chegar a função de rede virtualizada (chegar na máquina virtual ou *container*), uma parte da função executa, por exemplo, em um dispositivo que esteja no caminho, como um *switch* programável, ou até mesmo uma interface de rede, onde é realizado um pré-processamento dos pacotes que serão enviados para a VNF. Além dos benefícios com relação ao desempenho das funções de rede, o *offload* também permite uma maior flexibilidade na instanciação das funções de rede, permitindo sua execução em uma maior gama de dispositivos, como também pode trazer benefícios relacionados a segurança das funções de rede.

Embora o *offloading* de funções permita uma melhor utilização dos recursos da infraestrutura de rede e alto desempenho, sua implementação é complexa, dado que é necessária uma forma de orquestração entre as funções de rede e os dispositivos programáveis (YAMAZAKI et al., 2014), além da necessidade de ferramentas específicas para o desenvolvimento das funções de rede em si e de versões para diferentes arquiteturas, através de linguagens como P4 ou eBPF [1]. Esforços atuais que permitem o *offload* de funções, como Metron (BARBETTE et al., 2018) e E2 (PALKAR et al., 2015) são focados em Cadeias de Funções de Serviços (*Service Function Chaining* - SFC) ou limitam-se a regras de encaminhamento, enquanto que outros trabalhos (LI et al., 2016) (RINTA-AHO; KARLSTEDT; DESAI, 2012) fornecem implementações de funções de rede com suporte a *offload* para *Field Programmable Gate Array - FPGAs*, de forma que no momento não existe uma solução genérica orientada a elementos de uma VNF que possibilite a implantação de funções virtualizadas de rede com suporte a *offloading* no plano de dados programável.

Neste contexto, este trabalho tem como objetivo propor uma solução para a implementação de funções virtualizadas de rede com suporte a *offloading* parcial de elementos para dispositivos programáveis. Esta solução é composta por dois componentes principais: uma arquitetura de desenvolvimento de funções de rede virtualizadas com suporte a *offload*, e um gerenciador capaz de comunicar-se com a função de rede e a infraestrutura de dispositivos programáveis, e configurar o ambiente necessário para que o *offload* possa ocorrer. Além disso, foi realizada uma revisão de literatura para identificar o estado da arte em técnicas e plataformas com suporte a *offload*. Assim, as principais contribuições deste trabalho são as seguintes: (i) uma arquitetura genérica para o desenvolvimento de funções de rede com suporte a *offload*; (ii) uma revisão sobre arquiteturas de VNFs e soluções de *offload* existentes; (iii) um agente de *offload* para o Click

[1] https://www.iovisor.org/technology/ebpf

Modular Router e; (iv) um gerenciador capaz de se comunicar com VNFs e a infraestrutura de rede para configurar o ambiente necessário e realizar o processo de *offload*.

O restante desta dissertação está organizado da seguinte forma: No capítulo 2 é apresentada uma revisão de literatura, explorando conceitos de paradigmas necessários para o entendimento do restante da dissertação. No capítulo 3, inicialmente são apresentados e discutidos conceitos sobre arquiteturas de funções de rede, como sua modularidade, bem como porque é necessário e como pode ser realizado o *offload* destas funções, apresentando também trabalhos relacionados ao tema da dissertação. O capítulo 4 apresenta a arquitetura proposta, identificando a função de cada um dos componentes e seus pontos de comunicação. O capítulo 5 apresenta a implementação da plataforma de VNF com suporte a *offload*, e do gerenciador de *offload*. O capítulo 6 apresenta a metodologia utilizada para a avaliação, o cenário de testes, e uma discussão sobre os resultados obtidos. Por fim, o capítulo 7 apresenta as conclusões obtidas nesta dissertação e possíveis trabalhos futuros.

# 2 REVISÃO DE LITERATURA

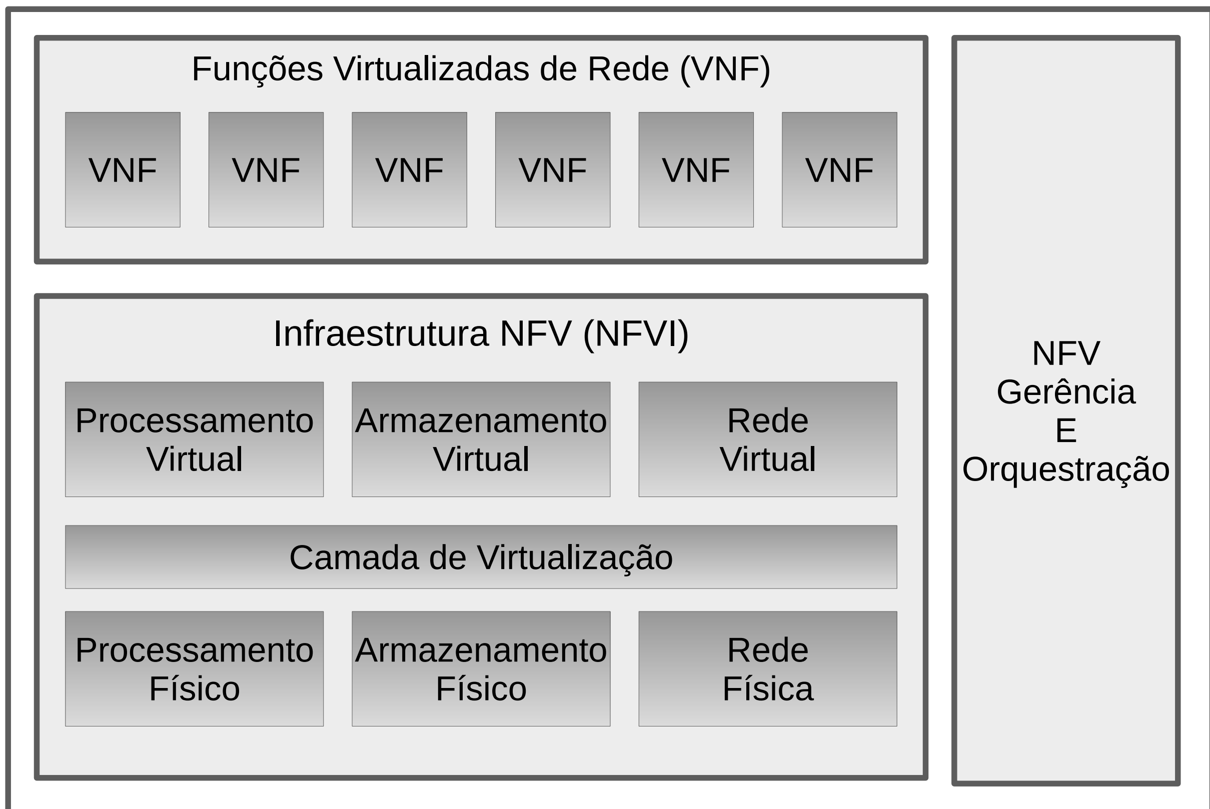
Este capítulo apresenta uma revisão de literatura sobre temas relevantes ao teor desta dissertação. Para que seja possível entender os benefícios trazidos ao realizar um *offload* de uma função de rede é necessário o conhecimento de alguns conceitos dos paradigmas de Virtualização de Funções de Rede (NFV), Redes Definidas por Software (SDN) e Plano de Dados Programáveis (PDP). Assim, é apresentada a motivação que levou ao desenvolvimento destes paradigmas, quais os benefícios trazidos por eles, e uma visão geral sobre sua arquitetura e principais conceitos. Após, é discutida a relação entre os paradigmas, e como sua utilização em conjunto é possível.

## 2.1 VIRTUALIZAÇÃO DE FUNÇÕES DE REDE

As infraestruturas de redes atuais dependem cada vez mais da utilização de *middleboxes* em sua topologia para atender as demandas dos usuários. *Middleboxes* consistem em um conjunto de *software* e *hardware* especializados para executar funções dentro da infraestrutura, e ganharam popularidade por apresentarem vantagens como alto desempenho e segurança. No entanto, a sua utilização em redes de computadores cada vez maiores e mais complexas resulta em diversos problemas, como aumento nos custos operacionais, sua inflexibilidade com relação ao suporte de novos protocolos e tecnologias, problemas de configuração e gerência de equipamentos de fabricantes diferentes, bem como dificuldades em escalabilidade vertical e horizontal (SHERRY et al., 2012). Além disso, com a adoção de novas tecnologias como 5G e IoT, além de novos serviços oferecidos por provedores de serviço, como *Cloud Computing*, redes precisam permitir um grau cada vez maior de programabilidade, e *middleboxes* impedem ou dificultam sua implantação (Yousaf et al., 2017).

O conceito de Virtualização de Funções de Rede (*Network Function Virtualization* - NFV) foi introduzido em 2012 através de um *white paper* escrito por membros de diversas empresas de telecomunicações e publicado pelo *European Telecommunications Standards Institute - ETSI*. Neste *white paper*, os autores descrevem os problemas decorrentes da utilização de *middleboxes* e apresentam a ideia de desacoplar o *software* das funções de rede do seu *hardware*, utilizando servidores e técnicas de virtualização já existentes para executar estas funções. Isto permite uma maior flexibilidade no desenvolvimento e manutenção das funções de rede, redu-

Figura 1 – Arquitetura de alto nível do paradigma NFV.




Fonte: do Autor.

zindo custos operacionais e de capital, e simplificando tarefas como manutenção e migração de funções (ETSI, 2012).

Assim, foi proposta uma arquitetura de referência para o paradigma NFV dividida em três grandes blocos funcionais: *NFV Infrastructure - NFVI*, *NFV Management e Orchestration - NFV MANO* e *Virtual Network Functions - VNF*. Cada um dos blocos é composto por uma série de componentes internos, trabalhando em conjunto para definir a funcionalidade do bloco. A Figura 1 apresenta uma visão geral desta arquitetura.

De acordo com a definição do ETSI (NFV-INF, 2015), o bloco NFVI representa a totalidade de recursos de *hardware* e *software* que compõem o ambiente onde VNFs são implantadas, como servidores de virtualização, armazenamento, dispositivos de rede e *hypervisors*. A implantação desta infraestrutura é realizada através de nós distribuídos em vários locais para suportar os requisitos de latência e localidade das VNFs. Como existem diversos casos de uso para VNFs, a infraestrutura deve ser capaz de se reconfigurar para atender os requisitos das funções. Devido à complexidade dos elementos representados neste bloco, ele divide-se em três domínios: *Compute Domain*, responsável por disponibilizar os recursos computacionais e de

armazenamento para o *hypervisor*, e recursos de rede físicos para o domínio de infraestrutura de rede; *Hypervisor Domain*, que deve mediar os recursos computacionais através de um *hypervisor* e transformar os recursos físicos em recursos virtualizados, disponibilizando máquinas virtuais, *containers* e interfaces de rede para as VNFs e; *Infrastructure Network Domain*, que permite a interconexão entre componentes de uma VNF distribuída ou entre diferente VNFs, a comunicação destas com sistemas de orquestração e gerência, bem como a conexão da infraestrutura com a rede externa.

Com a separação da função de rede do seu *hardware* proprietário, o gerenciamento desta função, antes localizado junto ao *hardware* precisa ser modificado, já que agora funções são executadas em uma infraestrutura genérica e compartilhada. Também são trazidas novas formas de conectar estas funções, como *links* virtuais, e o encadeamento destas funções para formar serviços mais complexos, conhecido como encadeamento de funções de rede (*Service Function Chaining* - SFC). Assim, o bloco funcional MANO (NFV-MAN, 2014) foi proposto pelo ETSI, com o papel de gerenciar a NFVI e orquestrar a alocação de recursos utilizados pelas VNFs. Além da definição de repositórios de dados, como catálogos de VNFs e informações sobre instâncias e recursos computacionais, três componentes principais de gerenciamento são definidos no MANO: o *Virtualised Infrastructure Manager* (VIM) deve realizar o controle e gerenciamento dos recursos disponibilizados pela NFVI, orquestrar a alocação e recuperação de recursos computacionais, gerenciar a associação dos recursos físicos aos virtualizados e também dar suporte ao gerenciamento de SFCs, criando e mantendo *links* e sub-redes virtualizadas; o *VNF Manager* é responsável pelo ciclo de vida das instâncias de VNFs, como sua instanciação e destruição, configuração e atualização, *scaling in and out* e recuperação de métricas. Por fim, o *NFV Orchestrator* realiza a orquestração e validação dos recursos da NFVI entre múltiplos VIMs, o ciclo de vida e gerenciamento da topologia de serviços de rede compostas por várias VNFs, a instanciação e gerenciamento dos VNFMs e gerenciamento de políticas dos serviços de rede.

O último bloco funcional da arquitetura define uma *Virtualised Network Function* - VNF como uma função de rede capaz de ser executada na NFVI e gerenciada pelo MANO (NFV-SWA, 2014). Uma VNF consiste em um conjunto de *software* desenvolvido com o objetivo de processar e encaminhar pacotes, podendo também ser uma função legada que foi virtualizada. De acordo com a arquitetura do ETSI, uma VNF pode ser constituída de vários elementos menores denominados *VNF Component* - VNFC. Cada VNFC executa uma função específica

dentro da VNF, como receber pacotes, realizar algum tipo de processamento ou se comunicar com o VNFM. Embora o conceito seja semelhante ao de VNFCs, o conceito de SFC refere-se exclusivamente ao encadeamento de VNFs completas, de modo que cada VNF pode funcionar individualmente, mas quando encadeada fornece um serviço mais avançado. VNFCs, por sua vez, não podem ser utilizados de maneira individual, pois normalmente executam apenas uma pequena função dentro da VNF, de forma que sua utilização é dependente de elementos anteriores ou posteriores para realizar parte do processamento. O bloco VNF também engloba um módulo específico chamado de *Element Management System - EMS*, que recebe ações de gerenciamento ou monitoramento do MANO e executa estas ações internamente na VNF. O desenvolvimento interno de VNFs não é padronizado, embora propostas neste sentido venham sendo estudadas. A definição de uma VNF é feita através de um VNF *package*, onde o desenvolvedor estrutura os VNFCs em uma ou mais imagens, e produz um VNF *descriptor* descrevendo a função, pontos de comunicação, bem como interfaces específicas de gerenciamento.

NFV apresenta uma arquitetura que engloba diversas necessidades e cenários dos operadores de rede, apresentando mapeamentos desde as camadas mais baixas da infraestrutura, até camadas de mais alto nível, como a própria definição das VNFs. Devido a isto, a utilização de NFV traz diversos benefícios relacionados ao gerenciamento e aproveitamento de recursos, facilitando a gerência para os operadores de rede, e permitindo uma visão mais precisa das necessidades do ambiente. Por representar uma mudança profunda na implantação de infraestruturas, ainda existem muitas áreas de pesquisa dentro do paradigma com problemas a serem resolvidos para sua utilização em ambientes de produção.

O problema de desempenho das funções de rede é de muita relevância para a adoção de NFV. *Middleboxes* criam diversos problemas dentro de uma rede de computadores, quebrando inclusive princípios da arquitetura da Internet, como o fato de que nós devem apenas processar os pacotes que são direcionados a eles, o que inviabilizaria tecnologias como *Network Address Translation - NAT* e *proxies*. No entanto, o desempenho de *middleboxes* se apresenta como um dos principais pontos para sua adoção, visto a necessidade cada vez maior de conexões de alta velocidade para usuários no mundo todo, bem como o oferecimento de novos serviços, como *streaming*, que requerem cada vez mais banda. Atualmente, grandes empresas de tecnologia têm, inclusive, desenvolvido *hardware* proprietário para utilização interna, pois os dispositivos disponíveis comercialmente ainda não são capazes de comportar as necessidades destas empresas (WIRED, 2015).

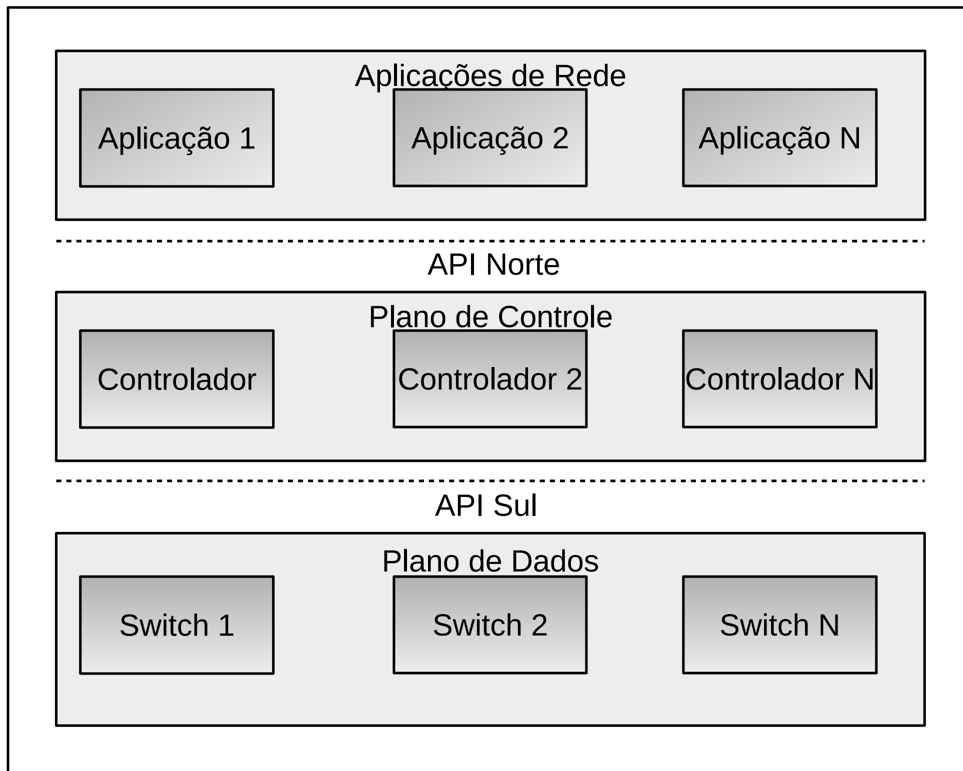
Considerando a importância do desempenho nas infraestruturas atuais, a adoção de NFV pode ser afetada gravemente ou até impossibilitada caso não seja possível um nível de desempenho pelo menos semelhante ao das infraestruturas atuais. Desta forma, diversos esforços vêm sendo direcionados para a resolução deste problema. O ETSI apresenta na visão geral da NFVI (NFV-INF, 2015) quatro desafios a serem vencidos para a implantação de NFV: (i) fatores de *hardware* e *software* que podem influenciar no desempenho; (ii) a interconexão entre as VNFs; (iii) a portabilidade trazida pela virtualização ao custo de desempenho e; (iv) o gerenciamento do ambiente onde as VNFs serão executadas. Devido à complexidade da arquitetura, e de seus diversos componentes, bem como a variedade de ferramentas que podem ser adotadas em NFV, existem diversos pontos onde otimizações podem influenciar no desempenho das VNFs. Na infraestrutura em si, por exemplo, a utilização de interfaces de rede e *switches* com maior capacidade e suporte a tecnologias como SDN ajuda a mitigar possíveis gargalos na comunicação entre dispositivos e servidores. Dentro dos servidores, por sua vez, a camada de virtualização é responsável por uma grande parte da perda de desempenho, assim como a própria pilha de rede do sistema operacional, e de componentes do servidor como memórias e CPU.

