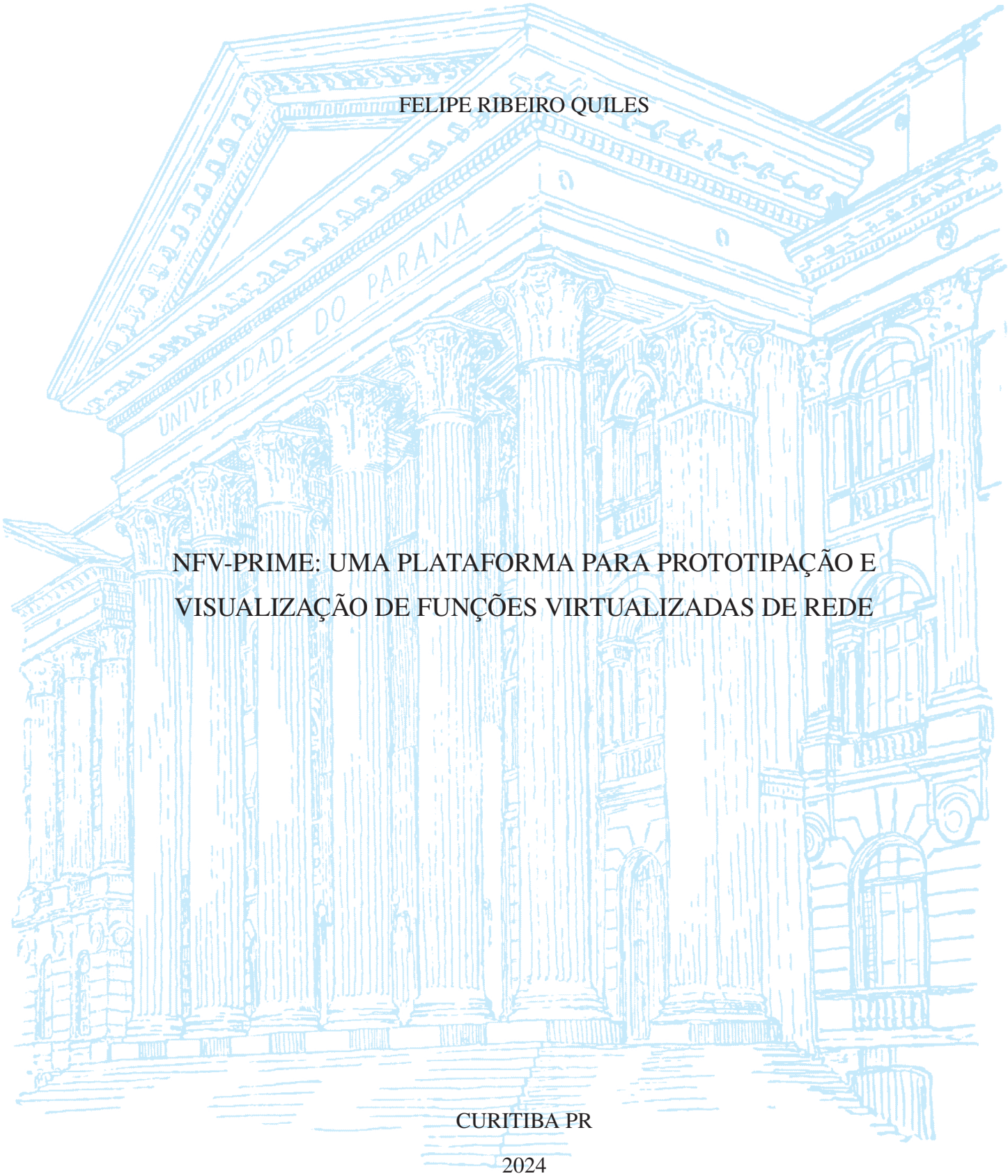
UNIVERSIDADE FEDERAL DO PARANÁ

FELIPE RIBEIRO QUILES



NFV-PRIME: UMA PLATAFORMA PARA PROTOTIPAÇÃO E VISUALIZAÇÃO DE FUNÇÕES VIRTUALIZADAS DE REDE

CURITIBA PR

2024

FELIPE RIBEIRO QUILES

NFV-PRIME: UMA PLATAFORMA PARA PROTOTIPAÇÃO E VISUALIZAÇÃO DE FUNÇÕES VIRTUALIZADAS DE REDE

Dissertação de Mestrado apresentada como requisito parcial à obtenção do grau de Mestre do Programa de Pós Graduação em Informática, Setor de Ciências Exatas, da Universidade Federal do Paraná.

Área de concentração: *Ciência da Computação*.

Orientador: Elias Procópio Duarte Jr.

Coorientador: Vinícius Fülber Garcia.

CURITIBA PR

2024

DADOS INTERNACIONAIS DE CATALOGAÇÃO NA PUBLICAÇÃO (CIP)
UNIVERSIDADE FEDERAL DO PARANÁ
SISTEMA DE BIBLIOTECAS – BIBLIOTECA DE CIÊNCIA E TECNOLOGIA

Quiles, Felipe Ribeiro
NFV-Prime: uma plataforma para prototipação e visualização de funções virtualizadas de rede / Felipe Ribeiro Quiles. – Curitiba, 2024.
1 recurso on-line : PDF.

Dissertação (Mestrado) - Universidade Federal do Paraná, Setor de Ciências Exatas, Programa de Pós-Graduação em Informática.

Orientador: Elias Procópio Duarte Júnior
Coorientador: Vinícius Fülber Garcia

1. Redes – Virtualização. 2. Prototipagem. 3. Virtualização de Funções de Rede. I. Universidade Federal do Paraná. II. Programa de Pós-Graduação em Informática. III. Duarte Júnior, Elias Procópio. IV. Garcia, Vinícius Fülber. V. Título.

Bibliotecário: Elias Barbosa da Silva CRB-9/1894

TERMO DE APROVAÇÃO

Os membros da Banca Examinadora designada pelo Colegiado do Programa de Pós-Graduação INFORMÁTICA da Universidade Federal do Paraná foram convocados para realizar a arguição da Dissertação de Mestrado de **FELIPE RIBEIRO QUILES** intitulada: **NFV-PRIME: UMA PLATAFORMA PARA PROTOTIPAÇÃO E VISUALIZAÇÃO DE FUNÇÕES VIRTUALIZADAS DE REDE**, sob orientação do Prof. Dr. ELIAS PROCÓPIO DUARTE JÚNIOR, que após terem inquirido o aluno e realizada a avaliação do trabalho, são de parecer pela sua APROVAÇÃO no rito de defesa.

A outorga do título de mestre está sujeita à homologação pelo colegiado, ao atendimento de todas as indicações e correções solicitadas pela banca e ao pleno atendimento das demandas regimentais do Programa de Pós-Graduação.

CURITIBA, 25 de Novembro de 2024.

Assinatura Eletrônica
28/11/2024 09:52:29.0
ELIAS PROCÓPIO DUARTE JÚNIOR
Presidente da Banca Examinadora

Assinatura Eletrônica
27/11/2024 08:01:55.0
CARLOS RANIERY PAULA DOS SANTOS
Avaliador Externo (UNIVERSIDADE FEDERAL DE SANTA MARIA)

Assinatura Eletrônica
04/12/2024 09:15:30.0
GIOVANNI VENÂNCIO DE SOUZA
Avaliador Interno (UNIVERSIDADE FEDERAL DO PARANÁ)

Assinatura Eletrônica
27/11/2024 16:39:20.0
VINÍCIUS FÜLBER GARCIA
Coorientador(a) (UNIVERSIDADE FEDERAL DO PARANÁ)

RESUMO

A tecnologia de Virtualização de Funções de Rede (*Network Functions Virtualization* - NFV) tem ganhado atenção como alternativa para a implementação de *middleboxes* em software, utilizando tecnologias de virtualização. *Middleboxes* implementam Funções de Rede (*Network Functions* - NF) e são disponibilizados tradicionalmente em hardware dedicado. As VNFs (*Virtualized Network Functions*) são *softwares* executados em *hardware* de prateleira, o que flexibiliza o gerenciamento da rede e diminui custos de capital e operação. Este trabalho propõe o NFV-Prime, uma plataforma para prototipação e visualização de VNFs, a qual visa facilitar a introdução do tema para novos usuários, e também, o processo de criar, instanciar e visualizar resultados de execução das VNFs na rede. Assim, o principal objetivo da plataforma é instigar e facilitar a exploração das possibilidades de implementações de NFs que o paradigma possibilita. A NFV-Prime consiste dos seguintes componentes: Editor, Gerenciador de Interfaces Virtuais de Rede, Gerador de Tráfego e Visualização. O Editor fornece ao usuário a possibilidade de desenvolver o código fonte de sua VNF diretamente na plataforma. O Gerenciador de Interfaces Virtuais de Rede possibilita ao usuário a criação e remoção das interfaces de rede. Já o Gerador de Tráfego disponibiliza ao usuário opções de configuração e parametrização de tráfegos que serão gerados na rede, podendo ser gerados pela ferramenta *Nping* ou pelo próprio usuário. Por fim, o componente de Visualização possibilita a análise gráfica de estatísticas da rede e também da VNF em execução em tempo real. Para a criação da plataforma NFV-Prime foi necessário o desenvolvimento tanto do *front-end* quanto do *back-end*. Para o *back-end* foi desenvolvida uma API REST (*Representational State Transfer*) utilizando a linguagem Python 3, devido à sua flexibilidade e amplo suporte em diversas plataformas, a qual foi programada com auxílio do *framework Flask*. Já para o desenvolvimento do *front-end* da aplicação *Web* utilizou-se o *Typescript*, o qual é uma linguagem que tem como base o *Javascript* sendo adicionado tipagem estática, em conjunto de HTML (*HyperText Markup Language*) e CSS (*Cascading Style Sheets*), sendo a plataforma desenvolvida utilizando o *framework* React. Também foi escolhido um sistema gerenciador de banco de dados objeto relacional, sendo o PostgreSQL. A NFV-Prime foi avaliada e comparada a uma NFV-MANO genérica, segundo especificação da ETSI, através da utilização do modelo *Human Error Assessment and Reduction Technique* (HEART). A fim de demonstrar o funcionamento do NFV-Prime, são apresentados quatro estudos de caso, os quais implementam os seguintes mecanismos: *Stateless Load Balancer*, *Stateful Load Balancer*, *Deep Packet Inspection* - DPI e *Leaky Bucket*. Além da descrição da implementação das VNFs correspondentes, são apresentados resultados da sua execução.

Palavras-chave: Funções Virtualizadas de Rede. Prototipação. Visualização de VNFs

ABSTRACT

Network Functions Virtualization (NFV) technology has been gaining attention as an alternative for implementing middleboxes in software, using virtualization technologies. Middleboxes implement Network Functions (NF) and are traditionally available on dedicated hardware. Virtualized Network Functions (VNFs) are software running on off-the-shelf hardware, which improves the flexibility and reduces capital and operating costs. This work proposes NFV-Prime, a platform for prototyping and visualizing VNFs, which aims at facilitating the introduction of the subject to new users, as well as the creation, instantiation and visualization of the results of VNF execution. Thus, the main objective of the platform is to instigate and facilitate the exploration of alternatives for implementing NFs that the paradigm makes possible. NFV-Prime consists of the following modules: Editor, Virtual Network Interface Manager, Traffic Generator and Visualization. The Editor provides the user with the possibility of developing the source code for their VNF directly on the platform. The Virtual Network Interface Manager allows the user to create and remove network interfaces. The Traffic Generator provides the user with options for configuring and parameterizing the traffic that is the input for the function, which can be either generated by the Nping tool or by the user. Finally, the Visualization module shows the output of the VNF running in real time as well as network statistics. To create the NFV-Prime platform, it was necessary to develop both the front-end and the back-end. For the back-end, a REST (Representational State Transfer) API was developed using the Python 3 language, due to its flexibility and broad support for various platforms, and was programmed using the Flask framework. The React framework was used to develop the NFV-Prime front-end as a web application. The Typescript language was employed, which is a language based on Javascript with the addition of static typing, together with HTML (HyperText Markup Language) and CSS (Cascading Style Sheets). PostgreSQL was chosen as the relational object database of the platform. The NFV-Prime was evaluated and compared to a generic NFV-MANO, according to the ETSI design, using the Human Error Assessment and Reduction Technique (HEART) model. In order to demonstrate how NFV-Prime works, four case studies are presented, which implement four different functions: Stateless Load Balancer, Stateful Load Balancer, Deep Packet Inspection - DPI and Leaky Bucket. In addition to describing the implementation of the corresponding VNFs, results of their execution are presented.

Keywords: Network Functions Virtualization. Prototyping. Visualizing VNFs.

SUMÁRIO

1 INTRODUÇÃO

As redes modernas, incluindo a Internet e redes celulares, utilizam diferentes Funções de Rede (*Network Functions* - NF) para a execução das mais diversas tarefas. Firewalls, dispositivos NAT (*Network Address Translation*) e sistemas de detecção de intrusão são exemplos dessas funções. Historicamente, as NFs são implementadas e executadas em dispositivos de hardware dedicado (Sherry et al., 2012a). Isso faz com que haja um alto nível de acoplamento entre *software* e *hardware* (Sekar et al., 2012), implicando em dificuldades de manutenção e escalabilidade das NFs.

O paradigma de Virtualização de Funções de Rede (*Network Functions Virtualization* - NFV) objetiva propor soluções aos desafios citados através do uso de tecnologias de virtualização. No paradigma NFV, as funções de rede são, portanto, desacopladas do hardware físico que as executa (Yi et al., 2018; Rodríguez-Haro et al., 2012). Dessa forma, as NFs são implementadas e operacionalizadas como dispositivos virtuais. Esses dispositivos virtuais são chamados de Funções Virtualizadas de Rede (*Virtualized Network Function* - VNF). Assim, o paradigma NFV se torna uma alternativa ao uso de *hardware* dedicado, flexibilizando e simplificando o projeto, desenvolvimento e manutenção de NFs (Cui et al., 2012).

Para viabilizar a adoção e implementação do paradigma NFV, a *European Telecommunications Standards Institute* (ETSI) (ETSI, 2014) propôs a arquitetura de referência NFV-MANO (*NFV Management and Orchestration*) com o intuito de padronizar e interoperabilizar o uso da tecnologia NFV. Desta forma, o ambiente NFV é composto por funções de rede executando em palataformas virtualizadas, executando em uma Infraestrutura Virtualizada (Virtualized Infrastructure - VI) e sendo gerenciadas pelo NFV-MANO. Em particular, existem diversas plataformas que implementam o modelo NFV-MANO. Entre elas encontram-se o Tacker (OPENSTACK, 2016), OSM (ETSI, 2020), OpenBaton (FRAUNHOFER-FOKUS e TU-BERLIN, 2017), Vines (Flauzino et al., 2021). Vale destacar que existem também as chamadas plataformas de execução de VNFs, que proveem recursos que permitem a execução de uma função de rede. Exemplos de plataformas de execução de VNF incluem o ClickOS (Martins et al., 2014), Click-On OSv (da Cruz Marcuzzo et al., 2017), COVEN (Garcia et al., 2019b) e Mini-NFV (Castillo-Lema et al., 2019).

Entretanto, as plataformas citadas anteriormente possuem um ambiente de difícil instalação e uso para novos usuários, principalmente aqueles que ainda não tiveram contato com o paradigma NFV. Isso se deve ao fato dessas plataformas utilizarem ferramentas complexas, a exemplo do Mini-NFV que utiliza o Mininet [1], as quais novos usuários não estão familiarizados, aumentando a curva de aprendizado necessária para utilizar as plataformas de prototipação e

[1] http://mininet.org/

execução de VNFs. Outro fator, é que a distribuição dessas soluções ocorrem, em sua maioria, por meio de *containers* e imagens de máquinas virtuais, como é o caso do Click-On-OSv, sendo necessário ao usuário manipular esses arquivos para conseguir acessar essas plataformas, adicionando outra etapa a qual o usuário precisa ter domínio. Todos os fatores listados tornam a utilização dessas plataformas complexa para novos usuários.

Diante do cenário apresentado, esse trabalho propõe a arquitetura da plataforma NFV-Prime, tendo como principal objetivo introduzir, de forma prática e facilitada, o paradigma NFV para novos desenvolvedores e usuários. Para isso, o NFV-Prime simplifica o processo de criar, instanciar e visualizar resultados de execução das VNFs em uma rede de testes. A arquitetura do NFV-Prime é subdividida em 9 módulos, sendo que cada módulo é responsável por executar operações específicas de tratamento, para a instanciação da plataforma, configuração de tráfego, gerenciamento de interfaces virtuais de rede e monitoramento do ambiente. Os módulos foram projetados para terem um baixo nível de acoplamento entre si, fato que permite substituições de maneira natural de acordo com a evolução e surgimento de novas tecnologias e ferramentas.

Como prova de conceito, a arquitetura proposta foi implementada através da plataforma NFV-Prime, que contém quatro grandes componentes funcionais: Editor, Gerenciador de Interfaces Virtuais de Rede, Gerador de Tráfego e Visualização. Os componentes da plataforma se propõem a serem didáticos, criando um *pipeline* de configuração e execução intuitivo para os usuários, reduzindo ao máximo a probabilidade de erros que possam ocorrer nos processos de configuração e execução de um ambiente NFV. Para demonstrar isso, a plataforma NFV-Prime foi avaliada, em três cenários distintos, através da utilização do modelo *Human Error Assessment and Reduction Technique* (HEART), onde foi possível detectar uma reduçao de 66,6% no número de casos possíveis de erros, além de uma significativa redução no impacto dos erros remanescentes no gerenciamento do ciclo de vida de uma VNF, quando comparado com o uso de um NFV-MANO genérico, conforme especificação da ETSI.

Para demonstrar e avaliar as funcionalidades dos módulos da arquitetura e da plataforma implementada, o NFV-prime foi testado em quatro estudos de caso. O primeiro apresenta uma VNF que implementa um mecanismo simplificado de balanceamento de carga (*load balancer*), em modo *stateless*. O segundo, implementa, também, um balanceador de carga, mas no modo *stateful*. O terceiro apresenta uma VNF que implementa um algoritmo de inspeção profunda de pacotes (*Deep Packet Inspection* - DPI). Por fim, no quarto estudo de caso é apresentada uma VNF que implementa um mecanismo de *shaping* de tráfego conhecido como *leaky-bucket*. Para realização dos estudos de caso foram exploradas as possibilidades de configuração do gerador de tráfego que a plataforma NFV-Prime disponibiliza, sendo a parametrização dos tráfegos diferentes entre todos os estudos de caso. Ao analisar os resultados das execuções das VNFs é possível confirmar a adequabilidade dos módulos propostos para suportar um ambiente de

prototipação do paradigma NFV, além de demonstrar a robustez da implementação da plataforma NFV-Prime, que obteve os resultados esperados em todos os estudos de caso executados.

O restante deste trabalho está organizado da seguinte forma. O Capítulo 2 introduz o paradigma NFV e apresenta trabalhos relacionados ao desenvolvimento de plataformas para a utilização do paradigma NFV. O Capítulo 3 apresenta uma visão geral sobre a arquitetura planejada para a plataforma NFV-Prime e sua implementação. No Capítulo 4, discorre-se sobre a avaliação e comparação de condições de erro da plataforma NFV-Prime em relação a uma NFV-MANO genérica, segundo especificação da ETSI. O Capítulo 5 detalha os quatro estudos de casos, discorrendo-se sobre as implementações de suas VNFs, parametrizações e configurações do gerador de tráfego, e também visualização e análise dos resultados obtidos nas execuções dos cenários criados. Por fim, a conclusão é apresentada no Capítulo 6. Os códigos-fonte das funções de rede implementadas nos estudos de caso estão disponíveis nos apêndices.

2 VIRTUALIZAÇÃO DE FUNÇÕES DE REDE

Este capítulo apresenta os principais conceitos de Virtualização de Funções de Rede (*Network Functions Virtualization* - NFV). A Seção 2.1 introduz e discorre sobre as motivações e vantagens da utilização do paradigma NFV. A Seção 2.2 descreve a arquitetura de referência NFV proposta pela *European Telecommunications Standards Institute* (ETSI). A Seção 2.3 descreve os serviços virtuaizados de rede. Já na Seção 2.4 são apresentados trabalhos relacionados à plataforma NFV-Prime.

2.1 INTRODUÇÃO À VIRTUALIZAÇÃO DE FUNÇÕES DE REDE

As redes de computadores modernas dependem de diversas Funções de Rede (*Network Function* - NF) para operar. Funções de rede realizam tarefas bem definidas no núcleo da rede, tendo *interfaces* externas e comportamentos operacionais bem definidos (Rosa et al., 2014). *Firewalls*, dispositivos NAT (*Network Address Translation*), Redes Virtuais Privadas (*Virtual Private Network* - VPN) e Inspeção Profunda de Pacotes (*Deep Packet Inspection* - DPI) são exemplos de funções de rede. Tradicionalmente, as NFs são implementadas em dispositivos de *hardware* dedicados, havendo portanto o acoplamento entre o *software* e o *hardware* (Sekar et al., 2012).

Neste contexto, para utilizar de NFs é necessário adquirir e manter tais dispositivos, fato que impacta negativamente nos custos da rede, já que eles representam uma parcela relevante das despesas operacionais (OPEX - *OPerational EXpenditures*) e de capital (CAPEX - *CAPital EXpenditures*) (Cotroneo et al., 2014). Além disso, esses dispositivos apresentam alta complexidade ao se considerar os processos de implantação, gerência e solução de problemas. Esse fato se deve, em particular, ao uso de *hardware* proprietário e da necessidade de configurações específicas (Sherry et al., 2012b). Por último, o uso de *hardware* dedicado muitas vezes não possibilita o redimensionamento dinâmico de recursos de rede conforme a demanda. Dessa forma, podem tanto ocorrer desperdícios como a falta de recursos computacionais na medida em que a carga da rede varia (Venâncio e Duarte Jr, 2020).

Nesse contexto, surge o paradigma NFV, o qual objetiva implementar NFs em *software* utilizando tecnologias de virtualização. No paradigma NFV, as funções de rede são desenvolvidas em um plano de *software*, sendo desacopladas do *hardware*. Sendo assim, NFs são implementadas e operacionalizadas como dispositivos virtuais. Esses dispositivos virtuais, quando executam uma NF, são chamados de Funções Virtualizadas de Rede (*Virtualized Network Function* - VNF). Usando essa abordagem, as NFs podem ser executadas em servidores comerciais de prateleira. Além disso, múltiplas VNFs podem coexistir no mesmo equipamento através da instanciação e

gerência de várias instâncias virtuais, estas criadas a partir de máquinas virtuais ou contêineres. Através dessas e outras características, o paradigma NFV se torna uma alternativa atraente ao uso de *hardware* dedicado, flexibilizando e simplificando o projeto, desenvolvimento e manutenção de NFs (Cui et al., 2012).

