Renan Freire Tavares

sfcFC: Uma abordagem de implementação de SFC em ambientes de nuvem com funções de rede utilizando Fastclick

Vitória, ES

2019

Renan Freire Tavares

sfcFC: Uma abordagem de implementação de SFC em ambientes de nuvem com funções de rede utilizando Fastclick

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Informática da Universidade Federal do Espírito Santo, como requisito parcial para obtenção do Grau de Mestre em Informática.

Universidade Federal do Espírito Santo – UFES

Centro Tecnológico

Programa de Pós-Graduação em Informática

Orientador: Prof. Dr. Maxwell Eduardo Monteiro

Coorientador: Prof. Dr. Rodolfo da Silva Villaça

Vitória, ES

2019

Renan Freire Tavares
sfcFC: Uma abordagem de implementação de SFC em ambientes de nuvem com funções de rede utilizando Fastclick/ Renan Freire Tavares. – Vitória, ES, 2019-
93 p. : il. (algumas color.) ; 30 cm.

Orientador: Prof. Dr. Maxwell Eduardo Monteiro

Dissertação de Mestrado – Universidade Federal do Espírito Santo – UFES
Centro Tecnológico
Programa de Pós-Graduação em Informática, 2019.

1. SFC. 2. Fastclick. 3. NFV. 4. Computação em Nuvem. I. Tavares, Renan Freire. II. Universidade Federal do Espírito Santo. IV. sfcFC: Uma abordagem de implementação de SFC em ambientes de nuvem com funções de rede utilizando Fastclick

CDU 02:141:005.7

Renan Freire Tavares

sfcFC: Uma abordagem de implementação de SFC em ambientes de nuvem com funções de rede utilizando Fastclick

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Informática da Universidade Federal do Espírito Santo, como requisito parcial para obtenção do Grau de Mestre em Informática.

Trabalho aprovado.

Prof. Dr. Maxwell Eduardo Monteiro
Orientador

Prof. Dr. Rodolfo da Silva Villaça
Coorientador

Prof. Dr. Vinícius Fernandes Soares Mota
Membro Interno

Prof. Dr. Gilmar Luiz Vassoler
Membro Externo

Vitória, ES
2019

Dedico este trabalho aos que acreditam na ciência e na sua capacidade de transformar o mundo num lugar melhor

Agradecimentos

Aos meus pais, Aquiles e Iêda, por todo o suporte e à toda a minha família pelo apoio.

Aos meus amigos e colegas de trabalho pela compreensão, principalmente nestes últimos meses.

Aos companheiros do LabNerds pelo apoio nesta caminhada.

Aos meus orientadores Maxwell e Rodolfo pelo amparo durante a pesquisa, sempre no incentivo para que pudesse obter o meu melhor.

À banca composta pelos professores Vinícius e Gilmar pela colaboração.

À Fundação de Amparo à Pesquisa e Inovação do Espírito Santo (FAPES) e ao projeto GT-NOSFVERATO pela viabilização dos estudos através de bolsa.

“It’s louder than words
This thing that we do
Louder than words
The way it unfurls
It’s louder than words
The sum of our parts
The beat of our hearts
Is louder than words“
(David Gilmour, Polly Samson)

Resumo

Ao encadeamento de funções de rede, também denominado de *Service Function Chaining* (SFC), dá-se a responsabilidade de direcionar o tráfego de rede a fim de cumprir os mais variados requisitos antes que os dados sejam entregues ao destinatário. Dado o crescente aumento na demanda de recursos computacionais, novos paradigmas foram criados a fim de se adaptar a essa nova realidade. Com base na Computação em Nuvem a abordagem *Network Function Vitualization* (NFV) permitiu que as funções de rede, outrora implementadas nos dispositivos de hardware, fossem implementadas de forma virtual. Um exemplo de ferramenta que entrega tais funções de forma simples é o Fastclick. Este, por sua vez, é uma derivação da linguagem Click com importantes ajustes de performance. SFC anda em conjunto com NFV a fim de atender as novas exigências da Computação. Deste modo, o modelo anterior de baixa capacidade de modificação e bastante dependente de fabricantes de hardware, foi substituído por este mais flexível e escalável. O tema vem sendo constantemente estudado pela academia e tratado de forma especial pela indústria. Entretanto, atualmente ainda há uma carência de ferramentas que cumpram a tarefa de implementar e disponibilizar essas funções forma simples. Neste intuito, este trabalho visa entregar o SFC em ambientes de nuvem aproveitando-se da simplicidade de construção de *Virtual Network Functions* (VNFs) do Fastclick. O resultado obtido foi a elaboração do sfcFC, uma abordagem independente de plataforma que, baseada no protocolo que dá suporte ao SFC, o *Network Service Header* (NSH), entrega o SFC em ambientes de redes virtuais de forma programável e aberta.

Palavras-chaves: SFC, Fastclick, NFV e Computação em Nuvem.

Abstract

Service Function Chaining (SFC) is responsible for steering network traffic in order to meet the most varied SLAs before data is delivered to the nd client. With the increasing demand for computational resources, new paradigms were created to adapt to this new reality. Once implemented on hardware devices, the Network Function Vitualization (NFV) approach has allowed network functions to be implemented virtually with the support of Cloud Computing. An example of a tool that simply delivers such functions is Fastclick. This one is a derivation of the Click language with important performance improvements. SFC goes hand in hand with NFV to meet new computing requirements. Thus, the ossified earlier model that has low modifiability and is largely dependent on hardware manufacturers has been replaced by the more flexible and scalable one. The SFC has been constantly studied by the academy and treated in a special way by the industry. However, currently there is still a lack of tools that fulfill the task of implementing and making these functions simple. This paper aims to deliver SFC in cloud environments by taking advantage of the simplicity of building Fastclick Virtual Network Functions (VNFs). The result was the development of sfcFC, a platform-independent approach. The sfcFC is based on the protocol that supports SFC, Network Service Header (NSH), and delivers SFC in virtual network environments in a programmable and open source manner.

Keywords: SFC, Fastclick, NFV and Cloud Computing.

Lista de ilustrações

Figura 1 – Arquitetura NFV (Adaptado de (ETSI, 2013)) 29
Figura 2 – Fluxograma dos elementos Click em um roteador (Adaptado de (KOHLER et al., 2000)) 32
Figura 3 – Topologia usada como exemplo do `sfcFC` 42
Figura 4 – Encapsulamento do cabeçalho `sfcFC` ao pacote IP 42
Figura 5 – Diagrama de Atividades correspondente ao `sfcFC-Classifier` 43
Figura 6 – Diagrama de Atividades correspondente ao SFF 43
Figura 7 – Diagrama de Atividades correspondente aos SFs 43
Figura 8 – Representação do Primeiro SFP 44
Figura 9 – Representação do Segundo SFP 45
Figura 10 – Representação do Terceiro SFP 46
Figura 11 – Topologia no Openstack 51
Figura 12 – SFP 1 - Exemplo de requisição capturada pelo Wireshark na interface do `sfcFC-Classifier` (1) com a rede SFC 51
Figura 13 – SFP 1 - Exemplo de requisição, capturada pelo Wireshark, passando pela interface do `Forwarder` (5) 53
Figura 14 – SFP 1 - Requisição passando pelo *firewall* (4) capturada pelo Wireshark 54
Figura 15 – SFP 1 - Pacotes capturados pelo balanceador *Round Robin* (3) capturados pelo Wireshark 54
Figura 16 – SFP 2 - Requisições capturadas pelo Wireshark na interface do `sfcFC-Classifier` (1) com a rede SFC 55
Figura 17 – SFP 2 - Exemplo de requisição capturada pelo Wireshark, passando pela interface do `forwarder` (5) 56
Figura 18 – SFP 2 - Requisição passando pelo *firewall* (4) capturada pelo Wireshark 56
Figura 19 – SFP 2 - Requisição passando pelo balanceador *Source* IP (2) capturada pelo Wireshark 56
Figura 20 – SFP 3 - Requisições capturadas pelo Wireshark na interface do `sfcFC-Classifier` (1) com a rede SFC 57
Figura 21 – SFP 3 - Exemplo de requisição, capturada pelo Wireshark, passando pela interface do `forwarder` (5) 58
Figura 22 – SFP 3 - Requisição passando pelo *firewall* (4) capturada pelo Wireshark 58
Figura 23 – Requisições do Cliente para o SFP 4 capturadas pelo Wireshark negadas pelo *firewall* (4) 59
Figura 24 – SFP 4 - Início da requisição capturada pelo Wireshark na interface do `sfcFC-Classifier` (1) com a rede SFC 59