## 2.2 REDES DEFINIDAS POR SOFTWARE

Tradicionalmente, cada dispositivo de rede possui o plano de controle e o plano de dados integrados e executando em conjunto no *hardware* do dispositivo. Para realizar a configuração da rede, operadores devem configurar individualmente os dispositivos de rede ou utilizar ferramentas proprietárias de configuração, em um processo manual que pode ocasionar erros de configuração e que se torna inviável para grandes topologias. Visando contornar este problema, em (CASADO et al., 2007) apresentou-se uma proposta de arquitetura onde o plano de controle dos dispositivos era separado do *hardware* do equipamento (plano de dados) e executado em uma entidade externa. Este conceito inicial evoluiu para o paradigma de Redes Definidas por Software (*Software Defined Network* - SDN).

De acordo com a arquitetura definida pela *Open Networking Foundation* (FOUNDATION, 2014), o objetivo de SDN é prover interfaces abertas que permitam o desenvolvimento de sistemas capazes de controlar a conectividade provida por recursos de rede, bem como o fluxo de dados entre estes dispositivos. De modo geral, SDN possui três princípios arquiteturais: o plano de dados e o plano de controle dos dispositivos devem ser separados, o controle da topologia deve ser logicamente centralizado, e a arquitetura deve expor os recursos de rede

Figura 2 – Arquitetura SDN




Fonte: do Autor.

e estados de forma abstraída para aplicações externas.

Assim, a Figura 2 representa a arquitetura SDN dividida em três camadas. Na camada do plano de dados são representados os dispositivos de rede que compõem a infraestrutura, com seus respectivos recursos para encaminhamento e processamento dos fluxos de rede, que são abstraídos a partir da capacidade física dos dispositivos. Em SDN, os dispositivos de rede tornam-se simples encaminhadores de pacotes, expondo os seus recursos e recebendo regras do plano de controle através de um protocolo.

No plano de controle está localizado o controlador SDN, responsável por centralizar toda a lógica de controle dos dispositivos, disponibilizar informações sobre a topologia e receber configurações através de uma API Norte (*Northbound API*), por onde se comunica com aplicações de rede. Também é responsável por traduzir e implantar os comandos destas aplicações para os dispositivos da infraestrutura através de uma API Sul (*Southbound API*), onde utiliza um protocolo de comunicação padronizado, além de servir como um ponto único de comunicação com outras paradigmas, como NFV. Um controlador SDN é um serviço que pode ser implantado de diversas formas, como um processo executando em um servidor ou como um sistema operacional, neste caso também chamado de *Networking Operating System*, e de forma

distribuída, já que por centralizar toda a lógica da rede, o controlador se torna um ponto único de falha.

As aplicações de rede definem e implementam funcionalidades no controlador, como roteamento e configuração dos dispositivos. Em uma rede SDN, operadores de rede e aplicações não possuem acesso direto aos dispositivos, já que esta função é realizada pelo controlador. Um controlador por si só não possui nenhuma funcionalidade, sendo dependente de aplicações de rede para realizar a configuração da topologia, embora a maioria dos controladores já possuam diversas aplicações integradas, como *drivers* para dispositivos específicos ou suporte a protocolos bastante utilizados. Desenvolvedores, por sua vez, utilizam a API Norte para a comunicação das aplicações com o controlador, onde podem ser utilizadas abstrações que serão interpretadas pelo controlador e traduzidas para dispositivos específicos. Dado que o controlador abstrai a infraestrutura da rede, aplicações podem ser desenvolvidas orientadas a serviços ou políticas globais da rede. Assim, não é necessário conhecimento sobre o *hardware* ou APIs proprietárias dos dispositivos de rede, simplificando o desenvolvimento de aplicações capazes de suportar uma maior gama de dispositivos e permitindo uma maior flexibilidade na implantação de novas funções.

A comunicação entre o controlador SDN e os dispositivos é realizada através da API Sul utilizando um protocolo padronizado, sendo que atualmente o protocolo OpenFlow (MCKEOWN et al., 2008) é o padrão da indústria. O protocolo OpenFlow permite que o controlador modifique diretamente tabelas de encaminhamento dos dispositivos, também chamadas de *flow tables*, através de ações para adicionar, modificar ou remover correspondências nas tabelas. Estas tabelas são utilizadas pelo plano de dados para tomar decisões sobre os pacotes processados. O protocolo também permite que o dispositivo de rede, quando não encontra uma correspondência para o fluxo na sua tabela, solicite ao controlador uma ação a ser tomada. A padronização do protocolo de comunicação entre o controlador e os dispositivos permite o controle de infraestruturas heterogêneas, sendo que o ônus para suportar o protocolo recai sobre os fabricantes dos dispositivos. Novas aplicações podem ser desenvolvidas de forma mais genérica devido às abstrações do controlador, e seu processo de implantação resume-se a importar e ativa-las no controlador, não havendo necessidade de modificações nos dispositivos físicos.

A utilização de SDN traz diversos benefícios para os operadores de rede e provedores de serviços (FOUNDATION, 2012), como o controle centralizado de infraestruturas heterogêneas, compostas por dispositivos de vários fabricantes, uma redução na complexidade de configu-

ração da rede, devido à possibilidade de automação de tarefas manuais, maior agilidade na implantação de novas tecnologias, bem como um aumento na segurança e confiabilidade das infraestruturas.

De modo geral, os benefícios trazidos por SDN possuem pontos em comum com os trazidos por NFV. Embora em certos pontos os dois paradigmas se sobrepõem, sua utilização em conjunto traz benefícios a ambas arquiteturas. Por exemplo, a configuração de interconexões de SFCs pode ser realizada de forma autônoma se a infraestrutura de rede possuir suporte a SDN. O controlador SDN pode se comunicar ou até mesmo fazer parte do MANO, integrando também o controle da rede com o orquestrador NFV.

## 2.3 PLANO DE DADOS PROGRAMÁVEIS

O alto desempenho de *middleboxes* é relacionado ao seu desenvolvimento baseado no uso de circuitos integrados (ASICs) customizados para executarem instruções básicas relacionadas ao processamento de pacotes de forma extremamente rápida. Nestes circuitos, fabricantes definem, por exemplo, como será realizada a análise do cabeçalho dos pacotes (*parsing*) e quais protocolos serão suportados, qual o algoritmo a ser utilizado para encaminhar o pacote para uma porta específica (*forwarding*), como os pacotes serão organizados e enfileirados antes de serem encaminhados (*queuing* e *scheduling*), e quais ações serão tomadas de acordo com regras definidas pelo plano de controle (BIFULCO; RÉTVÁRI, 2018).

Considerando que o processo de projetar e fabricar circuitos é complexo e custoso, fabricantes costumam adicionar o suporte a novas funcionalidades apenas quando existe uma alta demanda por elas devido aos custos associados ao desenvolvimento. Também, funcionalidades e protocolos menos populares, mesmo quando não utilizados pelo operador de rede, consomem recursos da mesma forma, já que a adição ou alteração destas funcionalidades não é possível após o dispositivo entrar em uso. Associado a este circuito estão memórias de conteúdo endereçável (*Content-Addressable Memory* - CAM), onde ficam armazenadas tabelas com configurações definidas pelo operador de rede através do plano de controle.

Uma das formas de implementar este circuito é através de uma abstração chamada de *match+action pipeline*. Esta abstração descreve o plano de dados como uma sequência de *lookup tables* organizadas em uma estrutura hierárquica, onde campos pré-definidos do cabeçalho dos pacotes são utilizados nas tabelas para localizar uma ação de processamento correspondente. O desenvolvedor configura o comportamento do plano de dados através da modificação

das tabelas armazenadas na TCAM, adicionando, modificando ou removendo entradas na tabela através de uma API como o OpenFlow. O circuito realiza a leitura destas tabelas para decidir de que forma os pacotes serão tratados. Este circuito, em conjunto com os algoritmos e ações realizados, é chamado de plano de dados, ou ainda, *fast path*.

O paradigma SDN, ao separar o plano de controle do dispositivo e definir um protocolo padronizado para a comunicação com o plano de dados, permitiu uma maior liberdade no controle dos dispositivos de rede. Através do OpenFlow, operadores de rede configuram políticas e regras genéricas que podem ser aplicadas para qualquer plano de dados suportado, independente de sua arquitetura interna. No entanto, o operador de rede não é capaz de acessar e modificar diretamente o plano de dados, e para isso utiliza o plano de controle. O plano de controle, também chamado de *slow path*, permite gravar informações como regras de roteamento e encaminhamento, além de fazer a leitura de contadores.

Estes dispositivos de rede, mesmo com suporte a OpenFlow para controle, possuem o plano de dados estático, pois a interação com plano de controle é limitada a alterações em regras de encaminhamento, não sendo possível alterar a forma como é realizada a leitura dos cabeçalhos (e por consequência quais protocolos podem ser tratados) ou como as ações são aplicadas. O plano de dados estático também limita a usabilidade do próprio OpenFlow, que precisa deixar explícito na sua definição quais campos de cabeçalho e ações que devem ser suportadas, para que assim sejam implementados pelo fabricante.

Dadas as limitações impostas pelo uso de um plano de dados estático, o conceito de *Reconfigurable Match Tables* - RMT (BOSSHART et al., 2013) foi proposto. A utilização de RMT estende o conceito de *match+action pipeline* para que suporte múltiplas tabelas de tamanhos arbitrários, com cada tabela configurável para suportar campos de cabeçalho variáveis. Ao invés de estender o protocolo OpenFlow repetidamente com novos campos e funcionalidades, RMT permite a reconfiguração do plano de dados de quatro formas: a definição dos campos pode ser alterada e novos campos adicionados; o número, a hierarquia e o tamanho das tabelas podem ser especificados, limitado apenas pelos recursos físicos disponíveis; novas ações podem ser definidas e; pacotes modificados podem ser colocados em filas específicas, com algoritmos diferentes para cada fila. Através desta definição, os autores conseguem descrever uma arquitetura que é flexível e reconfigurável, mas ainda restrita o suficiente para ser implementada utilizando, por exemplo, *Field Programmable Gate Arrays* - FPGA, que possuem um desempenho inferior ao de ASICs, no entanto ainda muito superior quando comparado a CPUs genéricas ou *software*.

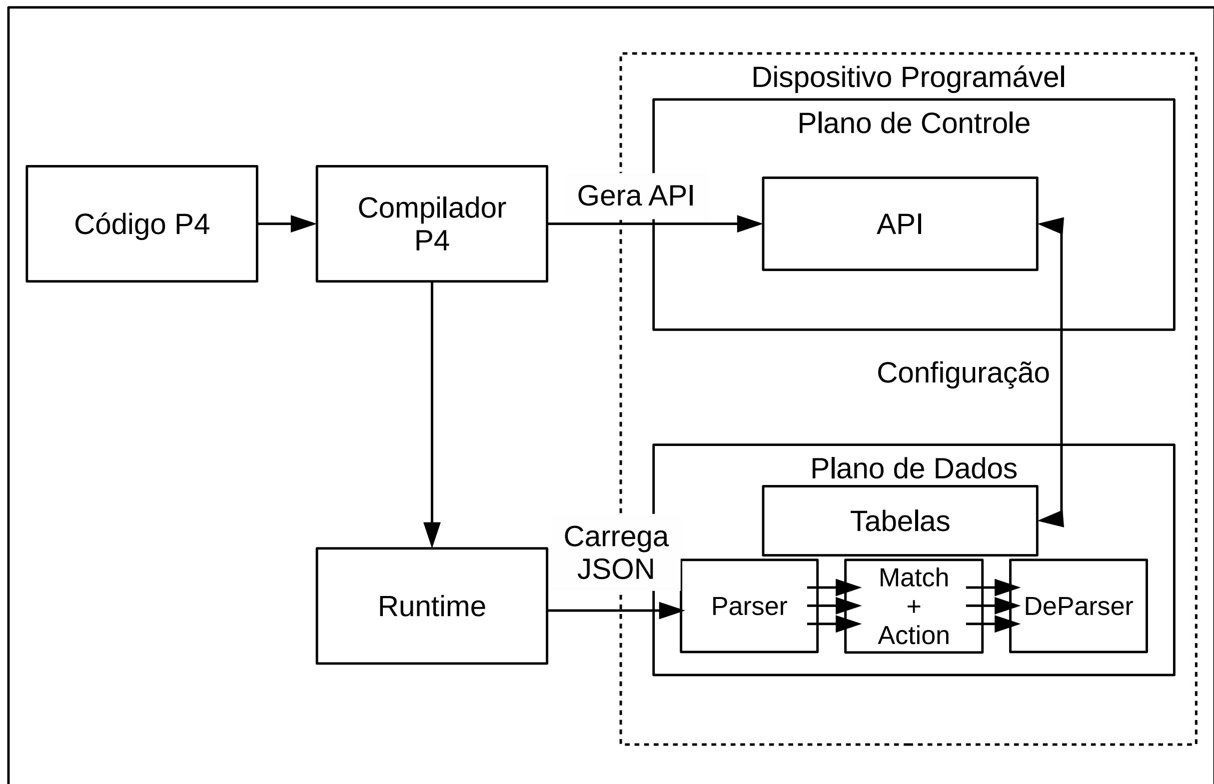
Enquanto o modelo RMT especifica uma arquitetura de plano de dados programável, a programação ainda precisava ser feita utilizando linguagens de baixo nível específicas dos circuitos. Com o objetivo de oferecer uma linguagem de alto nível para a programação destes dispositivos, a linguagem P4 foi especificada. P4 (BOSSHART et al., 2014) permite a descrição de processadores de pacotes através de abstrações e estruturas simples, capazes de expressar o funcionamento de um plano de dados genérico, buscando atingir três objetivos: permitir que o plano de dados seja reconfigurável, independência de protocolo, sendo capaz de suportar protocolos atuais e que venham a ser definidos, e independência de arquitetura, não sendo necessário conhecimento da arquitetura do *hardware* especializado para utilizá-lo.

A linguagem P4 é utilizada para desenvolvimento em cima do modelo de RMT, assumindo que o dispositivo suporte um *parser* programável, e que ações são compostas por primitivas independente de protocolos, mas suportadas pelo dispositivo. Isto permite que a linguagem não dependa de hardware ou arquitetura específicos dos dispositivos de rede, de modo que desenvolvedores podem criar programas genéricos, enquanto fabricantes devem disponibilizar um compilador para converter o código em P4 para a sua arquitetura. Além disso, fabricantes também podem disponibilizar extensões e APIs proprietárias que podem ser utilizadas em conjunto com P4 para estender suas funcionalidades.

A Figura 3 apresenta o processo de configuração de um dispositivo programável com suporte a P4. Inicialmente, um código fonte contendo informações referentes ao comportamento do plano de dados é compilado para a linguagem proprietária do dispositivo, utilizando um *backend* disponibilizado pelo fabricante.

A configuração do dispositivo é dividida em dois estágios: *Configure* e *Populate*. No primeiro estágio, é realizada a configuração do plano de dados, como a configuração do *parser*, a ordem dos estágios *match+action*, e quais cabeçalhos serão suportados. Um código P4 define elementos como um *parser*, onde o cabeçalho dos pacotes é lido e enviado para o *pipeline* correto, um ou mais *pipelines match+action*, onde são definidas ações a serem tomadas para os pacotes que encontram uma correspondência em *flow tables*, e um *deparser*, onde o pacote é remontado e enviado de volta para a rede. Nota-se que em plano de dados programáveis, embora blocos possam compartilhar metadados entre si, a arquitetura é tradicionalmente *stateless*, ou seja, metadados de pacotes anteriores não influenciam no processamento de novos pacotes, impossibilitando que funções necessariamente *stateful* possam ser implantadas apenas no plano de dados. O código com a definição destes elementos é então compilado em um formato inter-

Figura 3 – Configuração de um dispositivo programável.




Fonte: do Autor.

mediário e carregado no dispositivo. Este processo aplica-se apenas nos casos onde o plano de dados do dispositivo é programável, pois embora P4 apresente suporte para dispositivos estáticos, o estágio de compilação nestes casos consiste apenas em verificar se o dispositivo possui suporte as estruturas definidas no código.

O segundo estágio é relacionado à operação do dispositivo. Durante o primeiro estágio, o dispositivo não é capaz de processar pacotes. Após a configuração do plano de dados, o segundo estágio consiste nas operações tradicionalmente realizadas pelo plano de controle. O mesmo código que gerou o plano de dados também gera uma API para o plano de controle, de onde o operador de rede pode popular o conteúdo das tabelas, da mesma forma que se faz em um dispositivo tradicional. Esta mesma API comunica-se internamente com o plano de dados modificando o conteúdo das tabelas. Dispositivos programados em P4 são controlados por um controlador SDN, e embora possuam suporte para utilização de Openflow, limitações relacionadas ao projeto e extensibilidade do protocolo tornam sua utilização complexa e desencorajada.

Assim, foi desenvolvida a especificação do P4 Runtime [2], uma nova forma de comunicação entre controladores SDN e dispositivos programáveis. Quando o Openflow foi projetado,

[2] https://p4.org/p4-runtime/

dispositivos possuíam plano de dados estáticos, logo foi definido um protocolo que era capaz de configurar as tabelas de encaminhamento de forma remota baseando-se em planos de dados estáticos, ou seja, as ações a serem tomadas, bem como os protocolos a serem suportados foram definidos no projeto do Openflow. A principal diferença do P4 Runtime para o Openflow é que é possível controlar qualquer tipo de plano de dados implantado, com ações e protocolos customizados. Isto permite uma maior liberdade ao desenvolvedor na hora de projetar o plano de dados e aplicações SDN que interagem com ele.

A programabilidade do plano de dados, quando usada em conjunto com SDN e NFV permite que toda a infraestrutura de rede seja definida através de *software* e de forma agnóstica ao substrato físico onde está sendo executada. Projetistas de rede podem desenvolver e testar topologias de forma completamente virtualizada antes de colocá-las em produção. O plano de dados também pode ser utilizado para executar de forma parcial funções de rede complexas que antes necessitavam de processadores genéricos para serem executadas, afetando seu desempenho. Um exemplo do poder do plano de dados programável é a utilização de *in band-telemetry*. Ao invés dos dispositivos coletarem os dados de telemetria da rede e enviarem a um local centralizado utilizando um protocolo como SNMP ou Netflow, estes dados podem ser enviados junto com os fluxos de rede que estão passando pela infraestrutura para serem analisados por algum agente no caminho do tráfego, reduzindo assim o *overhead* causado por protocolos de gerenciamento.