Dessa forma, a ETSI (NFVISG, 2013) fundamentou o paradigma NFV em cinco requisitos de alto nível, sendo: portabilidade, desempenho, integração, gerência e orquestração escalabilidade. Portabilidade é o requisito referente à capacidade de transferir os sistemas desenvolvidos para diferentes ambientes NFV. Assim, é possível executá-los sem a necessidade de alterar características de seu funcionamento. Desempenho refere-se ao desafio de mitigar a degradação de performance que pode ocorrer ao utilizar o paradigma NFV. Essa degradação se deve ao fato das NFs serem implementadas em *software* ao invés de *hardware* especializado. Integração define que, independentemente de tecnologias, plataformas e fornecedores, as funções e serviços de rede devem ser executados e integrados em harmonia no ambiente NFV. Gerência e orquestração estabelece que, no mínimo, as operações protocolares devem ser implementadas e disponibilizadas aos agentes aos quais se destinam. Por fim, escalabilidade refere-se ao redimensionamento sob demanda do ambiente NFV, realizado na tentativa de evitar gargalos de processamento e também desperdício de recursos computacionais.

A Figura 2.1 apresenta uma comparação entre a estrutura das redes tradicionais, que utilizam *hardwares* dedicados, e as redes baseadas no paradigma NFV, que utilizam VNFs. Na figura VNFs apresentam o prefixo "v" de virtual, como *vFirewalls*, vDPI, vVPN e vNAT.

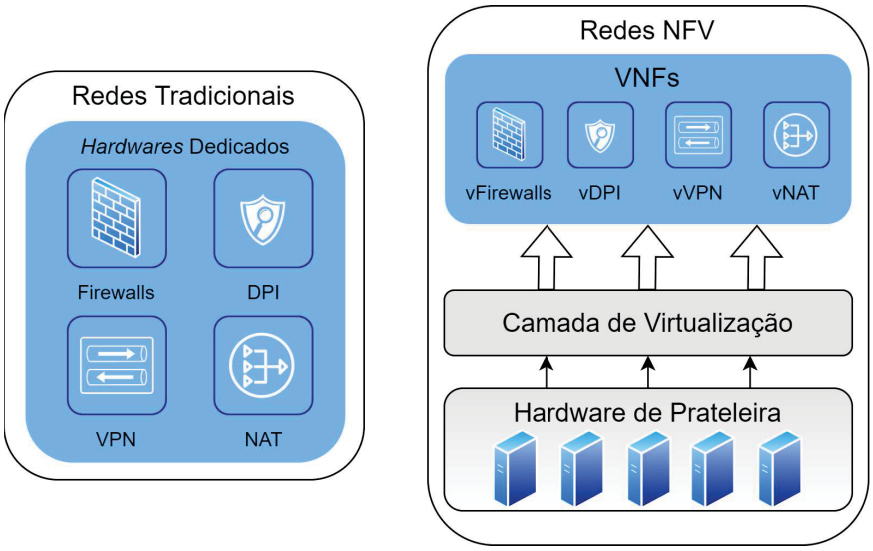


Figura 2.1: Comparação da estrutura entre as redes tradicionais e redes baseadas no paradigma NFV.

Ao se comparar as redes tradicionais às redes baseadas no paradigma NFV, verificam-se algumas vantagens das últimas. A primeira vantagem é a não necessidade de *hardwares* especializados. Desta maneira, a inclusão de novas VNFs na rede, bem como alterações e atualizações das já existentes, representam menor impacto nos custos de capital e operacionais

(OPEX e CAPEX), visto que não há necessidade de adquirir e instalar fisicamente novos equipamentos na rede. De forma geral, destaca-se a flexibilização e agilidade no processo de implantação e operação da rede. (Mijumbi et al., 2015). Além disso, as VNFs tipicamente demandam menos energia e espaço físico já que estão desacopladas do *hardware* (Han et al., 2015). Entretanto, também existem desvantagens e desafios que devem ser explorados.

Uma desvantagem encontrada é a redução do desempenho em comparação às alternativas executadas em *hardware* especializado. Isso se deve ao fato das VNFs estarem implementadas em software e a introdução da camada de virtualização no processo. Apesar dos benefícios citados, o paradigma NFV enfrenta dificuldades quando se trata de ferramentas intuitivas para a construção de VNFs. Tais ferramentas visam o desenvolvimento rápido de protótipos, funcionando, também, como uma ferramenta de aprendizagem do paradigma NFV. Alternativas disponíveis sofrem com a complexidade para executar as VNFs na rede, além de representarem um desafio para a utilização por usuários leigos.

2.2 ARQUITETURA DE REFERÊNCIA PARA NFV

A ETSI (ETSI, 2014), com o intuito de padronizar o uso da tecnologia NFV, propôs a arquitetura de referência NFV-MANO (*NFV Management and Orchestration*), ilustrada na Figura 2.2. São mostrados os três principais blocos: VNFs, Infraestrutura NFV (*NFV Infrastructure* - NFVI) e Gerência e Orquestração NFV (*NFV Management and Orchestration* - NFV-MANO, bloco NFV-MANO propriamente dito). Esses blocos são descritos a seguir.

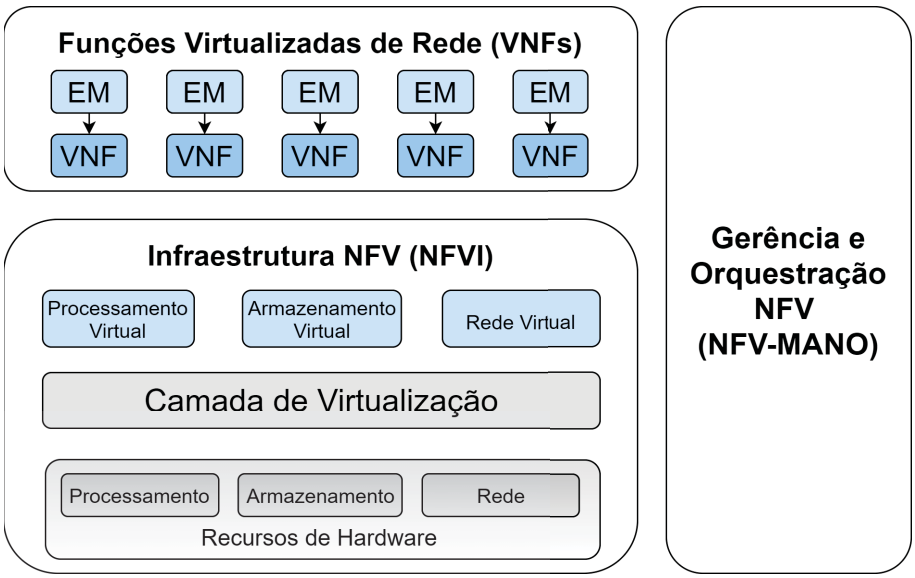


Figura 2.2: Arquitetura de referência NFV proposta pela ETSI.

O bloco NFVI é a representação da infraestrutura virtualizada em que as VNFs são executadas. Uma parte do NFVI é relacionada aos recursos computacionais físicos, ou seja, recursos de *hardware*, sendo esses responsáveis pelo processamento, armazenamento e rede

utilizados pelas VNFs. Entretanto, para que as VNFs possam usufruir desses recursos físicos há a introdução de uma Camada de Virtualização, responsável por abstrair os recursos físicos em recursos virtuais, desacoplando, assim, o *software* do *hardware*. Na Camada de Virtualização, o *hypervisor*, tem como responsabilidade a criação e gerência dos dispositivos virtuais, a exemplos de Máquinas Virtuais (*Virtual Machines* - VM) e *containers*. Dessa forma, a estratégia adotada pela ETSI para a arquitetura NFV permite a execução independente e isolada de cada VNF.

O segundo bloco presente na arquitetura NFV é o das VNFs. Ele corresponde à representação das instâncias dos dispositivos virtuais, os quais são responsáveis pela execução, sobre o NFVI, das NFs. No bloco, cada VNF é associada a um Descritor VNF (VNFD - VNF *Descriptor*), no qual as VNFs são especificadas em termos dos seus requisitos operacionais e de implantação. Assim, os VNFDs possuem informações que podem ser utilizadas pelo bloco NFV-MANO, como, por exemplo, políticas de realocação de recursos computacionais conforme a demanda. Além disso, as VNFs vinculam-se a um Gerenciador de Elemento (*Element Manager* - EM), que será responsável pelas funcionalidades de gerência FCAPS (*Fault, Configuration, Accounting, Performance and Security Management*) da instância virtual.

Por fim, o NFV-MANO é o bloco responsável por orquestrar e gerenciar o funcionamento das VNFs. O NFV-MANO é dividido em três módulos: Orquestrador NFV (NFV *Orchestrator* - NFVO), Gerenciador de VNF (VNF *Manager* - VNFM) e Gerenciador da Infraestrutura Virtualizada (*Virtualized Infrastructure Manager* - VIM). Além desses, também existe o módulo OSS/BSS (*Operations Support System/Business Support System*), que não faz parte, estritamente, da arquitetura, mas representa os sistemas externos que interagem com ela (Fulber-Garcia et al., 2020a). A Figura 2.3 mostra as conexões e divisões na arquitetura do NFV-MANO.

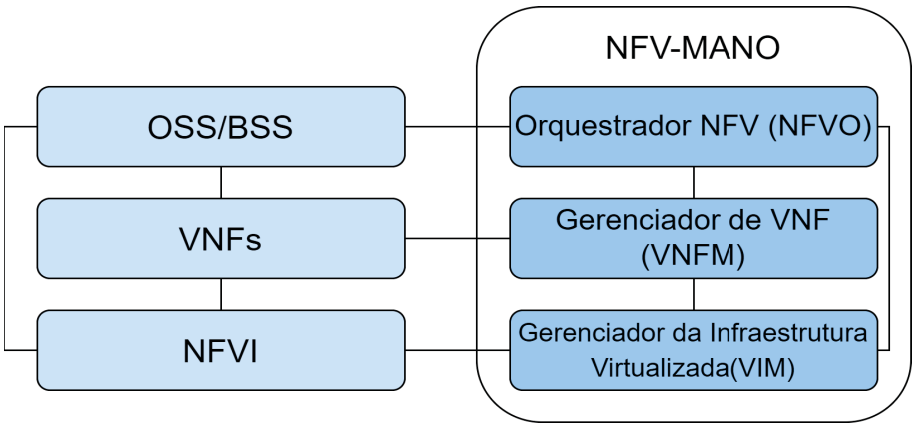


Figura 2.3: Módulos e conexões do bloco NFV-MANO.

O módulo VNFM é responsável, principalmente, pelo gerenciamento do ciclo de vida das VNFs. Ao VNFM são atribuídas as funções de instanciação, remoção, atualização, configuração e escalonamento de VNFs, além do monitoramento da utilização de recursos e notificações de falhas. Desta forma, o VNFM deve ter acesso aos VNFDs para poder executar funcionalidades correspondentes, além de ter acesso aos repositórios e as diferentes versões das VNFs, para

que todas as operações sejam suportadas. O acesso à VNF pode ocorrer diretamente ou através de um Sistema Gerenciador de Elemento (*Element Management System*). O VNFM permite a configuração e execução do software responsável pela execução da VNF. (Venâncio et al., 2021). Por fim, a arquitetura proposta pela ETSI permite a utilização de um ou mais VNFMs, desde que as operações realizadas por eles sejam suportadas por qualquer VNF, ou seja, independentemente da sua implementação (ETSI, 2014).

O módulo VIM, por sua vez, é responsável pelo controle e gerência de recursos computacionais da infraestrutura virtualizada, ou seja, do processamento, armazenamento e rede. Assim, o VIM cria os dispositivos virtuais, modifica a alocação de recursos computacionais e monitora a infraestrutura. A arquitetura do NFV-MANO proposta pela ETSI permite mais de uma instância de VIM, assim como no caso do VNFM, sendo que cada VIM é responsável por gerenciar uma infraestrutura virtualizada diferente (Mijumbi et al., 2016).

O módulo NFVO oferece uma abstração de alto nível para gerenciar e monitorar serviços virtuais compostos por múltiplas VNFs. Além disso, o módulo também é responsável pelo redimensionamento dinâmico dos recursos da VNF de acordo com sua demanda, pelo gerenciamento das políticas e pela autorização e validação das requisições realizadas pelo NFVI. O NFVO também realiza a orquestração de múltiplas funções visando implantar um serviço composto. Para isso, o NFVO fornece os recursos abstratos do NFVI através de vários VIMs, além de automatizar o provisionamento de diferentes VNFs e topologias que compreendem um serviço. No entanto, as VNFs são efetivamente controladas pelo VNFM, que pode receber comandos do NFVO e configurar individualmente as VNFs.

Por fim, tem-se o módulo OSS/BSS, que não faz parte estritamente da arquitetura proposta pela ETSI, mas representa os sistemas e aplicações externas que se comunicam com o sistema NFV. O uso desse módulo se deve ao fato de que o paradigma NFV não permite que sistemas externos gerenciem as VNFs diretamente. Desta forma, o OSS/BSS é responsável por realizar a comunicação, através de *interfaces* padronizadas, de um sistema externo com o bloco NFV-MANO para serem executadas, entre outras, requisições de instanciação, atualização e remoção de VNFs.

Atualmente, diversas plataformas que implementam a arquitetura de referência NFV-MANO estão disponíveis. Entre elas encontram-se o Tacker (OPENSTACK, 2016), OSM (ETSI, 2020), OpenBaton (FRAUNHOFER-FOKUS e TU-BERLIN, 2017) e Vines (Flauzino et al., 2021). Existem também as chamadas plataformas de execução de VNFs, que proveem recursos que permitem a execução de NFs em instâncias virtuais. Exemplos de plataformas de execução de VNF incluem o ClickOS (Martins et al., 2014), Click-On-OSv (da Cruz Marcuzzo et al., 2017), COVEN (Garcia et al., 2019b) e NIEP (Tavares et al., 2018). Associados a estas plataformas existem também sistemas que implementam estratégias eficientes para a instalação de funções

e serviços de rede no substrato físico disponível, exemplos incluem o CUSCO (Fulber-Garcia et al., 2020b) e Piecing NFV (Luizelli et al., 2015).

2.3 SERVIÇOS VIRTUALIZADOS DE REDE

Um dos principais aspectos da tecnologia NFV é a possibilidade de contruir serviços de rede complexos. Serviços de rede consistem na composição de múltiplas VNFs. As VNFs podem ser organizadas de diversas maneiras, sendo que cada combinação pode prover um serviço de rede diferente. Assim, através das VNFs flui tráfego de rede, na ordem em que o serviço é especificado. Tanto o IETF, quanto a ETSI tem feito esforços para padronizar os serviços de rede dentro do contexto NFV (Fulber-Garcia et al., 2020a). Tendo em vista a sua popularidade, este trabalho foca nos padrões do IETF, o qual apresenta o conceito de Cadeia de Função de Serviço (*Service Function Chain* - SFC) (Halpern e Pignataro, 2015). Uma SFC é definida pela sequência de funções através da qual o tráfego de rede é direcionado. Na Figura 2.4 é apresentada a arquitetura de SFC proposta pela IETF, sendo possível visualizar o fluxo do tráfego da rede em uma SFC.

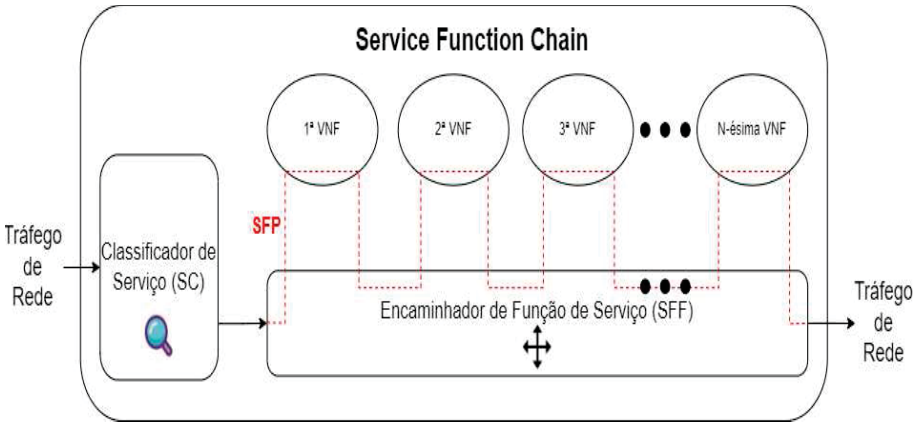


Figura 2.4: Arquitetura da SFC proposta pela IETF.

A arquitetura proposta pela IETF permite a criação, especificação e manutenção de SFCs. A arquitetura é composta por dois componentes, o Classificador de Serviço (*Service Classifier* - SC) e o Encaminhador de Função de Serviço (*Service Function Forwarder* - SFF), além das funções de rede. O SC é o componente a ser colocado no início da SFC, sendo o nodo de ingresso do tráfego da rede. A partir da classificação, o pacote é encapsulado sendo acrescentado o header NSH (*Network Service Header*) (Quinn et al., 2018) que, entre outras informações indica o SFC *Path* (SFP), caminho de VNFs através do qual o tráfego da SFC será encaminhado. Desta forma, a decisão de direcionar um determinado tráfego de rede para a próxima VNF é realizado através das informações encapsuladas pelo SC, diferentemente do roteamento convencional, que utiliza o endereço IP (Huff et al., 2018).

Após o tráfego de rede ser classificado pelo SC, o mesmo é direcionado ao SFF. Ao receber os pacotes, o SFF realiza o encaminhamento desses para as VNFs específicas, de acordo com o SFP definido. Desta maneira, o SFF realiza o envio do tráfego para a primeira VNF do SFC. A VNF em questão recebe os pacotes e realiza os processamentos necessários, devolvendo ao SFF o tráfego de rede processado. Por sua vez, o SFF recebe, novamente, os pacotes, encaminhando-os para a próxima VNF. Este processo de reenvio do trafégo alterando a VNF de destino é realizado até que se obtenha os pacotes processados pela última VNF, finalizando a execução da SFC. Desta forma, o SFF remove o encapsulamento realizado pelo SC e envia o tráfego para o destino final (Venâncio et al., 2023). Vale ressaltar que, há a possibilidade de pacotes serem consumidos pelas VNFs, podendo ocorrer, por exemplo, ao empregar uma VNF de *Firewall*, barrando pacotes.

2.4 TRABALHOS RELACIONADOS

Esta seção apresenta alguns dos trabalhos relacionados à plataforma NFV-Prime. Marcuzzo et al. (2017) propõem uma plataforma, chamada de Click-on-OSv para instanciação e execução de middleboxes baseadas em Click na rede. Para utilizar a plataforma, o usuário envia sua aplicação, a NF, para o Click-on-OSv que, assim, instancia e executa a mesma em seu ambiente virtualizado, coletando informações do funcionamento da VNF instanciada, que são retornadas ao usuário, sendo mostradas através de gráficos e estatísticas. A plataforma é disponibilizada através de uma imagem de máquina virtual baseada em OSv, em conjunto com sua documentação. A linguagem de programação utilizada é a linguagem do Click Modular Router, sendo utilizados seus módulos para o acesso à interface de rede da máquina.

Bondan et al. (2019) propõem o FENDE, um ecossistema e *marketplace*, que visa a distribuição e execução de VNFs, além da composição de serviços de cadeias de funções SFCs. O FENDE foi desenvolvido visando três tipos de usuários: desenvolvedores, revisores e clientes. Assim, provê todas as operações necessárias para um *marketplace*. Os autores consideram que o FENDE é a primeira solução de *marketplace* do mercado que possibilita aos usuários adquirirem e executarem VNFs e SFCs, tendo combinado em uma única solução capacidades de marketplace, gerenciamento e infraestrutura.

Tavares et al. (2018) propõem uma plataforma para desenvolver e testar VNFs e infraestruturas baseadas no paradigma NFV, chamada *NFV Infrastructure Emulation Platform* (NIEP). O NIEP foi construído utilizando o emulador de rede Mininet em conjunto com a plataforma de desenvolvimento de VNFs Click-on-OSv. O NIEP visa prover ao usuário um ambiente de emulação NFV completo, permitindo a execução de testes utilizando as VNFs desenvolvidas em um cenário controlado antes de colocá-las em ambientes de produção. Isso se deve ao fato do NIEP permitir ao usuário a criação de cenários NFV emulados e heterogêneos.

Sendo assim, o NIEP é capaz de simular cenários complexos, variando o número de hosts, switches e VNFs, bem como a sua topologia, a qual é definida através do Mininet.

Peuster et al. (2016) apresentam uma plataforma de prototipagem NFV, chamada de *Multi Datacenter servIce ChaIN Emulator* (MeDICINE), que visa executar em um ambiente com múltiplos pontos de presença (*Multi-Points of Presence* - Multi-PoP) NFs aptas ao ambiente de produção, sendo essas disponibilizadas através de contêineres. Através de sistemas de gerenciamento e orquestração conectados à plataforma por meio de interfaces padronizadas é possível controlar as NFs. Desta forma, desenvolvedores têm acesso a um ambiente realista para prototipar e testar serviços de rede complexos. O MeDICINE foi desenvolvido tendo como base a ferramenta Containernet[1], a qual estende a estrutura de emulação do Mininet e permite a utilização de contêineres padronizados como instâncias de computação dentro da rede emulada. A utilização de contêineres para a execução das NFs, permite aos desenvolvedores testar seus serviços para posterior execução em ambientes de produção.

Peuster et al. (2018) introduzem o Containernet 2.0, uma plataforma *open-source* de prototipagem NFV, a qual visa auxiliar na criação e execução, localmente, de SFCs complexos. A plataforma se destaca por suportar SFCs híbridas, sendo possível a cadeia ser composta por VNFs baseadas em contêineres e VNFs baseadas em máquinas virtuais. A instanciação de VMs ocorre através da utilização de um *script* em *Python*, o qual recebe a imagem da VM a ser instanciada por parâmetro. Destarte, o Containernet 2.0 simplifica o desenvolvimento de serviços de rede híbridos. A plataforma pode ser instalada e executada de forma local, possibilitando ao usuário iniciar o desenvolvimento de seus serviços de rede de forma rápida.

Castillo-Lema et al. (2019) apresentam o *framework* Mini-NFV para Orquestração NFV, um Gerenciador de VNF de uso geral para implementar e operar funções de rede virtuais e serviços de rede no Mininet utilizando modelos TOSCA NFV padronizados baseados na Arquitetura de *Framework* ETSI MANO. O objetivo principal do Mini-NFV é remover do programador a tarefa de configurar todo um ambiente de encadeamento de serviços, fazendo com que o mesmo possa dedicar seu tempo inteiramente no desenvolvimento das VNFs. Além disso, o uso do Mini-NFV facilita a transição de emulação para o mundo real, reforçando a replicabilidade e reprodutibilidade do paradigma NFV. Um ponto positivo encontrado é que o Mini-NFV se desenvolve e aperfeiçoa em conjunto ao Mininet.

[1] https://containernet.github.io/

3 A PLATAFORMA NFV-PRIME

Neste capítulo é descrita a plataforma NFV-Prime para prototipação e visualização de funções virtualizadas de rede. Na Seção 3.1 é apresentada a arquitetura do NFV-Prime e na Seção 3.2 é descrita a implementação da plataforma.