Figura 25 – SFP 4 - Final da requisição capturada pelo Wireshark na interface do `sfcFC-Classifier` (1) com a rede SFC 60
Figura 26 – SFP 4 - Exemplo de início de uma requisição, capturada pelo Wireshark, passando pela interface do `forwarder` (5) 60
Figura 27 – SFP 4 - Início da requisição passando pelo *firewall* (4) capturada pelo Wireshark 61
Figura 28 – Taxa de perda(%) de pacotes na abordagem com o uso de SFC 62
Figura 29 – Taxa de perda(%) de pacotes na abordagem sem o uso de SFC 62
Figura 30 – Comparativo da Taxa de Perda(%) de pacotes nos ambientes com e sem uso do SFC 63

Lista de tabelas

Tabela 1 – Principais componentes do Openstack (OPENSTACK, 2019) 28
Tabela 2 – Principais elementos do Click (KOHLER et al., 2000) 31
Tabela 3 – Tabela de IPs utilizados no Openstack durante os Testes com base na Figura 11 50

Lista de abreviaturas e siglas

SFC	Service Function Chaining
SLA	Service Level Agreement
NFV	Network Function Vitualization
SDN	Software Defined Network
VNF	Virtual Network Funtion
NSH	Network Service Header
QoS	Quality of Service
ETSI	European Telecommunications Standard Institute
IP	Internet Protocol
DPDK	Data Plane Development Kit
VIM	Virtualized Infrastructure Manager
IETF	Internet Research Task Force
RFC	Request for Comments
API	Application Programming Interface
CAPEX	Capital Expenditure
OPEX	Operational Expenditure
NFVI	Network Function Virtualization Infrastructure
MANO	Management and Orchestration
ARP	Address Resolution Protocol
SFF	Service Function Forwarder
SF	Service Function
SPI	Service Path Identifier
SI	Service Index
ONF	Open Networking Foundation
IaaS	Infrastructure as a Service

Sumário

1 INTRODUÇÃO 23
1.1 Objetivos 24
1.2 Estrutura do Texto 24

2 REVISÃO DA LITERATURA 27
2.1 Computação em Nuvem 27
2.2 *Network Function Virtualization* 28
2.2.1 Click 30
2.2.2 DPDK 33
2.2.3 Fastclick 33
2.3 *Service Function Chaining* 34

3 TRABALHOS RELACIONADOS 37
3.1 *Compact* SFC *Header* 37
3.2 Neo-NSH 38
3.3 PhantomSFC 38
3.4 Neutron SFC 39

4 PROPOSTA DO `SFCFC` 41
4.1 Projeto da Arquitetura 41
4.1.1 `sfcFC-Classifier` 42
4.1.2 `Forwarder` 43
4.1.3 *Service Functions* 43
4.2 Elaboração da Execução do `sfcFC` 44

5 AVALIAÇÃO DO `SFCFC` 49
5.1 Teste para o primeiro SFP 49
5.2 Teste para o segundo SFP 55
5.3 Teste para o terceiro SFP 57
5.4 Teste para o quarto SFP 58
5.5 Avaliação de Desempenho do `sfcFC` 61
5.5.1 Teste comparativo avaliando a perda de pacotes 62

6 CONCLUSÃO 65
6.1 Considerações Finais 65
6.2 Trabalhos Futuros 66

REFERÊNCIAS 67

APÊNDICES 69

APÊNDICE A – PASSO-A-PASSO DA INSTALAÇÃO DO FASTCLICK E DPDK NO OPENSTACK 71

APÊNDICE B – CÓDIGO CLICK DO SFCFC-CLASSIFIER 75

APÊNDICE C – CÓDIGO CLICK DO FORWARDER 79

APÊNDICE D – CÓDIGO CLICK DO FIREWALL 83

APÊNDICE E – CÓDIGO CLICK DO BALANCEADOR *SOURCE* IP 87

APÊNDICE F – CÓDIGO CLICK DO BALANCEADOR *ROUND ROBIN* 91

1 Introdução

Nos últimos anos, devido ao aumento exponencial do consumo, e consequentemente, da demanda de recursos computacionais, o setor de redes de computadores vem passando por grandes mudanças, tendo como principal foco a Computação em Nuvem. As novas exigências modificaram as redes nos *Data Centers*, levaram ao aumento no número de dispositivos e na forte dependência entre os mesmos e seus fabricantes. Entretanto, a introdução de novas funcionalidades em uma rede baseada em equipamentos físicos se mostrou complicada, demorada e cara. Além disto, visto que os serviços atuais dependem de uma cadeia de funcionalidades de rede, a reutilização destas funções associadas à dispositivos físicos e seus fabricantes acabou se tornando uma tarefa ainda mais difícil (JOHN et al., 2013).

A área de redes encontrou nas abordagens *Software Defined Network*(SDN) e *Network Function Virtualization*(NFV) a possibilidade de flexibilizar e escalar recursos garantindo a qualidade, realizando o que antes era custoso de forma simples e robusta. Desta forma, a interconexão de serviços, também conhecida como *Service Function Chaining*(SFC), fez uso destas tecnologias a fim de prover um gerenciamento flexível do tráfego de rede, fornecendo soluções para classificá-los e, ao mesmo tempo, aplicando políticas adequadas de acordo com os requisitos de serviço (MEDHAT et al., 2016).

A ideia principal do NFV é o desacoplamento entre as funções de rede e os dispositivos físicos. Isto significa que uma função de rede, como um *firewall*, pode ser executada como uma instância de software simples em qualquer plataforma de hardware. Desta forma, um determinado serviço pode ser decomposto em um conjunto de funções de rede (VNFs), que podem ser implementadas em um ou mais servidores físicos. Estas podem, então, ser realocadas e instanciadas em diferentes locais da rede sem necessariamente exigir a compra e instalação de novo hardware (MIJUMBI et al., 2015). Neste intuito, Fastclick (BARBETTE; SOLDANI; MATHY, 2015) permite a construção de VNFs em arquiteturas x86, fazendo uso do Click (KOHLER et al., 2000). Por sua vez, Click utiliza-se de uma composição de elementos - estrutura mínima de sua arquitetura - para realizar ações simples no âmbito de protocolos de rede, tais como decrementar o TTL de um pacote IP, por exemplo (BARBETTE; SOLDANI; MATHY, 2015). Deste modo, Fastclick se mostra como uma das tecnologias que torna factível a proposta NFV de desacoplar as funções de rede do hardware. E uma vez que há cerca de quinhentos elementos Click já disponíveis para a construção de roteadores, a tarefa do programador de redes é de certa forma facilitada.