## 2.4 DISCUSSÃO

Os paradigmas apresentados nesse capítulo possuem em comum o objetivo de trazer programabilidade e flexibilidade às infraestruturas independente de onde serão executadas, permitindo que topologias sejam projetadas e testadas de forma completamente virtualizada, para então serem implantadas em ambientes de produção. Para isso, são utilizadas tanto tecnologias já existentes, como é o caso da virtualização, e novas tecnologias propostas, como a programabilidade do plano de dados. Como estes paradigmas apresentam mudanças fundamentais na arquitetura de redes e infraestruturas, ainda existem diversos desafios que precisam ser resolvidos para que sua implantação seja possível.

Embora sejam dois paradigmas distintos, NFV e SDN apresentam benefícios para ambos paradigmas quando utilizados em conjunto. NFV possui em seu bloco funcional NFVI a definição de recursos de rede, que podem ser virtualizados ou não, e que devem ser gerenciados

pelo MANO. O MANO, por sua vez, define o conceito de *Network Controllers*, que permitem interfaces programáveis para configurar dispositivos de rede dentro da infraestrutura. De fato, quando houve a definição do paradigma NFV, já se esperava que SDN poderia agir como um *NFV Enabler*, uma tecnologia que auxilia a implantação de NFV.

Por permitir esta interface de programação e automação dos dispositivos de rede, o uso de SDN é fundamental para algumas vantagens que NFV traz, como a migração e *offloading* de funções, a escalabilidade horizontal e vertical, e a composição de serviços utilizando SFCs que dependem do substrato de rede suportar reconfiguração e automação dinâmicas. Em todos estes casos, é necessário um esforço em conjunto entre o VIM e o controlador, para que ao instanciar as VNFs, elas sejam conectadas corretamente entre si e com a rede externa.

O principal objetivo deste capítulo foi introduzir conceitos fundamentais destes paradigmas, bem como sua relação, para que seja possível entender o contexto do *offloading* de funções, os requisitos da arquitetura para suportá-lo, e os benefícios trazidos por ele, o que será apresentado no capítulo seguinte.

# 3 OFFLOADING DE FUNÇÕES VIRTUALIZADAS DE REDE

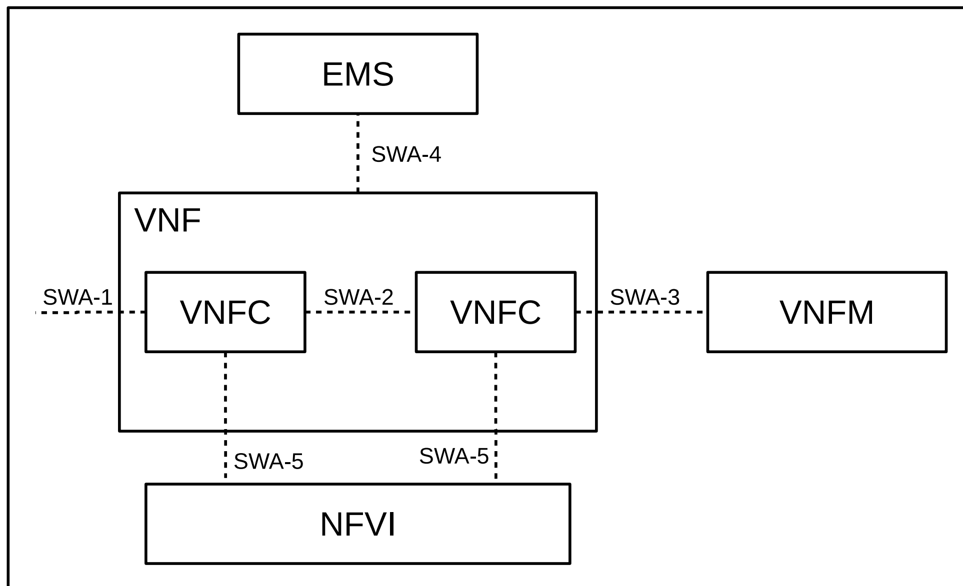
Neste capítulo são apresentados conceitos relacionados ao *offloading* de funções de rede no contexto de NFV e SDN. Um dos requisitos que torna possível o *offload* de uma função de rede é uma arquitetura de VNF compatível. Assim, inicialmente é realizada uma discussão sobre a arquitetura de VNFs, e de que forma isto pode afetar a viabilidade de *offloading*. Após, o conceito de técnicas de aceleração e *offloading* em ambientes NFV e SDN é introduzido. Por fim, são apresentados e discutidos trabalhos relacionados ao tema desta dissertação.

## 3.1 ARQUITETURA INTERNA DE VNFS

De acordo com a arquitetura publicada pela ETSI, VNFs (NFV-SWA, 2014) são funções de rede capazes de serem executadas em uma NFVI e gerenciadas por um NFVO, possuindo um ou mais componentes internos chamados de VNFCs e diversas interfaces com outras partes da arquitetura, como apresentado na Figura 4. A interface SWA-1 refere-se a conexão da VNF com outras VNFs, PNFs ou *endpoints*. A interface SWA-3 conecta a VNF com o bloco de orquestração, mais precisamente o VNF *Manager*, servindo como uma interface de gerência. A interface SWA-4, por sua vez, conecta a VNF ao EMS, podendo ser utilizada para uma gerência mais avançada ou coleção de métricas. A interface SWA-5, por sua vez, refere-se ao ponto de conexão da VNF ou VNFCs com a NFVI, servindo como uma abstração de todas as conexões entre a VNF e a NFVI. Por fim, a interface SWA-2 representa uma interface lógica para comunicação entre VNFCs.

VNFs não precisam necessariamente executar todos seus VNFCs em uma mesma instância, sendo possível a criação de VNFC *Instances* sobre múltiplos nós de computação dentro da NFVI. Neste caso, as interfaces lógicas dos VNFCs (SWA-2) utilizam o substrato da rede através das interfaces SWA-5 para conectarem-se entre si. Em uma situação onde os VNFCs são executados em conjunto no mesmo servidor ou até mesmo máquinas virtuais, a interface SWA-2 pode ser realizada através de tecnologias como memória compartilhada ou até mesmo comunicação entre processos. Para os sistemas de gerenciamento disponíveis no MANO, estas múltiplas instâncias devem comportar-se e são controladas como uma única VNF, os VNFCs não sendo visíveis da perspectiva do usuário e com seu ciclo de vida atrelado ao da VNF. Por fim, considerando as especificidades de VNFCs, a formalização da interface SWA-2 não é de-

Figura 4 – Visão interna de uma VNF



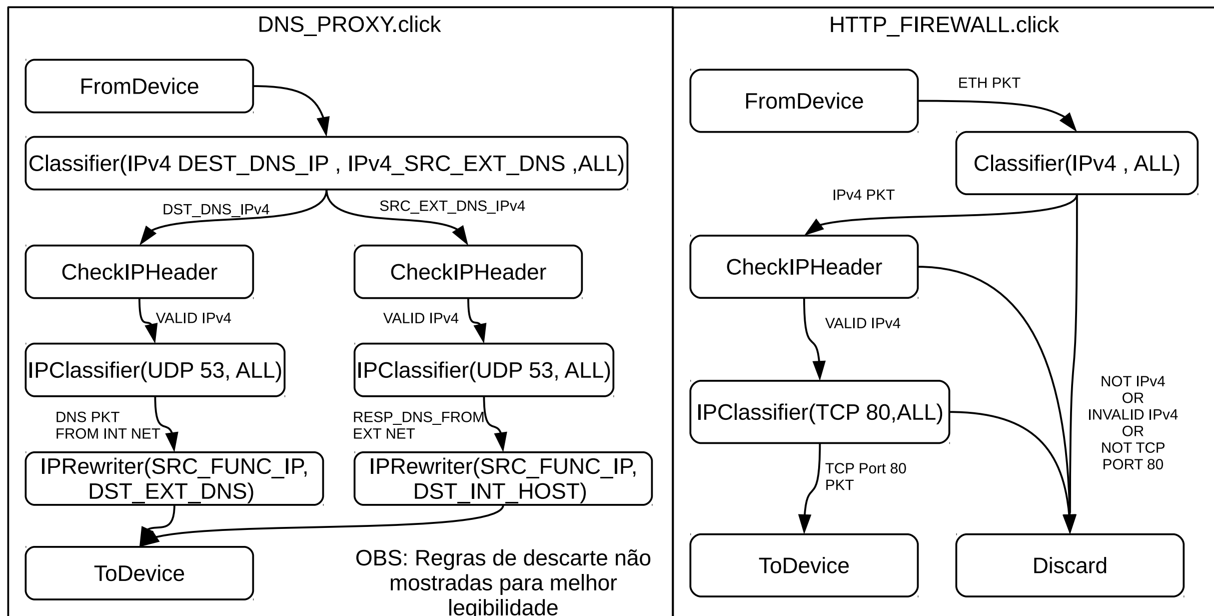

Fonte: adaptado de (NFV-SWA, 2014).

finida pelo ETSI. Desenvolvedores tem então a opção de implementar esta interface interna de acordo com as necessidades da VNF, focando por exemplo em questões de desempenho como latência e capacidade, ou então em questões de segurança.

Por não existir uma formalização da arquitetura interna de VNFs além dos VNFCs, fabricantes podem decidir a melhor forma de implementá-las, bastando apenas adequar as interfaces externas para a gerência e infraestrutura. Desta forma, funções legadas utilizando o *software* de *middleboxes* podem ser portadas e transformadas em uma VNF pelos fabricantes. Embora seja relevante para um período de transição entre infraestruturas atuais e NFV, estas VNFs são, assim como sua origem, monolíticas e proprietárias, oferecendo um menor controle sobre os recursos utilizados devido as diversas funcionalidades em uma mesma instância, e não permitindo acesso ou modificações em sua arquitetura interna. Em *middleboxes*, recursos físicos não utilizados pelo *software* ficam ociosos, pois não há forma de compartilhar eles com outros elementos na rede.

Já em infraestruturas NFV, recursos da NFVI são compartilhados entre VNFs, de modo que funções legadas resultam em desperdício de recursos já que, mesmo que apenas algumas funcionalidades da instância sejam utilizadas, todo o sistema precisa estar em execução (SEKAR et al., 2012). Assim, com o objetivo de não tornar infraestruturas NFV em apenas mais um ambiente para executar *middleboxes* monolíticos e proprietários, propostas de arquiteturas internas de VNFs levando em conta e estendendo o conceito de VNFCs vêm sendo apresentadas,

Figura 5 – Exemplos de funções de rede.




Fonte: do Autor.

permitindo mais flexibilidade na implantação de VNFs, e um melhor controle das funcionalidades executadas e utilização de recursos.

A arquitetura apresentada em (Garcia et al., 2019) define VNFs compostas por seis componentes, interconectados entre si por um roteador interno. Cada módulo é responsável por uma função específica e com interfaces bem definidas, permitindo que novos componentes sejam adicionados ou substituídos conforme a evolução e disponibilidade de novas tecnologias. VNFs implementadas seguindo esta arquitetura podem consumir menos recursos, já que apenas a função necessária é implantada (por exemplo, um módulo para dar suporte a NSH é utilizado apenas se a VNF faz parte de uma SFC), e também são mais flexíveis, já que componentes mais adequados a infraestrutura do usuário podem ser implantados e substituídos sem a necessidade de modificar o restante da VNF.

Outra vantagem trazida pelo uso de VNFs modulares refere-se ao reuso dos componentes. A Figura 5 apresenta dois exemplos de funções de rede construídas utilizando um *framework* de desenvolvimento chamado Click Modular Router (KOHLER et al., 2000a). Neste *framework* elementos são pequenas partes de códigos que desempenham funcionalidades específicas, semelhante ao conceito de VNFCs. Cada elemento possui uma ou mais entradas e saídas, além de regras definidas pelo usuário na declaração dos elementos. Funções de rede são compostas através da conexão destes elementos, onde o pacote entra, é processado e encaminhado para a saída, que por sua vez está conectada a um outro elemento ou a uma interface de

rede.

As funções apresentadas como exemplos consistem de uma *proxy* para requisições DNS e um *firewall* simples que permite apenas tráfego HTTP. Os elementos *FromDevice* e *ToDevice* referem-se a entrada e saída de tráfego na função, através de uma interface de rede. Mesmo que ambas funções possuam propósitos completamente diferentes, é possível visualizar que diversos elementos são comuns às duas. Por exemplo, ambas necessitam fazer uma classificação do tráfego recebido através do elemento *Classifier*, verificar a validade do cabeçalho IP (*CheckIPHeader*), e fazer classificação na camada IP (*IPClassifier*). Este comportamento se estende para outros tipos de funções de rede.

A arquitetura apresentada em (Chowdhury et al., 2019) permite a composição de VNFs e SFCs através de componentes reusáveis, leves e que podem ser implantados de forma independente, chamados de *micro-NFs*. A motivação para esta arquitetura é justamente o fato de que várias funções de rede possuem elementos em comum, que quando implementados como componentes únicos podem ser compartilhados entre diferentes VNFs. Em uma implementação monolítica, estes componentes deveriam ser implementados e otimizados individualmente para cada função de rede.

Arquiteturas de VNFs baseadas em componentes são compostas por elementos internos simples e bem definidos, que podem ser migrados sem afetar o restante da arquitetura. Elementos também podem ser reutilizados e substituídos de acordo com as necessidades dos usuários. Funções de rede mais complexas, como funções *stateful*, não podem ser implementadas completamente em FPGAs, devido a limitações inerentes das arquiteturas de dispositivos programáveis. A programação de FPGAs também é notoriamente mais complexa quando comparada a linguagens de alto nível, mesmo utilizando abstrações como a linguagem P4, possuindo também uma limitação de recursos disponíveis. Assim, arquiteturas baseadas em componentes são ideais para casos de *offload*, permitindo que apenas alguns elementos sejam transferidos da VNF para o dispositivo programável, sem que seja necessário reescrever toda a função. Pelos elementos também possuírem funções simples e bem definidas, sua implementação em P4 ou linguagens de baixo nível também é facilitada.

## 3.2 OFFLOADING

Funções virtualizadas de rede, como o próprio nome já diz, são executadas de forma virtualizada em servidores através da utilização de um *hypervisor*. CPUs genéricas são projeta-

das para oferecer um bom desempenho para os mais variados tipos de utilização, enquanto que ASICs são projetados desde seu início com um propósito específico. A utilização destes circuitos em *middleboxes* permitem um alto desempenho, de forma que, ao migrar funções de rede para um ambiente virtualizado, é esperado que ocorra uma queda no desempenho da função.

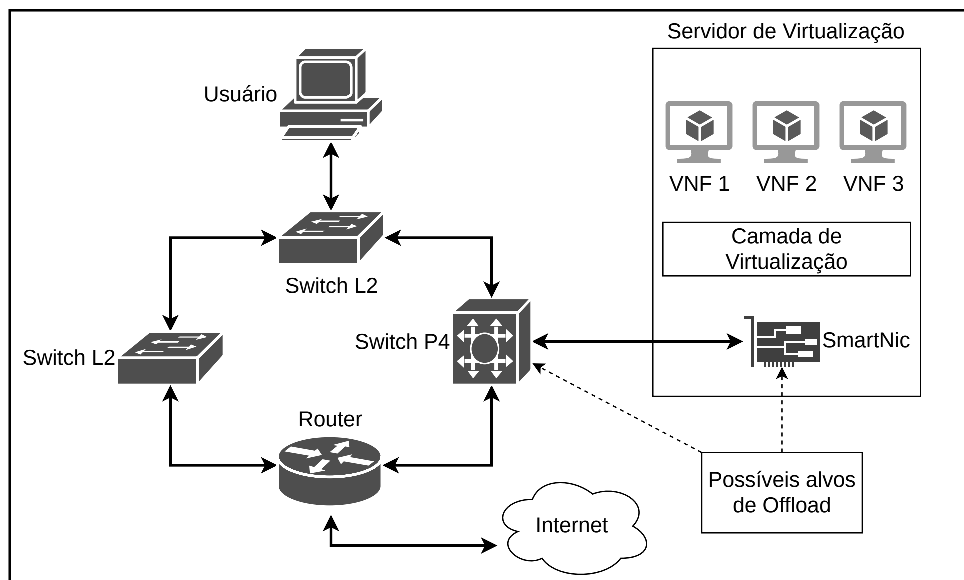
Como o desempenho de funções de rede é um aspecto importante para sua adoção, esforços para mitigar esta perda de desempenho associado a virtualização vem sendo estudados. Dada a complexidade de uma infraestrutura NFV, existem diversos pontos onde técnicas de aceleração do processamento de pacotes podem ser aplicados, como nos dispositivos de rede, servidores de virtualização, ou até mesmo na própria VNF. Estas técnicas podem ser divididas em dois grandes grupos: técnicas que implementam otimizações através de *software*, podendo ou não serem auxiliadas por *hardware*, e técnicas onde o processamento ou parte dele é delegado a outro processador (LINGUAGLOSSA et al., 2019).

O primeiro grupo é subdividido em técnicas desenvolvidas puramente em software, como *polling*, processamento em lotes ou paralelo e *zero-copy*, implementadas através de *drivers* ou otimizações na própria pilha de rede das VNFs, e técnicas que necessitam de suporte do *hardware*, como VT-d, IOMMU e SR-IOV. A utilização destas normalmente se dá através de *frameworks* de aceleração de pacotes como DPDK (INTEL, 2014), Netmap (RIZZO, 2012) e PF_RING. A principal vantagem destas técnicas com relação ao outro grupo está no fato de possuírem uma menor complexidade de implantação, devido à disponibilidade de APIs e pilhas de rede prontas para suportá-las, e o fato de, mesmo quando executadas em *hardware* onde não há suporte para funcionalidades mais avançadas, ainda apresentarem um desempenho superior ao de pilhas de rede tradicionais, com um ganho de desempenho razoável para infraestruturas em processo de transição para NFV (MARCUZZO et al., 2018).

No segundo grupo estão contidas técnicas onde a função de rede, ou uma parte dela, deixa de ser executada completamente pelo processador principal e passa a ser executada em outro processador dedicado. Este processo é denominado *offload* e não é exclusivo ao paradigma NFV, sendo utilizado também em outras áreas da computação. Neste caso, como pode ser visto na Figura 6, antes do tráfego chegar à função de rede virtualizada, uma parte da função executa em um dispositivo que esteja no caminho, como um *switch* programável, onde é realizado um pré-processamento dos pacotes que serão enviados para a VNF.