3.1 NFV-PRIME: ARQUITETURA

Neste trabalho é proposta uma arquitetura genérica, abrangente e expansível para a plataforma NFV-Prime. A Figura 3.1 ilustra a arquitetura, a qual é constituída por nove módulos fundamentais: (i) Interface de Acesso (IA), (ii) Interface Gráfica de Usuário (GUI), (iii) Gerenciador de Acesso (GA), (iv) Gerenciador da Plataforma (GP), (v) Gerenciador do Banco de Dados (GBD), (vi) Monitor (M), (vii) Gerenciador da Rede (GR), (viii) Gerenciador de Tráfego (GT) e (ix) Aplicação (App). Cada módulo é responsável por executar operações específicas de tratamento, encaminhamento e/ou aplicação de requisições, sejam essas referentes à instanciação da aplicação, configuração de tráfego, gerenciamento de interfaces virtuais de rede e monitoramento da plataforma - representados por setas contínuas - ou pela relação entre as funcionalidades dos componentes, ou seja, a interação que ocorre entre um ou mais módulos da plataforma - representadas por linhas tracejadas.

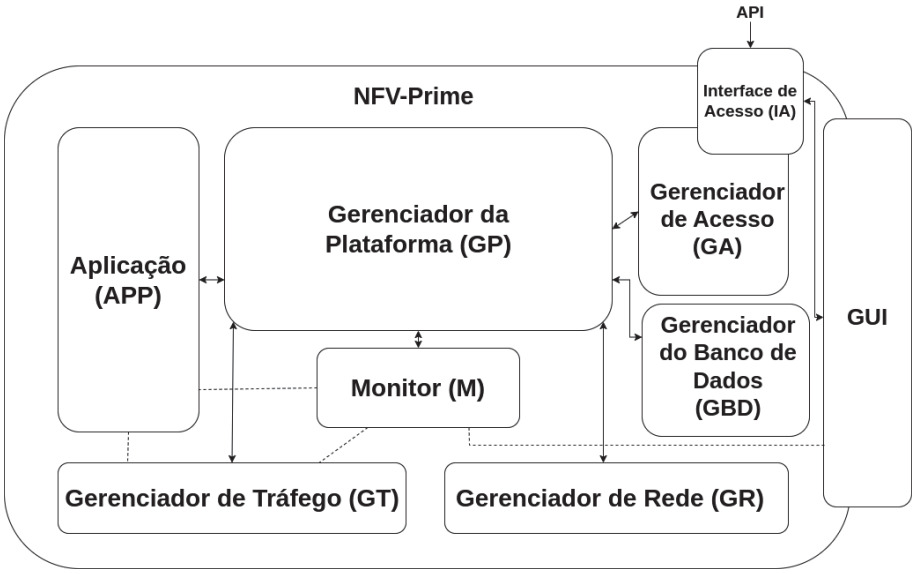


Figura 3.1: Arquitetura da plataforma NFV-Prime.

Os módulos da arquitetura NFV-Prime foram projetados para terem um baixo nível de acoplamento entre si, sendo que o funcionamento de uma solução e a cooperação entre módulos são garantidos a partir da comunicação entre eles. Assim, módulos individuais podem

ser substituídos de maneira natural de acordo com a evolução e surgimento de novas tecnologias e ferramentas. As descricões de competências e dos relacionamentos dos módulos da arquitetura NFV-Prime propostos são apresentados a seguir.

Interface de Acesso: é o módulo responsável pelo recebimento de requisições externas à arquitetura NFV-Prime, podendo essas serem originadas diretamente pela API própria do NFV-Prime, ou pela GUI, a qual realiza o consumo da API. Ao receber uma requisição, a interface de acesso realiza uma breve verificação da requisição, filtrando requisições que se encontrem incompletas ou contenham erro em sua construção, descartando requisições que não se enquadrem ao padrão proposto pela API do NFV-Prime. Se a requisição estiver conforme a API, é realizado seu encaminhamento para o Gerenciador de Acesso, módulo interno do NFV-Prime.

Interface Gráfica de Usuário: é o módulo responsável por apresentar o NFV-Prime ao usuário, trazendo uma interface gráfica que se comunica com outros módulos da plataforma via requisições, as quais devem ser construídas segundo sua API padrão. A requisição construída pelo GUI é encaminhada ao módulo Interface de Acesso. O GUI deve apresentar ao usuário as quatro principais funcionalidades do NFV-Prime: edição, gerenciamento de interfaces virtuais de rede, parametrização e geração de tráfego de rede e monitoramento de interfaces e VNFs.

Gerenciador de Acesso: é o módulo que recebe requisições da Interface de Acesso. O Gerenciador de Acesso é responsável pela reorganização das requisições, possibilitando que essas sejam interpretadas corretamente pelo módulo Gerenciador da Plataforma. Assim, é realizada a separação da funcionalidade que está sendo requisitada entre função e parâmetros. Por fim, o Gerenciador de Acesso reencaminha as requisições de forma organizada e simplificada, facilitando seu entendimento pelo Gerenciador da Plataforma.

Gerenciador da Plataforma: é o módulo responsável, principalmente, pelo ciclo de vida dos processos da plataforma NFV-Prime. Para isso, o módulo interpreta a requisição advinda do Gerenciador de Acesso, identifica o módulo responsável pela funcionalidade e encaminha um sinal de início ou parada. As ações indicadas pelo Gerenciador da Plataforma podem ser de instanciação ou remoção de recursos e VNFs. O Gerenciador da Plataforma realiza o gerenciamento dos seguintes módulos: Gerenciador do Banco de Dados, Monitor, Gerenciador de Rede, Gerenciador de Tráfego e Aplicação. Portanto, assim que é interpretada a requisição, o gerenciador da plataforma realiza seu encaminhamento para o módulo responsável por executar a ação requisitada, a qual deve ser efetivada imediatamente.

Gerenciador do Banco de Dados: é o módulo responsável pelo armazenamento, atualização, remoção e controle dos dados da plataforma. Dito isso, existe uma relação estreita com o Gerenciador da Plataforma, que recebe uma requisição e realiza consultas, quando necessário, ao módulo Gerenciador do Banco de Dados. Desta forma, a funcionalidade requisitada é executada com os parâmetros corretos. Da mesma forma, o Gerenciador da Plataforma, após

confirmação da execução da funcionalidade requisitada pelo módulo responsável, envia, se necessário, dados para serem armazenados ou atualizados no banco para posterior acesso.

Monitor: é responsável por monitorar os módulos de Aplicação, Gerenciador de Rede e Gerenciador de Tráfego, utilizando VNFs. O Monitor realiza uma coleta de dados acerca dos processos que estão em execução nesses módulos. Assim, o Monitor repassa os dados coletados para o módulo Gerenciador da Plataforma, que por sua vez aciona o módulo Gerenciador do Banco de Dados para atualizar e armazenar os dados em questão. Portanto, o Monitor tem papel fundamental no NFV-Prime, visto que é a partir de sua função que se torna possível atingir de forma prática a execução desejada da plataforma.

Gerenciador de Rede: é o módulo responsável pelo gerenciamento dos recursos de rede. A principal funcionalidade do Gerenciador de Rede é, quando requisitado pelo Gerenciador da Plataforma, instanciar, remover e manter interfaces virtuais de rede. A partir da ação do Gerenciador de Rede as interfaces são disponibilizadas na plataforma, podendo receber tráfego de rede. Desta forma, existe uma correlação entre o Gerenciador de Rede, Gerenciador de Tráfego e Aplicação, na forma de que a funcionalidade de um impacta diretamente nas possibilidades de execução dos outros.

Gerenciador de Tráfego: é o módulo responsável por gerenciar a configuração e geração de tráfego sintético. Ou seja, sua principal funcionalidade é a inicialização e parada do fluxo da rede. O Gerenciador de Tráfego recebe do Gerenciador da Plataforma a parametrização que será utilizada para a geração do fluxo da rede. Esses parâmetros são escolhidos pelo usuário da plataforma a partir da GUI, ou diretamente pela API da plataforma NFV-Prime.

Aplicação: é o módulo responsável pelo gerenciamento das VNFs desenvolvidas pelo usuário. Ou seja, tem como funcionalidade instanciar e remover a VNF da rede, de acordo com as requisições do Gerenciador da Plataforma. Resumidamente, a VNF desenvolvida pelo usuário é repassada ao Gerenciador da Plataforma através da API do NFV-Prime, a qual pode ser acessada diretamente ou a partir do módulo GUI. Quando interpretada a informação e identificada a necessidade do módulo de Aplicação, esse é acionado. Assim que o módulo de aplicação é requisitado, a ação informada pelo Gerenciador da Plataforma, seja de instanciar ou remover a VNF na rede, é executada imediatamente.

De maneira geral, quando uma requisição, originada pelo GUI ou API, é recebida pela Interface de Acesso, é realizada uma filtragem de requisições que não estejam em conformidade aos formatos estabelecidos pelo NFV-Prime. Estando em conformidade, a Interface de Acesso prossegue o encaminhamento do pacote para o Gerenciador de Acesso, o qual procede com a verificação acerca da requisição, sendo considerado o conjunto de operações disponibilizadas pela plataforma, além dos parâmetros esperados para a requisição. Caso a requisicão seja válida e autorizada, é realizado seu encaminhamento para o Gerenciador da Plataforma. O Gerenciador da Plataforma interpreta a requisição e, se necessário, realiza uma consulta ao

Gerenciador do Banco de Dados para completar as informações necessárias para execução da funcionalidade. Após, é definido qual dos módulos finais, dentre Monitor, Gerenciador de Rede, Gerenciador de Tráfego e Aplicação é o responsável por executar a funcionalidade que está sendo requisitada, encaminhando-a para o módulo em questão. Por fim, ao receber a requisição, o módulo final executa-a de maneira imediata, retornando dados de sua execução, se necessário, para o Gerenciador da Plataforma, que pode acionar novamente o Gerenciador de Banco de Dados para atualização e armazenamento desses dados. Os resultados das requisições fazem o caminho contrário ao das requisições, sendo seu destino final o GUI, que pode utilizá-los de diferentes formas.

3.2 NFV-PRIME: IMPLEMENTAÇÃO

A plataforma NFV-Prime foi implementada para ser uma Aplicação Web (*Web Application*). Essa escolha visa facilitar o acesso à plataforma, visando uma ampla divulgação tanto da ferramenta quanto do próprio paradigma NFV. Dito isso, priorizou-se uma interface que seja amigável para os usuários, além de acelerar os processos de criação, instanciação e visualização de VNFs. O principal objetivo da plataforma é instigar e facilitar a exploração das possibilidades de implementações de NFs que o paradigma possibilita. Além disso, o modelo de Aplicação Web possibilita a utilização de serviços que proveem processamento em nuvem, podendo a aplicação ser hospedada em um servidor com alta disponibilidade de recursos, viabilizando na escalabilidade da plataforma.

Para a criação da plataforma NFV-Prime foi necessário o desenvolvimento tanto do *front-end* quanto do *back-end*, além de um banco de dados dedicado. Para o *back-end* foi desenvolvida uma *API Rest* com auxílio do *framework* Flask [1], que utiliza a linguagem *Python 3*, escolhida devido à sua flexibilidade e amplo suporte em diversas plataformas. Já para o desenvolvimento do *front-end* da Aplicação Web utilizou-se o Typescript[2], o qual é uma linguagem que tem como base o Javascript sendo adicionada tipagem estática, em conjunto de HTML e CSS, sendo a plataforma desenvolvida utilizando o *framework* React [3]. O sistema gerenciador de banco de dados objeto relacional escolhido foi o PostgreSQL[4].

A plataforma foi construída visando facilitar e agilizar os processos envolvidos na criação, instanciação e visualização das VNFs. Para isso, o *front-end* da plataforma NFV-Prime foi dividido em quatro componentes principais, sendo eles: Editor, Gerenciador de Interfaces Virtuais de Rede, Gerador de Tráfego e Visualização. Os componentes, na ordem apresentada, formam um *pipeline* de montagem do cenário, que ao final pode ser executado. Além do *pipeline*, a NFV-Prime também permite as funcionalidades de importar e exportar, ambas utilizando

[1] https://flask-ptbr.readthedocs.io/en/latest/
[2] https://www.typescriptlang.org/docs/
[3] https://react.dev/
[4] https://www.postgresql.org/

arquivos de JSON. Essas funcionalidades possibilitam ao usuário a reutilização de funções de forma rápida e prática. A Aplicação Web e seus componentes podem ser visualizados na Figura 3.2. O componente Editor possibilita ao usuário desenvolver, em um editor de texto da aplicação, o código fonte de sua NF, que precisa ser, necessariamente, em *Python 3*, para que possa ser executada posteriormente na rede. A plataforma também possibilita a importação de arquivos de extensão "*.py*", facilitando o processo de criação da NF, e abrindo a possibilidade do usuário utilizar ferramentas de desenvolvimento já estabelecidas no mercado, a exemplo do *Visual Studio Code* (VSCode) [5]. Além disso, também são disponibilizadas quatro exemplos de NFs, sendo elas *Leaky Bucket*, *Load Balancer (Stateless)*, *Load Balancer (Stateful)* e *Deep Packet Inspection* - DPI, para que o usuário possa experimentar o sistema. Fato importante a ressaltar é que o editor permite ao usuário a personalização dessas NFs predefinidas. O componente Editor é apresentado na Figura 3.3.

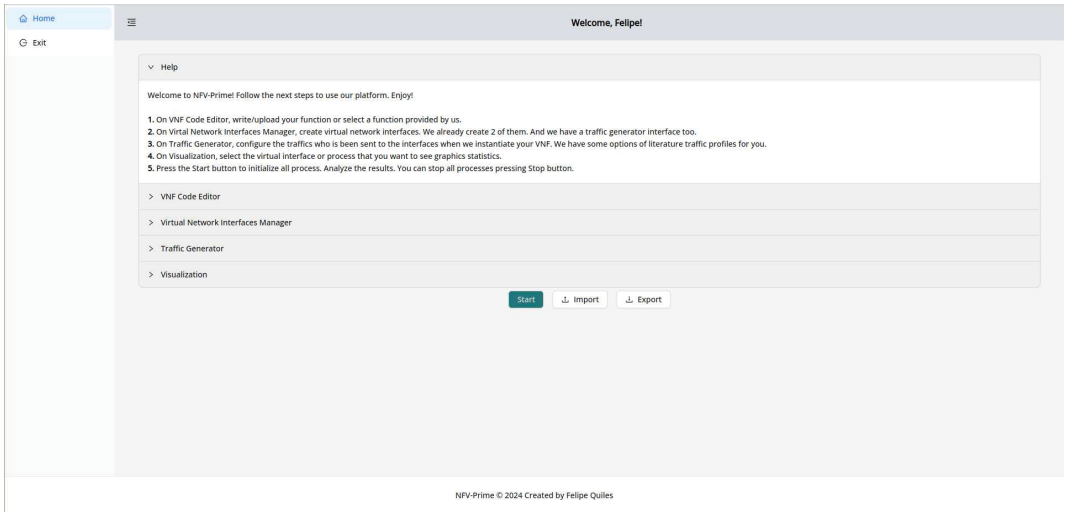


Figura 3.2: Plataforma NFV-Prime.

O Gerenciador de Interfaces Virtuais de Rede permite ao usuário gerir a quantidade de interfaces de rede que estarão presentes no NFV-Prime. Por padrão, o NFV-Prime inicia com três interfaces virtuais de rede instanciadas, sendo que duas são nomeadas dummy_1 e dummy_2, enquanto a outra é a interface responsável pela geração do tráfego na rede, cuja identificação é *dummy_traffic_de_generator*. O usuário pode remover ou adicionar novas *dummies* à rede, sendo que ao adicionar uma nova interface seu identificador será o prefixo "*dummy_*" seguido do número da *dummy* que segue ordem ascendente. Para a criação de uma interface virtual de rede foi utilizado o utilitário netns [6], o qual cria um ambiente de rede separado em uma mesma máquina, sendo o ambiente principal (*host*), o qual executa a VNF, conectado ao ambiente gerado pelo netns (*net namespace*), simbolizando os *endpoints* da rede. No NFV-Prime foram criados dois ambientes: o do gerador de tráfego, responsável pela *dummy* geradora de tráfego, e o das demais

[5] https://code.visualstudio.com/
[6] https://github.com/Lekensteyn/netns

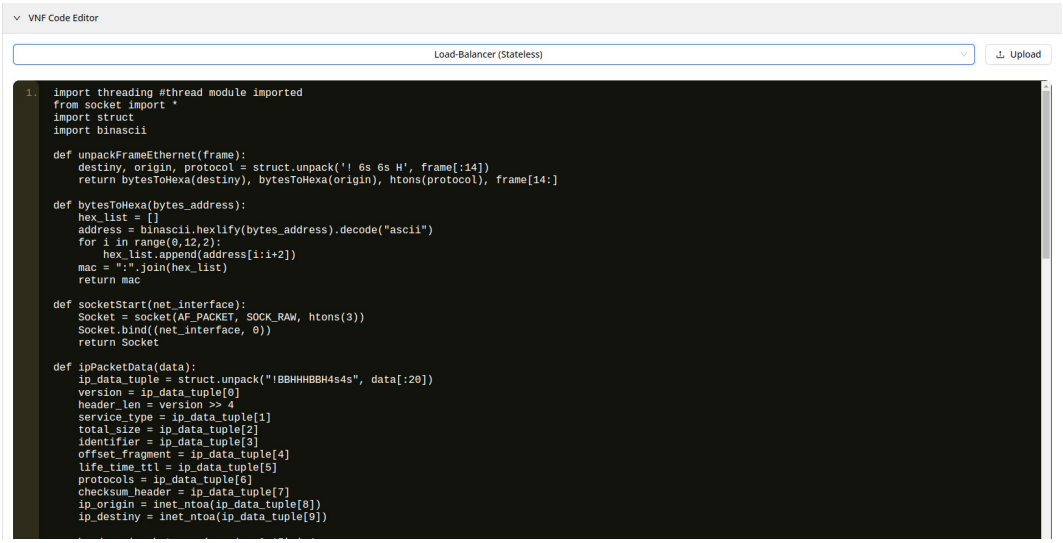


Figura 3.3: Plataforma NFV-Prime: Componente Editor.

dummies. Assim, ao criar uma interface de rede virtual, a mesma terá os seguintes atributos definidos: o host_name, sendo o nome da interface virtual no *host*; o netnamespace_name, sendo o nome da interface no ambiente criado pelo utilitário *netns*; o host_IP é o IP definido no ambiente *host*; o netnamespace_IP é o IP relacionado ao netnamespace para a respectiva interface; o host_ethernet e o netnamespace_ethernet são as configurações de endereço MAC para o *host* e *netnamespace*, respectivamente. Os atributos das interfaces virtuais de rede podem ser utilizados no desenvolvimento da VNF para diferentes objetivos. O componente de Gerenciamento de Interfaces Virtuais de Rede é apresentado na Figura 3.4.

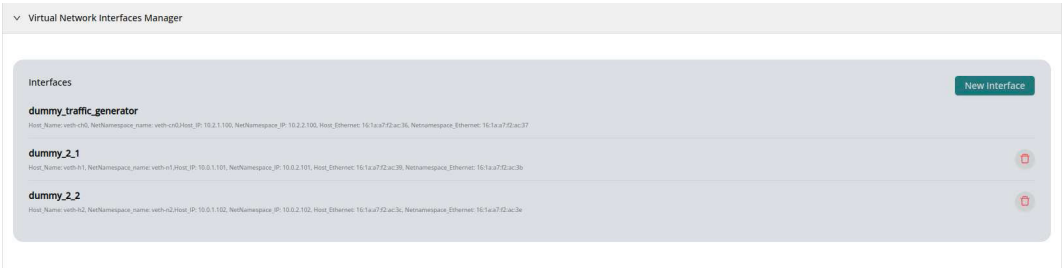


Figura 3.4: Plataforma NFV-Prime: Componente Gerenciamento de Interfaces Viruais de Rede.

O Gerador de Tráfego, por sua vez, possibilita a configuração e parametrização de tráfegos de rede sintéticos que serão executados na rede em conjunto à VNF desenvolvida pelo usuário. A geração do tráfego no NFV-Prime pode ocorrer de duas formas diferentes, a primeira, modo automático, é através da utilização da ferramenta *Nping*, versão 0.7.8, a qual possibilita gerar tráfego sintético na rede, além de disponibilizar um leque de possibilidades de parametrização do mesmo. Nesse caso, a parametrização do tráfego pode acontecer através da taxa de transmissão de pacotes por segundo na rede (*rate*); do tamanho do pacote que será enviado (em bytes); do número de pacotes enviados na rede; do *delay* (em milissegundos) ao enviar os pacotes; da porta para a qual os pacotes serão enviados; da interface virtual de rede ao

qual o tráfego gerado será direcionado; e, por fim, do gatilho temporal (*trigger*) que dispara o tráfego após um intervalo definido. Nesse caso, para auxiliar o usuário, a plataforma NFV-Prime disponibiliza a configuração, predefinida, de dois perfis de tráfego, sendo eles o elefante (Hamdan et al., 2020) e o guepardo (Maji et al., 2017).

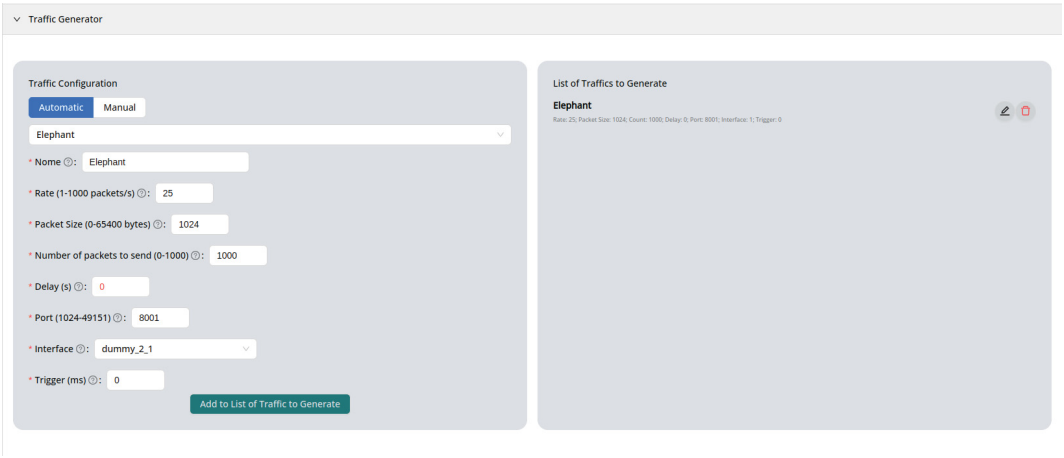


Figura 3.5: Plataforma NFV-Prime: Componente Gerador de Tráfego - Modo Automático.