Amparado pela IETF e suas RFCs 7665 (HALPERN; PIGNATARO, 2015) e

8300 (QUINN; ELZUR; PIGNATARO, 2018), o SFC é outra tecnologia motivadora para implementação de NFV. Isto porque o SFC possibilita a construção de múltiplas cadeias de VNFs, permitindo que os acordos de serviço sejam atendidos de acordo com a demanda do usuário. Sendo assim, é necessário que se estude a teoria sob uma perspectiva prática. O tema ainda carece de estudo no que tange ao encadeamento e a alocação dinâmica de recursos, bem como a prestação de serviços de ponta a ponta, dentre outros. O encadeamento dinâmico, um dos pontos que necessitam de atenção dentro do tema, é um aspecto de suma importância para o modelo SFC. Dado que o caminho seguido pela solicitação de um usuário pode ser diferente dependendo da aplicação requisitada, a rede precisa estar devidamente programada para atender esta demanda. Além disto, as decisões de encaminhamento dos elementos do SFC devem ser eficientes. Embora uma quantidade significativa de pesquisa tenha sido realizada, os principais problemas, tais como: (i) o encadeamento dinâmico, (ii) a ordenação de funções numa cadeia; (iii) ferramentas que tomem em conta acordos a nível de serviço (SLA) e a Qualidade do Serviço (QoS), (iv) que sejam resilientes, (v) que consumam menos energia elétrica, (vi) seguras, (vii) distribuíveis e, (viii) que visem redes móveis e sem fio ainda estão em aberto da perspectiva do SFC (BHAMARE et al., 2016; MIRJALILY; ZHIQUAN, 2018).

Com intuito de resolver algumas destas carências técnicas que a abordagem SFC ainda possui, este trabalho pretende construir uma aplicação que entregue o SFC em ambientes de nuvem com o apoio do Fastclick. Construindo um cabeçalho baseado no *Network Service Header* (NSH), o `sfcFC`, nome dado ao trabalho, é uma abordagem distribuível e independente de plataforma de nuvem que entrega o SFC em ambientes virtuais de Rede.

1.1 Objetivos

O objetivo geral deste trabalho é implementar e avaliar uma solução de SFC para ambientes de nuvem de forma programável e aberta. Este objetivo é dividido entre os seguintes objetivos específicos:

- Construir elementos do SFC com o Fastclick;
- Desenvolver um método que permita o encadeamento dinâmico das VNFs;
- Montar cenários de teste que permitam a validação da proposta;

1.2 Estrutura do Texto

A partir deste, o trabalho possui a seguinte organização:

- O Capítulo 2 aborda os temas estruturais para a construção do trabalho;
- O Capítulo 3 menciona os trabalhos relacionados e posiciona o trabalho na literatura da área;
- O Capítulo 4 apresenta a proposta do `sfcFC`;
- O Capítulo 5 expõe os resultados obtidos com a avaliação da implementação do `sfcFC`;
- O Capítulo 6 relata a análise do trabalho como um todo, pontuando todos os possíveis tópicos abordados e trabalhos futuros.

2 Revisão da Literatura

Este capítulo introduz os conceitos que embasam o desenvolvimento deste trabalho, desde a definição de Computação em Nuvem, base para os temas de *Network Function Chaining* (NFV) e *Service Function Chaining* (SFC), Openstack, *Data Plane Development Kit* (DPDK) e Fastclick, que são tecnologias utilizadas no desenvolvimento deste trabalho.

2.1 Computação em Nuvem

A necessidade de escalar recursos dado o aumento da demanda computacional encontrou na computação em nuvem a sua saída. Isso porque este paradigma oferece algumas qualidades descritas em Mell, Grance et al. (2011), tais como: provisionamento de serviços sob demanda, acesso distribuído e multiplataforma, associação dinâmica e elasticidade de recursos. Há de se acrescentar também o aumento na capacidade de controle destes recursos, que estão sempre sendo devidamente monitorados nativamente pelas plataformas de nuvem.

O Openstack(OPENSTACK, 2019) atua no contexto de computação em nuvem provendo uma Infraestrutura como Serviço (IaaS). Mell, Grance et al. (2011) identifica como principal atributo deste modelo a disponibilização de recursos de hardware e software de forma completa ao usuário. Desta forma, pode-se controlar, por exemplo: o processamento, memória, rede, armazenamento e, também, o sistema operacional das instâncias de computação. Já no contexto de NFV, dentro da arquitetura proposta pela ETSI, o Openstack atua como um VIM.

Este sistema operacional em nuvem se caracteriza pela modularização de componentes. Isto é possível dado a utilização de APIs e protocolos de comunicação por mensagem, onde cada componente se comunica de forma simples e fácil de manter. Deste modo, o Openstack é composto pelos componentes dispostos na Tabela 1.

Tabela 1 – Principais componentes do Openstack (OPENSTACK, 2019)

Componente	Função
Keystone	Responsável pela autenticação dos módulos do Openstack
Glance	Regula as imagens de sistema para as instâncias da nuvem
Nova	Serviço de computação do Openstack. Coordena o ciclo de vida das instâncias da nuvem
Neutron	Gerencia a rede do Openstack
Horizon	Oferece uma interface web para controle de módulos e recursos
Tacker	Baseado nas especificações ETSI para o NFV, este módulo é responsável por orquestrar serviços utilizando VNFs

2.2 *Network Function Virtualization*

O setor de telecomunicações, desde que foi concebido, tem utilizado dispositivos de rede de forma ampla. Com licenciamento de software proprietário e dependente de seus fabricantes, estes equipamentos dificultam a necessidade atual de cumprir requisitos cada vez mais exigentes de escalabilidade e elasticidade. Isso porque, adquirir novos dispositivos físicos demanda um alto custo, bem como aumenta as despesas operacionais para implantação dos mesmos. A estes custos dá se o nome de CAPEX e OPEX, respectivamente (MIJUMBI et al., 2015).

O paradigma NFV foi planejado devido a esta necessidade de se implantar redes cada vez mais dinâmicas. Com o amparo da computação em nuvem, o NFV introduz três principais características (HAN et al., 2015):

- Desacoplamento de software e hardware: esta divisão dá tanto ao software como ao hardware autonomia de desenvolvimento. Uma vez que um não depende do outro, ambos podem ser mantidos de forma separada;

- Desenvolvimento flexível de funções de rede: no paradigma NFV diversas funções podem ser desenvolvidas em um conjunto de diferentes recursos físicos e em locais distribuídos;

- Provisionamento dinâmico de serviços: administradores de rede podem determinar o escalonamento de funções de rede de forma ágil e programável de acordo com os requisitos estabelecidos.