Embora de uma forma limitada, interfaces de rede atuais já realizam o *offload* de certas partes dos pacotes tratados. Uma grande parte do tráfego recebido e encaminhado hoje em dia

Figura 6 – Possíveis locais onde elementos de uma função podem ser executados em cenários de *offload*.




Fonte: do Autor.

consiste de protocolos TCP e UDP sobre pacotes IP. Entre as tarefas necessárias para que seja possível enviar e receber este tráfego, está o cálculo de *checksum* dos pacotes, o tratamento de fluxos segmentados (maiores que o MTU da rede), e por vezes funções de criptografia. Por serem tarefas tão comuns, fabricantes implementam nas interfaces de rede circuitos estáticos específicos para realizar estas tarefas, diminuindo a carga sobre o processador principal. Uma outra técnica de *offload* utiliza GPUs para processar pacotes de forma paralela. A arquitetura de GPUs é otimizada para executar operações simples em paralelo, de modo que pacotes podem então serem organizados em lotes e enviados para a GPU executar o processamento necessário em cima deles. Porém, dependendo do tipo de tráfego encaminhado para ser processado, é necessária a reordenação destes pacotes quando eles voltam da GPU.

Uma outra alternativa que permite o offload de tarefas mais complexas com maior flexibilidade é o uso de dispositivos programáveis baseados em FPGAs. FPGAs apresentam vantagens sobre outras formas de *offload* por apresentarem um desempenho mais próximo do de ASICs, um menor consumo de energia quando comparado a GPUs, e permitirem reprogramabilidade dos circuitos após sua implantação. Esta reprogramabilidade é possível por FPGAs serem compostas por circuitos genéricos interconectados entre si que podem ser configurados para executarem tarefas específicas. *Switches* programáveis, como os baseados no design de RMT, e SmartNICs podem utilizar FPGAs em sua implementação. Tradicionalmente, a programação destas FPGAs é feita através de uma linguagem de baixo nível como Verilog ou VHDL,

sendo que FPGAs de fabricantes diferentes possuem arquiteturas e bibliotecas proprietárias, gerando uma certa complexidade no desenvolvimento de funções de rede baseadas em FPGAs. Uma alternativa ao uso de linguagens de baixo nível é utilizar linguagens como P4, que embora limitem o uso da FPGA para tarefas de processamento de pacotes *stateless*, o mesmo código pode ser compilado para diversas arquiteturas diferentes.

A segurança dos dispositivos programáveis também é um benefício trazido pelo *offload*. Por executarem de forma virtualizada tendo como hospedeiro sistemas operacionais tradicionais, VNFs estão propensas a ataques caso o hospedeiro seja comprometido. Além disso, uma outra forma de execução de VNFs utiliza *containers* como sua plataforma de execução, de modo que o *kernel* é compartilhado entre o hospedeiro e a VNF (Yang; Fung, 2016). Dispositivos programáveis baseados em FPGA possuem uma arquitetura simplificada em comparação com servidores, executando apenas um *firmware* ao invés de um sistema operacional completo. Isto pode trazer uma camada de segurança contra ataques a estes dispositivos, visto que utilizam arquiteturas menos comuns e, assim, menos visadas por atacantes.

Um último benefício trazido pelo *offload* tem relação com o conceito de *placement*. A área de *placement* ou alocação de recursos de NFV (Gil Herrera; Botero, 2016) busca encontrar formas de melhor utilizar os recursos da rede para instanciar VNFs, levando em consideração limitações como balanceamento de carga, consumo de energia e tolerância a falhas. O *offload* de funções contribui com esta área de duas formas: (i) permitindo que partes das funções de rede executem em dispositivos diferentes e (ii) aumentando a quantidade de dispositivos capazes de suportar a execução de VNFs. Através destas contribuições, algoritmos de *placement* têm uma maior flexibilidade na escolha de onde as VNFs serão executadas.

A utilização do *offload* como uma forma de mitigar a perda de desempenho decorrente da virtualização de funções de rede permite o desenvolvimento de funções híbridas, onde uma parte da função executa em um dispositivo programável (através de FPGAs), enquanto que tarefas mais complexas, que utilizem mais recursos ou funcionalidades não disponibilizadas por dispositivos programáveis podem continuar executando de forma virtualizada. No entanto, antes que possa ser definido um modelo de *offload*, alguns cuidados devem ser tomados referentes a limitações desta técnica.

Técnicas de *offloading* são tradicionalmente implementadas na forma de *bump-in-the-wire*, ou seja, o processamento é realizado antes de o tráfego entrar ou após sair do processador principal. Isto ocorre pois existe um custo associado a transferência dos dados entre os pro-

cessadores, de modo que o *offload* de elementos que estejam entre elementos executados no processador principal resulta em um maior *overhead* relacionado à transferência, podendo até anular o desempenho ganho ao realizar o *offload*. Outra questão refere-se a limitações nas arquiteturas de FPGA, pois embora sejam circuitos programáveis, não são capazes de executar todas tarefas que uma CPU genérica executa, além de possuírem recursos mais limitados.

Nota-se também que o uso de *offload* no contexto de NFV não está restrito à execução de elementos de uma função. A NFVI também pode se aproveitar de FPGAs para implementar *offload* nas camadas física ou virtual, de forma transparente para as funções executadas. No entanto, esta abordagem está fora do escopo desta dissertação, que tem como objetivo permitir que VNFs utilizem técnicas de *offload* para melhorar seu desempenho.

## 3.3 TRABALHOS RELACIONADOS

Considerando o contexto apresentado nas seções anteriores, nesta seção são apresentados trabalhos relacionados ao tema desta dissertação. Inicialmente são apresentados trabalhos sobre o desenvolvimento de *frameworks* onde haja suporte a *offloading*. Embora os artigos correspondam a *offloading* em FPGAs, os conceitos apresentados podem ser aplicados em um cenário de *offload* para plano de dados programável. Após, foram buscados trabalhos referentes a orquestração entre VNFs e planos de dados programáveis. Estes conceitos são necessários para que seja possível desenvolver uma solução capaz de comunicar-se com ambos ambientes NFV e SDN para orquestrar o *offload*.

### 3.3.1 Plataformas de VNFs com suporte a *offload*

Em Chimpp (RUBOW et al., 2010), é apresentado um ambiente de desenvolvimento de funções para *hardware* reconfigurável, tendo como alvo a plataforma NetFPGA. Uma arquitetura modular modelada a partir de conceitos do Click Modular Router, incluindo a sua linguagem, foram utilizados no desenvolvimento. Este ambiente também é integrado ao simulador OMNeT++ para testes, de modo que o objetivo é simplificar tarefas de experimentação com plataformas NetFPGA, permitindo também a simulação de funções híbridas executando em CPU e FPGA.

Tomando como base os elementos disponíveis no Click Modular Router, são implementadas versões destes elementos para a plataforma NetFPGA de forma manual, sendo que os

desenvolvedores buscaram implementar os elementos mais utilizados. Um processo do Click executando em CPU também pode ser associado aos elementos executando na NetFPGA, funcionando como um plano de controle da implementação. Conceitos da linguagem Click são utilizados para a definição de uma linguagem estendida, com a qual podem ser definidas instâncias dos elementos e suas conexões. Neste código também é definido um *package*, que contém um grupo com definições e *buses* (um conjunto de declarações `wire` em Verilog), contidos em um arquivo XML. Quando o código da linguagem é processado gera-se uma definição em Verilog que pode ser implantada na plataforma NetFPGA.

A comunicação entre uma instância do Click executando em CPU e os elementos executando na FPGA é feita de forma diferente para o tráfego de pacotes e para o controle dos elementos. O tráfego é compartilhado através de elementos *FromDevice* e *ToDevice* do Click, já que a placa NetFPGA possui módulos de *kernel* que permitem ela ser vista como uma interface de rede normal para o sistema. A comunicação para o controle dos elementos, por sua vez, é feita através de elementos Click especializados capazes de escrever em registradores definidos pelo usuário dentro da FPGA, sendo que alguns destes também são disponibilizados pelos autores.

Em Click2NetFPGA (RINTA-AHO; KARLSTEDT; DESAI, 2012), os autores abordam a utilização de High Level Synthesis (*i.e.,* transformar algoritmos em linguagens de alto nível em um projeto de *hardware*) para converter um roteador Click em módulos executáveis na plataforma NetFPGA de forma automática, através de uma *toolchain* baseada em LLVM.

O processo de conversão consiste em cinco passos. Inicialmente elementos Click desenvolvidos em C++ são compilados em objetos linkáveis pelo compilador, sendo que este processo só precisa ser feito se houver alterações no código-fonte do Click. Após, uma ferramenta chamada `click2llvm` realiza a leitura da configuração Click definida pelo usuário e carrega uma instância do Click em memória. A ferramenta então lê os valores inicializados e grava eles como constantes no formato de módulo LLVM IR, junto com o código dos elementos. Este módulo representa uma instância em execução do Click, e após passa por várias transformações e otimizações através do *backend* AHIR, que transforma *bytecode* LLVM em VHDL. Por fim, este código VHDL é combinado com arquivos Verilog da SDK da NetFPGA, gerando uma *netlist*, que por sua vez pode ser implementada na NetFPGA.

Desta forma, todo o roteador Click é executado em FPGA, de forma que não há uma instância do Click executando em CPU. O código também não pode ser reconfigurado dinami-

camente, já que mudanças na configuração requerem o processo de compilação da *toolchain* novamente. Por fim, o desempenho atingido por esta conversão automática de código é de 30 a 50% do desempenho obtido se os elementos fossem implementados manualmente.

ClickNP (LI et al., 2016) é um *framework* de desenvolvimento baseado no Click Modular Router com suporte a execução de elementos, semelhante ao Chimpp. O ClickNP é construído tomando como base a arquitetura *Catapult Shell*, que já dispõe de várias abstrações para FPGA como gerenciamento de memória, DMA e suporte a Ethernet. Uma instância do ClickNP possui um processo *host* que se comunica com a FPGA através de uma biblioteca, expondo um canal de comunicação PCIe que permite baixa latência e alta vazão para comunicação entre o processo executando em CPU e a FPGA.

O processo *host* possui uma *thread* de gerenciamento e várias *worker threads*. A *thread* de gerência é responsável por chamar bibliotecas de HLS para passar parâmetros aos elementos implementados em FPGA, gerenciar o ciclo de vida destes elementos, bem como a conexão entre os elementos executando na FPGA e na CPU. A implementação dos elementos em FPGA é feita manualmente, sendo que os autores disponibilizam mais de 100 elementos já implementados. Por ser construído utilizando a arquitetura *Catapult Shell*, utilizando HLS é possível compilar o código de alto nível para diversos modelos de FPGA ou em binários executáveis em CPU.

A escolha dos elementos para serem executados em CPU ou FPGA se dá através de anotações na configuração Click, que também é estendida para declarar elementos e *handlers* para os módulos executados em FPGA. O *framework* também se aproveita do paralelismo disponível em FPGAs, de modo que os módulos podem ser executados em paralelo.

### 3.3.2 Orquestração de *offload* em NFV

Metron (BARBETTE et al., 2018) é uma plataforma NFV para o *placement* e *dispatching* de SFCs, capaz de eliminar transferência de dados entre núcleos diferentes. A plataforma é capaz de fazer *offload* de parte do processamento de pacotes para dispositivos de rede, utilizar técnicas de *smart tagging* para configurar e explorar a afinidade de classes de tráfego. Através da utilização do controlador SDN ONOS (BERDE et al., 2014), a plataforma é capaz de gerenciar redes heterogêneas com OpenFlow ou P4.

Ao realizar a implantação de uma SFC, inicialmente a plataforma faz um *parse* dos elementos que compõem as VNFs. Após, os elementos das VNFs são combinados e sintetizados

em um único grafo de encaminhamento, ao mesmo tempo em que associa funções *stateless*, como regras de encaminhamento, a *switches* OpenFlow que estejam no caminho do tráfego, convertendo estas operações para regras OpenFlow de forma automática. O restante das operações da SFC são potencialmente *stateful*, e por isso são executadas em um servidor de virtualização, divididas por núcleos.

Um dos diferenciais da plataforma é a capacidade de, quando regras OpenFlow são aplicadas para encaminhamento do tráfego nos *switches*, uma anotação é colocada determinando a classe do tráfego. Ao chegar no servidor de virtualização, a interface de rede faz a leitura destas anotações e encaminha cada tipo de tráfego para um núcleo de processamento específico, eliminando a necessidade de comunicação entre núcleos no servidor.

O *framework* utilizado para sintetizar as SFCs é o SNF (KATSIKAS et al., 2016), que é capaz de derivar classes de pacotes a partir de tráfego sendo encaminhado em uma SFC, sintetizando uma nova SFC equivalente a original sem redundância e com otimizações no encaminhamento. O processo de otimização do *framework* consolida todas operações de leitura para uma dada classe de tráfego em um único elemento, descarta o mais cedo possível classes de tráfego que resultam em descartes, e associa cada classe de tráfego com um elemento de escrita. Outra otimização refere-se ao compartilhamento de elementos comuns para diferentes classes de tráfego.

A arquitetura da plataforma consiste em um controlador e um agente do plano de dados. O controlador Metron é implementado como uma aplicação no controlador ONOS, escolhido por já prover *drivers* para protocolos populares como OpenFlow, P4 e Netconf. O agente do plano de dados é uma aplicação que executa nos servidores de virtualização, baseada no Click Modular Router. Ao ocorrer uma decisão de implantação de uma SFC, o controlador é responsável por sintetizar a SFC e configurar a rede, aplicando as regras OpenFlow, e delegar ao agente a configuração da interface de rede e da instância do Click Modular Router no servidor de virtualização.

O *framework* E2 (PALKAR et al., 2015) implementa uma ferramenta capaz de realizar tarefas comuns, mas não triviais relacionadas ao processamento de pacotes, como *placement*, *elastic scaling*, balanceamento de carga, gerenciamento, entre outros. O objetivo é permitir que operadores de redes possam focar em tarefas principais enquanto consolida tarefas de gerenciamento.

No E2, um único controlador é responsável tanto pelas tarefas de gerenciamento das

NFs como tarefas de gerenciamento de recursos. A operação interna é baseada no conceito de *pipelets*, que definem qual tráfego deve ser processado por quais NFs, mas não onde ou como o processamento ocorre na infraestrutura. Sua arquitetura é composta por três módulos principais: *E2 Manager*, responsável pela operação em geral da infraestrutura, um *Server agent* instanciado e responsável pela configuração de cada servidor, e *E2 Dataplane* (E2D), que permite flexibilidade nas conexões entre as NFs.

Em cada servidor, existe uma implementação do E2D que utiliza uma plataforma chamada de SoftNIC, um switch programável que funciona com o conceito de módulos conectados em um grafo direcional, semelhante ao Click Modular Router. A SoftNIC permite a execução de módulos de processamento entre as portas virtuais (vports) disponibilizadas as NFs e as portas físicas (pports). Módulos para monitoramento e balanceamento de carga, classificação de pacotes e tunelamento entre NFs são implementados na SoftNIC e servem como uma camada de gerenciamento do framework entre as NFs e as interfaces físicas.

O E2 Manager, por sua vez, é responsável pelo *placement*, interconexão entre servidores, scaling e garantir afinidade entre as VNFs. O processo de *placement* inicia com a combinação de vários *pipelets* em um grafo unificado para cada NF, semelhante ao processo realizado pelo Metron. Este grafo é então convertido para um grafo onde cada nó representa uma instância, cada instância é mapeada para um servidor onde será executada. Caso seja possível, o *offload* de uma função pode ser realizado para um *switch*, desde que esta NF esteja diretamente conectada a uma porta do *switch*, e o *switch* tenha recursos disponíveis. Neste caso, o *offload* a ser feito refere-se apenas a regras de encaminhamento de tráfego *trafic steering*, e a configuração é realizada pelo E2 Manager via OpenFlow.

## 3.4 DISCUSSÃO

Após a revisão de literatura, foram identificados os principais trabalhos relacionados ao *offload* de funções virtualizadas de rede, que se dividem em plataformas de desenvolvimento com suporte a *offload*, e arquiteturas de orquestração de *offload* em infraestruturas SDN e NFV. Também buscou-se identificar as principais vantagens trazidas pelas arquiteturas pesquisadas, e as deficiências existentes na área.

A Tabela 1 apresenta os principais conceitos de cada um dos trabalhos encontrados referentes as plataformas de desenvolvimento de VNFs com suporte a *offload*. É possível visualizar que existe apenas uma plataforma que faz a conversão automática dos elementos para a lingua-

Tabela 1 – Características principais dos trabalhos relacionados.

| Framework | Execução | Comunicação | Linguagem de Entrada | Conversão de Código | Plataforma |
|---|---|---|---|---|---|
| Chimpp | CPU-FPGA | Rede e SDK | Click com extensões | Manual | NetFPGA |
| Click2 NetFPGA | Somente FPGA | - | Click | Automática | NetFPGA |
| ClickNP | CPU-FPGA | Canal PCIe | Click com extensões | Manual | Qualquer FPGA via HLS |

Fonte: do Autor.

gem de *offload*, resultando em um desempenho inferior a uma implementação manual, de modo que a abordagem de implementar elementos mais comuns manualmente apresenta um melhor desempenho. Todas as soluções também utilizam como entrada a linguagem Click, embora modificada, devido a sua expressividade e sua grande quantidade de elementos prontos. As soluções Chimpp e ClickNP também permitem uma execução híbrida entre elementos executando em FPGA e na CPU, dando mais flexibilidade na composição de funções.

A escolha do Click Modular Router como o ponto inicial de construção das plataformas deve-se a sua estrutura modular, onde elementos podem ser facilmente migrados para outro processador e continuarem funcionando em conjunto com a função executando no processador principal. Todos os trabalhos também possuem suporte a FPGAs, mas apenas a uma arquitetura, como NetFPGA, ou então precisam utilizar HLS para suportar arquiteturas diferentes. Além disso, a arquitetura das plataformas foi projetada tendo em vista que o dispositivo FPGA está sendo executado no mesmo servidor onde as funções de rede executam, já que utilizam canais de comunicação como o *bus* PCI ou APIs proprietárias de interação com a FPGA. Isto torna estas arquiteturas inviáveis de serem executadas em dispositivos de rede programáveis na infraestrutura, pois estes possuem arquiteturas diferentes, dependentes do fabricante, e podem ser implantados em diferentes pontos na rede, nem sempre diretamente conectados aos servidores de virtualização onde as VNFs estão executando.

Com relação às plataformas de orquestração de *offload*, ambas plataformas são capazes de fazer o offload para dispositivos de rede, não sendo necessariamente dispositivos programáveis. O protocolo OpenFlow é utilizado para a instalação das regras de encaminhamento, bem como de funcionalidades simples. O suporte a *offload* na plataforma Metron é implementado através de uma camada de abstração inserida no servidor de virtualização, enquanto a plataforma E2 não define especificamente como o *offload* é realizado. As plataformas trabalham

com o contexto de *offload* a nível de SFCs, e embora o Metron seja capaz de sintetizar grafos de encaminhamento a partir de uma configuração Click, e consequentemente separar elementos internos de uma função, isto é feito antes de as funções serem implantadas, de forma que a opção pelo *offload* não pode ser feita de forma dinâmica.