Já a segunda opção ao utilizar o Gerador de Tráfego, modo manual, possibilita ao usuário desenvolver e instanciar na rede seu próprio gerador de tráfego, sendo que esse precisa, necessariamente, utilizar a linguagem *Python 3*. Assim como no componente Editor, também é possível ao usuário fazer a importação de arquivos de extensão "*.py*". Ao finalizar a configuração de um tráfego, o usuário deve adicioná-lo à lista de tráfegos para que o mesmo seja gerado na rede quando a plataforma NFV-Prime for realizar a execução ao final do *pipeline*. Vale ressaltar que enquanto a plataforma NFV-Prime não estiver executando a VNF na rede é possível editar ou remover tráfegos que estejam presentes na lista do componente. O Gerador de Tráfego pode ser visualizado na Figura 3.6.

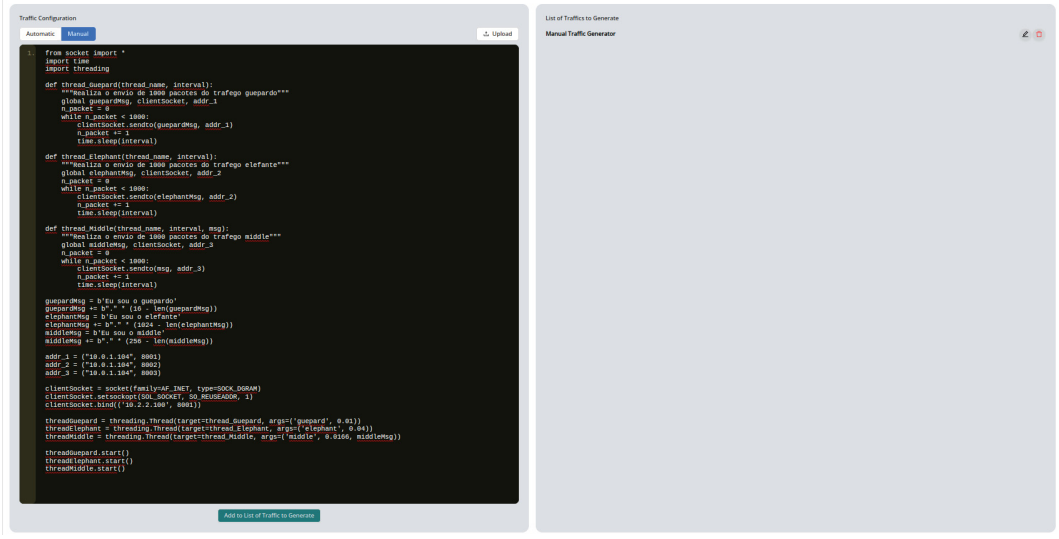


Figura 3.6: Plataforma NFV-Prime: Componente Gerador de Tráfego - Modo Manual.

Já o componente de Visualização é responsável por mostrar em tempo real estatísticas da interface virtual de rede escolhida, sendo mostrados ao usuário em um gráfico a taxa de transmissão e recebimento, TX e RX respectivamente, da interface de rede. Os gráficos servem para mostrar possíveis alterações entre o tráfego de entrada e de saída das interfaces a depender do funcionamento da VNF que está em execução na plataforma. Assim, é possível analisar os impactos que os diversos mecanismos de rede podem gerar no tráfego quando instanciados à rede em diferentes cenários. As estatísticas de TX e RX são obtidas através de um *sniffer* de rede, desenvolvido em *Python3*, utilizando da biblioteca "*socket*". O *sniffer* é inicializado em conjunto a cada nova interface virtual de rede e analisa os pacotes que chegam a essa. Com essas informações o programa atualiza e armazena, a cada segundo, no banco de dados da aplicação, a taxa de envio e recepção para cada interface virtual ativa no NFV-Prime. Além disso, também é possível visualizar a relação de uso da memória (MEM) e da unidade central de processamento (CPU) da VNF instanciada na rede, essa estatística é obtida através de VNFs auxiliares ao NFV-Prime, que observam a VNF do usuário a cada segundo, desde o momento em que a mesma é instanciada na rede. A Figura 3.7 apresenta o componente de Visualização no ambiente da plataforma NFV-Prime.

Figura 3.7: Plataforma NFV-Prime: Componente Visualização.

O diagrama que exemplifica o funcionamento da plataforma NFV-Prime é mostrado na Figura 3.8, sendo a sequência destacada em cor vermelha o uso recomendado da plataforma. Os primeiros passos na plataforma são o desenvolvimento ou importação da VNF, a qual será executada como VNF no ambiente virtual. Se já existirem interfaces virtuais de rede suficientes para a execução correta da VNF, o usuário pode prosseguir para a configuração dos tráfegos que serão gerados ao executar a VNF desenvolvida. Por outro lado, se não existirem, ou houver necessidade de adição, uma nova interface virtual de rede deve ser instanciada, repetindo essa ação até que a quantidade de interfaces seja suficiente para a execução correta da VNF. Após finalizar a parametrização dos tráfegos de rede, o botão de *start* deve ser pressionado para que, no *back-end* seja criado o arquivo "*.py*" a partir do código fonte, o qual será executado no ambiente virtualizado. Além disso, a plataforma também executa os comandos da ferramenta *Nping* através

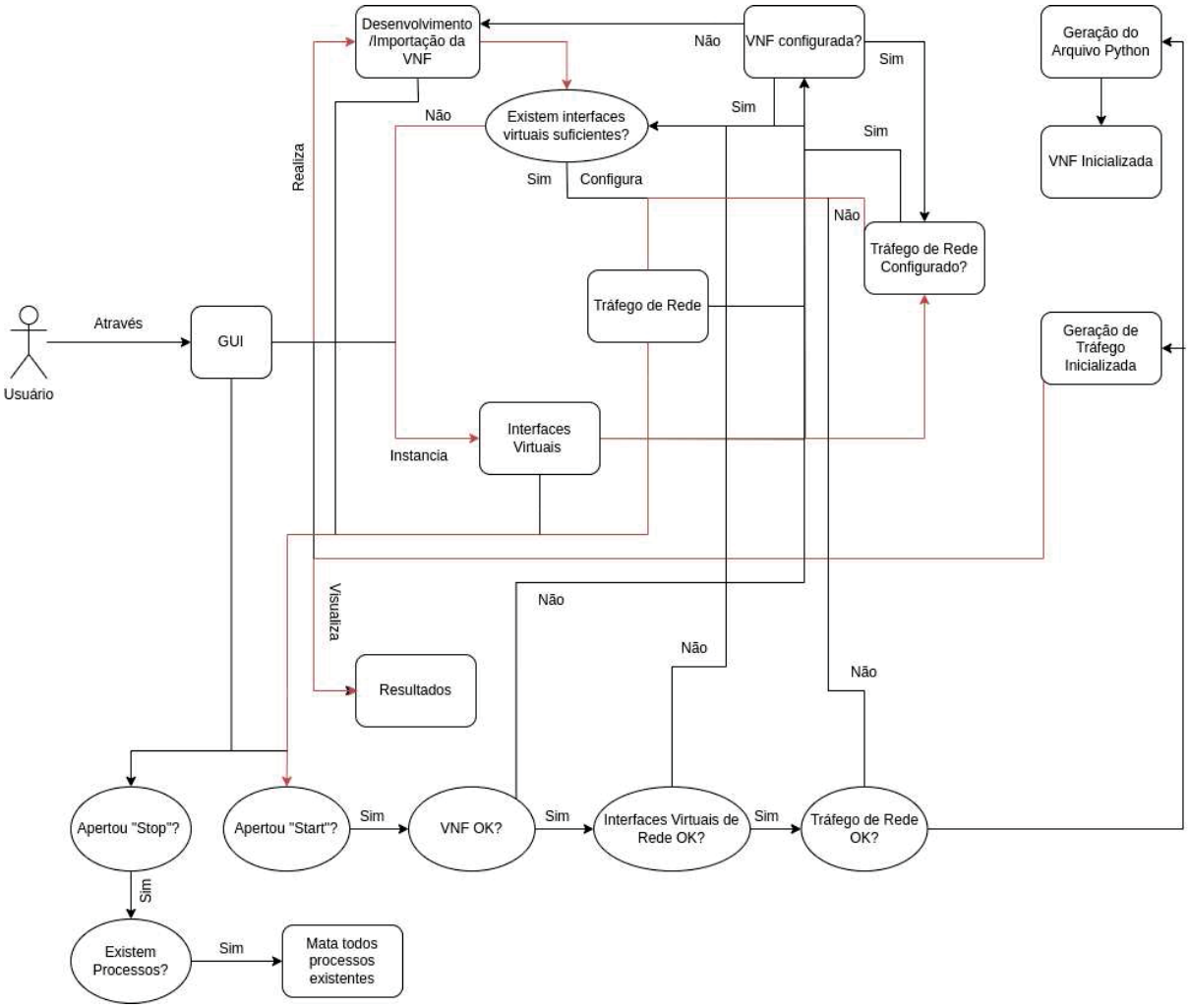


Figura 3.8: Diagrama de funcionamento do NFV-Prime.

de chamadas de sistema ao sistema (biblioteca *sys*), para os tráfegos automatizados, e instancia os geradores de tráfegos manuais na rede de maneira análoga à VNF. Após instanciada a VNF e inicializados os fluxos de pacotes, é possível visualizar nos gráficos as estatísticas da interface, apresentando as métricas de RX e TX, de acordo com a VNF que está em execução. Também é possível visualizar os impactos da VNF sobre a máquina na qual está sendo executada, quando selecionado o modo de visualização da VNF, mostrando a relação de uso de memória e CPU. Por fim, a plataforma permite parar a execução, sendo que todos os componentes da aplicação são parados ao mesmo tempo, necessariamente.

4 NFV-PRIME: AVALIAÇÃO E COMPARAÇÃO DE CONDIÇÕES DE ERRO

A fim de avaliar a plataforma NFV-Prime foi realizada uma comparação entre a interface proposta e a implementação de uma arquitetura NFV-MANO genérica, segundo especificação da ETSI, e flexível (Garcia et al., 2019a), disponibilizada em imagem de disco. O método utilizado para realizar a comparação entre as implementações foi o modelo *Human Error Assessment and Reduction Technique* (HEART) (Alexander, 2017). O modelo visa avaliar uma tarefa genérica, determinando um nível de incerteza associada à tarefa descrita. Para definir o nível de incerteza da tarefa, o modelo fornece uma tabela de valores, levando em conta a experiência esperada do usuário ao qual realizará a tarefa.

No HEART, também é necessário identificar as *Error Producing Conditions* (EPCs) presentes na tarefa e definir sua proporção de efeito. Assim, é possível calcular o impacto das EPCs com base em seu multiplicador, oriundo da tabela de EPCs do modelo HEART, e na proporção de efeito, a qual foi determinada durante a aplicação do modelo. O produtório dos valores de impacto obtido para todas as EPCs resultam no impacto total. Por fim, utiliza-se o valor do impacto total e a incerteza da tarefa para calcular a probabilidade de ocorrer pelo menos 1 erro na execução da tarefa. Entretanto, utilizou-se uma versão simplificada do modelo HEART, descartando a probabilidade de erro para obter uma generalização do resultado do processo, focando o resultado em uma etapa anterior a obtenção da probabilidade. Sendo assim, o impacto total e a quantidade de EPCs para as tarefas nos cenáios estabelecidos são utilizados para realizar a comparação. O nível de experiência do usuário foi apenas considerado na determinação da proporção de efeito das EPCs.

Três cenários serão utilizados para a comparação entre as plataformas de gerenciamento de ciclo de vida de VNFs. No primeiro cenário será avaliado o processo de inicialização das implementações, ou seja, desde a obtenção dos arquivos base até o acesso a implementação, estando apta ao uso. Nas Tabelas 4.1 e 4.2 são apresentados os EPCs e suas respectivas proporções e impactos para a NFV-MANO genérica e a NFV-Prime, respectivamente. Ao analisar os resultados de impacto total obtidos, 530189106,9 para a NFV-MANO genérica, calculados a partir de 11 EPCs e 18,879 para a NFV-Prime, calculados a partir de 3 EPCs. Ao analisar os valores obtidos, percebe-se que o valor obtido para a implementação da NFV-Prime é, expressivamente, inferior ao obtido para a NFV-MANO genérica. Esse fato ocorre pela questão de a NFV-MANO genérica ser disponibilizada em imagem de disco, fator que aumenta consideravelmente a dificuldade na instalação e potencialidade de erros, tendo em vista que a configuração do ambiente para a instalação deve ser feita manualmente. Enquanto a NFV-Prime é desenvolvida para ser uma Aplicação Web, fato que facilita o acesso e uso da palataforma,

diminuindo as possibilidades de erros que possam ocorrer em suas configurações por parte do usuário.

ID	Descrição	Multiplicador	Proporção	Impacto
1	Desconhecimento de uma situação o que é potencialmente importante mas que só ocorre raramente ou que é novo	17	0,5	9
4	Um meio de suprimir ou sobrescrever informações ou recursos superiores que é facilmente acessível	9	0,8	7,4
5	Sem meios de exibir informações espaciais e funcionais de forma intuitiva para o operador	8	0,2	2,4
6	Conflito entre a visão do designer e o modelo do operador	8	0,6	5,2
7	Sem forma óbvia de reverter operações	8	0,25	2,75
12	Discrepância entre risco real e risco perceptível	4	0,1	1,3
13	Feedback precário do sistema	4	0,15	1,45
14	Sem confirmação clara da ação realizada por parte do sistema	3	0,3	1,6
17	Falta de verificação ou testes do resultado	3	0,5	2
19	Sem diversidade de informações entrada para verificações de veracidade	2.5	0,8	7.4
20	Uma incompatibilidade entre o nível de ensino na realização de tarefas de um indivíduo e os requisitos da tarefa	2	0,05	1,05
Impacto Total	530189106,9	-		

Tabela 4.1: 1º Cenário: EPCs para a NFV-MANO genérica.

ID	Descrição	Multiplicador	Proporção	Impacto
1	Desconhecimento de uma situação o que é potencialmente importante mas que só ocorre raramente ou que é novo	17	0,2	4,2
6	Conflito entre a visão do designer e o modelo do operador	8	0,3	3,1
13	Feedback precário do sistema	4	0,15	1,45
Impacto Total	18,879	-		

Tabela 4.2: 1º Cenário: EPCs para a NFV-Prime.

No segundo cenário é analisada a tarefa de instanciar na rede uma VNF e inicializar o tráfego de rede. Nas Tabelas 4.3 e 4.4 são apresentados os EPCs e suas respectivas proporções

e impactos na realização dessa tarefa através de um NFV-MANO genérica e a NFV-Prime, respectivamente. Para a execução dessa tarefa, a NFV-MANO genérica obteve um impacto total de 20036094190777 a partir de 12 EPCs, enquanto a NFV-Prime obteve 79,52112, a partir de 5 EPCs. Essa diferença de valores ocorre principalmente pelo fato de a NFV-Prime relizar o uso de *scripts* pré definidos para realizar a criação e/ou remoção de interfaces virtuais de rede, para a instanciação da VNF à rede, para a definição e parametrização do tráfego de rede que será gerado ao inicializar a VNF. Desta forma, a NFV-Prime remove, em grande parte, a participação do usuário nesses processos, restando ao usuário a seleção de parâmetros ou apenas cliques em botões, diminuindo a possibilidade de erros acontecerem. Enquanto que o valor substancialmente maior da NFV-MANO genérica, em comparação com o NFV-Prime, se refere a todos os processos serem manuais e dependerem do usuário, potencializando os erros.

ID	Descrição	Multiplicador	Proporção	Impacto
1	Desconhecimento de uma situação o que é potencialmente importante mas que só ocorre raramente ou que é novo	17	0,5	9
4	Um meio de suprimir ou sobrescrever informações ou recursos superiores que é facilmente acessível	9	0,8	7,4
5	Sem meios de exibir informações espaciais e funcionais de forma intuitiva para o operador	8	0,2	2,4
7	Sem forma óbvia de reverter operações	8	0,25	2,75
10	A necessidade de transferência específica conhecimento de tarefa em tarefa sem perda	5.5	1	5.5
12	Discrepância entre risco real e risco perceptível	4	0,7	3,1
13	Feedback precário do sistema	4	0,15	1,45
14	Sem confirmação clara da ação realizada por parte do sistema	3	0,3	1,6
17	Falta de verificação ou testes do resultado	3	0,5	2
19	Sem diversidade de informações entrada para verificações de veracidade	2.5	0,8	7.4
20	Uma incompatibilidade entre o nível de ensino na realização de tarefas de um indivíduo e os requisitos da tarefa	2	0,35	1,35
21	Um incentivo para usar outros procedimentos mais perigosos	2	0,4	1,4
Impacto Total	20036094190777	-		

Tabela 4.3: 2º Cenário: EPCs para a NFV-MANO genérica.

ID	Descrição	Multiplicador	Proporção	Impacto
1	Desconhecimento de uma situação o que é potencialmente importante mas que só ocorre raramente ou que é novo	17	0,2	4,2
4	Um meio de suprimir ou sobrescrever informações ou recursos superiores que é facilmente acessível	9	0,4	4,2
13	Feedback precário do sistema	4	0,6	2,8
17	Falta de verificação ou testes do resultado	3	0,2	1,4
20	Uma incompatibilidade entre o nível de ensino na realização de tarefas de um indivíduo e os requisitos da tarefa	2	0,15	1,15
Impacto Total	79,52112	-		

Tabela 4.4: 2º Cenário: EPCs para a NFV-Prime.

No terceiro cenário é analisada a tarefa de visualização de estatísticas de rede e de consumo de recursos computacionais VNF em execução. Nas Tabelas 4.5 e 4.6 são apresentados os EPCs e suas respectivas proporções e impactos na realização dessa tarefa para a NFV-MANO genérica e a NFV-Prime, respectivamente. Para a execução dessa tarefa, a NFV-MANO genérica obteve um impacto total de 2568,56886 a partir de 7 EPCs. Esse valor se deve ao fato de uma NFV-MANO genérica não possuir em suas competências a obrigatoriedade de prover a visualização ou de refinar métricas monitoradas, sendo necessário a utilização de subsistemas terceiros, como Zabbix (LLC, 2001), ou acoplados ao próprio NFV-MANO. Enquanto isso, a NFV-Prime obteve um impacto total de 6,09, calculado a partir de 2 EPCs. Isso se deve ao fato de a plataforma possuir um módulo nativo e simplificado para a visualização de consumo de recursos conputacionais e estatísticas da rede, tendo esse módulo finalidades exclusivamente didáticas.

Os resultados mostram que a diferença no impacto total entre os cenários decorre, principalmente, da visualização intuitiva, da automatização de processos e da implemetação como Aplicação Web. Esses aspectos correspondem ao aumento na quantidade de EPCs de alto impacto presentes na Tabelas 4.1, 4.3 e 4.5, mas ausentes nas Tabela 4.2, 4.4 e 4.6, que são referentes à NFV-Prime. Além disso, alguns dos erros evitados pela interface NFV-Prime são principalmente ligados à automação dos processos, utilizando *scripts* para criação de interfaces virtuais de rede e instanciação das VNFs à rede, além de também reduzir as opções de escolha para o usuário, facilitando e agilizando os processos envolvidos para criar, instanciar e visualizar VNFs. Assim, conforme as análises realizadas, verifica-se que a NFV-Prime simplifica os processos citados acima, sendo capaz de prevenir que o usuário cometa erros durante todo o processo, desde a inicialização da NFV-Prime até a obtenção e visualização de estatísticas.

ID	Descrição	Multiplicador	Proporção	Impacto
1	Desconhecimento de uma situação o que é potencialmente importante mas que só ocorre raramente ou que é novo	17	0,5	9
4	Um meio de suprimir ou sobrescrever informações ou recursos superiores que é facilmente acessível	9	0,8	7,4
5	Sem meios de exibir informações espaciais e funcionais de forma intuitiva para o operador	8	0,2	2,4
7	Sem forma óbvia de reverter operações	8	0,25	2,75
12	Discrepância entre risco real e risco perceptível	4	0,7	3,1
13	Feedback precário do sistema	4	0,15	1,45
20	Uma incompatibilidade entre o nível de ensino na realização de tarefas de um indivíduo e os requisitos da tarefa	2	0,3	1,3
Impacto Total	2568,56886	-		

Tabela 4.5: 3º Cenário: EPCs para a NFV-MANO genérica.

ID	Descrição	Multiplicador	Proporção	Impacto
4	Um meio de suprimir ou sobrescrever informações ou recursos superiores que é facilmente acessível	9	0,4	4,2
13	Feedback precário do sistema	4	0,25	1,45
Impacto Total	6,09	-		

Tabela 4.6: 3º Cenário: EPCs para a NFV-Prime.

5 NFV-PRIME: ESTUDOS DE CASO

Neste capítulo são apresentados os estudos de casos realizados na plataforma NFV-Prime, bem como seus detalhes de configuração e funcionamento. A Seção 5.1 apresenta uma VNF que implementa um mecanismo simplificado de balanceamento de carga (*load balancer*), em modo *stateless*. A Seção 5.2, implementa, também, um balanceador de carga, mas no modo *stateful*. A Seção 5.3 demonstra uma VNF que implementa um algoritmo de inspeção profunda de pacotes (*Deep Packet Inspection* - DPI). Por fim, a Seção 5.4 apresenta uma VNF que implementa um mecanismo de *shaping* de tráfego conhecido como *leaky-bucket*.

5.1 PRIMEIRO ESTUDO DE CASO: *STATELESS LOAD BALANCER*

Para o primeiro estudo de caso foi desenvolvido e instanciado na rede um mecanismo simplificado de balanceamento de carga, conhecido como *Load Balancer*. O mecanismo recebe o tráfego de rede e realiza o balanceamento de carga, distribuindo os pacotes e/ou fluxos entre os diferentes *endpoints* existentes. Dessa forma, ao receber um pacote, o *Load Balancer* verifica, naquele momento, qual a melhor opção de *endpoint* de acordo com suas políticas, realizando o encaminhamento do pacotes recebidos.

No primeiro estudo de caso analisa-se o modo *stateless* da função de *Load Balancer*, a qual não realiza distinção de fluxos, executando o balanceamento da rede orientando-se por pacotes. Para o estudo de caso foram utilizadas três interfaces de rede virtuais, a "*dummy_1*", "*dummy_2*" e "*dummy_3*". Houve também a parametrização do tráfego, utilizando o *Nping* versão 0.7.8 para geração de tráfego. Os parâmetros de tráfego utilizados foram: taxa de transmissão igual a 25 pacotes por segundo; pacotes no tamanho de 1024 *bytes*; quantidade de pacotes transmitidos igual a 1000; porta destino 8001; interface de destino igual a *dummy_1*; sem atraso nos pacotes; e sem gatilho de inicialização. A configuração do tráfego e das interfaces virtuais de rede pode ser visualizada nas Figuras 5.1 e 5.2, respectivamente.

Figura 5.1: *Stateless Load Balancer*: parametrização dos tráfegos de rede.