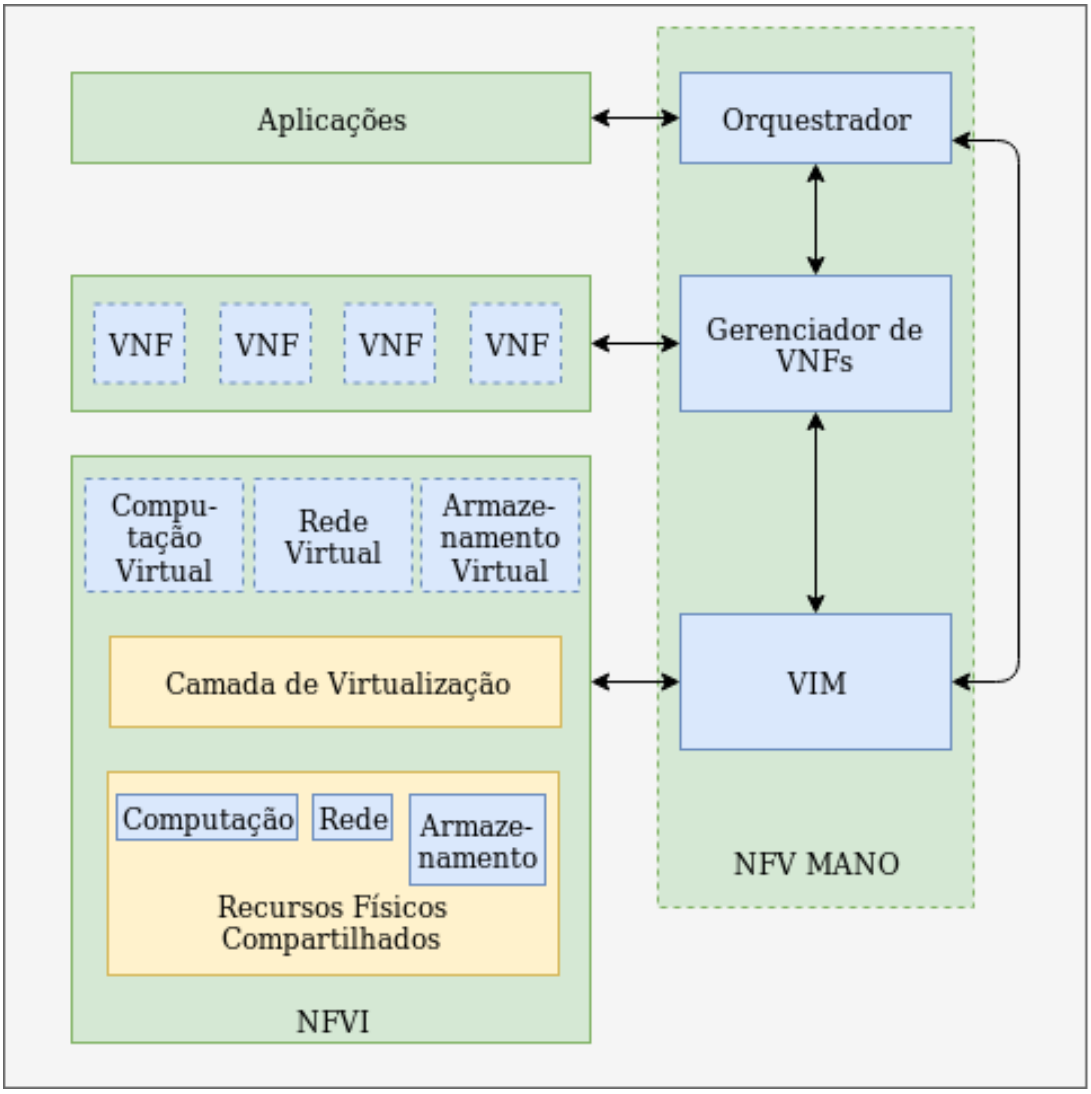


Figura 1 – Arquitetura NFV (Adaptado de (ETSI, 2013))

A concepção do NFV se deu no ano de 2012 onde a indústria se atentou para os problemas já descritos e, então decidiram selecionar a ETSI para ser o instituto responsável pela especificação do paradigma (MIJUMBI et al., 2015). A ETSI propôs a seguinte arquitetura ao NFV, exemplificada na Figura 1, cujo os quatro principais blocos estão descritos a seguir (HAN et al., 2015):

- Orquestrador - responsável pelo gerenciamento e orquestração dos recursos de software e hardware virtualizado;
- NFV MANO - cumpre as tarefas de instanciação, escalonamento, atualização e encerramento das VNFs;
- NFVI - abstrai os recursos físicos utilizados e garante que o ciclo de vida da VNF seja independente dos mesmos, oferecendo interfaces adequadas para tal;

- VIM - parte encarregada de prover recursos virtuais de computação, rede e armazenamento. Sendo capaz de gerenciar os mesmos e controlar suas interações com as VNFs.

2.2.1 Click

Click (KOHLER et al., 2000) é uma arquitetura de software modular destinada a construção de roteadores e funções de rede virtuais. A ideia foi concebida no início dos anos 2000 já com a proposta de flexibilizar o gerenciamento dos roteadores. Desde então já havia a preocupação com a capacidade de elasticidade e escalabilidade dos recursos dado aumento da demanda dos requisitos computacionais. Com esta arquitetura é possível construir VNFs de forma simples fazendo o uso de blocos conectáveis denominados "elementos".

Os elementos Click foram construídos com base na linguagem de programação C++ e têm como inspiração os próprios roteadores tradicionais. Isto porque a conexão pode ser iniciada tanto pela origem como pelo destino. Além disto, os elementos são conectados e orientados ao fluxo de pacotes. As principais propriedades dos elementos, são:

- Classe - A base do elemento é sua classe. Esta especifica o tipo de processamento que o pacote irá receber quando passar pelo mesmo.
- Portas - Um elemento contém uma ou mais portas de entrada e saída. Toda conexão tem como origem a porta de saída de um elemento e como destino a porta de entrada de outro.
- Configuração - Cada elemento possui alguns atributos opcionais que podem modificar o seu comportamento a fim de se ajustar a determinada demanda requisitada.

A base do código Click depende de alguns elementos primários dispostos na Tabela 2. Tais elementos se conectam, basicamente, conforme a Figura 2, onde o processo indicado como Pacote IP representa um ou mais elementos Click responsáveis pelo processamento de pacotes IP.

Listagem 2.1 – Exemplo de código do elemento Click AddressInfo.

```
1 AddressInfo(net0 10.0.0.5 10.0.0.0/24 FA:16:3E:2C:5C:46);
```

Listagem 2.2 – Exemplo de código do elemento Click Classifier.

```
1 // click router packet classifiers
2 c1,c2,c3,c4 :: Classifier(
3     12/0806 20/0001, // ARP Requests/queries out 0
4     12/0806 20/0002, // ARP Replies out 1
5     12/0800, // IP Packets out 2
6     -); // other packets - out 3
```

A sintaxe básica da linguagem desenvolvida para a arquitetura click envolve:

- Definição de constantes - Uso da palavra `define`. Exemplo: define($IFNET0 ens6), declara-se a constante IFNET o valor ens6.