As soluções de orquestração encontradas também apresentam certas restrições com relação a forma que o *offload* é realizado. No caso da plataforma E2 existe uma dependência da arquitetura SoftNIC que precisa ser executada no servidor de virtualização, enquanto que na plataforma Metron, a síntese das SFCs realizada pelo SNF não permite que o operador de rede escolha os elementos que serão alvos de *offload*, e obriga os usuários a definirem *tags* para diferentes classes de tráfego. Além disso, por serem plataformas orientadas a SFC, sua efetividade com apenas uma VNF é limitada ou até impossível.

Uma deficiência clara encontrada nos trabalhos relacionados é o fato que, embora exista um consenso sobre a utilização da linguagem Click como a origem para cenários de *offload*, a linguagem e a arquitetura de destino variam desde FPGAs até a utilização de protocolos como OpenFlow. A utilização de uma linguagem padronizada para dispositivos programáveis, como a linguagem P4, permite que o *offload* possa ser realizado para várias arquiteturas diferentes. Outra limitação refere-se à necessidade de modificar o servidor de virtualização para suportar as ferramentas necessárias, como a utilização de uma placa FPGA, ou as camadas de abstração fornecidas pelas plataformas de orquestração. Por fim, uma plataforma de *offload* orientada ao conceito de SFCs não é adequada para casos onde apenas uma VNF seja utilizada. Embora grande parte dos serviços disponibilizados por um provedor sejam implementados através de SFCs, isto não significa que todos eles devem suportar *offload*, visto que os recursos disponibilizados por dispositivos programáveis são escassos e, por isso, devem ser reservados para VNFs mais complexas onde o *offload* pode trazer mais benefícios.

# 4 ARQUITETURA DE *OFFLOAD* PARCIAL DE VNFS

Como apresentado no capítulo anterior, soluções atuais de *offload* possuem limitações relacionadas às arquiteturas de destino, suportando arquiteturas específicas ao invés de uma linguagem padronizada. Estas soluções também permitem o *offload* apenas de VNFs completas, não sendo capazes de separar e migrar componentes internos de uma VNF, visto que são focadas em cenários de SFC, e não possuem uma forma de realizar o gerenciamento interno das VNFs. Desta forma, este trabalho busca permitir o *offload* parcial de uma VNF, mais precisamente de seus componentes internos, para dispositivos disponíveis na rede com suporte a programabilidade do plano de dados.

A abordagem utilizada neste trabalho consiste de um módulo interno de suporte a *offload* adicionado em uma plataforma de desenvolvimento de VNFs modular, e uma arquitetura com módulos responsáveis por gerenciar diferentes partes do processo de *offload*.

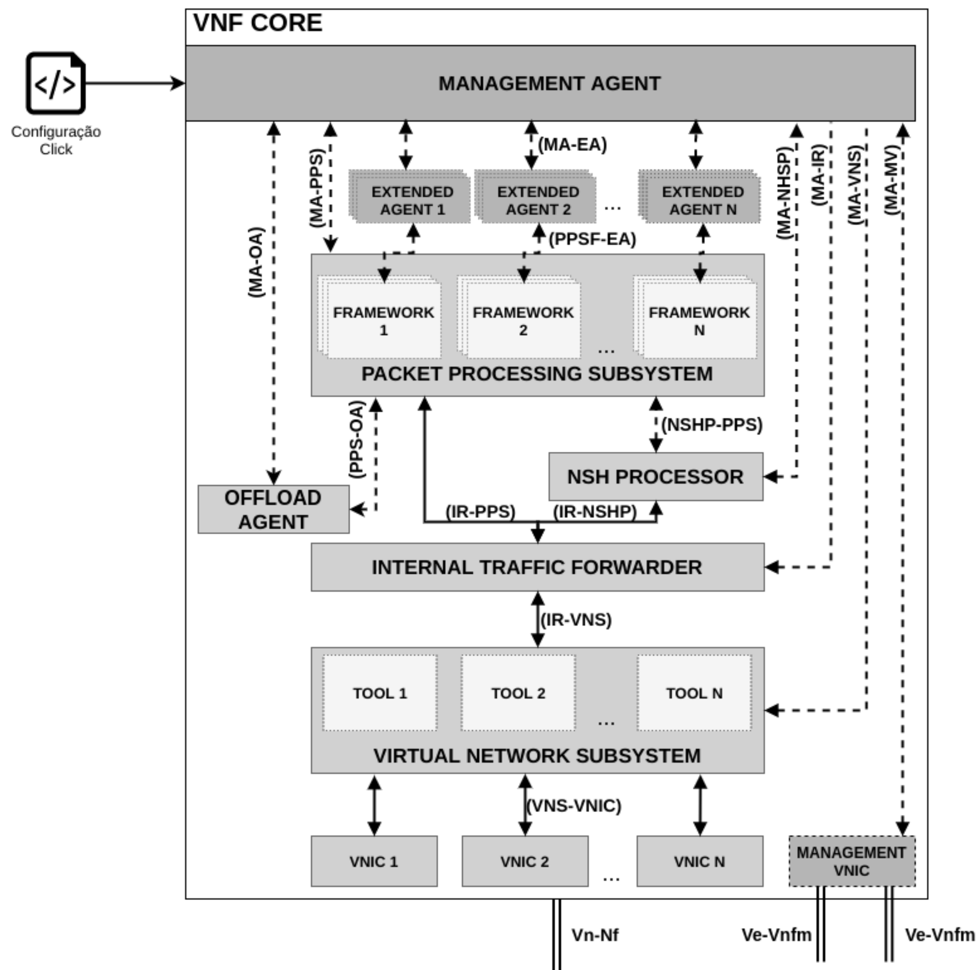
Inicialmente é apresenta a arquitetura de uma VNF modular com suporte a *offload*, elencando as decisões de projeto que levaram a definição do módulo. Após, a arquitetura de gerenciamento do processo de *offload* é apresentada, com uma breve descrição dos módulos que a compõe, e de que forma o processo é realizado. Por fim, benefícios e limitações da arquitetura proposta são identificados e discutidos.

## 4.1 ARQUITETURA DA VNF COM SUPORTE A OFFLOAD

Considerando os trabalhos relacionados descritos no capítulo anterior, e identificadas suas vantagens e desvantagens, nesta seção é apresentada uma arquitetura interna de VNFs com suporte a *offload* parcial de seus elementos. Como a modularidade da plataforma é um aspecto importante no suporte ao *offload*, optou-se por uma arquitetura flexível e genérica o suficiente para que novas funcionalidades possam ser adicionadas sem necessidade de maiores modificações.

Desta forma, a arquitetura proposta em (Garcia et al., 2019) mostrou-se adequada aos propósitos desta dissertação, por possuir uma extensibilidade nativa, módulos com funcionalidades e conexões bem definidas, bem como *frameworks* de processamento de pacotes modulares, de modo que as modificações necessárias para adicionar suporte ao *offload* nesta arquitetura consistem de um novo módulo, denominado *Offload Agent*, e alterações no agente de gerencia-

Figura 7 – Arquitetura da VNF proposta




Fonte: adaptado de (Garcia et al., 2019).

mento.

A visão interna da arquitetura é apresentada na Figura 7. No contexto desta dissertação, um novo módulo para o *offload* foi desenvolvido e adicionado na arquitetura. No entanto, a apresentação do restante dos módulos é relevante para o entendimento do contexto onde o novo módulo foi definido.

Assim, a arquitetura é composta por seis módulos principais executados sobre um sistema operacional virtualizado: *Virtual Network Subsystem - VNS*, *Internal Traffic Forwarder - ITF*, *NSH Processor - NSHP*, *Packet Processing Subsystem - PPS*, *Management Agent - MA* e *Extended Agents - EA*. O VNS é responsável pela configuração de baixo nível das interfaces de rede virtualizadas externas, e por enviar e receber pacotes da infraestrutura. Sua implementação pode ser feita com aceleradores de pacotes como DPDK e Netmap. O ITF é o roteador de pacotes interno da VNF. Através de uma configuração recebida pelo MA, este módulo realiza conexões com os elementos que serão utilizados no processamento e configura a ordem de

encaminhamento dos pacotes internamente. O NSHP é um módulo de execução opcional para tratamento de cabeçalhos NSH, utilizado quando a função faz parte de uma SFC. Sua função é realizar operações nos *frames*, como remover, reinserir ou atualizar o cabeçalho antes de enviar ou após receber os pacotes do PPS. O PPS corresponde aos *frameworks* de implementação e desenvolvimento de funções de rede, como VPP ou Click Modular Router, suportando também implementações de funções de rede em linguagens como C++ e Python. A arquitetura também suporta que vários *frameworks* diferentes sejam encadeados. O MA é responsável por monitorar e controlar internamente a execução dos módulos da arquitetura. O MA recebe do VNFM um *VNF Package*, instancia os módulos e os configura de acordo com o descritor. Durante a execução da VNF, o MA também recupera dados de monitoramento e controla o ciclo de vida dos módulos da VNF. O EA é um agente estendido associado ao MA. Funções de rede mais avançadas podem ter configurações ou recuperar dados específicos, que não estão disponíveis no MA. O EA permite a implementação destas funções em um módulo separado, associado a configuração da VNF, que pode ser chamado pelo MA.

A instanciação de uma VNF inicia com o MA recebendo um *VNF Package* através da conexão MA-MV. Após validar e extrair as informações necessárias, o MA solicita a criação de canais de comunicação entre os módulos que serão utilizados (MA-IR), inicializa os agentes estendidos (MA-EA), configura o VNS e PPS e inicia sua execução (MA-VNS, MA-PPS), sendo que a configuração mínima para instanciação é pelo menos um VNS e um PPS. Na instanciação do PPS, por exemplo, o MA recebe no *VNF Package* um descritor informando qual o *framework* a ser utilizado, e o arquivo de configuração deste *framework*. Se houver algum gerenciamento especializado do PPS, o MA delega ao EA (PPSF-EA). Quando é necessário o uso de NSH, o MA solicita a reconfiguração dos canais de comunicação e inicia o NSHP (MA-NSHP). Todo o tráfego que chega nas interfaces de rede é recebido primeiramente no VNS (VNS-VNIC), que por sua vez encaminha ao IR (IR-VNS). A partir do IR, o tráfego pode ser encaminhado diretamente ao PPS (IR-PPS) ou, no caso de uma SFC, encaminhado antes ao NSHP (IR-NSHP) e depois ao PPS (NSHP-PPS).

### 4.1.1 *Offload Agent*

Com o objetivo de tornar a arquitetura compatível com cenários de *offload*, um novo módulo denominado *Offload Agent* - OA é proposto. Embora a arquitetura preveja o uso de novos *frameworks* para o VNS, PPS e EA, a função especializada do OA não se encaixa na

definição destes módulos, sendo mais semelhante a forma que o módulo NSHP é implementado na arquitetura.

Para que a implementação do módulo não cause incompatibilidade com os conceitos e o restante dos módulos da arquitetura, e também levando em consideração as propostas de *offload* apresentadas anteriormente, algumas decisões de projeto foram tomadas:

- **A linguagem utilizada pela VNF para o desenvolvimento de funções não deve ser alterada ou estendida.** Soluções de *offload* apresentadas anteriormente, mesmo que permitam uma maior gama de cenários de comunicação entre a FPGA e a instância da CPU, também adicionam novas estruturas e configurações na linguagem de desenvolvimento de funções, resultando em uma maior dificuldade na implementação das funções, e incompatibilidade com funções legadas.

- **Dispositivos programáveis alvos de *offload* não devem ser configurados pelo agente ou pelo operador de rede.** Soluções como Metron e E2 oferecem diversos benefícios, no entanto sua alta complexidade de configuração e instanciação podem dificultar a adoção das plataformas. Nesta dissertação propõe-se uma plataforma que, embora não resolva todos os problemas, possa ser utilizada por usuários que não tenham muita experiência com dispositivos programáveis.

- **A VNF deve continuar suportando cenários onde não há necessidade de *offload*.** A arquitetura base é genérica o suficiente para suportar diversos cenários de uso, incluindo SFCs. Desta forma, em cenários onde o *offload* não é necessário ou pode ocasionar problemas ao ambiente da função, o módulo deve ser desativado sem prejuízo ao restante da VNF.

- **O agente de *offload* deve fazer apenas um papel de configuração, não devendo processar tráfego.** Embora isto cause limitações com relação a classificação do tráfego ao chegar na VNF, colocar um componente a mais para processar tráfego dentro da arquitetura pode anular os benefícios trazidos pelo *offload* de elementos.

Assim, o módulo OA realiza a análise da configuração a ser executada pela VNF, e gera uma versão modificada da função original sem os elementos que sofrerão *offload*. O OA também precisa extrair informações que serão utilizadas pela arquitetura de gerenciamento, como por exemplo, a ordem dos elementos, seu nome, e as regras associadas a eles. Considerando que a

plataforma de VNF suporta diferentes aplicações para executar as funções, cada aplicação deve ter sua forma de analisar os dados.

Através desta arquitetura definida para o OA, o *offload* é possível de forma transparente para o restante dos módulos da VNF, não sendo necessárias modificações na configuração dos outros módulos, por exemplo. O VNFM continua sendo o responsável pelo ciclo de vida da VNF, e deve receber a informação sobre quando o processo foi finalizado com sucesso para que possa inicializar a VNF. O restante do processo de *offload* é realizado pelo gerenciador apresentado a seguir.

## 4.2 ARQUITETURA DO GERENCIADOR DE OFFLOAD

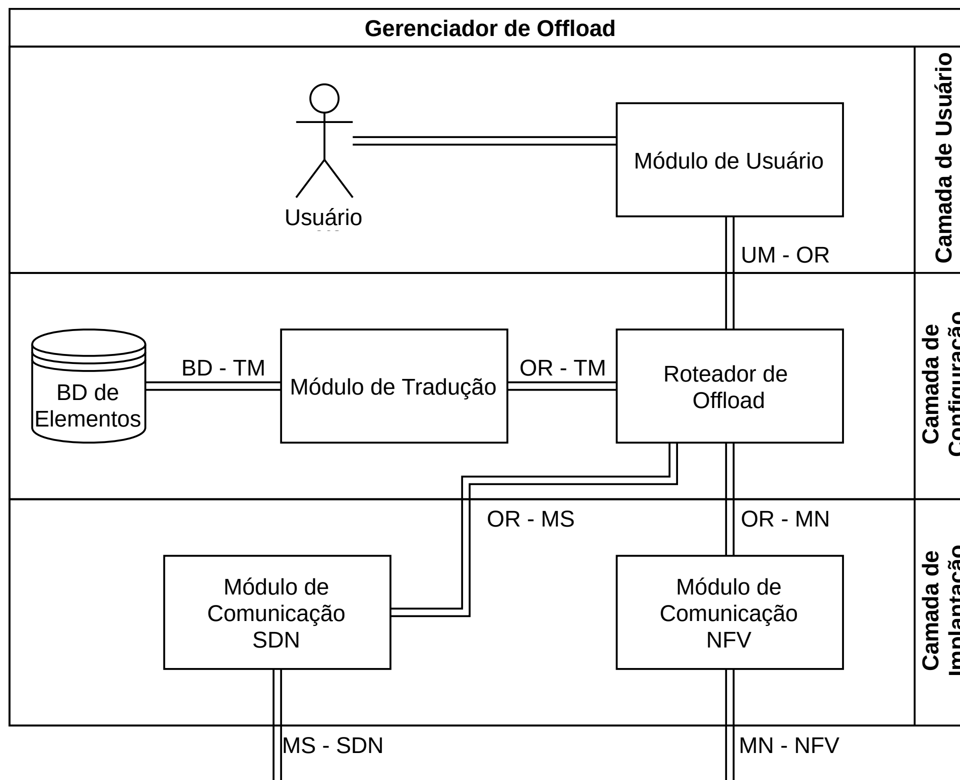
Para que o processo de *offload* seja possível, a VNF precisa do auxílio de um sistema de gerenciamento capaz de recuperar informações e configurar outros componentes da infraestrutura de rede. Assim, uma arquitetura de um sistema de gerenciamento do processo de *offload* de VNFs é proposta. A configuração da infraestrutura é um processo complexo e composto por várias etapas, dependente também da comunicação com outros elementos da rede. Estas etapas foram identificadas e são descritas brevemente a seguir.

- **Interação com o usuário ou orquestrador de alto nível:** A decisão sobre quando o *offload* deve ser realizado pode levar em consideração diversos fatores, como necessidades específicas do usuário ou a reconfiguração da rede por motivos de desempenho e segurança, estabelecidos a partir de diversas métricas. Esta decisão pode ser feita por um operador de rede (de forma manual) ou por um orquestrador de alto nível, e não está no escopo desta dissertação. Desta forma, uma interface por onde estas requisições são recebidas e informações são disponibilizadas para um agente externo se faz necessária.

- **Recuperação de informações sobre o estado da rede:** Nesta etapa, o gerenciador deve coletar informações das VNFs e da infraestrutura SDN necessárias para continuar o processo. Entre as informações necessárias, estão portas onde os dispositivos programáveis e VNFs estão conectados, a disponibilidade de recursos dos dispositivos programáveis, e quais funções são suportadas por eles, visto que um dos objetivos da proposta é suportar diversas arquiteturas utilizando uma linguagem comum. Controladores SDN disponibilizam estas informações em um formato próprio que deve ser tratado pelo coletor.

- **Tradução da lógica da função:** Pelo fato do *offload* ser realizado entre duas tecnologias

Figura 8 – Arquitetura do gerenciador de Offload




Fonte: do Autor.

onde possivelmente as linguagens de programação sejam incompatíveis, é preciso definir uma forma de traduzir a lógica da função entre estas tecnologias.

- **Configuração dos dispositivos e rotas:** Finalmente, a última etapa consiste na configuração dos dispositivos que farão parte do *offload*, e o estabelecimento de comunicação entre eles utilizando a infraestrutura disponível.

Assim, a arquitetura apresentada na Figura 8 é proposta. O gerenciador é dividido em cinco módulos, cada um responsável por uma parte diferente do processo. Uma breve descrição de cada módulo é apresentada a seguir.

- **Módulo de Usuário (User Module - UM):** O módulo de usuário é a interface externa do sistema com o operador de rede ou orquestrador. Neste módulo, o usuário pode fazer solicitações para iniciar e parar um offload, e recuperar métricas relacionadas ao processo. O módulo de usuário também pode ser utilizado para adicionar novas implementações de elementos no banco de elementos do tradutor. Como nativamente apenas requisições manuais são suportadas (feitas por um usuário), este módulo é implementado de forma separada para que possa ser estendido por outros desenvolvedores.