Ao apertar o botão de inicialização, a VNF de balanceamento de carga é instanciada na rede, recebendo os pacotes e retransmitindo-os de forma uniforme para os diferentes destinos finais. O código fonte desenvolvido para o *Stateless Load Balancer*, utilizando a linguagem Python 3 e a biblioteca *socket*, pode ser visualizado no Apêndice A.1. Primeiramente, é feita a

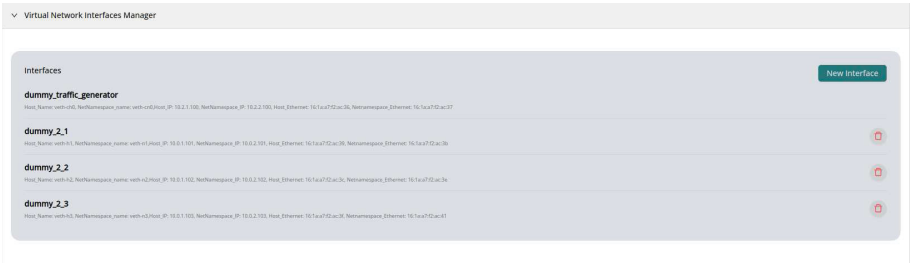


Figura 5.2: *Stateless Load Balancer*: configuração das interfaces virtuais.

instanciação de um *socket* de rede, o *clientSocket*, o qual representa a interface que receberá o fluxo da rede, nesse caso identificada como “veth-ch0”, correspondendo à interface de rede virtual geradora de tráfego no *host*. Além disso, são instanciados mais três *sockets* correspondentes às *dummies* do NFV-Prime, utilizando seus *host_names*. Então, o programa se resume em apenas uma *thread* executando a função *thread_LoadBalancer*, a qual consiste na repetição infinita do recebimento de pacotes pelo *clientSocket*. Ao receber um pacote, o algoritmo realiza seu desempacotamento e verifica se o IP destino do pacote é idêntico ao *host_IP* da interface escolhida, se sim, verifica-se, utilizando-se de um contador de pacotes e do operador de resto da divisão por 3, qual das três interfaces virtuais de rede deve receber o pacote, sendo enviado para a *dummy* escolhida em seguida. Existe, também, a função *saveInfo* de *debbuging* do mecanismo. Essa função atua quando a variável *debug* está ativa, fazendo com que, ao receber 1000 pacotes, o programa seja parado, gerando um arquivo “.csv” com a quantidade de pacotes transmitidos, descartados pelo mecanismo, além da quantidade de pacotes recebidos por cada *dummy* do experimento.

Na Figura 5.3 apresenta a distribuição dos 1000 pacotes transmitidos entre as três interfaces virtuais de rede. A *dummy_1*, com maior prioridade, recebeu 334 pacotes, ou seja, 33,4% dos pacotes transmitidos pelo gerador de tráfego, enquanto as outras 2 interfaces receberam 333 pacotes cada, número que representa 33,3% do total de pacotes transmitidos. Vale ressaltar que, por se tratar de apenas um fluxo de pacotes sendo balanceado na rede a métrica “RX” das três interfaces de rede virtuais ficam com valores próximos, sendo de aproximadamente 8192 *bytes*.

Com isso, esse estudo de caso demonstrou a atuação do mecanismo de balanceamento de rede orientado a pacotes, sendo possível visualizar através dos gráficos a recepção e retransmissão dos pacotes pela VNF. Além disso, foi possível visualizar o balanceamento da rede através do gráfico da distribuição do recebimento de pacotes das interfaces de rede virtuais utilizadas. Assim, destaca-se que, a plataforma NFV-Prime consegue de forma simples mostrar os efeitos que a VNF do *Stateless Load Balancer* causa ao ser instanciada em uma rede.

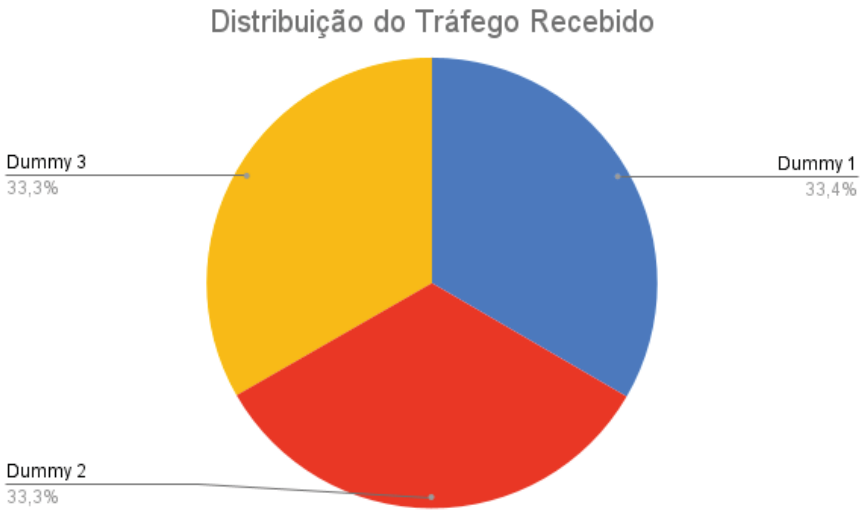


Figura 5.3: *Stateless Load Balancer*: gráfico da distribuição do tráfego recebido entre as *dummies*.

5.2 SEGUNDO ESTUDO DE CASO: *STATEFUL LOAD BALANCER*

O algoritmo de *Stateful Load Balancer* objetiva balancear os fluxos de rede recebidos de forma uniforme entre os diferentes *endpoints* disponíveis, sendo que um fluxo é identificado pela chave IP destino e portas de origem e destino. Desta forma, ao receber um novo fluxo de rede, o algoritmo encaminha o primeiro pacote para a interface de rede virtual que esteja com a menor carga e, a partir desse momento, sempre que um pacote desse fluxo é recebido ele será encaminhado para essa interface. Para realizar este estudo de caso foram utilizadas quatro interfaces de rede virtuais, a "*dummy_1*", "*dummy_2*", "*dummy_3*" e "*dummy_4*", o qual é utilizado apenas como destino final do tráfego, não sendo endereçado como *endpoint* pela VNF. Houve também a parametrização dos tráfegos de rede, totalizando três configurações diferentes, sendo que dois, nomeados de guepardo e "middle", utilizam o *Nping* para gerar o tráfego, enquanto o outro é um gerador de tráfego manual, desenvolvido em Python 3, especificamente para este experimento.

Os parâmetros de tráfego utilizados para o guepardo foram: taxa de transmissão igual a 100 pacotes por segundo, pacotes no tamanho de 64 *bytes*, quantidade de pacotes transmitidos igual a 500, porta destino 8005 e interface destino igual a *dummy_4*, sem atraso nos pacotes e com gatilho de inicialização de 3000 milissegundos. O tráfego "middle" tem a seguinte configuração: taxa de transmissão igual a 25 pacotes por segundo, pacotes no tamanho de 256 *bytes*, quantidade de pacotes transmitidos igual a 250, porta destino 8004 e interface destino igual a *dummy_4*, sem atraso nos pacotes e com gatilho de inicialização de 1500 milissegundos. Já o programa em Python 3, que pode ser visualizado no Apêndice A.2.1, é responsável por gerar 3 tráfegos, sendo que todos realizam o envio de 1000 pacotes tendo como destino final a interface *dummy_4*, sendo que o primeiro tráfego tem a seguinte configuração: taxa de transmissão igual a 100

pacotes por segundo, pacotes no tamanho de 16 *bytes* e porta destino 8001. O segundo tráfego do programa é configurado da seguinte forma: taxa de transmissão igual a 25 pacotes por segundo, pacotes no tamanho de 1024 *bytes* e porta destino 8002. Por fim, o último tráfego gerado tem sua configuração da seguinte forma: taxa de transmissão igual a 60 pacotes por segundo, pacotes no tamanho de 256 *bytes* e porta destino 8003. A configuração do tráfego e das interfaces virtuais de rede podem ser visualizadas nas Figuras 5.4 e 5.5.

Figura 5.4: *Stateful Load Balancer*: parametrização do tráfego de rede.

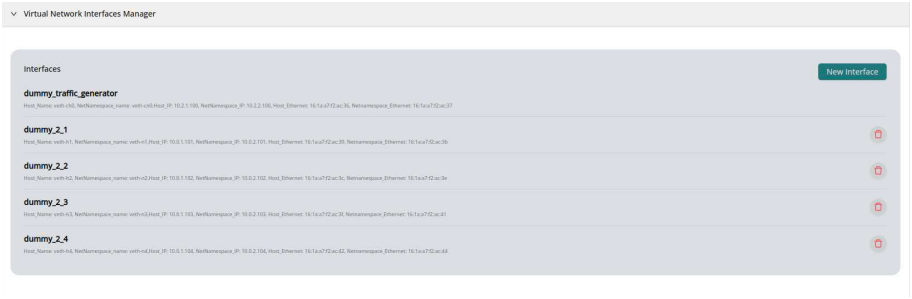


Figura 5.5: *Stateful Load Balancer*: configuração das interfaces virtuais.

Ao apertar o botão de iniciar, a VNF que realiza o balanceamento do tráfego é instanciada na rede, recebendo os fluxos e retransmitindo-os de forma uniforme para as três interfaces de rede virtuais, sendo a *dummy_1*, *dummy_2* e *dummy_3*. O código fonte desenvolvido para o *Stateful Load Balancer*, utilizando a linguagem Python 3 e a biblioteca *socket*, pode ser visualizado no Apêndice A.2. Primeiramente, há a instanciação de um *socket* de rede, o *clientSocket*, o qual representa a interface que receberá os fluxos da rede, nesse caso identificada como "veth-ch0", correspondente da interface de rede virtual geradora de tráfego no *host*. Além disso, são instanciados mais três *sockets* correspondentes às *dummies* citadas acima, utilizando seus *host_names*. Dito isso, o programa consiste em apenas uma *thread* executando a função *thread_LoadBalancer*, a qual consiste na repetição infinita do recebimento de pacotes pelo *clientSocket*.

Ao receber um pacote, a função realiza seu desempacotamento e verifica se o IP destino do pacote está nos padrões das interfaces do NFV-Prime. Em caso afirmativo, verifica-se, utilizando a chave do fluxo, IP destino e portas de origem e destino, se ele já está sendo direcionado para uma das interfaces; se estiver, o pacote do fluxo é encaminhado imediatamente

para a interface em questão. Entretanto, se o pacote analisado é o primeiro do fluxo, então um contador de fluxos e o operador de resto da divisão por 3 são utilizados para descobrir qual das três *dummies* passará a receber esse fluxo, sendo o pacote enviado para a *dummy* escolhida em seguida. Existem, também, as funções *saveInfo* e a *thread_Time* de *debbuging* do mecanismo. Essas funções atuam quando a variável *debug* está ativa, a primeira faz com que ao receber 3750 pacotes, o programa seja parado, gerando um arquivo ".csv" com a quantidade de pacotes transmitidos pelo mecanismo por cada *dummy* do experimento. Enquanto isso, a segunda função analisa a quantidade de *bytes* que está sendo transmitida para cada *dummy* por segundo, salvando essas informações em um arquivo ".csv" auxiliar.

Na Figura 5.6 é possível visualizar a distribuição dos 3750 pacotes na rede. A *dummy_1*, com maior prioridade, recebeu 1250 pacotes, ou seja, 33,3% dos pacotes transmitidos pelo gerador de tráfego, sendo o fluxo "middle" somado a um dos fluxos do gerador manual. Enquanto isso, a *dummy_2* recebeu 1500 pacotes, sendo 40% do total de pacotes transmitidos, recebendo os tráfegos guepardo e um dos tráfegos manuais. Por fim, a *dummy_3* recebeu 1000 pacotes, representando 26,7% do número total de pacotes transmitidos, ou seja, recebeu apenas um fluxo do gerador de tráfego manual.

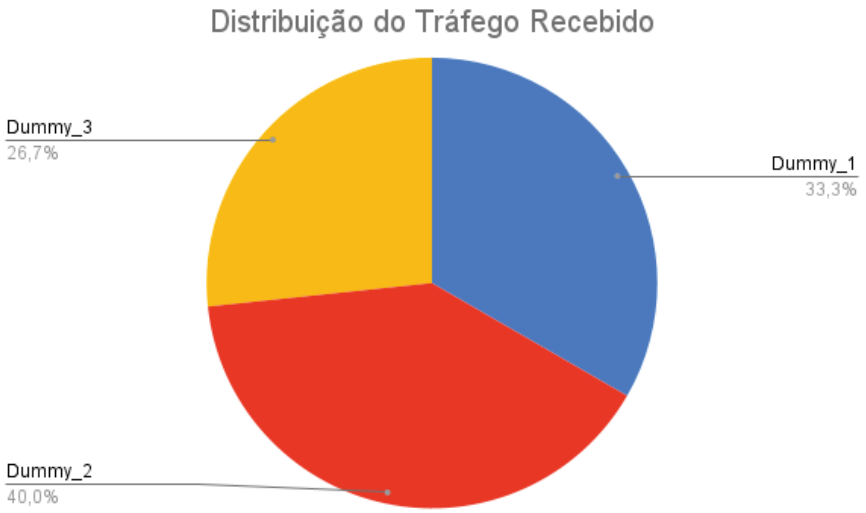


Figura 5.6: Stateful Load Balancer: gráfico da distribuição do tráfego recebido entre as *dummies*.

Já nas Figuras 5.7 e 5.8 são apresentadas as taxas médias do "RX" das *dummies* por segundo e a cada segundo, respectivamente, enquanto a VNF está em execução. Ao analisar as figuras, é possível verificar que, ao todo, apesar da *dummy_1* receber mais pacotes que a *dummy_3*, sua taxa média de recebimento, em *bytes*, é menor, sendo de 6230,76 *bytes*. Esse fato ocorre devido aos fluxos recebidos pela *dummy_1* terem uma taxa de transmissão de *bytes* por segundo inferior aos demais, sendo recebidos dois tráfegos com pacotes de tamanho 16 *bytes* e 256 *bytes*. Além disso, seus fluxos terminam mais rápido que os outros, tendo uma duração de aproximadamente 14 segundos, ou seja, tráfegos que têm uma taxa de transmissão maior que das

demais. A *dummy_2* foi a interface com o maior RX por segundo do experimento, tendo em média 25844,98 *bytes* por segundo recebidos. Além disso, foi a interface que teve o fluxo mais duradouro, de aproximadamente 32 segundos. Dito isso, é possível concluir que seus fluxos tem pacotes com tamanhos maiores, sendo de 64 e 1024 bytes e uma taxa de transmissão menor que das demais. Já a *dummy_3* teve um RX médio de 14222,22 *bytes* por segundo, indicando que seus pacotes têm tamanho de 256 *bytes* e *rate* intermediário quando comparado aos pacotes dos outros fluxos. Fato destacável é perceber o aumento e a diminuição do RX médio quando um fluxo é encerrado ou seu recebimento é iniciado pela interface de rede, sendo possível notar esse comportamento tanto na *dummy_1* (segundo 1.5), quanto na *dummy_2*. Na *dummy_2* é possível notar que a partir do terceiro segundo de execução a última interface citada começou a receber mais de um fluxo, tendo um aumento no valor da RX de 25533,86 *bytes* para 29956,26 *bytes* (em média no 3º e 4° segundo, respectivamente).

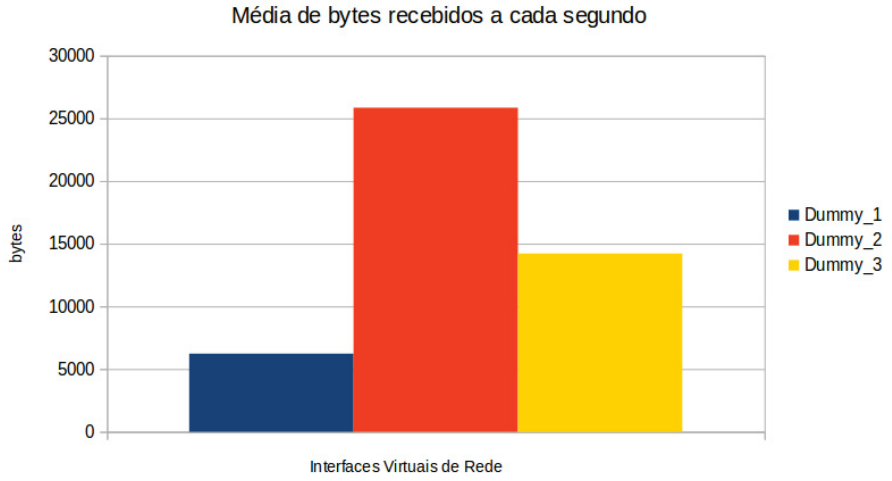


Figura 5.7: *Stateful Load Balancer*: gráfico da média de *bytes* recebidos a cada segundo pelas interfaces.

Destarte, esse estudo de caso demonstrou a atuação do mecanismo de balanceamento de carga orientado a fluxos de pacotes, sendo possível visualizar através dos gráficos a recepção e transmissão dos fluxos pela VNF em execução para as interfaces virtuais de rede da plataforma NFV-Prime. Essa transmissão dos pacotes pela VNF pode ser visualizada através do gráfico da distribuição do recebimento de pacotes das interfaces de rede virtuais utilizadas no estudo de caso. Além de apresentar a informação das taxas de recebimento médias por segundo e a cada segundo de cada *dummy*, comprovando a execução e funcionalidade da VNF, a qual realiza o balanceamento de carga nos termos propostos.

5.3 TERCEIRO ESTUDO DE CASO: INSPEÇÃO PROFUNDA DE PACOTES - DPI

No terceiro estudo de caso, foi escolhida uma função de inspeção profunda de pacotes (*Deep Packet Inspection* - DPI), que é um mecanismo de classificação de tráfego. O DPI realiza

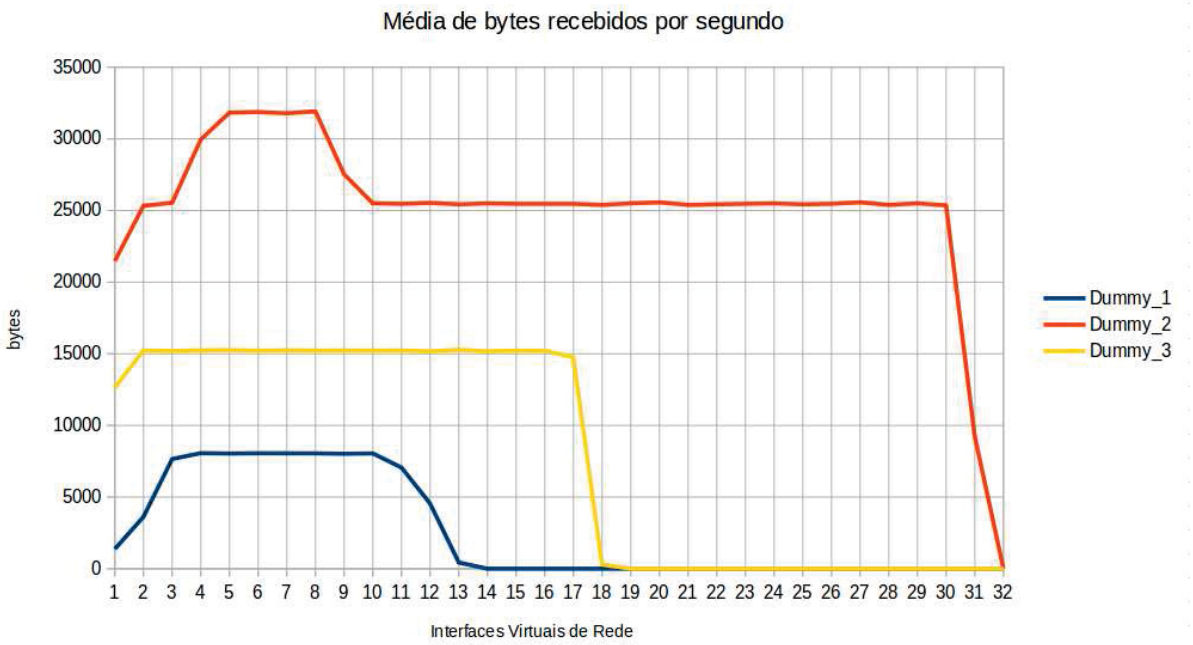


Figura 5.8: *Stateful Load Balancer*: gráfico da média de *bytes* recebidos por segundo pelas interfaces.

uma varredura no conteúdo do pacote em busca de padrões e assinaturas (Bujlow et al., 2015). Neste estudo, utiliza-se uma versão *brute force* do DPI, ou seja, os pacotes recebidos pela VNF são varridos em sua totalidade, buscando padrões pré estabelecidos e, se encontrados, o encaminhamento do pacote para o *endpoint* não é realizado.

Em seguida, é descrita a implementação do DPI, apresentado no Apêndice A.3, executado na rede como uma VNF. Primeiramente, ocorre a instanciação de um *socket* de rede, o *clientSocket*, o qual representa a interface que receberá o fluxo da rede, nesse caso a geradora de tráfego, cujo *hostname* é "veth-ch0". Além disso, também é instanciado o *socket* de rede da interface virtual de rede da plataforma NFV-Prime que receberá o fluxo, *server_interface*, sendo essa a "*dummy_1*", Figura 5.9. Após, é inicializada uma *thread*, sendo executada a função *thread_DPI*, a qual consiste na repetição infinita do recebimento de pacotes pelo *clientSocket*. Ao receber um pacote, seu desempacotamento é realizado, verificando se o IP destino do pacote está nos padrões das interfaces do NFV-Prime. Em caso afirmativo, o conteúdo do pacote é varrido em busca de expressões regulares com o auxílio da biblioteca "re" do Python 3. Se um padrão buscado for encontrado, a função que implementa o DPI realizará o descarte do pacote. Entretanto, se não encontrar, realizará o encaminhamento do pacote para seu respectivo *endpoint*. Existe, também, a função *saveInfo* de *debbuging* do mecanismo. Essa função atua quando a variável *debug* está ativa, fazendo com que, ao receber 1000 pacotes, o programa seja parado, gerando um arquivo ".csv" com a quantidade de pacotes transmitidos, descartados pelo mecanismo, além da quantidade de pacotes recebidos por cada *dummy* do experimento.

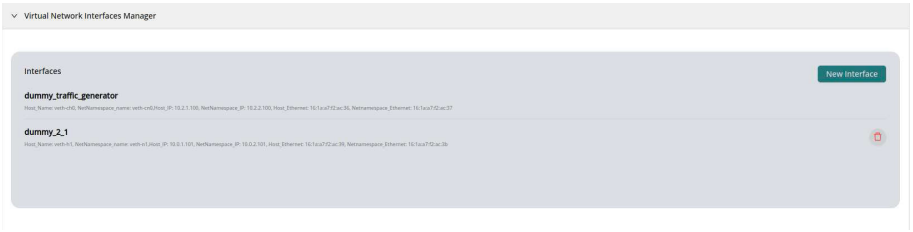


Figura 5.9: *DPI*: configuração das interfaces virtuais.