Tabela 2 – Principais elementos do Click (KOHLER et al., 2000)

Elemento	Função
`Queue`	Os elementos não possuem capacidade de armazenar por si só. Para isto, o elemento `Queue` foi desenvolvido com o intuito de conservar os pacotes numa fila, auxiliando outros elementos e de roteadores Click de modo geral.
`AdressInfo`	Armazena dados de endereços IP, MAC e máscara de rede. Com este elemento é possível atribuir um nome e acessar os endereços de forma mais legível e intuitiva. Com o trecho de código na Listagem 2.1, por exemplo, é possível identificar endereço IP, MAC e máscara de rede, acessando net0:ip, net0:ipnet e net0:eth, respectivamente.
`FromDevice`	Lê os pacotes da interface de rede indicada no parâmetro do elemento.
`ToDevice]`	Envia os pacotes pela interface de rede designada no parâmetro do elemento.
`FromDPDKDevice`	Lê os pacotes da interface de rede, configurada com DPDK, indicada no parâmetro do elemento.
`ToDPDKDevice`	Envia os pacotes pela interface de rede, configurada com DPDK, designada no parâmetro do elemento.
`Classifier`	Um dos principais elementos do Click, o *Classifier* recebe em sua entrada os pacotes e classifica-os de acordo com o *Ethertype* determinado nos parâmetros do elemento. O trecho de código na Listagem 2.2 define quatro saídas: a primeira para as consultas ARP, a segunda para as respostas ARP, a terceira para os pacotes IP e a última para os demais pacotes.
`ARPQuerier`	Os pacotes que chegam na entrada 0 deste elemento devem ser pacotes IP com seu endereço de destino definido. Se um endereço MAC já for conhecido pelo destino, o pacote IP será empacotado em um cabeçalho *Ethernet* e enviado para sua saída 0. Caso contrário, o pacote IP será salvo e uma consulta ARP será enviada. Se uma resposta ARP chegar à entrada 1 para um endereço IP requisitado pela máquina, o mapeamento dos endereços IP e MAC é registrado e todos os pacotes IP serão enviados.
`ARPResponder`	Este elemento encaminha uma resposta ARP caso o endereço seja conhecido. Se não, o pacote é enviado a saída 1, caso exista.

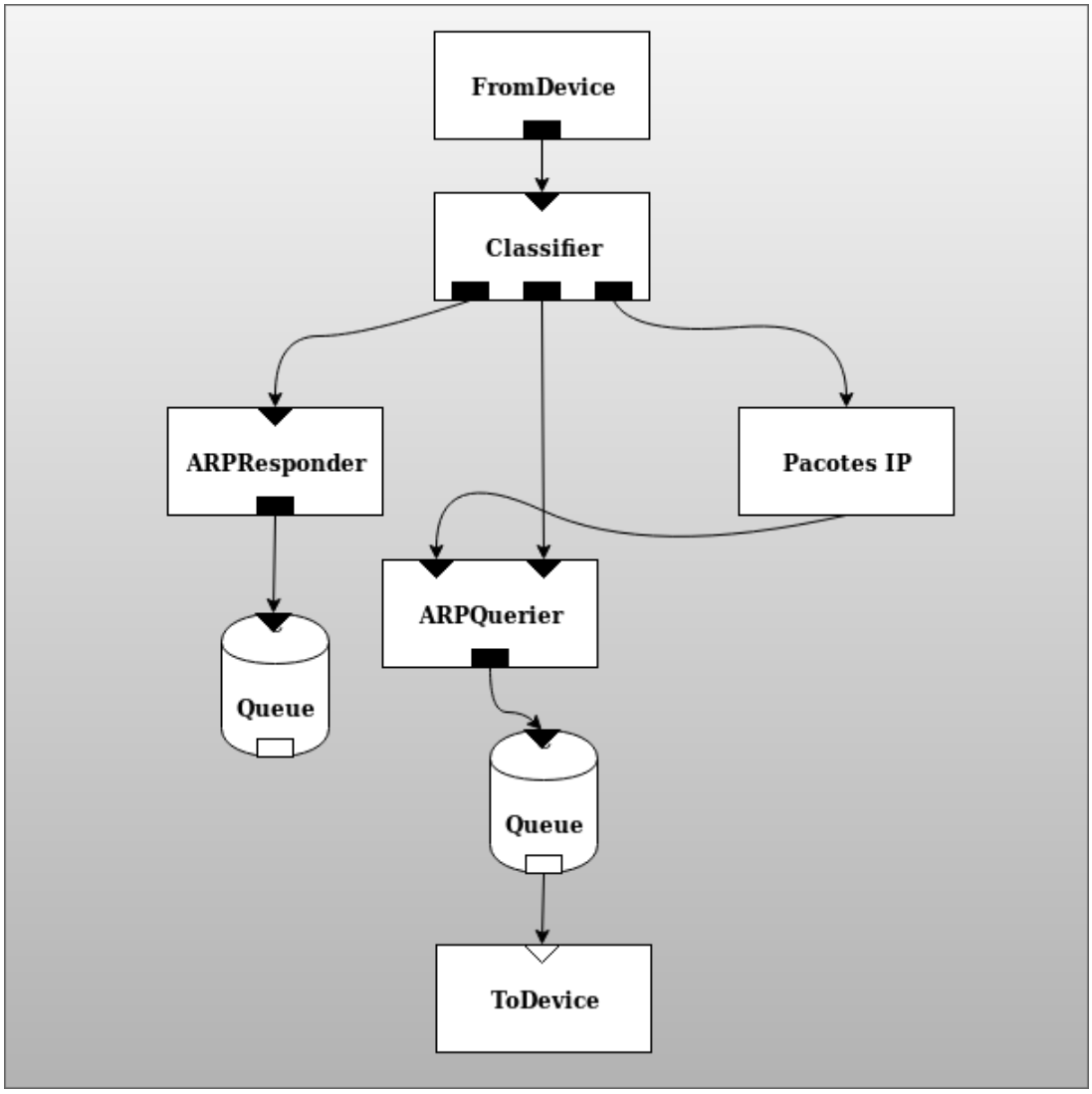


Figura 2 – Fluxograma dos elementos Click em um roteador (Adaptado de (KOHLER et al., 2000))

- Declaração de Variáveis - Utilização de dois pontos em sequência. Exemplo: arpq1 :: \ texttt{ARPQuerier(net0), atribui-se a variável arpq1 ao elemento `ARPQuerier`.

- Indicação de fluxo - Para indicar o fluxo utiliza-se a expressão '->'. Exemplo: $sink1$ :: `ARPPrint`$()->$`Queue`$(1024)->$`ToDevice`$(\$IFNET0)$, assinala que o fluxo passa pelos elementos `ARPPrint`, `Queue` e `ToDevice` em sequência. E este fluxo é atribuído a variável sink1.

- Indicação de porta - Denotação da porta entre colchetes. Exemplo: `FromDevice`$(\$IFNET0)->[0]c1$ onde o tráfego que chega é enviado a porta de entrada de um elemento referenciado pela variável c1 do tipo `classifier`. Em $c1[2]->$`Strip`$(14)->$`CheckIPHeader`$()->[0]sfcclassifier$ o tráfego IP que chega ao `classifier` c1 sai pela porta 2 e segue o fluxo definido.

2.2.2 DPDK

Também no intuito de auxiliar a construção de VNFs em plataformas x86, o DPDK tem como característica principal a não utilização de interrupções para, por exemplo, avisar a chegada de pacotes em determinada interface. Ou seja, os processos DPDK operam em modo de *polling*, fazendo com que a CPU opere em função das VNFs verificando constantemente quando a mesma precisa de processamento.