- **Roteador de Offload (Offload Router - OR):** O OR funciona como um ponto central de comunicação entre os outros módulos. Ele faz o roteamento das mensagens de controle da arquitetura entre os módulos adjacentes. Ao receber uma solicitação do módulo de usuário, o OR prepara as mensagens de acordo com a configuração recebida e encaminha, por exemplo, requisições sobre o estado da rede para o MN e o MS, e requisições referentes a tradução dos elementos para o módulo de tradução.

- **Módulo de Tradução (Translation Module - TM):** O módulo de tradução é responsável por gerar uma função que será executada na arquitetura de destino a partir de informações obtidas na origem. O módulo possui um banco de dados com implementações de módulos providas pelo usuário, que são utilizadas na geração do código para o dispositivo destino. A tradução se dá através da correlação dos nomes dos elementos e regras obtidos na origem com implementações existentes no banco de dados, de forma que a implementação da função de destino é diferente da origem, mas logicamente deve realizar as mesmas operações. Para a tradução das regras entre diferentes arquiteturas, uma tabela de equivalência entre os tipos pode ser adicionada pelo usuário.

- **Módulo de Comunicação SDN (SDN Module - SM) e NFV (NFV Module - MN):** A comunicação com a infraestrutura é realizada através dos módulos de comunicação. O módulo SDN conecta-se com o controlador da infraestrutura, recuperando informações sobre a topologia e dispositivos programáveis, e executando as operações de instalação dos elementos de rede e regras. O módulo NFV, por sua vez, conecta-se diretamente com as VNFs ou com o VNFM da infraestrutura, e após ao agente interno de *offload* executando nas VNFs. Sua implementação em separado deve-se ao fato que controladores SDN e NFV não possuem uma interface de comunicação padronizada, sendo necessárias implementações diferentes para cada plataforma.

O processo de *offload* inicia com o usuário utilizando o UM para fazer a requisição, informando obrigatoriamente dados como os dispositivos de origem e destino, e opcionalmente implementações de módulos e uma tabela de tradução de regras. O UM encaminha estas requisições ao módulo OR, que prepara as requisições aos outros módulos. Dos módulos de comunicação (MS e MN), são recuperadas a topologia e dados dos dispositivos da rede, e as informações obtidas pelo agente interno da VNF, respectivamente. A instanciação da VNF é de responsabilidade do VNFM ou do usuário, pois o gerenciador não é capaz de instanciar a

VNF. Isto deve-se ao fato de VNFs com suporte a *offload* fazerem parte do catálogo de VNFs da infraestrutura, portanto podendo ser executadas sem um ambiente de *offload*.

Após, o OR comunica-se com o módulo de tradução (TM) para iniciar a geração da nova função para a arquitetura de destino. O código gerado é então verificado e compilado para a arquitetura de destino, resultando em uma função pronta para ser instalada. A função então é enviada de volta ao OR para que possa ser instalada e configurada no dispositivo através do MS, utilizando funcionalidades disponíveis no controlador SDN. Após a instalação e instanciação da função no dispositivo, o OR solicita a instalação das regras de encaminhamento de tráfego ao MS e informa o MN que o processo está pronto e a função pode ser iniciada. A comunicação entre os módulos é realizada de forma assíncrona, sendo que o OR, após enviar uma requisição para qualquer um dos módulos, só inicia a próxima etapa após receber a confirmação de que a etapa anterior foi concluída com sucesso.

## 4.3 DISCUSSÃO E LIMITAÇÕES

Ao projetar a arquitetura proposta, levou-se em consideração as principais ideias das plataformas já existentes, bem como dos paradigmas utilizados. Para a definição da VNFs, optou-se por estender uma plataforma de construção de VNFs modular e flexível, mantendo os conceitos originais de generalidade e flexibilidade da arquitetura base, e adicionando um novo módulo que não interfere no funcionamento do restante da plataforma. Embora algumas limitações tenham sido impostas por esta abordagem, como a de não modificar a linguagem de configuração da VNF, e não permitir que o módulo processe tráfego, esta arquitetura modificada é compatível com funções legadas e cenários onde não há a necessidade de *offload*.

Com relação à arquitetura do gerenciador de *offload*, após a definição dos requisitos necessários, uma abordagem simplificada e modular, em consonância com a arquitetura da VNF mostrou-se a mais apropriada, permitindo que o gerenciador suporte agentes externos para várias de suas tarefas, como a decisão sobre quando o *offload* deve ocorrer, a escolha dos dispositivos que farão parte do processo, bem como a tradução de regras entre os diferentes paradigmas. Visto que existem várias pesquisas referentes a estas áreas, um enorme grau de flexibilidade é trazido ao gerenciador, que em sua essência traz uma forma de integrar estes outros trabalhos de uma forma que seu uso seja possível para realizar cenários de *offload*. Além disso, a modularidade traz benefícios ao suporte de novas ferramentas e tecnologias, como por exemplo o módulo de usuário que pode ser modificado para suportar orquestradores de alto nível, e os

módulos de comunicação, que podem suportar diferentes controladores ou MANOs.

De fato, uma plataforma capaz de realizar todas estas tarefas de forma autocontida, como as apresentadas no capítulo anterior, trazem benefícios relacionados ao seu desempenho e a sua implantação nas infraestruturas, junto com o suporte para VNFs que o operador de rede não possua acesso interno (*black boxes*). No entanto, um último benefício trazido pela arquitetura proposta refere-se ao seu potencial de uso em plataformas de testes. Atualmente, como será visto no próximo capítulo, plataformas capazes de simular cenários de *offload* são escassas (a única encontrada é a que será utilizada nos testes) e, pela arquitetura proposta ser capaz de executar sem a necessidade de uma infraestrutura NFV implantada de forma completa, apresenta-se como uma boa alternativa para ser utilizada em simulações e ambientes de testes, o que dado a complexidade do processo de *offload*, é uma necessidade cada vez maior com a popularização de infraestruturas programáveis.

# 5 DESENVOLVIMENTO DO PROTÓTIPO

Para realizar a validação e testes de funcionamento da arquitetura proposta, um protótipo contendo as principais funcionalidades foi desenvolvido. Novamente, a implementação consiste de dois componentes, a plataforma de VNF e o gerenciador de *offload*. Para a plataforma de VNF, apenas o módulo *Offload Agent* foi desenvolvido, visto que já existe uma implementação da arquitetura base. Já com relação aos módulos do gerenciador, foram implementadas as funções necessárias para a execução do processo de *offload* para um cenário específico de testes, descrito no capítulo seguinte.

## 5.1 PLATAFORMA DE VNF

A plataforma de VNF utiliza o protótipo COVEN[3], desenvolvido no contexto do artigo de (Garcia et al., 2019). O COVEN é um protótipo que implementa os principais módulos da plataforma, e que pode ser instanciado em um sistema operacional tradicional. A implementação de IR e VNS disponíveis no COVEN é baseada em *sockets* L2, enquanto que PPSs podem ser implementados utilizando Click, Python, JAVA e C. Nos módulos que já estão implementados, não houve alteração no código, bastando apenas clonar o repositório e instalar as dependências necessárias.

O COVEN também possui o módulo de gerenciamento desenvolvido, no entanto ele comunica-se com os outros módulos apenas localmente, não possuindo suporte a gerenciamento de forma remota. Como isto é necessário para a inicialização do OA e recuperação de informações, um serviço REST capaz de servir requisições remotas baseado no EMS Agent[4] foi utilizado. Este serviço utiliza o swagger-codegen[5] para gerar uma API REST com servidor Python Flask[6] que pode então ser associada a comandos executados no sistema operacional, sendo então utilizada para comandar o gerenciador interno de forma remota, chamar a execução do OA, e retornar os dados produzidos por ele.

A configuração do COVEN utiliza um arquivo de configuração YAML, onde são definidos os PPS utilizados, as portas de entrada e saída da VNF, e arquivos de configuração exclusivos para cada PPS. Os arquivos devem estar localmente na VNF para o processo de con-

[3] http://github.com/ViniGarcia/COVEN
[4] https://github.com/lmarcuzzo/ems_agent
[5] https://swagger.io/tools/swagger-codegen/
[6] https://palletsprojects.com/p/flask/

figuração, podendo ser enviados através do EMS Agent. O arquivo de configuração Click é o arquivo utilizado pelo OA para gerar a nova função e recuperar informações, e devido a isto possui um nome e local fixo dentro do sistema operacional.

Basicamente, a implementação desenvolvida durante esta dissertação consiste do OA que será apresentado a seguir, e de algumas funções de suporte no `ems_agent`:

- `execute_offload_agent`: Faz a chamada do *script* `offload_agent.py` dentro da VNF.
- `get_offload_element_list`: Retorna o arquivo `offload_element_list.json`, que é a lista gerada pelo *script* anterior.
- `coven_startup`: Chamada de inicialização do gerenciador interno do COVEN.
- `revert_offload`: Substitui o arquivo de configuração Click pelo arquivo original (antes do *offload*).

Outras funções utilizadas, como funções de leitura e escrita de arquivos, verificação de processo por PID, e logs do sistema já estavam implementadas no EMS Agent. Por fim, ao invés de executar os serviços em uma máquina virtual, um **container** Docker utilizando o sistema operacional `debian:slim` como base. Isto é necessário pois a plataforma de testes utilizada não possui suporte a máquinas virtuais, apenas *containers*.

### 5.1.1 Offload Agent

A implementação do OA se dá através de um *script* que analisa a configuração da função a ser executada no PPS, gera uma nova função com os elementos que sofrerão *offload* removidos, e extrai informações relevantes para serem enviadas ao gerenciador de *offload*, como o nome dos elementos, sua ordem de execução dentro do PPS, e as regras associadas a eles. No protótipo, o OA foi implementado apenas para funções de rede baseadas no Click Modular Router. O suporte a novos processadores de pacotes é possível devido a modularidade do OA, no entanto isto requer o desenvolvimento de um *front-end* de comunicação entre o OA e o processador de pacote, dado que a forma de recuperação de informações, bem como a linguagem utilizada é diferente entre os processadores.

A linguagem Click é bastante expressiva, suportando várias formas de chamada dos elementos e das regras de configuração, o que torna a implementação de um *parser* completo

em um protótipo básico inviável. A sintaxe da linguagem Click (KOHLER et al., 2000b) define diversos operadores, conexões, grupos de elementos, elementos compostos e sobrecarga de operadores. A implementação de todos estes construtores requer o desenvolvimento de um *parser* complexo, sendo que alguns operadores não podem ser facilmente adaptados para outras linguagens mais restritivas. Desta forma, a implementação do agente focou em duas formas de declaração que são mais tipicamente utilizadas pelo Click: a declaração de elementos e a declaração de conectores. Estas declarações foram escolhidas pois, após uma análise nas configurações de exemplos no repositório do Click [7], foi possível visualizar que eram as mais utilizadas para a declaração de funções.

A declaração de um elemento é feita da seguinte forma:

```
Elemento1(regras) :: alias
Elemento1(regras) :: alias2
```

Esta declaração, chamada de prefixada, é realizada antes da declaração de conectores, e define o nome do elemento, as regras associadas, e o nome pelo qual ele será chamado na declaração de conectores. A declaração dos conectores, por sua vez, se faz da seguinte forma:

```
Elemento1(regras) → alias → Elemento2(regras)
```

Os elementos podem ser declarados diretamente com os conectores, ou então através dos *aliases* definidos anteriormente. O símbolo → indica uma conexão entre elementos, sendo uma declaração de entrada quando está antes e uma conexão de saída quando está após. A ordem dos elementos é dada simplesmente pela sua posição na declaração de conectores, por exemplo, o Elemento1 é o primeiro e assim por diante.

O *script* implementado, ao receber uma solicitação de *offload*, ao invés de enviar o arquivo de configuração da função diretamente ao PPS envia ela ao OA, que faz a leitura do arquivo e armazena os elementos declarados de forma pré-fixada em um dicionário onde a chave é o *alias*, para posterior substituição nos conectores. O OA então carrega a lista de elementos com suporte a *offload* disponibilizada pelo gerenciador de *offload*, e salva a configuração original com um nome diferente (para ser restaurada caso o processo não seja concluído). Após, a primeira informação extraída refere-se aos elementos *FromDevice* e *ToDevice*, que funcionam respectivamente como a entrada e saída de dados na função. A regra associada a estes elementos nada mais é do que o nome da interface de rede de onde os dados chega e para onde são enviados, sendo que com esta informação é possível identificar o endereço MAC das interfaces,

[7] https://github.com/kohler/click/tree/master/conf

necessário para o encaminhamento pelo dispositivo programável. Estes elementos também não serão removidos do código gerado, pois ainda se fazem necessários.

A *string* de configuração dos conectores é lida e separada pelo símbolo $\rightarrow$, sendo que os *aliases* são substituídos pela sua declaração pré-fixada. Os elementos divididos são então comparados com a lista de elementos recebida e, caso o elemento seja encontrado, é adicionado em uma nova lista (lista de elementos para *offload*) que será enviada ao gerenciador de *offload*. A nova configuração sem os elementos que sofrerão *offload* é salva na VNF, e a outra lista (com os elementos que sofrerão *offload*) é disponibilizada através de uma chamada REST.

## 5.2 *FRAMEWORK* DE *OFFLOAD*

A seguir, os módulos do gerenciador de *offload* implementados são apresentados. A linguagem Python foi utilizada para sua implementação, sendo que a comunicação entre os módulos é feita através de chamadas REST. Embora no ambiente de desenvolvimento do protótipo os módulos sejam executados no mesmo sistema operacional, a escolha de comunicação entre eles via REST permite sua execução distribuída. Por exemplo, o OM, que é o ponto central de conexão entre os outros módulos pode ser executado em uma VNF instanciada na infraestrutura junto com o BD de elementos e o módulo de tradução, enquanto que o módulo de comunicação SDN pode ser executado como uma aplicação SDN no controlador, o módulo de comunicação NFV de forma análoga no MANO, e o módulo de usuário executado no próprio computador do operador de rede.

Uma descrição de como cada um dos módulos foi implementado, bem como as funções externas disponibilizadas por eles são apresentadas a seguir.

### 5.2.1 Módulo de Usuário

O módulo de usuário também foi implementado como um *script*, por onde o operador de rede ou orquestrador envia as requisições para o gerenciador de *offload*. Como falado anteriormente, a implantação de forma separada do gerenciador permite que o suporte a novas plataformas possa ser desenvolvido sem a necessidade de modificar o gerenciador. Chamadas no módulo são realizadas da seguinte forma:

```
UserModule.py ip_do_gerenciador função parâmetros
```

As funções disponíveis inicialmente são as seguintes:

Figura 9 – Configuração de *Offload* passada ao agente.

```
DEVICES:
  - SRC_DEVICE:
    - mgmt_ip: 10.0.0.1
    - type: coven
  - DST_DEVICE:
    - name: bmv2-s1
    - type: P4Runtime
  SDN_CONTROLLER:
  - mgmt_ip: 10.0.1.1
  - type: onos
```

Fonte: do Autor.

- `start_offload`: Esta função toma um arquivo de configuração de *offload* como parâmetro e envia ela ao gerenciador de offload, retornando se o *offload* foi possível ou não, e um identificador único dele.
- `get_offload_metrics`: Recebe como parâmetro o identificador único do *offload*, e retorna se o *offload* está ativo. Esta função também pode ser utilizada para retornar métricas específicas para um orquestrador.
- `revert_offload`: Recebe um identificador de *offload*, e solicita ao gerenciador que o *offload* seja parado e a função revertida para a original. Retorna se a operação foi concluída com sucesso ou não.
- `insert_new_element`: Recebe como parâmetro um arquivo de implementação do elemento e o nome a ser dado, e insere ele no BD de elementos. Retorna se a operação foi concluída com sucesso.
- `remove_element`: Recebe o nome do elemento e o remove do banco. Retorna se a operação foi concluída com sucesso.

A biblioteca `python-requests` [8] é utilizada para fazer e receber as solicitações do gerenciador. Funções adicionais podem ser implementadas neste módulo para dar suporte a um orquestrador de alto nível, como por exemplo a tradução de arquivos de configuração ou de métricas recebidas.

Um exemplo do arquivo de configuração utilizado está apresentado na Figura 9. Esta configuração representa a forma mais simplificada de *offload*, onde se define apenas a origem

[8] https://github.com/psf/requests

e o destino da função. Para isto, assume-se que a função Click e a VNF já estão instanciadas, assim como o controlador SDN já está acessível. Como parâmetros opcionais, também podem ser adicionadas uma lista de elementos de configuração a serem colocados no banco de dados, e uma tabela de equivalência para a conversão dos formatos.

### 5.2.2 Módulo de Tradução e Banco de Dados de Elementos

A linguagem Python também foi utilizada para o desenvolvimento do módulo de tradução, enquanto que o banco de dados utiliza SQLite3 [9]. Junto ao código do módulo, estão o arquivo do banco de dados (`element.db`), um arquivo base de uma arquitetura de *switch* P4 (baseado no código disponível em [10]), e um arquivo com o mapeamento dos tipos entre arquiteturas diferentes. Como o módulo realiza a compilação de código P4, o ambiente onde ele será executado deve possuir instalada a *toolchain* de compilação P4.

Para a interação com o banco de dados, o módulo implementa funções básicas de acesso ao banco, como escrita, leitura e atualização, que podem ser chamadas internamente para a recuperação dos elementos, ou externamente para incluir e apagar elementos pelo módulo de usuário. Já para o processo de tradução, o módulo implementa uma função que recebe do gerenciador a lista de elementos, busca estes elementos no banco de dados, adiciona eles ao código base do *switch* e realiza a compilação, resultando em um código P4 compilado e, por fim, traduz as regras para o formato de destino do dispositivo.

O tradutor inicia com um código base da arquitetura do dispositivo, que pode ser customizado se disponibilizado pelo usuário, ou então utiliza um código padrão provido pela plataforma, e começa a organizar os elementos no código. O módulo tenta encaixar os elementos no código e, se conseguir, realiza a compilação tendo como alvo a arquitetura do dispositivo identificada pelo usuário ou pelas informações obtidas pelo módulo de gerenciamento de offload. Após o código conseguir ser compilado para a arquitetura alvo, verifica-se se as regras obtidas da VNF são compatíveis com o dispositivo de destino. Caso não sejam, é necessário que haja uma tabela de equivalência entre tipos para que o tradutor realize a tradução.