Neste estudo de caso, foi utilizado um gerador de tráfego manual, ou seja, um algoritmo, em Python 3, para a geração de tráfego sintético da rede. O gerador de tráfego tem duas versões, sendo as duas parametrizáveis. Na primeira versão, apresentada no Apêndice A.3.1, o gerador consiste em uma *thread* que realiza o envio de 1000 pacotes para a interface virtual de rede conhecida como *dummy_1*. Entretanto, existe uma probabilidade, parametrizável, de envio de um pacote que contenha o padrão determinado, podendo o padrão ser colocado no início, meio ou fim do conteúdo do pacote. Para isso, foi utilizada a biblioteca "random" e definida uma semente de aleatorização comum para que todas as execuções gerem a mesma sequência de pacotes. Nesse caso, foram executados cenários nos quais pacotes transmitidos pelo gerador tinham 10%, 30%, 60% e 100% de chance de conter o padrão a ser buscado pelo DPI. Na Figura 5.10, é possível verificar quantos pacotes contêm o padrão nos quatro cenários destacados, e também a quantidade que contêm o padrão no início, meio ou fim. Ao analisar, nota-se que a quantidade de pacotes dentre as possibilidades que contenham o padrão buscado, é distribuída de forma proporcional, sendo que na execução com 60% de probabilidade de conterem o padrão, 593 pacotes tiveram o padrão identificado, onde 217 com o padrão no início, 199 contendo o padrão no meio e 177 pacotes com o padrão identificado no final de seu *payload*.

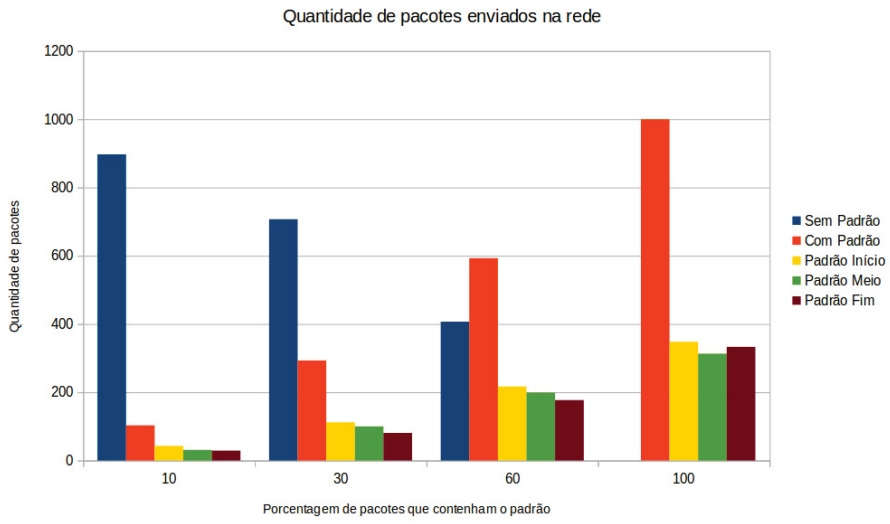


Figura 5.10: DPI: quantidade de pacotes enviados e sua distribuição nos quatro cenários descritos.

Na Figura 5.11 é apresentado o tempo médio de duração, em microssegundos, para o DPI fazer o processamento e realizar a respectiva ação de encaminhamento ou descarte do pacote. A VNF foi executada para tráfegos com 10%, 30%, 60% e 100% de probabilidade dos pacotes conterem o padrão. Os resultados permitem concluir que a média de tempo para o processamento dos pacotes que não contêm o padrão é sempre maior que a de processamento dos pacotes que contêm o padrão, que pode estar localizado no *payload* em qualquer das posições definidas. O tráfego de 10% de probabilidade, transmite 897 pacotes sem o padrão e 103 pacotes com o padrão, sendo a média de tempo para que o pacote seja processado de 2 microssegundos. Já a média de tempo para o procesamento dos que contém o padrão é de 0,2 microssegundos, enquanto para o processamentos de todos os pacotes é de 1,82 microssegundos. Entretanto, verificando os resultados obtidos para o tráfego com 60% de probabilidade, no qual foram transmitidos 407 pacotes sem o padrão e 593 pacotes contendo o padrão, o tempo médio do processamento desses pacotes é de 2,02 e 0,03 microssegundos, respectivamente. Enquanto isso, essa média para o processamento não distinguindo os pacotes, foi de 0,84 microssegundos.

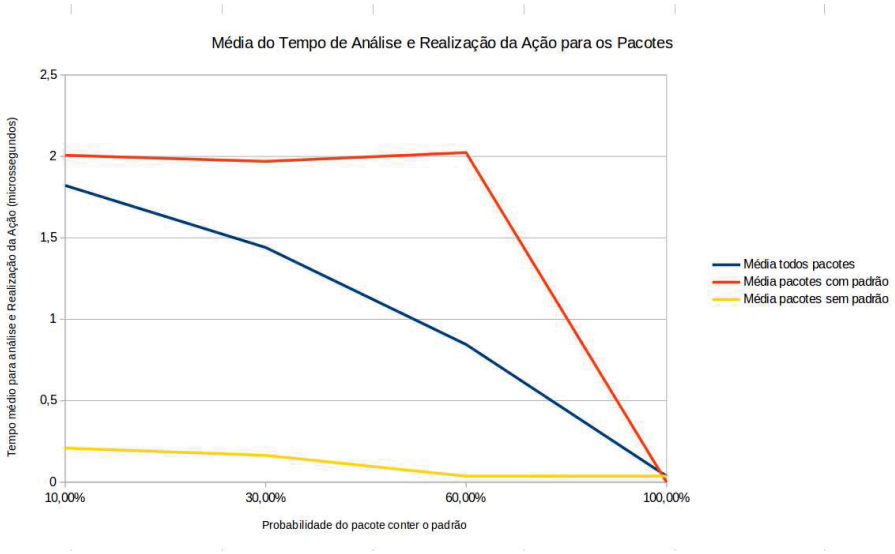


Figura 5.11: DPI: tempo médio de duração do processamento do pacote.

Já na segunda versão, apresentada no Apêndice A.3.2, o gerador consiste em uma *thread* que realiza o envio de 1000 pacotes para a interface virtual de rede conhecida como *dummy_1*. Entretanto, todos os pacote transmitidos contêm o padrão a ser procurado pelo DPI, sendo parametrizável o local em que esse padrão será inserido em todos os pacotes, podendo ser no início, meio ou fim do *payload*. Assim, três cenários foram executados, sendo que em cada cenário o padrão está em posições diferentes do *payload*, para verificar os impactos na performance da rede. Na Figura 5.12, é possível verificar a média de tempo que a função DPI levou para realizar a busca ao padrão e posteriormente a ação correspondente. É possível notar que, pacotes com o padrão a ser buscado no início, meio e fim do *payload* obtiveram médias de tempo de processamento por pacote de 0,0334; 0,0341; e 0,0349 microssegundos,

respectivamente. Por se tratar do envio de 1000 pacotes, tem-se que no total foram necessários 33,4; 34,1; e 34,9 microssegundos para analisar e realizar a ação necessária para todos os pacotes do tráfego. Assim, é possível afirmar que quando o padrão está no final do *payload*, a função DPI necessita de um tempo maior para o pacote tenha o seu padrão identificado. Já quando o padrão encontra-se no início, esse é identificado de forma mais rápida. Fato esse que ocorre por conta da biblioteca "re" do Python utilizar um algoritmo que percorre o pacote do início para o seu fim.

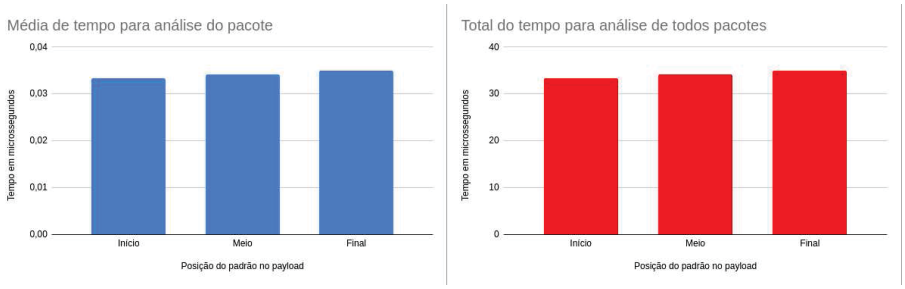


Figura 5.12: DPI: gráficos da média e total de tempo para processamento dos pacotes.

Em síntese, as figuras apresentadas demonstram a execução VNF que implementa o DPI na rede, fazendo com que pacotes que contenham o padrão sejam descartados. Através dos gráficos também foi possível afirmar que a posição em que o conteúdo procurado está, impacta diretamente na performance da VNF de DPI. Desta forma, ao incluí-la na rede, a mesma pode impactar negativamente no *jitter*.

5.4 QUARTO ESTUDO DE CASO: *LEAKY-BUCKET*

O quarto estudo de caso considerou o uso de uma função de *Leaky Bucket* (LB), que é um mecanismo de *shaping*, ou seja, realiza um controle sobre o fluxo de dados de uma rede. No algoritmo do LB, temos o *bucket*, que é a representação de uma fila, o qual é preenchido com os pacotes recebidos pela interface de rede. Assim, a profundidade do *bucket* define a quantidade máxima de pacotes que podem ser enfileirados nele. Deste modo, ao se atingir a capacidade máxima, os pacotes que chegam não poderão mais ser armazenados e, portanto, serão descartados (Evans e Filsfils, 2010).

Ao se aplicar o LB em um tráfego de rede, é necessário que a fila de pacotes do *bucket* seja consumida. Dessa forma, o consumo ocorre considerando uma taxa constante (*rate*), determinando a quantidade de pacotes que será reduzida do *bucket*, como se o mesmo estivesse com um furo na sua base. A taxa é aplicada utilizando um intervalo fixo de tempo. Assim, a cada intervalo, uma quantidade de pacotes equivalente ao valor da taxa será drenada do *bucket*. A Figura 5.13 mostra o processo, já descrito, realizado pelo algoritmo do LB após o recebimento de um pacote.

Por fim, destaca-se que o uso do LB tem como principal objetivo suavizar picos de tráfego na rede. Assim, é possível adequar a vazão da rede a uma taxa constante definida no

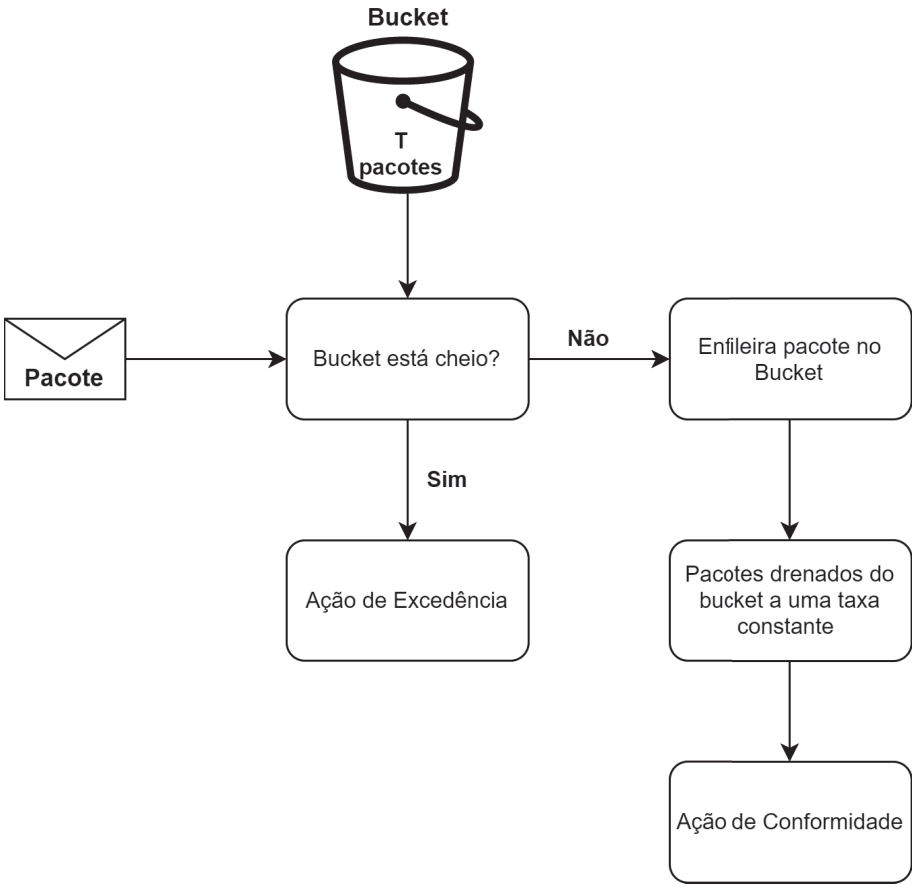


Figura 5.13: Fluxograma *Leaky Bucket*.

shaper. Em seguida é descrita a implementação do *Leaky Bucket*, executado na rede como uma VNF. Primeiramente, há parametrização do LB e a instanciação de um *socket* de rede, o *clientSocket*, o qual representa a interface que receberá o fluxo da rede, nesse caso a geradora de tráfego, cujo *hostname* é "veth-ch0". Após, são inicializadas duas *threads*, sendo que uma executa a função *thread_Time*, enquanto a outra executa a função *thread_LeakyBucket*. A primeira é responsável por realizar o consumo do *bucket*, através da função *consumeBucket*, a cada intervalo de tempo estabelecido. Neste instante são transmitidos alguns pacotes para o destino final, sendo esse o *server_interface*, interface de rede "*dummy_1*". O LB tem definida uma taxa máxima de envio por intervalo, sendo essa representada pela variável *packets_to_release*, não podendo ser ultrapassada. Desta forma, são enviados a cada intervalo, o máximo de pacotes possíveis.

A segunda *thread* é responsável por receber o tráfego de rede através do *socket* configurado, *clientSocket*. Vale lembrar que, no caso do LB, o *bucket* corresponde à fila, responsável por armazenar os pacotes até que sejam enviados ao destino final. Porém, o *bucket* tem um limite de pacotes que podem ser enfileirados, valor representado pela variável *bucket_max_size*. Desta forma, o LB, ao receber um pacote, realiza uma verificação do tamanho da fila, se existirem pacotes armazenados e a quantidade máxima ainda não foi atingida, o pacote é enfileirado. Por outro lado, se a quantidade máxima tiver sido atingida, o pacote recebido é

descartado pelo LB. Por fim, se a fila estiver vazia e ainda há a possibilidade de envio na janela do intervalo, o pacote é enviado ao destino final, não sendo armazenado. Existem, também, as funções de *debbuging* do mecanismo *analyzeBucket* e *saveInfo*, as quais são executadas apenas se a variável *debug* estiver ativa. A primeira função armazena, em um arquivo ".csv", informações sobre a quantidade de pacotes de cada tráfego está presente no *bucket* a cada segundo de execução. Enquanto isso, a segunda função atua quando a marca de, neste caso, 3000 pacotes recebidos é atingida, realizando a parada do programa e gerando um arquivo ".csv" com a quantidade de pacotes transmitidos e descartados pelo mecanismo. O código do LB está descrito no Apêndice A.4.

Neste estudo de caso, foram realizados dois experimentos com a função LB. Nos dois foram utilizados três fluxos de rede, com diferentes perfis de tráfego, sendo eles guepardo, elefante e "middle", meio termo entre o guepardo e o elefante, todos sendo gerados a partir da ferramenta *Nping*. A parametrização do guepardo é a seguinte: taxa de transmissão igual a 100 pacotes por segundo, pacotes no tamanho de 16 *bytes*, quantidade de pacotes transmitidos igual a 1000, porta destino 8001 e interface destino igual a *dummy_1*, sem atraso nos pacotes e com gatilho de inicialização igual a 0 milissegundos nos dois experimentos. O elefante teve a seguinte paramerização: taxa de transmissão igual a 25 pacotes por segundo, pacotes no tamanho de 1024 *bytes*, quantidade de pacotes transmitidos igual a 1000, porta destino 8001 e interface destino igual a *dummy_1*, sem atraso nos pacotes e com 15000 milissegundos de gatilho de inicialização no primeiro experimento e 0 milissegundos para o segundo experimento. Já o tráfego "middle" é configurado da seguinte forma: taxa de transmissão igual a 60 pacotes por segundo, pacotes no tamanho de 256 *bytes*, quantidade de pacotes transmitidos igual a 1000, porta destino 8001 e interface destino igual a *dummy_1*, sem atraso nos pacotes e com 60000 milissegundos de gatilho de inicialização no primeiro experimento e 0 milissegundos para o segundo experimento. Na Figura 5.14 e 5.15 é apresentada a configuração de tráfego utilizada para o primeiro e segundo experimento, respectivamente. Já na Figura 5.16 é apresentada a configuração das interfaces virtuais de rede que foi utilizada para ambos experimentos.

Figura 5.14: *Leaky Bucket*: parametrização dos tráfegos.

O primeiro experimento é realizado com os três tráfegos isolados, ou seja, a inicialização de um depende do término do outro. Assim, é possível verificar os padrões executados pelo LB para cada tráfego especificamente. Neste caso, o LB utilizado foi parametrizado para ter um

Figura 5.15: *Leaky Bucket*: parametrização dos tráfegos.

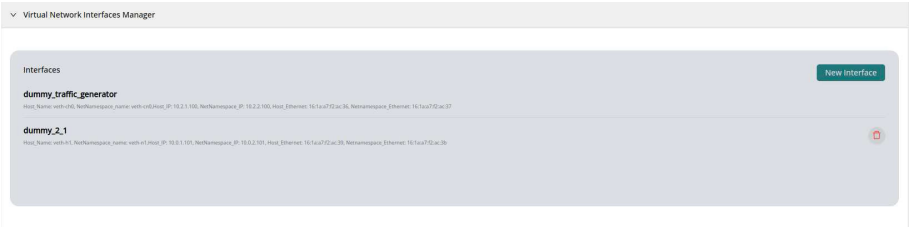


Figura 5.16: *Leaky Bucket*: configuração das interfaces virtuais.

bucket, tamanho da fila, de 100 pacotes sendo que a cada intervalo de 1 segundo, 50 pacotes são transmitidos. A Figura 5.17, apresenta os dados sobre as médias de transmissões realizadas pelo LB para cada perfil de tráfego. Ao analisá-lo é possível verificar que o perfil de tráfego guepardo realizou a transmissão de, em média, 614,86 pacotes, tendo um desvio padrão de 14,38 pacotes entre as execuções. Isso se deve ao fato de que o guepardo é o tráfego com maior quantidade de pacotes transmitidos por segundo, fazendo com que ele seja o maior prejudicado e tenha a menor quantidade de pacotes transmitidos dentre os três tráfegos. Já o tráfego "middle", obteve em média 822,26 pacotes transmitidos, porém seu desvio padrão entre as execuções foi o mais alto dentre os tráfegos, sendo de 42,94 pacotes, fato que se deve ao número de pacotes enviados por segundo pelo tráfego estar próximo à taxa de transmissão do LB, sendo assim qualquer dessincronia pode gerar um alto impacto ao tráfego. Por fim, o tráfego elefante realizou a transmissão, em média, de todos os seus pacotes, totalizando 1000 transmissões, visto que sua taxa de pacotes por segundo é menor que a taxa de transmissão por segundo do LB. Assim, este estudo obteve uma transmissão, em média, de 2437,13 pacotes a cada execução, com desvio padrão de 45,24, métrica alavancada principalmente pelo "middle".

Para o segundo experimento, gerou-se os três tráfegos concomitantemente. Neste caso, o LB foi parametrizado de quatro formas diferentes, sendo os valores escolhidos para o *rate* 50, 75, 100 e 150 pacotes por segundo e seus respectivos tamanho de *buckets* 100, 150, 200 e 300 pacotes, sendo a execução identificado pelo seguinte nome "*rate*/*bucket*". Na Figura 5.18 é possível visualizar a quantidade de pacotes que foram transmitidos, em média, pelo LB, para cada perfil de tráfego. Ao analisar o gráfico é possível verificar que o guepardo, assim como no primeiro experimento do LB, continua a ser o tráfego mais prejudicado, sendo transmitidos 333,03 pacotes na execução 50/100, aumentando para 492,03 pacotes quando 75/150, aumentando para 640,9

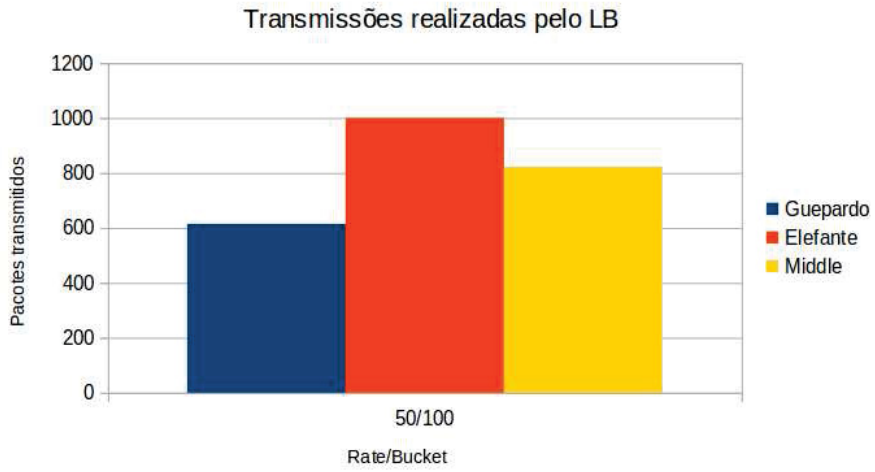


Figura 5.17: *Leaky Bucket*: gráfico de transmissões realizadas pelo LB para os diferentes perfis de tráfego.

pacotes na execução 100/200 e, finalmente, 934,1 pacotes na execução 150/300. Já o elefante, assim como no experimento passado, foi o tráfego que mais realizou transmissões obtendo 635,43 pacotes, 760,63 pacotes, 880,4 pacotes e 966,6 pacotes transmitidos nas execuções 50/100, 75/150, 100/200 e 150/300, respectivamente. Já o tráfego "middle" com o LB utilizando as seguintes parametrizações: 50/100, 75/150, 100/200 e 150/300, realizou a transmissão de 414,76 pacotes, 619,06 pacotes, 773,43 pacotes e 958,3 pacotes, respectivamente. Esses resultado são obtidos pelo fato de o elefante ser o tráfego mais duradouro, visto que o término dos fluxos guepardo e "middle" liberam a fila para que o fluxo elefante.

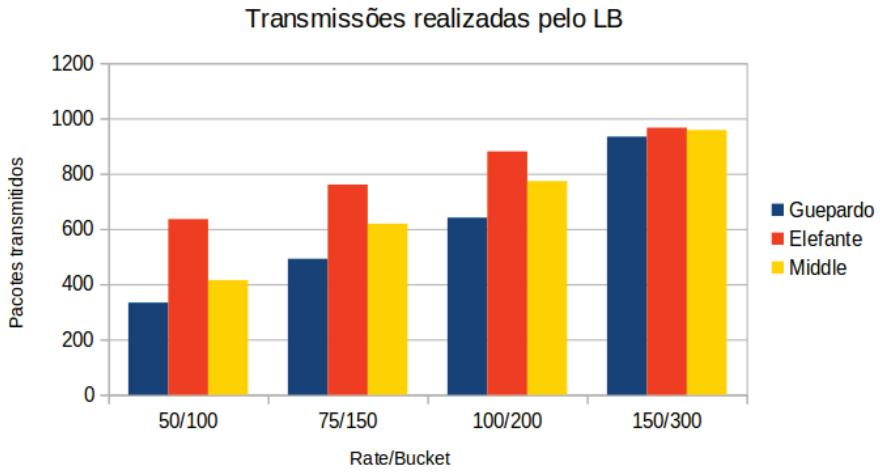


Figura 5.18: *Leaky Bucket*: quantidade de pacotes que foram transmitidos para cada perfil de tráfego.