Tecnicamente o DPDK oferece uma camada de abstração a fim de oferecer acesso aos recursos do sistema. Para gerenciar a memória, o DPDK oferece dois modelos, o `rte_malloc` e o `rte_mempool`. O primeiro, semelhantemente à função `malloc` da linguagem C, aloca objetos em páginas de memória maiores que as do padrão do sistema operacional. Essas páginas grandes, alinhadas com a memória cache num *socket* de acesso não uniforme à memória(NUMA), tem por objetivo aumentar o desempenho das aplicações. Já o `rte_mempool` consiste em um conjunto de objetos de tamanho fixo que podem ser alocados conforme demanda. Por fim, o DPDK oferece uma biblioteca chamada de *Poll Mode Driver* (PMD) para configurar as interfaces de rede e suas respectivas filas. Além disso, o PMD acessa os descritores RX e TX diretamente sem interrupções (com exceção das interrupções da alteração do status do link) a fim de receber, processar e entregar pacotes de forma eficiente (CERRATO; ANNARUMMA; RISSO, 2014).

2.2.3 Fastclick

Click foi publicado em um momento onde NFV não estava em evidência. Com o aumento das demandas de processamento de pacotes a nível de usuário, Barbette, Soldani e Mathy (2015) reuniu alguns *frameworks* e decidiu compará-los em termos de desempenho. O fruto deste trabalho rendeu a integração do Click com os frameworks DPDK e NETMAP (RIZZO, 2012).

Fastclick utiliza os mesmos elementos que o Click, a principal mudança é estrutural e no código fonte do Click. As principais modificações são:

- IO em lotes - Click processa todos os pacotes antes de chamar novamente o método que indica quantos pacotes estão na fila de entrada. Assim, os pacotes disponíveis em lotes são lidos transmitidos de uma única vez. As tarefas correspondentes a esta transmissão só entregam a CPU no final da rajada de pacotes. No entanto, Click reprograma a tarefa se um outro pacote puder ser processado. O Fastclick, por outro lado só reprograma esta tarefa se um outro lote de pacotes chegar.

- Modelo *Full Push* - No Click todos os pacotes que chegam são colocados em uma fila e processados por ordem de chegada. Os pacotes atravessam os elementos entre a fila e o elemento de saída, `ToDevice`, para então ser enviados para transmissão.

Esta abordagem permite que o trabalho seja dividido entre *threads* que lidem (i) com a parte entre o elemento `FromDevice` e a fila, e (ii) entre a fila e o elemento `ToDevice`. Outra vantagem do uso destas filas é que o elemento `ToDevice` só recebe pacotes se o mesmo possuir espaço. Entretanto, esta mesma abordagem possui as seguintes desvantagens: (i) várias *threads* podem escrever na fila. Logo, alguma forma de sincronização deverá ser utilizada, sobrecarregando o sistema de certa forma; e (ii) caso as *threads* sejam executadas em núcleos diferentes, poderão ocorrer erros de cache, resultando em perda de desempenho. O Fastclick, por sua vez, implementa o método *full push*, onde não há fila. As interfaces de rede atualmente possuem espaço suficiente para acomodar mais de 4096 pacotes, fazendo o uso de filas desnecessário. Deste modo, uma *thread* toma conta do processo do começo ao fim, ou seja, do elemento `FromDevice` ao `ToDevice`. Este modelo prova ser a maior contribuição do Fastclick. Há uma série de melhorias apontadas no trabalho do Fastclick para evitar possíveis falhas a partir desta funcionalidade.

- *Zero Copy* - Um pacote no Click possui os seus dados e metadados. Os dados do pacote são salvos em um *buffer*, mas alocar um *buffer* para cada pacote diminui o desempenho de processamento. Por este motivo, no Fastclick os pacotes são pré-alocados em *pools*. Quando um pacote é recebido, o seu conteúdo é copiado para o pool e o metadado é escrito em memória no objeto do pacote. Assim, o processamento é otimizado e direcionado a identificar quando o *buffer* está cheio e não quando um pacote é recebido. E já que um *buffer* nunca é copiado, este processo permite o encaminhamento *"Zero Copy"* de *buffers* na fila circular.

2.3 *Service Function Chaining*

O aumento da diversidade de aplicações e seus diversos requisitos em relação a qualidade de serviço, atrelado ao aumento do tráfego nas redes dos data centers, implicam em um natural aumento da necessidade de gerenciamento do uso dos serviços de rede. O Encadeamento de Funções de Serviço (SFC) surge como uma promissora tecnologia para gerenciamento flexível de serviços de rede, classificando fluxos de acordo com seus requisitos e políticas de uso. Sendo assim, por definição, o SFC é composto por um conjunto de Funções de Serviço (SFs) ordenadas por cadeias que manipulam, controlam e monitoram o tráfego de uma aplicação específica (MEDHAT et al., 2016).

Muitas redes de *data center* ainda possuem equipamentos físicos, como *firewalls*, NATs, etc. Ordenar cadeias de serviço que atendam às restrições da política de uso, capacidade computacional e rede, bem como ter latência e vazão aceitável para os usuários finais é uma tarefa extremamente difícil. Alterações dinâmicas (sob demanda) precisam ser feitas para essas cadeias de serviço, tais como a exclusão ou adição de novas funções virtuais

em função das demandas do usuário (BHAMARE et al., 2016). Com isto, recentemente SFC tem feito uso de tecnologias habilitadoras, tais como o SDN e NFV (MEDHAT et al., 2016).

As especificações do SFC se dividem entre grupos como ETSI e ONF. Um dos documentos de grande importância para o tema, a RFC 7665 (HALPERN; PIGNATARO, 2015) aponta as principais características relacionadas à arquitetura do SFC dentro dos padrões do NFV. Nesta RFC são definidos:

- *Service Function Paths* - São as cadeias de um SFC. Um SFP pode ser pré-determinada, selecionando exatamente por onde deve passar, ou deixar esta tarefa para outros componentes da arquitetura SFC;
- Encapsulamento - o encapsulamento SFC permite a seleção e identificação de SFPs, como também o compartilhamento de metadados;
- SF - Uma SF pode ser considerada um recurso dentro de um domínio que é parte de um serviço. Uma SF executa um serviço específico e pode ou não ter conhecimento do domínio SFC. As que possuem ciência do SFC podem lidar com o encapsulamento SFC, já as que não possuem necessitam de um *proxy*.
- SFF - O SFF é responsável pelo encaminhamento de pacotes recebidos da rede para as SFs utilizando-se das informações contidas no encapsulamento SFC. O tráfego das SFs eventualmente retorna ao SFF, que fica responsável por devolver os pacotes à rede. Algumas SFs, como *firewalls*, podem consumir um pacote e não devolvê-lo ao SFF no caso de bloqueio;
- SFC *Proxy* - Esta função fica entre o SFF e as SFs para os quais o SFF está direcionando tráfego. O *proxy* recebe pacotes do SFF em nome do SF, remove o encapsulamento SFC e, em seguida, entrega os pacotes para as SFs que não reconhecem o encapsulamento SFC. Na volta, ao receber os pacotes do SF, o encapsulamento SFC é reaplicado e o pacote encaminhado de volta ao SFF;
- Classificação - O tráfego da rede que satisfaz aos critérios de classificação é direcionado a um determinado SFP e encaminhado para as SFs que a compõem. A classificação pode possuir vários níveis de granularidade, desde as mais genéricas às mais restritivas, que tratam por portas de rede, por exemplo. Como consequência da classificação, os pacotes que chegam ao ambiente SFC são encapsulados e um SFP adequado é selecionado ou criado. Um pacote pode ser reclassificado e assim, ser encaminhado a outro SFP.