A linguagem P4 para a declaração dos elementos devido aos seus benefícios, permitindo a programação de dispositivos programáveis através de uma linguagem de alto nível, e padronizando a compilação entre diferentes arquiteturas. No contexto do módulo de tradução, um

[9] https://www.sqlite.org/
[10] https://github.com/opennetworkinglab/onos/blob/master/apps/p4-tutorial/pipeconf/src/main/resources/mytunnel.p4

Figura 10 – Exemplo de um classificador TCP implementado em P4.

```
//Dropa pacote
action drop() {
    mark_to_drop(standard_metadata);
}
//Envia para porta
action tcp_forward(macAddr_t dstAddr, egressSpec_t port) {
    //Seta porta de saida
    standard_metadata.egress_spec = port;
    //Seta mac de origem = switch e mac de destino
    hdr.ethernet.srcAddr = hdr.ethernet.dstAddr;
}
//Implementa match de protocolo TCP por porta
table tcp_fwd{
    key = {
        hdr.tcp.dstPort: exact;
    }
    actions = {
        tcp_forward;
        drop;
        NoAction;
    }
    size = 1024;
    default_action = NoAction();
}
```

Fonte: do Autor.

elemento é definido como uma tabela P4 e as suas *actions* associadas. Um exemplo de um elemento implementado em P4 é mostrado na Figura 10, seguindo o modelo *match+action*. Ações para descarte de pacotes (obrigatório) e encaminhamento do pacote são definidas, sendo que a ação de encaminhamento necessita do MAC de destino (obtido pelo OA) e da porta de saída do dispositivo (visto mais a frente). Já na tabela, é vista a comparação da porta do cabeçalho TCP para definir quando a ação será tomada.

A declaração do cabeçalho TCP é feita no próprio código P4, sendo que cada protocolo precisa ter sua declaração de cabeçalho. Na implementação utilizada no protótipo, foram implementados cabeçalhos para os protocolos Ethernet, ARP, ICMP, IPv4, TCP e UDP. Também é necessário colocar estes elementos dentro de um bloco de controle do código, sendo que neste caso todos são colocados no bloco *ingress*. Por fim, a ordem de aplicação das tabelas é implementada ao fim do bloco de controle, como mostrado no código da Figura 11.

A última função a ser realizada pelo módulo é a tradução das regras. Como um exemplo,

Figura 11 – Tabela adicionada ao *pipeline* P4.

```
apply {
    if (hdr.tcp.isValid()) {
        tcp_fwd.apply();
    }
}
```

Fonte: do Autor.

a adição de uma regra de encaminhamento para o elemento descrito anteriormente é feita da seguinte forma:

```
table_add tcp_fwd tcp_forward 80 => 00:00:00:00:00:00 1
```

A função table_add (implementada no switch) é utilizada na tabela tcp_fwd, para que a ação tcp_forward seja chamada quando a porta TCP for 80, e os parâmetros da ação (MAC e porta de saída) são dados após o =>. Ao final do processo de tradução, devem ter sido gerados um arquivo com o código compilado, e um arquivo com as regras formatadas.

Desta forma, as funções disponíveis para uso do módulo são as seguintes:

- `insert_element` e `remove_element`: Estas funções servem para, respectivamente, inserir e remover elementos no banco de dados, recebendo como parâmetro o código do elemento com seu nome como chave para inserção, e o nome do elemento como chave para remoção;
- `translate_function`: Recebe como parâmetro a lista de elementos e regras obtidos do OA, o MAC da interface de entrada da VNF, e a porta do dispositivo onde a VNF está conectada, e tenta compilar o código com base nestas informações.
- `get_p4_json`: Retorna o arquivo do código gerado pela função `translate_function`.
- `get_translated_rules`: Retorna as regras formatadas geradas pela função.

### 5.2.3 Módulos de comunicação

Dois módulos de comunicação foram implementados, o módulo de SDN e o módulo de NFV. O módulo de NFV é um *script* Python que faz requisições ao MANO ou a uma VNF em específico. No contexto do protótipo, este módulo apenas realiza requisições ao MA da VNF alvo, através de um endereço IP provido pelo gerenciador de *offload*, e retorna os dados a ele. A

implementação em separado deste módulo deve-se ao fato que, caso seja necessário conectar-se com orquestradores ou algum VNFM, possivelmente será necessário preparar requisições customizadas para solicitação de informações destas ferramentas. Como as funções do OA já foram apresentadas anteriormente, as funções do módulo NFV não serão apresentadas aqui.

O módulo SDN, por sua, vez, possui uma aplicação mais destacada no protótipo. Como a arquitetura requer um controlador SDN, não devendo conectar-se diretamente a dispositivos programáveis, uma API de requisições ao controlador ONOS foi implementada. Esta API é utilizada pelo gerenciador de *offload* para recuperar informações fundamentais, como os recursos dos dispositivos programáveis, e a localização dos *hosts* na infraestrutura. Desta forma, as seguintes funções são implementadas pelo módulo de SDN:

- `get_device_by_id`: Esta função recebe como parâmetro a ID do dispositivo programável (provida pelo arquivo de configuração no módulo de usuário), e solicita ao controlador via REST os dados deste dispositivo, como modelo e protocolo utilizado para conexão.
- `get_VNF_by_id`: Esta função recebe como parâmetro o endereço MAC da interface da VNF w disponibiliza diversas informações relacionadas a esta, como a porta onde está conectada e endereço IP da interface.
- `install_application`: Esta função recebe como parâmetro um código compilado em P4 e a ID do dispositivo e solicita a sua instalação ao controlador.
- `install_rules`: Esta função recebe como parâmetro a lista de regras disponibiliza pelo tradutor e instala elas no dispositivo alvo.

As funções de recuperação de dados são utilizadas inicialmente pelo gerenciador de *offload* para recuperar o endereço MAC da VNF, conseguindo com isso identificar através da função de localizar VNF a porta onde está conectada e seu endereço IP. Após são recuperadas informações do dispositivo programável alvo, como o protocolo utilizado (identificando seu modelo), e se está executando alguma configuração. Ao receber o código e as regras do gerenciador de *offload* (após a execução do módulo de tradução), as funções de instalação são utilizadas para enviar os dados ao dispositivo.

### 5.2.4 Módulo de gerenciamento de *offload*

Por fim, o módulo de gerenciamento de *offload* é implementado como um serviço de interconexão entre os outros módulos. A implementação utiliza a ferramenta *Swagger Codegen*, capaz de gerar definições de servidores para várias linguagens com base em uma especificação JSON. O módulo de offload executa um servidor REST que recebe chamadas do módulo de usuário e delega tarefas aos outros módulos. Por exemplo, é ele que solicita as informações sobre a VNF e sobre a topologia de rede dos módulos SDN e NFV, consolida estas informações e as envia ao módulo de tradução. Após o processo de tradução, ele solicita o código e as regras ao módulo de tradução, envia elas para instalação através do módulo SDN, e sinaliza o módulo NFV que o processo está concluído e a VNF pode ser iniciada.

# 6 AVALIAÇÃO E VALIDAÇÃO

Com um protótipo funcional implementado, neste capítulo é realizada a validação e avaliação da arquitetura proposta. Inicialmente são definidas e justificadas as métricas utilizadas para a avaliação. Após, são apresentadas as ferramentas necessárias para instanciar a infraestrutura de testes, bem como o cenário a ser utilizado na avaliação. Por fim, os resultados obtidos são apresentados e discutidos.

## 6.1 MÉTRICAS DE AVALIAÇÃO

*Offload* pode trazer diversos benefícios para infraestruturas de rede, por exemplo: segurança, flexibilidade e maior desempenho. Em específico, o ganho no desempenho trazido pela utilização de técnicas de *offloading* pode ser quantificado observando duas métricas principais (*i.e.*, vazão e latência), as quais serão consideradas ao longo deste capítulo.

- **Vazão** representa a maior taxa possível de encaminhamento pelos dispositivos que estão sendo testados (BRADNER; MCQUAID, 1999). Para a realização dos testes de vazão, um *host* de origem encaminha tráfego a um *host* de destino, de forma que os dispositivos avaliados estejam no caminho deste tráfego. Quanto maior a vazão obtida, maior a capacidade dos dispositivos de rede de processar e encaminhar tráfego. O teste deve ser repetido para diferentes tipos de pacote, para que seja avaliado não só a vazão, mas a capacidade de processamento dos dispositivos. A vazão foi escolhida para avaliação porque um dos benefícios de se executar parte de uma função de rede em um dispositivo programável é a sua maior capacidade de processar e encaminhar pacotes.

- **Latência**, ou atraso, representa o tempo que um pacote leva para percorrer a infraestrutura entre a origem e o destino. A latência pode ser dividida entre a latência de transmissão (o tempo que um pacote leva de um dispositivo ao outro) e a latência de processamento (o tempo que leva para um pacote ser processado no dispositivo). Duas principais formas de medir a latência incluem o tempo de ida de um ponto a outro (*One Way Delay*), ou então o tempo de ida e volta a partir de sua origem (*Round Trip Time*). Da mesma forma que a vazão, a redução na latência de processamento é um benefício trazido pelo *offload*, devendo ser avaliado.

Embora estas métricas tenham sido definidas originalmente para a avaliação de dispositivos físicos, elas também podem ser utilizadas para dispositivos virtualizados, representando de uma forma quantitativa o desempenho atingido pelos dispositivos.

## 6.2 FERRAMENTAS E AMBIENTE DE TESTES

O ambiente de testes é executado de forma completamente virtualizada em um computador com um processador Core i5-8350U@1.70Ghz, com TurboBoost até 3.60Ghz, 8 Gb de memória DDR4-2400 e sistema operacional Ubuntu 20.04. A composição do ambiente de testes requer a instanciação de diversas ferramentas, como a plataforma de testes FOP4 (baseada no Containernet), o controlador SDN ONOS, e a *toolchain* de compilação do dispositivo programável BMv2. Uma breve descrição das aplicações mais importantes será apresentada a seguir.

### 6.2.1 FOP4

A plataforma FOP4 (Moro et al., 2019) é um ambiente de testes desenvolvido como uma extensão da plataforma Containernet (Peuster; Kampmeyer; Karl, 2018) que permite a prototipação de cenários heterogêneos compostos por *containers* e dispositivos programáveis. A plataforma implementa suporte ao *switch* P4 BMv2, e também possui suporte a SmartNICs. Os dispositivos BMv2 podem então ser configurados através de um arquivo de configuração JSON produzido pelo compilador `bcc`[11], ou então por uma configuração *pipeconf*, utilizada pelo controlador SDN ONOS. Por se basear na plataforma Containernet, que por sua vez é um *fork* do Mininet, topologias podem facilmente ser implantadas e testadas, permitindo inclusive a utilização de VNFs através de *containers*. Atualmente, a plataforma FOP4 é a única disponível capaz de suportar testes com cenários de *offload*, e uma de suas limitações, e também do dispositivo BMv2, é o seu desempenho limitado, visto que toda a infraestrutura é executada de forma virtualizada. No entanto, a plataforma se mostra adequada para a validação de um *framework* de *offload*, por permitir que as principais funções do protótipo desenvolvido sejam testadas.

[11] https://github.com/iovisor/bcc

### 6.2.2 ONOS

O *Open Network Operating System* - ONOS (BERDE et al., 2014) é um controlador SDN distribuído desenvolvido com foco em escalabilidade e modularidade. O ONOS oferece uma visão e gerenciamento global da rede, por onde aplicações interagem com os dispositivos da infraestrutura. Diversas aplicações foram desenvolvidas para o controlador, que hoje é capaz de suportar diversos protocolos de configuração diferentes (*e.g, OpenFlow, NETCONF e P4Runtime*). Entre os aspectos que levaram a escolha do ONOS para a utilização no ambiente de testes, estão a disponibilidade de *drivers* para o *switch* BMv2 e o protocolo P4Runtime, sua grande quantidade de aplicações disponíveis (sendo que algumas, como a descoberta de *hosts* são obrigatórias para o funcionamento do *framework*), e a sua adoção em outras plataformas de *offload* como Metron (BARBETTE et al., 2018) e como o padrão da indústria, dado que diversos fabricantes implementam suporte ao ONOS.

### 6.2.3 Ferramentas de medição e monitoramento

Com relação as ferramentas utilizadas para avaliar as métricas definidas anteriormente, a ferramenta Iperf [12] foi utilizada para a medição de vazão da infraestrutura, enquanto que a ferramenta D-ITG [13] foi utilizada para a medição de latência, por permitir simular protocolos comumente usados em redes.

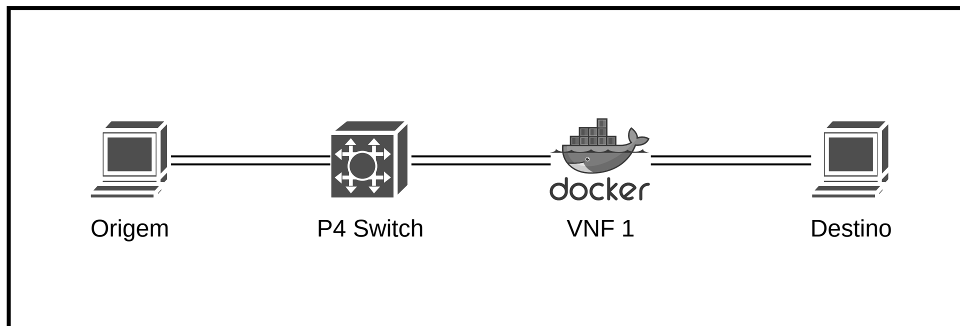
## 6.3 CENÁRIO DE AVALIAÇÃO E METODOLOGIA

Para a avaliação da plataforma, o cenário descrito na Figura 12 foi utilizado. Este cenário consiste de dois *hosts* conectados através de um *switch* P4 e uma VNF. Neste cenário, a VNF está executando uma função de classificação que permite apenas que pacotes IPv4 sejam encaminhados para a função de classificação seguinte, que permite apenas TCP na porta 23 e UDP. Esta topologia é executada na plataforma FOP4, sendo que a VNF é implantada como um *container* e conectado a plataforma FOP4 através de `linux-bridges`. A partir desta topologia, são realizados testes de vazão e de latência.

No teste de vazão, a ferramenta `iperf` foi utilizada, com pacotes UDP. De acordo com a RFC 2544, devem ser utilizados datagramas de diferentes tamanhos, entre 64 *Bytes*

[12] https://iperf.fr/
[13] http://traffic.comics.unina.it/software/ITG/index.php

Figura 12 – Cenário de teste de desempenho




Fonte: do Autor.

e o MTU máximo da rede (neste caso, 1470 *Bytes*). Inicialmente um teste é realizado com todos os elementos executando na VNF e o dispositivo programável fazendo apenas um papel de encaminhamento do tráfego. Após, o processo de *offload* é realizado, e o mesmo teste é repetido, com a diferença que agora o elemento de classificação IP não executa mais na VNF, e sim no dispositivo programável. Os resultados obtidos são então comparados para verificar o desempenho nos dois casos.

Já para a avaliação de latência, a ferramenta D-ITG foi utilizada. Com o objetivo de simular um tráfego menos sintético que o gerado pelo `iperf`, foram escolhidos o protocolo VoIP (RTP) para avaliação de latência em UDP, e Telnet para avaliação em TCP. Como o envio de pacotes pelo `iperf` é feito através de *bursts* (todos os pacotes são enviados em sequência), a utilização destes protocolos para avaliar a latência pode ajudar a identificar problemas com relação ao encaminhamento ou enfileiramento dos pacotes. As mesmas diferenças entre os testes de vazão se aplicam para o teste de latência (o elemento de classificação IP é executado primeiro na VNF e após no dispositivo).

Pelo fato de todo ambiente estar sendo executado de forma virtualizada, e devido ao dispositivo programável utilizado (BMv2), foram dedicados dois núcleos físicos ao processo do BMv2, enquanto que a VNF executada possui um núcleo físico a disposição. Os testes foram repetidos 30 vezes e foi possível atingir um intervalo de confiança de 95%.

As configurações Click utilizadas são mostradas em 13. Após a entrada do pacote na VNF, ele é classificado em camada dois (*i.e.*, observando o cabeçalho de enlace) pelo elemento *Classifier* e então encaminhado para validação do pacote. Caso seja válido, é realizada a classificação TCP ou UDP e encaminhamento para a interface de saída. Na configuração do cenário de *offload*, a função de classificação IP é removida da VNF.

Figura 13 – Configuração Click antes e após o processo de *offload*.

```
//Configuracao Click sem Offload

FromDevice(eth0) -> Classifier(12/0800,-) ->
CheckIPHeader(14) -> IPClassifier(src tcp port 23,
udp,-)-> ToDevice(eth1)

//Configuracao Click com Offload

FromDevice(eth0) -> CheckIPHeader(14) ->
IPClassifier(src tcp port 23,udp,-) -> ToDevice(eth1)
```

Fonte: do Autor.

## 6.4 RESULTADOS

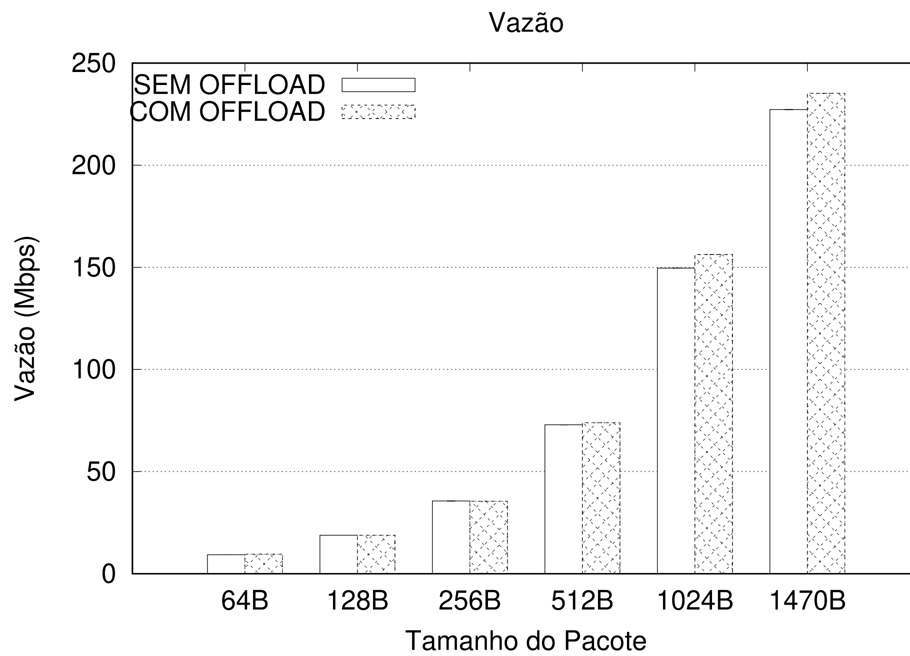
Na Figura 14 estão os resultados de vazão obtidos durante a realização dos testes. Para pacotes menores (64, 128 e 256B), não é possível visualizar uma diferença perceptível entre os dois cenários avaliados. Isto pode ser relacionado a baixa taxa de vazão obtidas nestes casos, pois a partir de 512B até 1470B (o MTU máximo) a diferença de desempenho entre os cenários é crescente. Dispositivos programáveis são capazes de processar tráfego na casa dos Gbps, e a tendência é que a diferença de desempenho seja maior nestas situações.