A Figura 5.19 reforça as análise apresentadas anteriormente ao apresentar a quantidade de pacotes dos fluxos presentes no *bucket* a cada segundo da execução com parâmetros 100/200. É possível verificar que existe uma padronização no preenchimento do *bucket*, podendo ser visto principalmente quando a VNF está recebendo pacotes dos três fluxos, sendo que mais da metade

do *bucket* é preenchido por pacotes do tráfego guepardo, tendo em média 106,67 pacotes, já o "middle" tem em média 66,67 pacotes e o elefante 26,67 pacotes. Com base nesses dados é interessante notar que a cada segundo, os tráfegos colocam a mesma quantidade de pacotes, sendo aproximadamente 50% da sua taxa de transmissão ao *bucket*. Também, ao analisar o desvio padrão, é possível constatar que o momento em que existe um distúrbio maior é justamente quando um dos tráfegos encerra seu fluxo, fazendo com que os fluxos restantes consigam ocupar ainda mais o *bucket*.

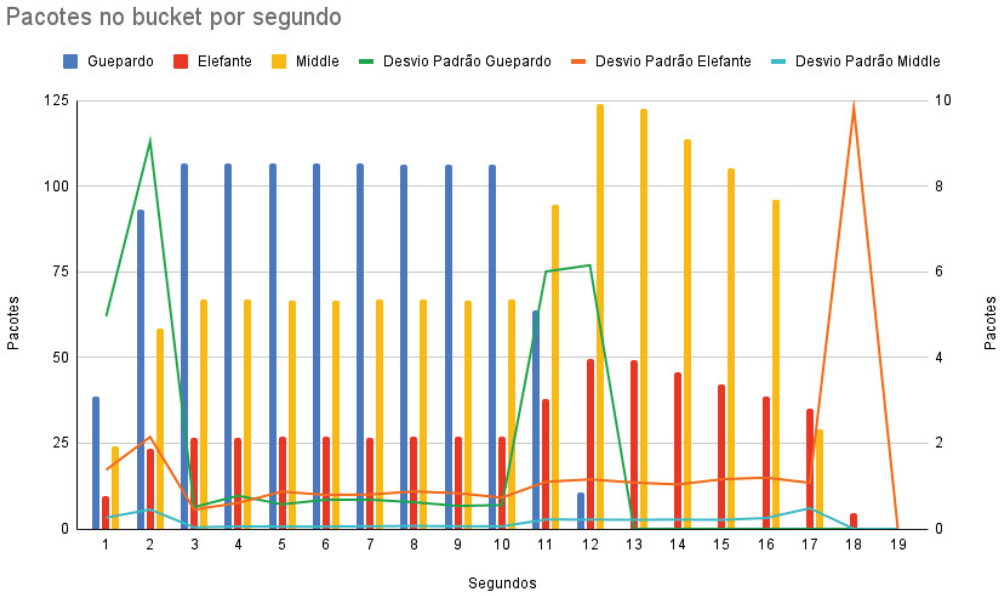


Figura 5.19: *Leaky Bucket*: gráfico da quantidade de pacotes no *bucket* por segundo na execução 100/200.

Em síntese, neste estudo de caso, as figuras demonstram a execução correta da VNF que implementa o LB na rede, fazendo com que se respeite a quantidade máxima de pacotes enviados por segundo. Se essa condição não for respeitada, o LB realiza o enfileiramento de pacotes no *bucket* e se essa ação não for possível, ocorre o descarte do pacote. Através dos gráficos também foi possível afirmar que os padrões estabelecido pelo LB para os tráfegos separados tendem a se manter quando os fluxos são concomitentes. Entretanto, quando o LB recebe os tráfegos em conjunto, os mesmos ocupam o *bucket* de forma que, neste estudo de caso, a cada segundo os fluxos tenham proporções parecidas de pacotes na fila, quando comparados às suas taxas de transmissão.

6 CONCLUSÃO

O uso de tecnologias NFV flexibiliza a implementação, implantação e gerenciamento de funções de rede. Este trabalho apresentou a plataforma NFV-Prime para prototipação e visualização de VNFs. Um de seus principais objetivos é facilitar a introdução ao tema para novos usuários. A plataforma consiste de quatro componentes, Editor, Gerenciador de Interfaces Virtuais de Rede, Gerador de Tráfego, e o Visualização, que permite acompanhar a execução da função de rede em tempo real. Vale reforçar que a plataforma NFV-Prime possibilita uma grande variedade de parametrizações e configurações dos tráfegos de rede que serão gerados durante a execução da VNF. Para apresentação da plataforma foram desenvolvidos quatro estudos de caso: *Stateless Load Balancer*, *Stateful Load Balancer*, *Deep Packet Inspection*, e *Leaky Bucket*. A escolha dos estudos de caso visa demonstrar que a plataforma foi desenvolvida para suportar diferentes perfis de NFs.

Trabalhos futuros incluem expandir a plataforma NFV-Prime, permitindo múltiplos usuários simultâneos. Nesse caso, será também necessário utilizar um serviço de autenticação. Além disso, também é objetivo a expansão das métricas do módulo Visualização. Também deve ser criado um componente de *debugging* da plataforma, que permite ao usuário verificar e analisar a execução da VNF através desse componente. É também objetivo a disponibilização do NFV-Prime, em larga escala, como uma Aplicação Web.

REFERÊNCIAS

Alexander, T. M. (2017). Human error assessment and reduction technique (heart) and human factor analysis and classification system (hfacs). Em *Collaboration on Quality in the Space and Defense Industries Forum (CQSDI)*, número KSC-E-DAA-TN40040.

Bondan, L., Franco, M. F., Marcuzzo, L., Venancio, G., Santos, R. L., Pfitscher, R. J., Scheid, E. J., Stiller, B., De Turck, F., Duarte, E. P. et al. (2019). Fende: marketplace-based distribution, execution, and life cycle management of vnfs. *IEEE Communications Magazine*, 57(1):13–19.

Bujlow, T., Carela-Español, V. e Barlet-Ros, P. (2015). Independent comparison of popular dpi tools for traffic classification. *Computer Networks*, 76:75–89.

Castillo-Lema, J., Neto, A. V., de Oliveira, F. e Kofuji, S. T. (2019). Mininet-nfv: Evolving mininet with oasis tosca nvf profiles towards reproducible nfv prototyping. Em *2019 IEEE Conference on Network Softwarization (NetSoft)*, páginas 506–512. IEEE.

Cotroneo, D., De Simone, L., Iannillo, A. K., Lanzaro, A., Natella, R., Fan, J. e Ping, W. (2014). Network function virtualization: Challenges and directions for reliability assurance. Em *International Symposium on Software Reliability Engineering Workshops*, páginas 37–42. IEEE.

Cui, C., Deng, H., Telekom, D., Michel, U. e Damker, H. (2012). Network functions virtualisation.

da Cruz Marcuzzo, L., Garcia, V. F., Cunha, V., Corujo, D., Barraca, J. P., Aguiar, R. L., Schaeffer-Filho, A. E., Granville, L. Z. e dos Santos, C. R. P. (2017). Click-on-osv: A platform for running click-based middleboxes. Em *IFIP/IEEE Symposium on Integrated Network and Service Management*, páginas 885–886.

ETSI (2020). Osm. `https://osm.etsi.org/`. Acessado em 27/10/2021.

ETSI, N. (2014). Network functions virtualisation (nfv); terminology for main concepts in nfv. *Group Specification, Dec*.

Evans, J. W. e Filsfils, C. (2010). *Deploying IP and MPLS QoS for multiservice networks: theory and practice*. Elsevier.

Flauzino, J., Fulber-Garcia, V., Huff, A., Venâncio, G. e Duarte Jr, E. P. (2021). Gerência e orquestração de funções e serviços de rede virtualizados em nuvem cloudstack. Em *Workshop de Gerência e Operação de Redes e Serviços*, páginas 82–95. SBC.

FRAUNHOFER-FOKUS e TU-BERLIN (2017). Open baton. `https://openbaton.github.io/`. Acessado em 27/10/2021.

Fulber-Garcia, V., Duarte Jr, E. P., Huff, A. e dos Santos, C. R. (2020a). Network service topology: Formalization, taxonomy and the custom specification model. *Computer Networks*, 178:107337.

Fulber-Garcia, V., Luizelli, M. C., dos Santos, C. R. P. e Duarte, E. P. (2020b). Cusco: a customizable solution for nfv composition. Em *International Conference on Advanced Information Networking and Applications*, páginas 204–216. Springer.

Garcia, V. F., Marcuzzo, L. d. C., Huff, A., Bondan, L., Nobre, J. C., Schaeffer-Filho, A., dos Santos, C. R., Granville, L. Z. e Duarte, E. P. (2019a). On the design of a flexible architecture for virtualized network function platforms. Em *2019 IEEE Global Communications Conference (GLOBECOM)*, páginas 1–6. IEEE.

Garcia, V. F., Marcuzzo, L. d. C., Huff, A., Bondan, L., Nobre, J. C., Schaeffer-Filho, A., dos Santos, C. R. P., Granville, L. Z. e Duarte, E. P. (2019b). On the design of a flexible architecture for virtualized network function platforms. Em *IEEE Global Communications Conference*, páginas 1–6.

Halpern, J. e Pignataro, C. (2015). Service function chaining (sfc) architecture. Relatório técnico.

Hamdan, M., Mohammed, B., Humayun, U., Abdelaziz, A., Khan, S., Ali, M. A., Imran, M. e Marsono, M. N. (2020). Flow-aware elephant flow detection for software-defined networks. *IEEE Access*, 8:72585–72597.

Han, B., Gopalakrishnan, V., Ji, L. e Lee, S. (2015). Network function virtualization: Challenges and opportunities for innovations. *IEEE Communications Magazine*, 53(2):90–97.

Huff, A., Venancio, G., Marcuzzo, L. d. C., Garcia, V. F., dos Santos, C. R. e Duarte, E. P. (2018). A holistic approach to define service chains using click-on-osv on different nfv platforms. Em *2018 IEEE Global Communications Conference (GLOBECOM)*, páginas 1–6. IEEE.

LLC, Z. (2001). Zabbix. `https://www.zabbix.com/`. Acessado em 27/10/2021.

Luizelli, M. C., Bays, L. R., Buriol, L. S., Barcellos, M. P. e Gaspary, L. P. (2015). Piecing together the nfv provisioning puzzle: Efficient placement and chaining of virtual network functions. Em *IFIP/IEEE International Symposium on Integrated Network Management*, páginas 98–106.

Maji, S., Veeraraghavan, M., Buchanan, M., Alali, F., Ros-Giral, J. e Commike, A. (2017). A high-speed cheetah flow identification network function (cfinf). Em *2017 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, páginas 1–7.

Marcuzzo, L. D. C., Garcia, V. F., Cunha, V., Corujo, D., Barraca, J. P., Aguiar, R. L., Schaeffer-Filho, A. E., Granville, L. Z. e dos Santos, C. R. (2017). Click-on-osv: A platform for running click-based middleboxes. Em *2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, páginas 885–886. IEEE.

Martins, J., Ahmed, M., Raiciu, C., Olteanu, V., Honda, M., Bifulco, R. e Huici, F. (2014). Clickos and the art of network function virtualization. Em *USENIX Symposium on Networked Systems Design and Implementation*, páginas 459–473, Seattle, WA. USENIX Association.

Mijumbi, R., Serrat, J., Gorricho, J.-L., Bouten, N., De Turck, F. e Boutaba, R. (2015). Network function virtualization: State-of-the-art and research challenges. *IEEE Communications surveys & tutorials*, 18(1):236–262.

Mijumbi, R., Serrat, J., Gorricho, J.-L., Latre, S., Charalambides, M. e Lopez, D. (2016). Management and orchestration challenges in network functions virtualization. *IEEE Communications Magazine*, 54(1):98–105.

NFVISG, E. (2013). *Network Function Virtualisation (NFV): architectural framework*. European Telecommunications Standards Institute.

OPENSTACK (2016). Tacker. `https://wiki.openstack.org/wiki/Tacker`. Acessado em 27/10/2021.

Peuster, M., Kampmeyer, J. e Karl, H. (2018). Containernet 2.0: A rapid prototyping platform for hybrid service function chains. Em *2018 4th IEEE Conference on Network Softwarization and Workshops (NetSoft)*, páginas 335–337. IEEE.

Peuster, M., Karl, H. e Van Rossem, S. (2016). Medicine: Rapid prototyping of production-ready network services in multi-pop environments. Em *2016 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, páginas 148–153. IEEE.

Quinn, P., Elzur, U. e Pignataro, C. (2018). Network service header (nsh). Relatório técnico.

Rodríguez-Haro, F., Freitag, F., Navarro, L., Hernánchez-sánchez, E., Farías-Mendoza, N., Guerrero-Ibáñez, J. A. e González-Potes, A. (2012). A summary of virtualization techniques. *Procedia Technology*, 3:267–272.

Rosa, R., Siqueira, M., Rothenberg, C. E., Barea, E. e Marcondes, C. (2014). Network function virtualization: Perspectivas, realidades e desafios. *Minicursos SBRC-Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos*.

Sekar, V., Egi, N., Ratnasamy, S., Reiter, M. K. e Shi, G. (2012). Design and implementation of a consolidated middlebox architecture. Em *9th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 12)*, páginas 323–336.

Sherry, J., Hasan, S., Scott, C., Krishnamurthy, A., Ratnasamy, S. e Sekar, V. (2012a). Making middleboxes someone else's problem: Network processing as a cloud service. *ACM SIGCOMM Computer Communication Review*, 42(4):13–24.

Sherry, J., Hasan, S., Scott, C., Krishnamurthy, A., Ratnasamy, S. e Sekar, V. (2012b). Making middleboxes someone else's problem: Network processing as a cloud service. *ACM SIGCOMM Computer Communication Review*, 42(4):13–24.

Tavares, T. N., da Cruz Marcuzzo, L., Garcia, V. F., de Souza, G. V., Franco, M. F., Bondan, L., De Turck, F., Granville, L. Z., Junior, E. P. D., dos Santos, C. R. P. et al. (2018). Niep: Nfv infrastructure emulation platform. Em *2018 IEEE 32nd International Conference on Advanced Information Networking and Applications (AINA)*, páginas 173–180. IEEE.

Venâncio, G. e Duarte Jr, E. P. (2020). Uma arquitetura de alta disponibilidade para funções virtualizadas de rede. Em *Anais do XXXVIII Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos*, páginas 407–420, Porto Alegre, RS, Brasil. SBC.

Venâncio, G., Fulber-Garcia, V., Alchieri, E. A. e Duarte Jr, E. P. (2023). Serviços virtualizados de rede confiáveis: Uma arquitetura para sfcs tolerantes a falhas e intrusão. Em *Anais do XXIV Workshop de Testes e Tolerância a Falhas*, páginas 1–14. SBC.

Venâncio, G., Garcia, V. F., da Cruz Marcuzzo, L., Tavares, T. N., Franco, M. F., Bondan, L., Schaeffer-Filho, A. E., Paula dos Santos, C. R., Granville, L. Z. e P. Duarte Jr, E. (2021). Beyond vnfm: Filling the gaps of the etsi vnf manager to fully support vnf life cycle operations. *International Journal of Network Management*, 31(5):e2068.

Yi, B., Wang, X., Li, K., Huang, M. et al. (2018). A comprehensive survey of network function virtualization. *Computer Networks*, 133:212–262.

APÊNDICE A – IMPLEMENTAÇÕES DAS FUNÇÕES VIRTUALIZADAS DE REDE E GERADORES DE TRÁFEGO MANUAIS

Neste apêndice são apresentados os códigos fontes das VNFs utilizadas nos estudos de caso da plataforma NFV-Prime. A Seção 5.1 apresenta a VNF do *load balancer*, em modo *stateless*. A Seção 5.2, apresenta o código do (*stateful load balancer* e o gerador de tráfego manual utilizado na execução. A Seção 5.3 apresenta o código fonte do (*Deep Packet Inspection* - DPI) e seus geradores de tráfegos manuais, sendo o misto e o isolado. Por fim, a Seção 5.4 apresenta a implementação da VNF do mecanismo *leaky-bucket*.

A.1 STATELESS LOAD BALANCER

```
import threading #thread module imported
from socket import *
import struct
import binascii

def unpackFrameEthernet(frame):
    """Desempacota os dados Ethernet do frame"""
    destiny, origin, protocol = struct.unpack('! 6s 6s H', frame[:14])
    return bytesToHexa(destiny), bytesToHexa(origin), htons(protocol),
        frame[14:]

def bytesToHexa(bytes_address):
    """Transforma bytes to hexadecimal"""
    hex_list = []
    address = binascii.hexlify(bytes_address).decode("ascii")
    for i in range(0,12,2):
        hex_list.append(address[i:i+2])
    mac = ":".join(hex_list)
    return mac

def socketStart(net_interface):
    """Inicializa o socket de rede em modo RAW na interface desejada"""
    Socket = socket(AF_PACKET, SOCK_RAW, htons(3))
    Socket.bind((net_interface, 0))
    return Socket

def ipPacketData(data):
    """Desempacota os dados IP recebidos"""
    ip_data_tuple = struct.unpack("!BBHHHBBH4s4s", data[:20])
```

```
30      version = ip_data_tuple[0]
31      header_len = version >> 4
32      service_type = ip_data_tuple[1]
33      total_size = ip_data_tuple[2]
34      identifier = ip_data_tuple[3]
35      offset_fragment = ip_data_tuple[4]
36      life_time_ttl = ip_data_tuple[5]
37      protocols = ip_data_tuple[6]
38      checksum_header = ip_data_tuple[7]
39      ip_origin = inet_ntoa(ip_data_tuple[8])
40      ip_destiny = inet_ntoa(ip_data_tuple[9])
41
42      header_size_bytes = (version & 15) * 4
43      return ip_origin, ip_destiny, data[header_size_bytes:]
44
45  def saveInfo():
46      """Funcao que salva as informacoes obtidas pelo algoritmo em
47      um arquivo .csv de saida"""
48      global n_dropped, n_transmitted, outputFile, n_dummy1, n_dummy2, n_dummy3
49
50      saida = '{};{};{};{};{}\n'.format(n_transmitted, n_dropped, n_dummy1,
51          n_dummy2, n_dummy3)
52      outputFile.write(saida)
53      outputFile.close()
54      semaphore.release()
55      exit()
56
57  def thread_LoadBalancer():
58      """Thread do LoadBalancer que ao receber um pacote o desempacota,
59      verificando o destino e realizando posteriormente o envio para a
60      melhor opcao de dummy"""
61
62      global clientSocket, semaphore, debug, n_dropped, n_transmitted,
63      n_dummy1, n_dummy2, n_dummy3, n_packet
64
65      while 1:
66          contentReceived = clientSocket.recv(65535)
67          mac_destino, mac_fonte, protocolo, carga_util =
68          unpackFrameEthernet(contentReceived)
69          ipOrigem, ipDestino, dados = ipPacketData(carga_util)
70          if debug:
71              total = n_transmitted + n_dropped
72              if total >= 1000:
73                  exit()
74          if ipDestino == '10.0.1.101':
```

```
75          print ("ipOrigem {} -> ipDestino {}".format(ipOrigem, ipDestino))
76          if (n_packet % 3 == 0):
77             senderSocket_h1.send(contentReceived)
78             n_dummy1 += 1
79          elif (n_packet % 3 == 1):
80             senderSocket_h2.send(contentReceived)
81             n_dummy2 += 1
82          elif (n_packet % 3 == 2):
83             senderSocket_h3.send(contentReceived)
84             n_dummy3 += 1
85          if debug:
86             print("Transmitindo pacote")
87             n_transmitted += 1
88             total = n_transmitted + n_dropped
89             if total >= 1000:
90                saveInfo()
91          n_packet += 1
92       semaphore.release()
93
94 interval = 1
95 debug = 1
96 n_packet = 0
97
98 if debug:
99    n_transmitted = 0
100   n_dropped = 0
101   n_dummy1= 0
102   n_dummy2= 0
103   n_dummy3 = 0
104   outputFile = open('/home/felipe/Desktop/testesMestrado/
105   loadBalancerStateless.csv', 'a')
106
107 clientSocket = socketStart('veth-ch0')
108 senderSocket_h1 = socketStart('veth-h1')
109 senderSocket_h2 = socketStart('veth-h2')
110 senderSocket_h3 = socketStart('veth-h3')
111
112 semaphore = threading.Semaphore(1)
113 load_balancer = threading.Thread(target=thread_LoadBalancer, args=())
114
115 load_balancer.start()
```

A.2 STATEFUL LOAD BALANCER

```
import threading #thread module imported
from socket import *
import struct
import binascii
import time

destinies = {}

def unpackFrameEthernet(frame):
   """Desempacota os dados Ethernet do frame"""
   destiny, origin, protocol = struct.unpack('! 6s 6s H', frame[:14])
   return bytesToHexa(destiny), bytesToHexa(origin), htons(protocol),
   frame[14:]

def bytesToHexa(bytes_address):
   """Transforma bytes to hexadecimal"""
   hex_list = []
   address = binascii.hexlify(bytes_address).decode("ascii")
   for i in range(0,12,2):
      hex_list.append(address[i:i+2])
   mac = ":".join(hex_list)
   return mac

def socketStart(net_interface):
   """Inicializa o socket de rede em modo RAW na interface desejada"""
   Socket = socket(AF_PACKET, SOCK_RAW, htons(3))
   Socket.bind((net_interface, 0))
   return Socket

def ipPacketData(data):
   """Desempacota os dados IP recebidos"""
   ip_data_tuple = struct.unpack("!BBHHHBBH4s4s", data[:20])
   version = ip_data_tuple[0]
   header_len = version >> 4
   service_type = ip_data_tuple[1]
   total_size = ip_data_tuple[2]
   identifier = ip_data_tuple[3]
   offset_fragment = ip_data_tuple[4]
   life_time_ttl = ip_data_tuple[5]
   protocols = ip_data_tuple[6]
   checksum_header = ip_data_tuple[7]
   ip_origin = inet_ntoa(ip_data_tuple[8])
   ip_destiny = inet_ntoa(ip_data_tuple[9])