São princípios gerais de uma arquitetura de implementação de SFC:

- Independência da topologia - Nenhuma modificação na topologia de rede é necessária para implantar o SFC;
- Separação dos planos - a realização das SFPs deve ser separada do processamento e manipulação de pacotes;
- Classificação - O tráfego que satisfaz determinada regra é encaminhado a uma SFP específica;
- Metadados Compartilhados - Metadados podem ser compartilhados entre elementos do SFC e também fora do ambiente SFC;
- Independência - A criação, modificação ou remoção de uma SFC não pode impactar outras. O mesmo vale para as SFPs;
- Pontos de controle heterogêneos - A arquitetura permite que as funções de serviço que compõe uma cadeia usem mecanismos independentes para lidar com políticas e critérios de classificação locais.

Outra RFC que apoia a implementação do SFC é a de número 8300 (QUINN; ELZUR; PIGNATARO, 2018). Nesta proposta, a especificação aborda o encapsulamento SFC com o protocolo *Network Service Header* (NSH). O NSH é dividido em três macro elementos: o (i) *Base Header*, o (ii) e (iii) *Service Path Header* e o *Context Header*.

O *Base Header* é encarregado de carregar informações acerca do protocolo NSH, tais como sua versão, o máximo de saltos numa SFP e o protocolo que acompanha o pacote após o cabeçalho NSH. Por sua vez, o *Service Path Header* contém os principais dados do protocolo. O *Service Path Identification* (SPI) identifica exclusivamente um SFP. O Classificador define este identificador de acordo com o resultado da classificação. Já o *Service Index* (SI) fornece a localização dentro de um SFP. Pode-se configurar o valor inicial do SI de acordo com a demanda do sistema. No entanto, por padrão, inicialmente o classificador deve definir o SI para 255 que vai sendo decrementado pelo valor unitário ao passo que for executado pelas SFs constituintes do SFP. Por fim, o *Context Header* carrega metadados do protocolo.

A combinação entre os elementos do SFC e os dados de SPI e SI proveem a construção de um ambiente adequado à operação do SFC. Ambas RFCs também elucidam alguns pontos que ambientes SFC devem controlar, como a fragmentação de pacotes, por exemplo, dado o encapsulamento, além de controles como: segurança, balanceamento de carga, redundância, resiliência dos recursos, etc.

3 Trabalhos Relacionados

Este capítulo apresenta trabalhos relacionados e destaca as principais contribuições desta dissertação. Visto que abordagens SFC utilizam tanto NFV como SDN ou uma combinação de ambas, a seguir estão elencados os trabalhos que utilizam como base as RFCs 7665 (HALPERN; PIGNATARO, 2015) e 8300 (QUINN; ELZUR; PIGNATARO, 2018) para desenvolver um ambiente com estas características.

3.1 *Compact* SFC *Header*

O objetivo do *Compact* SFC *Header* (HANTOUTI; BENAMAR; TALEB, 2018) é simplificar o encaminhamento do SFC reduzindo o tamanho do pacote e evitando a reclassificação do tráfego em um SFF. As informações do SFC ficam no campo do endereço MAC de origem. Uma vez que este não é modificado durante o tráfego. Deste modo, o tamanho do pacote permanece o mesmo. A estrutura do cabeçalho consiste em: i) um identificador da cadeia de serviços, ou seja, o SFP; ii) a próxima função de serviço, ou seja, o SI; e iii) um contador que é decrementado após a visita ao SF e inicializado com o número de SFs que o fluxo de tráfego deve atravessar. Este último serve para identificar a posição da próxima função de serviço e expira após a visita ao último SFF, antes do envio para o destino.

O desenvolvimento desta arquitetura se deve aos requisitos reais de infraestruturas, tais como:

- Flexibilidade - A abordagem se utiliza da vantagem de poder ser implementada em um controlador SDN, ganhando assim a flexibilidade da programabilidade de ser implementada tanto em funções físicas e virtuais.
- Escalabilidade - O *Compact* SFC *Header* codifica as informações relevantes no endereço MAC. Portanto, não é adicionado nenhum *bit* adicional aos pacotes e, ao mesmo tempo, um número significativo de cadeias e funções podem ser desenvolvidas.
- Sobrecarga - O SFC e a implementação do NSH introduzem sobrecarga à rede, principalmente no que tange ao aumento do tamanho do pacote e às operações de encapsulamento e desencapsulamento. Como o trabalho não introduz um novo cabeçalho ao pacote, isto é evitado.

Ao passo que o *Compact* SFC *Header* promete avanços importantes em termos de eficiência, centralizar as operações em torno de um controlador tende a gerar um ponto

único de falha caso não sejam adotadas políticas de escalonamento adequadas.

3.2 Neo-NSH

Na abordagem Neo-NSH (KULKARNI et al., 2017), os SFFs usam os campos SPI e SI para representar o tipo de serviço e deixam que a própria rede escolha dinamicamente a melhor SF com base no tipo da função de rede e nas informações de contexto. Deste modo, os SFFs necessitam fazer uso de um controlador SDN inteligente para escolher a instância de serviço apropriada. Nesta abordagem os classificadores são responsáveis por inserir o ID do serviço.

Os principais benefícios do *Neo*-NSH são: i) a capacidade de balancear a carga do tráfego nas SFs. ii) a adição e remoção de SFs não afeta o SPI. Assim, a rede se torna mais ágil visto que também não há necessidade de calcular e comunicar os rótulos SPI aos SFFs.

A desvantagem desta proposta é a perda de visibilidade do caminho de ponta a ponta. Isso porque a lista de SFs para determinado SPI não pode ser determinada estaticamente, dado que um mesmo SPI pode mapear diferentes caminhos.

3.3 PhantomSFC

A arquitetura proposta em Castanho et al. (2018) foi desenvolvida para cumprir os seguintes requisitos:

1. Independência do esquema de transporte;
2. Abordagem agnóstica em relação ao tipo de rede;
3. Elasticidade;
4. Baixo impacto na performance da rede.

O PhantomSFC associa cada elemento SFC a uma instância virtual. Isto possibilita que a rede seja usada apenas para fins de comunicação. Além disto, a virtualização dos componentes do SFC atende ao requisito de escalabilidade, pois cada componente pode ser implementado por várias instâncias distribuídas que podem ser dimensionadas de acordo com a demanda. Por fim, com intuito de gerenciar o ambiente, há um controlador SFC logicamente centralizado responsável por registrar, gerenciar elementos arquiteturais, além de lidar com a criação, remoção e modificação de cadeias.

Instanciar cada elemento em um elemento virtual causa, entretanto, uma sobrecarga na rede. A fim de minimizar isto, foi utilizada a biblioteca DPDK devido ao seu bom

desempenho no processamento de pacotes.

O trabalho PhantomSFC é um dos trabalhos que mais se assemelha ao sfcFC. No entanto, a construção dos elementos do SFC dependem de um controle de ponteiros em C. Logo, no que tange a curva de aprendizado, este trabalho requer um nível de esforço maior num comparativo a construção de VNFs com elementos Click.