Embora a plataforma de VNF e o PPS escolhidos sejam capazes de processar tráfego com alto desempenho, a baixa vazão apresentada nos testes decorre do uso de um ambiente virtualizado executando toda a infraestrutura e compartilhando recursos, além do uso do *switch* de referência BMv2, o qual foi desenvolvido com o propósito de validar o funcionamento de funções, não sendo adequado para ser utilizado em ambientes de produção ou competir com dispositivos físicos ou *carrier-grade*.

Os testes de latência realizados são apresentados na Figura 15. Ambos protocolos apresentam uma latência inferior a 2 ms, sendo que o protocolo VoIP apresenta uma latência maior que o protocolo Telnet. Em ambos os testes (com e sem *offload*) não houve uma diferença significativa na latência do protocolo VoIP, pois a VNF não estava com recursos esgotados, ou seja, era capaz de processar todo o tráfego, e também porque o elemento o qual foi feito *offload* era responsável pela classificação IP, e não TCP ou UDP.

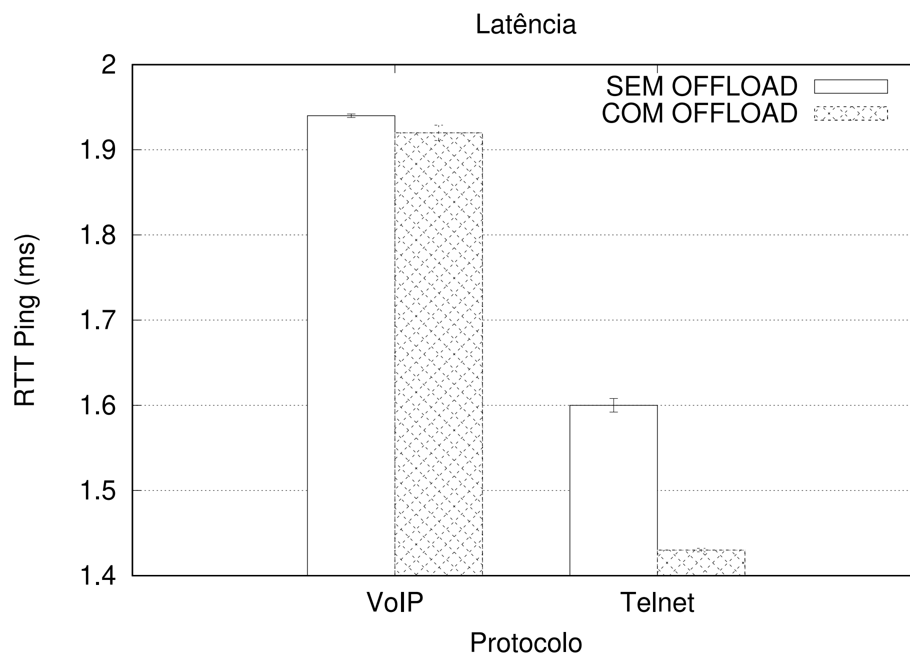
Já com relação ao protocolo Telnet, é possível visualizar uma diferença maior de desempenho no cenário com e sem *offload*, na ordem de 0,1 a 0,2 ms. Inicialmente não foi possível

Figura 14 – Testes de vazão para diferentes tamanhos de pacotes.




Fonte: do Autor.

Figura 15 – Latência para protocolos VoIP e Telnet




Fonte: do Autor.

identificar o que causou este desempenho diferente, visto que o cenário era semelhante ao do teste com VoIP, onde o elemento que sofreu *offload* era responsável por classificação IP e a VNF não estava saturada. Após análise do ambiente de testes, uma explicação plausível é relacionada ao modo de conexão entre o dispositivo programável e o *container* onde a VNF executa.

Na plataforma FOP4, a conexão entre o *container* e o restante da topologia é realizada através de *linux-bridges*, já que o container é executado de forma *standalone*, e por isso não pode compartilhar a pilha de rede do *host*. Como apresentado no capítulo 3, interfaces de rede físicas implementam uma forma limitada de *offload*, onde funções como o *checksum* TCP e UDP, cálculo da janela deslizante TCP e controle de congestão são implementadas em *hardware*.

A implementação destas funcionalidades é necessária pois não há garantias em uma infraestrutura física de que os pacotes chegarão ao destino da mesma forma que saíram da origem. No entanto, este problema não ocorre em ambientes virtualizados, já que os pacotes são encaminhados completamente em *software*, de modo que *bridges* são capazes de pular algumas destas etapas (como assumir que o *checksum* e a ordem dos pacotes esteja correta), implementando de certa forma este "*offload*".

No cenário onde a classificação IP é feita dentro do Click Modular Router, todas as etapas são executadas pois quem faz a validação do pacote é o Click, e não o *container* ou *host*. No entanto, ao retirar o elemento de classificação IP de dentro do Click, ele fica responsável apenas pelo encaminhamento TCP, permitindo que o *frame* Ethernet seja processado pelo *container* e, por consequência, as técnicas de *offload* da *bridge* sejam aplicadas.

É importante notar que isto não é um efeito do processo de *offload* implementado pela arquitetura proposta, mas sim de como o ambiente de testes foi implementado. No entanto, em cenários reais, onde o dispositivo programável não é virtualizado e sim conectado a VNF através de uma interface física, o mesmo efeito pode ocorrer, visto que a interface física poderá analisar o *frame* antes de encaminhá-lo a VNF.

Uma outra explicação pode ser relacionada a diferença na quantidade de pacotes enviada por cada um dos protocolos. Apesar de ter se definido o mesmo tempo de duração para ambos os testes, assim como a mesma quantidade de tráfego, cada protocolo simulou este tráfego de forma diferente, sendo possível que o atraso de processamento causado pela diferença na quantidade de pacotes tenha influenciado os resultados.

## 6.5 DISCUSSÃO

O cenário de testes utilizado consiste de uma infraestrutura mínima para o funcionamento da plataforma de *offload*, e foi executado de forma completamente virtualizada. Embora seja possível visualizar um aumento de desempenho quando utilizando o *offload*, uma avaliação mais aprofundada é necessária para avaliar o benefício trazido pelo *offload* com relação ao desempenho das funções. No entanto, para que seja possível avaliar o desempenho do *offload* em *hardware*, se faz necessária a obtenção de um dispositivo físico, o que não foi possível durante o desenvolvimento desta dissertação.

Apesar disso, o protótipo desenvolvido foi capaz de executar com sucesso todo o processo de implantação das funções de rede. Como todo o ambiente testado foi executado no mesmo computador de forma virtualizada, também não foi possível avaliar a comunicação entre os componentes instanciados de forma distribuída, nem o tempo de execução das etapas individuais, visto que diversos fatores (*e.g.*, o compartilhamento de recursos) tornariam os resultados inválidos.

# 7 CONCLUSÃO

Considerando as grandes mudanças de necessidades ocorrendo em infraestruturas de redes atuais, novas tecnologias e paradigmas vêm sendo desenvolvidos com o objetivo de permitir o suporte das redes para diferentes demandas. Paradigmas como NFV, SDN e PDP vêm se destacando por apresentarem uma grande mudança na forma como topologias são implantadas, permitindo uma maior flexibilidade na utilização de recursos e implementações de funções de rede.

No entanto, limitações relacionadas ao desempenho, segurança e flexibilidade podem afetar a adoção destas tecnologias, de modo que técnicas vêm sendo desenvolvidas com o objetivo de mitigar estes problemas. Uma das técnicas mais promissoras refere-se ao *offload* de parte do processamento destas funções para dispositivos programáveis, permitindo assim uma melhor utilização dos recursos da infraestrutura. No entanto, atualmente não existe uma solução amplamente adotada capaz de suportar os diferentes cenários de *offload* existentes.

Nesta dissertação foi proposta a arquitetura de um *framework* para o *offload* parcial de funções virtualizadas de rede em infraestruturas heterogêneas. Após uma revisão de literatura sobre os principais trabalhos existentes na área, foram identificados os requisitos necessários e uma arquitetura foi proposta. A arquitetura consiste de uma plataforma VNF modular com suporte a *offload*, e um gerenciador capaz de configurar o processo em conjunto com os paradigmas utilizados. A arquitetura da VNF baseia-se na plataforma proposta por (Garcia et al., 2019), sendo alterada para os propósitos da dissertação, enquanto que o gerenciador de *offload* é composto por vários módulos com funções distintas.

Após a apresentação da arquitetura, um protótipo foi desenvolvido e avaliado em um ambiente de testes virtualizado, com o objetivo de validar o funcionamento da arquitetura. Os resultados obtidos demonstram que a arquitetura e o protótipo desenvolvidos foram capazes de executar o processo de *offload* com sucesso, embora uma avaliação de desempenho não tenha obtido resultados válidos devido a limitações nos componentes do ambiente de testes utilizado.

Como trabalhos futuros, uma avaliação de desempenho mais exaustiva em um ambiente de testes propício se faz necessária, visando comprovar os benefícios trazidos pelo *offload*. A exploração dos outros benefícios, como segurança e flexibilidade, também pode ajudar a validar a arquitetura proposta. Melhorias no módulo de tradução da plataforma podem trazer benefícios no suporte de novas VNFs ou arquiteturas de dispositivos programáveis. Além disso,

a plataforma também é capaz de suportas ferramentas de simulação de infraestruturas de redes e, assim, permitir que cenários de *offload* possam ser testados, visto que atualmente não existem muitas opções de plataformas de teste com suporte a *offload*.

# REFERÊNCIAS

Akpakwu, G. A. et al. A Survey on 5G Networks for the Internet of Things: communication technologies and challenges. **IEEE Access**, [S.l.], v.6, p.3619–3647, 2018.

BARBETTE, T. et al. Metron : nfv service chains at the true speed of the underlying hardware. In: USENIX SYMPOSIUM ON NETWORKED SYSTEMS DESIGN AND IMPLEMENTATION (NSDI ’18), 15. **Proceedings...** [S.l.: s.n.], 2018.

BERDE, P. et al. ONOS: towards an open, distributed sdn os. In: THIRD WORKSHOP ON HOT TOPICS IN SOFTWARE DEFINED NETWORKING, New York, NY, USA. **Proceedings...** Association for Computing Machinery, 2014. p.1–6. (HotSDN ’14).

BIFULCO, R.; RÉTVÁRI, G. A Survey on the Programmable Data Plane: abstractions, architectures, and open problems. In: IEEE INTERNATIONAL CONFERENCE ON HIGH PERFORMANCE SWITCHING AND ROUTING, HPSR 2018, 2018. **Anais...** [S.l.: s.n.], 2018.

BOSSHART, P. et al. Forwarding Metamorphosis: fast programmable match-action processing in hardware for sdn. **SIGCOMM Comput. Commun. Rev.**, New York, NY, USA, v.43, n.4, p.99–110, Aug. 2013.

BOSSHART, P. et al. P4: programming protocol-independent packet processors. **SIGCOMM Comput. Commun. Rev.**, New York, NY, USA, v.44, n.3, p.87–95, July 2014.

BRADNER, S.; MCQUAID, J. **Benchmarking Methodology for Network Interconnect Devices**. [S.l.]: RFC Editor, 1999. RFC. (2544).

CASADO, M. et al. Ethane: taking control of the enterprise. In: CONFERENCE ON APPLICATIONS, TECHNOLOGIES, ARCHITECTURES, AND PROTOCOLS FOR COMPUTER COMMUNICATIONS, 2007., New York, NY, USA. **Proceedings...** Association for Computing Machinery, 2007. p.1–12. (SIGCOMM ’07).

Chowdhury, S. R. et al. $\mu$NF: a disaggregated packet processing architecture. In: IEEE CONFERENCE ON NETWORK SOFTWARIZATION (NETSOFT), 2019. **Anais...** [S.l.: s.n.], 2019. p.342–350.

ETSI. **Network Functions Virtualisation – Introductory White Paper**. Acesso em 04 set. 2016.

FOUNDATION, O. N. Software-defined networking: the new norm for networks. **ONF White Paper**, [S.l.], 2012.

FOUNDATION, O. N. **SDN Architecture**. 2014.

Garcia, V. F. et al. On the Design of a Flexible Architecture for Virtualized Network Function Platforms. In: IEEE GLOBAL COMMUNICATIONS CONFERENCE (GLOBECOM), 2019. **Anais...** [S.l.: s.n.], 2019. p.1–6.

Gil Herrera, J.; Botero, J. F. Resource Allocation in NFV: a comprehensive survey. **IEEE Transactions on Network and Service Management**, [S.l.], v.13, n.3, p.518–532, 2016.

Hawilo, H. et al. NFV: state of the art, challenges, and implementation in next generation mobile networks (vepc). **IEEE Network**, [S.l.], v.28, n.6, p.18–26, 2014.

INTEL, D. Data Plane Development Kit. **URL http://dpdk. org**, [S.l.], 2014.

KATSIKAS, G. P. et al. SNF: synthesizing high performance nfv service chains. **PeerJ Computer Science**, [S.l.], v.2, p.e98, 2016.

KOHLER, E. et al. The Click Modular Router. **ACM Trans. Comput. Syst.**, New York, NY, USA, v.18, n.3, p.263–297, Aug. 2000.

KOHLER, E. et al. The Click modular router. **ACM Transactions on Computer Systems (TOCS)**, [S.l.], v.18, n.3, p.263–297, 2000.

Kreutz, D. et al. Software-Defined Networking: a comprehensive survey. **Proceedings of the IEEE**, [S.l.], v.103, n.1, p.14–76, Jan 2015.

LI, B. et al. Clicknp: highly flexible and high performance network processing with reconfigurable hardware. In: ACM SIGCOMM CONFERENCE, 2016. **Proceedings...** [S.l.: s.n.], 2016. p.1–14.

LINGUAGLOSSA, L. et al. Survey of Performance Acceleration Techniques for Network Function Virtualization. **Proceedings of the IEEE**, [S.l.], v.107, n.4, p.746–764, 2019.

MARCUZZO, L. et al. Análise e comparação de técnicas e aceleradores de processamento de pacotes. **Revista Eletrônica Argentina-Brasil de Tecnologias da Informação e da Comunicação**, [S.l.], v.2, n.1, 2018.

MCKEOWN, N. et al. OpenFlow: enabling innovation in campus networks. **ACM SIGCOMM Computer Communication Review**, [S.l.], v.38, n.2, p.69–74, 2008.

Mijumbi, R. et al. Network Function Virtualization: state-of-the-art and research challenges. **IEEE Communications Surveys Tutorials**, [S.l.], v.18, n.1, p.236–262, 2016.

Moro, D. et al. FOP4: function offloading prototyping in heterogeneous and programmable network scenarios. In: IEEE CONFERENCE ON NETWORK FUNCTION VIRTUALIZATION AND SOFTWARE DEFINED NETWORKS (NFV-SDN), 2019. **Anais...** [S.l.: s.n.], 2019. p.1–6.

NFV-INF, E. G. **NFV-INF 001 - Network Functions Virtualisation (NFV);Infrastructure Overview**. 2015.

NFV-MAN, E. G. **NFV-MAN 001 - Network Functions Virtualisation (NFV);Management and Orchestration**. 2014.

NFV-SWA, E. G. **NFV-SWA 001 - Network Functions Virtualisation (NFV);Virtual Network Functions Architecture**. 2014.

PALKAR, S. et al. E2: a framework for nfv applications. In: SYMPOSIUM ON OPERATING SYSTEMS PRINCIPLES, 25. **Proceedings...** [S.l.: s.n.], 2015. p.121–136.

Peuster, M.; Kampmeyer, J.; Karl, H. Containernet 2.0: a rapid prototyping platform for hybrid service function chains. In: IEEE CONFERENCE ON NETWORK SOFTWARIZATION AND WORKSHOPS (NETSOFT), 2018. **Anais...** [S.l.: s.n.], 2018. p.335–337.

RINTA-AHO, T.; KARLSTEDT, M.; DESAI, M. P. The Click2NetFPGA Toolchain. In: PRESENTED AS PART OF THE 2012 USENIX ANNUAL TECHNICAL CONFERENCE (USENIX ATC 12), Boston, MA. **Anais...** USENIX, 2012. p.77–88.

RIZZO, L. netmap: a novel framework for fast packet i/o. In: USENIX ANNUAL TECHNICAL CONFERENCE (USENIX ATC 12), 2012., Boston, MA. **Anais...** USENIX Association, 2012. p.101–112.

RUBOW, E. et al. Chimpp: a click-based programming and simulation environment for reconfigurable networking hardware. In: ACM/IEEE SYMPOSIUM ON ARCHITECTURES FOR

NETWORKING AND COMMUNICATIONS SYSTEMS (ANCS), 2010. **Anais...** [S.l.: s.n.], 2010. p.1–10.

SEKAR, V. et al. Design and Implementation of a Consolidated Middlebox Architecture. In: PRESENTED AS PART OF THE 9TH USENIX SYMPOSIUM ON NETWORKED SYSTEMS DESIGN AND IMPLEMENTATION (NSDI 12), San Jose, CA. **Anais...** USENIX, 2012. p.323–336.

SHERRY, J. et al. Making Middleboxes Someone else's Problem: network processing as a cloud service. In: ACM SIGCOMM 2012 CONFERENCE ON APPLICATIONS, TECHNOLOGIES, ARCHITECTURES, AND PROTOCOLS FOR COMPUTER COMMUNICATION, New York, NY, USA. **Proceedings...** ACM, 2012. p.13–24. (SIGCOMM '12).

WALFISH, M. et al. Middleboxes No Longer Considered Harmful. In: CONFERENCE ON SYMPOSIUM ON OPERATING SYSTEMS DESIGN & IMPLEMENTATION - VOLUME 6, 6., Berkeley, CA, USA. **Proceedings...** USENIX Association, 2004. p.15–15. (OSDI'04).

WIRED. **Revealed**: the secret gear connecting google's online empire. Acesso em 10 set. 2020.

YAMAZAKI, K. et al. Accelerating SDN/NFV with Transparent Offloading Architecture. In: OPEN NETWORKING SUMMIT 2014 (ONS 2014), Santa Clara, CA. **Anais...** USENIX Association, 2014.

Yang, W.; Fung, C. A survey on security in network functions virtualization. In: IEEE NETSOFT CONFERENCE AND WORKSHOPS (NETSOFT), 2016. **Anais...** [S.l.: s.n.], 2016. p.15–19.

Yousaf, F. Z. et al. NFV and SDN—Key Technology Enablers for 5G Networks. **IEEE Journal on Selected Areas in Communications**, [S.l.], v.35, n.11, p.2468–2478, 2017.