```

```
45      header_size_bytes = (version & 15) * 4
46      return ip_origin, ip_destiny, data[header_size_bytes:]
47
48  def dados_pacote_udp(carga):
49      """Desempacota os dados UDP recebidos"""
50      tupla_dados_udp = struct.unpack('! H H H H', carga[:8])
51      porta_fonte = tupla_dados_udp[0]
52      porta_destino = tupla_dados_udp[1]
53      udp_len = tupla_dados_udp[2]
54      udp_checksum = tupla_dados_udp[3]
55
56      return porta_fonte, porta_destino, udp_len, udp_checksum, carga[8:]
57
58  def saveInfo():
59      """Funcao que salva as informacoes obtidas pelo algoritmo em
60      um arquivo .csv de saida"""
61
62      global n_dropped, n_transmitted, outputFile, outputFileBucket, n_dummy1,
63      n_dummy2, n_dummy3
64
65      saida = '{};{};{};{};{}\n'.format(n_transmitted, n_dropped, n_dummy1,
66      n_dummy2, n_dummy3)
67      outputFile.write(saida)
68      outputFileBucket.write('{};{};{};Finish;\n'.format(b_dummy1,
69      b_dummy2, b_dummy3))
70      outputFile.close()
71      semaphore.release()
72      outputFileBucket.close()
73      exit()
74
75  def thread_Time(thread_name, interval):
76      """ Thread que analisa a quantidade de bytes recebidos pelas dummies
77      a cada intervalo de tempo"""
78      global semaphore, b_dummy1, b_dummy2, b_dummy3, n_transmitted, n_dropped,
79      outputFileBucket
80
81      while 1: #Ver condicao do while
82          # semaphore.acquire()
83          saida = '{};{};{};\n'.format(b_dummy1, b_dummy2, b_dummy3)
84          outputFileBucket.write(saida)
85          b_dummy1 = 0
86          b_dummy2 = 0
87          b_dummy3 = 0
88          # semaphore.release()
89          time.sleep(interval)
```

```
90
91  def thread_LoadBalancer():
92      """Thread do LoadBalancer que ao receber um pacote o desempacota,
93      verificando o destino e realizando posteriormente o envio para a
94      melhor opcao de dummy"""
95      global clientSocket, semaphore, debug, n_dropped, n_transmitted,
96      n_dummy1, n_dummy2, n_dummy3, n_packet, b_dummy1, b_dummy2, b_dummy3
97
98      n_traffics = 0
99      while 1:
100         contentReceived = clientSocket.recv(65535)
101         mac_destino, mac_fonte, protocolo, carga_util =
102         unpackFrameEthernet(contentReceived)
103         ipOrigem, ipDestino, dados = ipPacketData(carga_util)
104         porta_fonte, porta_destino, udp_len, udp_checksum,
105         carga = dados_pacote_udp(dados)
106         if debug:
107            total = n_transmitted + n_dropped
108            if total >= 3750:
109               exit()
110         if '10.0.1.' in ipDestino:
111            semaphore.acquire()
112            packetSize = len(carga)
113            print ("ipOrigem {} -> ipDestino {}".format(ipOrigem,
114            ipDestino))
115            ip_destiny_port = '{}_{}_{}'.format(ipDestino, porta_destino,
116            porta_fonte)
117            if(destinies.get(ip_destiny_port)):
118               destinies[ip_destiny_port]['socket'].send(contentReceived)
119               if (destinies[ip_destiny_port]['socket'] == senderSocket_h1):
120                  n_dummy1 += 1
121                  b_dummy1 += packetSize
122               elif (destinies[ip_destiny_port]['socket'] == senderSocket_h2):
123                  n_dummy2 += 1
124                  b_dummy2 += packetSize
125               elif (destinies[ip_destiny_port]['socket'] == senderSocket_h3):
126                  n_dummy3 += 1
127                  b_dummy3 += packetSize
128            else:
129               if (n_traffics % 3 == 0):
130                  senderSocket_h1.send(contentReceived)
131                  destiny = {ip_destiny_port: {'socket': senderSocket_h1}}
132                  destinies.update(destiny)
133                  n_dummy1 += 1
134                  b_dummy1 += packetSize
```

```
135             elif (n_traffics % 3 == 1):
136                senderSocket_h2.send(contentReceived)
137                destiny = {ip_destiny_port: {'socket': senderSocket_h2 }}
138                destinies.update(destiny)
139                n_dummy2 += 1
140                b_dummy2 += packetSize
141             elif (n_traffics % 3 == 2):
142                senderSocket_h3.send(contentReceived)
143                destiny = {ip_destiny_port: {'socket': senderSocket_h3}}
144                destinies.update(destiny)
145                n_dummy3 += 1
146                b_dummy3 += packetSize
147             n_traffics += 1
148          if debug:
149             print("Transmitindo pacote")
150             n_transmitted += 1
151             total = n_transmitted + n_dropped
152             if total >= 3750:
153                saveInfo()
154          n_packet += 1
155       semaphore.release()
156
157 interval = 1
158 debug = 1
159 n_packet = 0
160
161 if debug:
162    n_transmitted = 0
163    n_dropped = 0
164    n_dummy1= 0
165    n_dummy2= 0
166    n_dummy3 = 0
167    b_dummy1= 0
168    b_dummy2= 0
169    b_dummy3 = 0
170    outputFile = open('/home/felipe/Desktop/testesMestrado/
171    loadBalancerStatefull.csv', 'a')
172    outputFileBucket = open('/home/felipe/Desktop/testesMestrado/
173    loadBalancerStatefullList.csv', 'a')
174    timer = threading.Thread(target=thread_Time, args=('timer', interval))
175    timer.start()
176
177 clientSocket = socketStart('veth-ch0')
178 senderSocket_h1 = socketStart('veth-h1')
179 senderSocket_h2 = socketStart('veth-h2')
```

```
senderSocket_h3 = socketStart('veth-h3')

semaphore = threading.Semaphore(1)
load_balancer = threading.Thread(target=thread_LoadBalancer, args=())

load_balancer.start()
```

A.2.1 Stateful Load Balancer: gerador de tráfego manual

```
from socket import *
import time
import threading

def thread_Guepard(thread_name, interval):
    """Realiza o envio de 1000 pacotes do trafego guepardo"""
    global guepardMsg, clientSocket, addr_1
    n_packet = 0
    while n_packet < 1000:
        clientSocket.sendto(guepardMsg, addr_1)
        n_packet += 1
        time.sleep(interval)

def thread_Elephant(thread_name, interval):
    """Realiza o envio de 1000 pacotes do trafego elefante"""
    global elephantMsg, clientSocket, addr_2
    n_packet = 0
    while n_packet < 1000:
        clientSocket.sendto(elephantMsg, addr_2)
        n_packet += 1
        time.sleep(interval)

def thread_Middle(thread_name, interval, msg):
    """Realiza o envio de 1000 pacotes do trafego middle"""
    global middleMsg, clientSocket, addr_3
    n_packet = 0
    while n_packet < 1000:
        clientSocket.sendto(msg, addr_3)
        n_packet += 1
        time.sleep(interval)

guepardMsg = b'Eu sou o guepardo'
guepardMsg += b"." * (16 - len(guepardMsg))
elephantMsg = b'Eu sou o elefante'
elephantMsg += b"." * (1024 - len(elephantMsg))
```

```
middleMsg = b'Eu sou o middle'
middleMsg += b"." * (256 - len(middleMsg))

addr_1 = ("10.0.1.104", 8001)
addr_2 = ("10.0.1.104", 8002)
addr_3 = ("10.0.1.104", 8003)

clientSocket = socket(family=AF_INET, type=SOCK_DGRAM)
clientSocket.setsockopt(SOL_SOCKET, SO_REUSEADDR, 1)
clientSocket.bind(('10.2.2.100', 8001))

threadGuepard = threading.Thread(target=thread_Guepard,
args=('guepard', 0.01))
threadElephant = threading.Thread(target=thread_Elephant,
args=('elephant', 0.04))
threadMiddle = threading.Thread(target=thread_Middle,
args=('middle', 0.0166, middleMsg))

threadGuepard.start()
threadElephant.start()
threadMiddle.start()
```

A.3 DEEP PACKET INSPECTION - DPI

```
import threading #thread module imported
from socket import *
import struct
import binascii
import re
import time

destinies = {}

def unpackFrameEthernet(frame):
    """Desempacota os dados Ethernet do frame"""
    destiny, origin, protocol = struct.unpack('! 6s 6s H', frame[:14])
    return bytesToHexa(destiny), bytesToHexa(origin), htons(protocol),
    frame[14:]

def bytesToHexa(bytes_address):
    """Transforma bytes to hexadecimal"""
    hex_list = []
    address = binascii.hexlify(bytes_address).decode("ascii")
    for i in range(0,12,2):
```

```
21        hex_list.append(address[i:i+2])
22    mac = ":".join(hex_list)
23    return mac
24
25 def socketStart(net_interface):
26    """Inicializa o socket de rede em modo RAW na interface desejada"""
27    Socket = socket(AF_PACKET, SOCK_RAW, htons(3))
28    Socket.bind((net_interface, 0))
29    return Socket
30
31 def ipPacketData(data):
32    """Desempacota os dados IP recebidos"""
33    ip_data_tuple = struct.unpack("!BBHHHBBH4s4s", data[:20])
34    version = ip_data_tuple[0]
35    header_len = version >> 4
36    service_type = ip_data_tuple[1]
37    total_size = ip_data_tuple[2]
38    identifier = ip_data_tuple[3]
39    offset_fragment = ip_data_tuple[4]
40    life_time_ttl = ip_data_tuple[5]
41    protocols = ip_data_tuple[6]
42    checksum_header = ip_data_tuple[7]
43    ip_origin = inet_ntoa(ip_data_tuple[8])
44    ip_destiny = inet_ntoa(ip_data_tuple[9])
45
46    header_size_bytes = (version & 15) * 4
47    return ip_origin, ip_destiny, data[header_size_bytes:]
48
49 def dados_pacote_udp(carga):
50    """Desempacota os dados UDP recebidos"""
51    tupla_dados_udp = struct.unpack('! H H H H', carga[:8])
52    porta_fonte = tupla_dados_udp[0]
53    porta_destino = tupla_dados_udp[1]
54    udp_len = tupla_dados_udp[2]
55    udp_checksum = tupla_dados_udp[3]
56
57    return porta_fonte, porta_destino, udp_len, udp_checksum,
58    carga[8:]
59
60 def saveInfo():
61    """Funcao que salva as informacoes obtidas pelo algoritmo em
62    um arquivo .csv de saida"""
63    global n_dropped, n_transmitted, outputFile, outputFileTimes
64
65    saida = '{};{};\n'.format(n_transmitted, n_dropped)
```

```
66      outputFile.write(saida)
67      outputFile.close()
68      outputFileTimes.write('\n')
69      outputFileTimes.close()
70      semaphore.release()
71      exit()
72
73  def thread_DPI():
74      """ Thread do DPI que ao receber um pacote o desempacota,
75      verificando seu destino, realizando a inspecao e
76      posteriormente a acao"""
77      global clientSocket, semaphore, debug, n_dropped, n_transmitted,
78      n_packet, outputFileTimes
79
80      while 1:
81          contentReceived = clientSocket.recv(65535)
82          mac_destino, mac_fonte, protocolo, carga_util =
83          unpackFrameEthernet(contentReceived)
84          ipOrigem, ipDestino, dados = ipPacketData(carga_util)
85          porta_fonte, porta_destino, udp_len, udp_checksum, carga =
86          dados_pacote_udp(dados)
87          if debug:
88              total = n_transmitted + n_dropped
89              if total >= 1000:
90                  exit()
91          if '10.0.1.101' in ipDestino:
92              stime = time.time()
93              patternFind = re.search(".*Estou infectado.*",
94              carga.decode("utf-8"))
95              if (not patternFind):
96                  senderSocket_h1.send(contentReceived)
97                  if debug:
98                      print("Transmitindo pacote")
99                      n_transmitted += 1
100                     etime = time.time()
101                     saida = '{};'.format(etime - stime)
102                     outputFileTimes.write(saida)
103                     total = n_transmitted + n_dropped
104                     if total >= 1000:
105                         saveInfo()
106             else:
107                 if debug:
108                     print("Pacote infectado, descartando")
109                     etime = time.time()
110                     saida = '{};'.format(etime - stime)
```

```
111                 outputFileTimes.write(saida)
112                 n_dropped += 1
113                 total = n_transmitted + n_dropped
114                 if total >= 1000:
115                     saveInfo()
116             n_packet += 1
117         semaphore.release()
118 
119 interval = 1
120 debug = 1
121 n_packet = 0
122 
123 if debug:
124     n_transmitted = 0
125     n_dropped = 0
126     outputFile = open('/home/felipe/Desktop/testesMestrado/dpi.csv',
127     'a')
128     outputFileTimes = open('/home/felipe/Desktop/testesMestrado/
129     dpiTimes.csv', 'a')
130 
131 
132 clientSocket = socketStart('veth-ch0')
133 senderSocket_h1 = socketStart('veth-h1')
134 
135 semaphore = threading.Semaphore(1)
136 dpi = threading.Thread(target=thread_DPI, args=())
137 
138 dpi.start()
```

A.3.1 Deep Packet Inspection: gerador de tráfego manual - misto

```
1  from socket import *
2  import time
3  import threading
4  import random
5  random.seed(10)
6  
7  def thread_Traffic(thread_name, interval, msg, probability):
8      """Realiza o envio de 1000 pacotes, podendo enviar
9      pacotes infectados"""
10     global clientSocket, addr_1, startMessage, middleMsg, endMsg
11     n_packet = 0
12     while n_packet < 1000:
13         number = random.randint(0, 100)
```

```
14        if number >= probability:
15            clientSocket.sendto(msg, addr_1)
16        else:
17            if (number % 3) == 0:
18                clientSocket.sendto(startMessage, addr_1)
19            elif (number % 3) == 1:
20                clientSocket.sendto(middleMsg, addr_1)
21            elif (number % 3) == 2:
22                clientSocket.sendto(endMsg, addr_1)
23        n_packet += 1
24        time.sleep(interval)
25
26  message = b'Estou limpo' + (b'.' * 1013)
27  startMessage = b'Estou infectado'
28  startMessage += b"." * (1024 - len(startMessage))
29  middleMsg = b"." * 512
30  middleMsg += b"Estou infectado" + b"." * (512 - len('Estou infectado'))
31  endMsg = b"." * (1024-len('Estou infectado'))
32  endMsg += b"Estou infectado"
33
34  addr_1 = ("10.0.1.101", 8001)
35
36  clientSocket = socket(family=AF_INET, type=SOCK_DGRAM)
37  clientSocket.setsockopt(SOL_SOCKET, SO_REUSEADDR, 1)
38  clientSocket.bind(('10.2.2.100', 8001))
39
40  threadTraffic = threading.Thread(target=thread_Traffic, args=
41  ('traffic_sender', 0.04, message, 10))
42
43  threadTraffic.start()
```

A.3.2 Deep Packet Inspection: gerador de tráfego manual - isolado

```
1   from socket import *
2   import time
3   import threading
4
5   def thread_Traffic(thread_name, interval, msg):
6       """Realiza o envio de 1000 pacotes do trafego"""
7       global clientSocket, addr_1
8       n_packet = 0
9       while n_packet < 1000:
10          clientSocket.sendto(msg, addr_1)
11          n_packet += 1
```

```
      time.sleep(interval)

startMessage = b'Estou infectado'
startMessage += b"." * (1024 - len(startMessage))
middleMsg = b"." * 512
middleMsg += b"Estou infectado" + (b"." * (1024 - len('Estou infectado')
- 512))
endMsg = b"." * 1009
endMsg += b"Estou infectado"

addr_1 = ("10.0.1.101", 8001)

clientSocket = socket(family=AF_INET, type=SOCK_DGRAM)
clientSocket.setsockopt(SOL_SOCKET, SO_REUSEADDR, 1)
clientSocket.bind(('10.2.2.100', 8001))

threadTraffic = threading.Thread(target=thread_Traffic, args=
('traffic_sender', 0.04, middleMsg))

threadTraffic.start()
```

A.4 LEAKY BUCKET

```
import threading #thread module imported
import time #time module
from socket import *
import struct
import binascii

def unpackFrameEthernet(frame):
   """Desempacota os dados Ethernet do frame"""
   destiny, origin, protocol = struct.unpack('! 6s 6s H', frame[:14])
   return bytesToHexa(destiny), bytesToHexa(origin),
   htons(protocol), frame[14:]

def bytesToHexa(bytes_address):
   """Transforma bytes to hexadecimal"""
   hex_list = []
   address = binascii.hexlify(bytes_address).decode("ascii")
   for i in range(0,12,2):
      hex_list.append(address[i:i+2])
   mac = ":".join(hex_list)
   return mac

```

```
22 def socketStart(net_interface):
23     """Inicializa o socket de rede em modo RAW na interface desejada"""
24     Socket = socket(AF_PACKET, SOCK_RAW, htons(3))
25     Socket.bind((net_interface, 0))
26     return Socket
27 
28 def ipPacketData(data):
29     """Desempacota os dados IP recebidos"""
30     ip_data_tuple = struct.unpack("!BBHHHBBH4s4s", data[:20])
31     version = ip_data_tuple[0]
32     header_len = version >> 4
33     service_type = ip_data_tuple[1]
34     total_size = ip_data_tuple[2]
35     identifier = ip_data_tuple[3]
36     offset_fragment = ip_data_tuple[4]
37     life_time_ttl = ip_data_tuple[5]
38     protocols = ip_data_tuple[6]
39     checksum_header = ip_data_tuple[7]
40     ip_origin = inet_ntoa(ip_data_tuple[8])
41     ip_destiny = inet_ntoa(ip_data_tuple[9])
42 
43     header_size_bytes = (version & 15) * 4
44     return ip_origin, ip_destiny, data[header_size_bytes:]
45 
46 def analyzeBucket():
47     """Funcao que analisa o bucket, obtendo a quantidade de pacotes
48     de cada tipo presente"""
49     global bucket, outputFileBucket
50 
51     b_guepard = 0
52     b_elephant = 0
53     b_middle = 0
54     for packet in bucket:
55         if packet[1] <= 100:
56             b_guepard += 1
57         elif packet[1] >= 1000:
58             b_elephant += 1
59         elif packet[1] > 100 and packet[1] < 1000:
60             b_middle += 1
61     saida = '{};{};{};\n'.format(b_guepard, b_elephant, b_middle)
62     outputFileBucket.write(saida)
63 
64 def consumeBucket():
65     """Funcao que consome o bucket de acordo com a quantidade
66     de pacotes que sao consumidos a cada intervao de tempo"""
```

```
67
68      global bucket, packets_to_release, clientSocket, debug,
69      n_transmitted, n_guepard, n_elephant, n_middle
70      if (len(bucket) == 0):
71          return
72      while (len(bucket) > 0) and (packets_to_release > 0):
73          semaphore.acquire()
74          contentReceived = bucket.pop(0)
75          senderSocket_h1.send(contentReceived[0])
76          print(contentReceived)
77          if debug:
78              print("Transmitindo pacote")
79              if contentReceived[1] <= 100:
80                  n_guepard += 1
81              elif contentReceived[1] >= 1000:
82                  n_elephant += 1
83              elif contentReceived[1] > 100 and contentReceived[1] < 1000:
84                  n_middle += 1
85              n_transmitted += 1
86              total = n_transmitted + n_dropped
87              if total >= 3000:
88                  saveInfos()
89          packets_to_release -= 1
90          semaphore.release()
91
92
93  def thread_Time(thread_name, interval):
94      """ Thread que reseta o consumo do bucket a cada intervalo
95      de tempo
96      interval -> intervalo de tempo para que sejam adicionados os tokens"""
97      global semaphore, packets_to_release, packets_to_release_value
98      while 1: #Ver condicao do while
99          semaphore.acquire()
100         packets_to_release = packets_to_release_value
101         semaphore.release()
102         analyzeBucket()
103         consumeBucket()
104         if debug:
105             total = n_transmitted + n_dropped
106             if total >= 3000:
107                 exit()
108
109         time.sleep(interval)
110
111 def saveInfos():
```

```
112     """Funcao que salva as informacoes obtidas pelo algoritmo em um
113     arquivo .csv de saida"""
114     global n_dropped, n_transmitted, outputFile, n_guepard,
115     n_elephant, n_middle
116
117     saida = '{};{};{};{};{};\n'.format(n_transmitted, n_dropped, n_guepard,
118     n_elephant, n_middle)
119     outputFile.write(saida)
120     outputFile.close()
121     outputFileBucket.close()
122     semaphore.release()
123     exit()
124
125 def thread_LeakyBucket():
126     """ Thread do LeakyBucket que ao receber um pacote enfileira,
127     transmite ou descarta o pacote, de acordo com seus parametros"""
128     global clientSocket, packets_to_release, bucket, semaphore,
129     bucket_max_size, debug, n_dropped, n_transmitted, n_guepard,
130     n_elephant, n_middle
131
132     while 1:
133         contentReceived = clientSocket.recv(65535)
134         mac_destino, mac_fonte, protocolo, carga_util =
135         unpackFrameEthernet(contentReceived)
136         ipOrigem, ipDestino, dados = ipPacketData(carga_util)
137         if debug:
138             total = n_transmitted + n_dropped
139             if total >= 3000:
140                 exit()
141         if len(bucket) > 0:
142             semaphore.acquire()
143             if len(bucket) < bucket_max_size:
144                 if ipDestino == '10.0.1.101':
145                     if debug:
146                         print("Adicionou na fila e bucket nao vazio")
147                     bucket.append([contentReceived, len(dados)])
148             else:
149                 if debug:
150                     print("Mensagem dropada")
151                     n_dropped += 1
152                     total = n_transmitted + n_dropped
153                     if total >= 3000:
154                         saveInfos()
155         else:
156             semaphore.acquire()
```

```
157             if packets_to_release > 0:
158                 if ipDestino == '10.0.1.101':
159                     print ("ipOrigem {} -> ipDestino {}".format(ipOrigem,
160                     ipDestino))
161                     senderSocket_h1.send(contentReceived)
162                     if debug:
163                         print("Transmitindo pacote")
164                         if len(dados) <= 100:
165                             n_guepard += 1
166                         elif len(dados) >= 1000:
167                             n_elephant += 1
168                         elif len(dados) > 100 and len(dados) < 1000:
169                             n_middle += 1
170 
171                         n_transmitted += 1
172                         total = n_transmitted + n_dropped
173                         if total >= 3000:
174                             saveInfos()
175                     packets_to_release -= 1
176             else:
177                 if ipDestino == '10.0.1.101':
178                     if debug:
179                         print("Adicionou na fila e bucket vazio")
180                     bucket.append([contentReceived, len(dados)])
181         semaphore.release()
182 
183 bucket = []
184 
185 packets_to_release = 50
186 bucket_max_size = 100
187 interval = 1
188 debug = 1
189 
190 packets_to_release_value = packets_to_release
191 
192 if debug:
193     n_transmitted = 0
194     n_dropped = 0
195     n_guepard= 0
196     n_elephant= 0
197     n_middle = 0
198     outputFile = open('/home/felipe/Desktop/testesMestrado
199     /leakybucket-{}.csv'.format(packets_to_release_value), 'w')
200     outputFileBucket = open('/home/felipe/Desktop/testesMestrado
201     /leakybucket-{}-bucket.csv'.format(packets_to_release_value), 'w')
```

```
202
203 clientSocket = socketStart('veth-ch0')
204 senderSocket_h1 = socketStart('veth-h1')
205
206 semaphore = threading.Semaphore(1)
207 timer = threading.Thread(target=thread_Time, args=('timer', interval))
208 leaky_bucket = threading.Thread(target=thread_LeakyBucket, args=())
209
210 timer.start()
211 leaky_bucket.start()
```