3.4 Neutron SFC

Vale ressaltar também o *networking*-sfc, que é um módulo do Openstack capaz de entregar o SFC. Esta abordagem faz uso do Tacker e de *scripts* TOSCA[1]. O Neutron enxerga a rede virtual através de portas, sendo estas estruturas básicas de sua arquitetura. Assim, cada interface virtual é um porta e é através delas que o SFC é realizado. A inclusão dessas portas em cadeia permite direcionar o tráfego através de uma ou mais instâncias que fornecem funções de serviço.

O módulo SFC para o Neutron possui várias limitações além de não realizar o SFC de acordo com as normas da ETSI. Por exemplo, a criação de cadeias entre duas redes virtuais é uma limitação[2] conhecida até a presente data de publicação deste trabalho. Além de ser limitado ao Openstack, o projeto utiliza de *scripts* TOSCA que possuem baixa legibilidade.

As brechas encontradas no `Neutron SFC` dão condições favoráveis para a implementação de uma aplicação que realize SFC no Openstack.

[1] http://docs.oasis-open.org/tosca/tosca-nfv/v1.0/tosca-nfv-v1.0.html
[2] https://ask.openstack.org/en/question/117018/cross-subnet-chain/

4 Proposta do sfcFC

Este trabalho tem como propósito implementar funções de rede programáveis em ambientes de nuvem com a abordagem do SFC. Deste modo, usou-se o Fastclick, apresentado neste trabalho no Capítulo 2, como ferramenta de implementação. Além de prover melhorias em relação ao Click, a ferramenta é constantemente atualizada por seus idealizadores, diferentemente do Click que já não recebe atualizações há alguns anos. Além disto, tanto Click quanto Fastclick permitem uma construção de funções de rede independente de plataforma, com isso podem ser utilizadas em qualquer hardware x86.

4.1 Projeto da Arquitetura

Como neste trabalho já foi mencionado o Classifier, elemento Click. O elemento classificador do SFC construído para o `sfcFC` será denominado `sfcFC-Classifier`. Assim, serão utilizados os seguintes componentes: `sfcFC-Classifier`, `forwarder`, a SF *firewall*, a SF balanceador de carga *Round Robin* e a SF balanceador de carga *source* IP. Os dois primeiros elementos fazem parte da arquitetura proposta pela RFC 7665, onde o `sfcFC-Classifier` é responsável por classificar os pacotes que chegam ao ambiente e encaminhá-los ao `forwarder`. Este, por sua vez, deverá encaminhar os pacotes para as SFs com base nas regras de encaminhamento pré-definidas. A Figura 3 apresenta uma topologia utilizada como exemplo para ilustrar o funcionamento do `sfcFC`. Nela é possível distinguir três ambientes: (i) dos clientes, (ii) do `sfcFC` com seus respectivos componentes e dos (iii) servidores, que oferecem aplicações web, por exemplo.

Para que a integração entre um ambiente de nuvem e o Fastclick viabilizasse a implementação do SFC, foi necessário desenvolver a ideia de um cabeçalho `sfcFC`. A construção deste cabeçalho se espelha na abordagem IPinIP, que por sua vez, é definida através da RFC 2003 (PERKINS, 1996). Logo, o cabeçalho `sfcFC` é uma abstração que na prática se utiliza de um cabeçalho UDP e, em seus campos de porta de origem e porta de destino, determina respectivamente os campos SPI e SI, definidos pela RFC 8300. Junto ao o cabeçalho UDP, o cabeçalho `sfcFC` também carrega um cabeçalho IP que armazena os endereços de origem e destino durante o SFP. Assim, ao chegar no ambiente `sfcFC` o pacote IP original será envelopado com o cabeçalho UDP/IP carregando informações sobre o encaminhamento das SFs na rede. A Figura 4 mostra como se dá o encapsulamento do pacote UDP/IP ao chegar ao ambiente `sfcFC`. O SPI, como já mencionado, identifica a cadeia (SFP). Ao passo que o SI identifica a etapa dentro do SFP. No `sfcFC` o SI é incrementado após passar por uma SF. Com isso, resumidamente o cabeçalho é composto por estes quatro campos: SFP, SP, IP origem e IP destino.

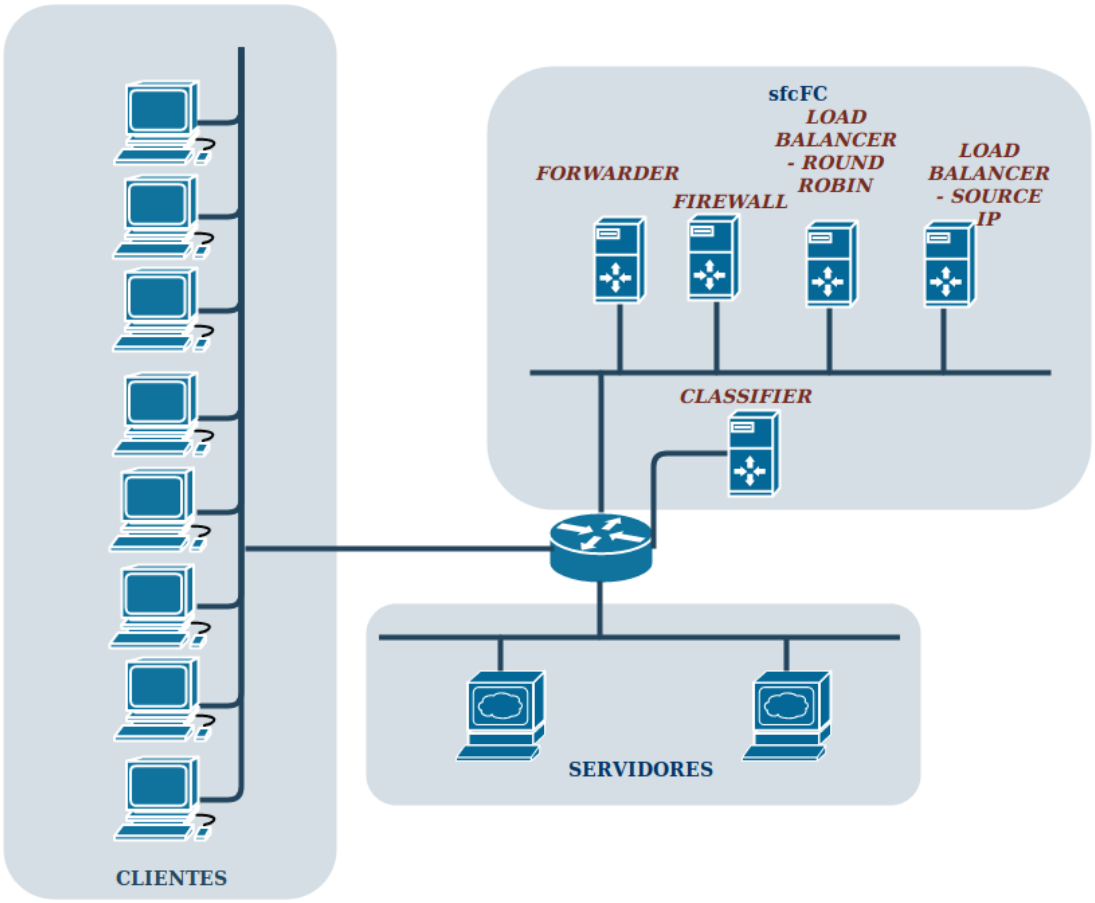


Figura 3 – Topologia usada como exemplo do sfcFC

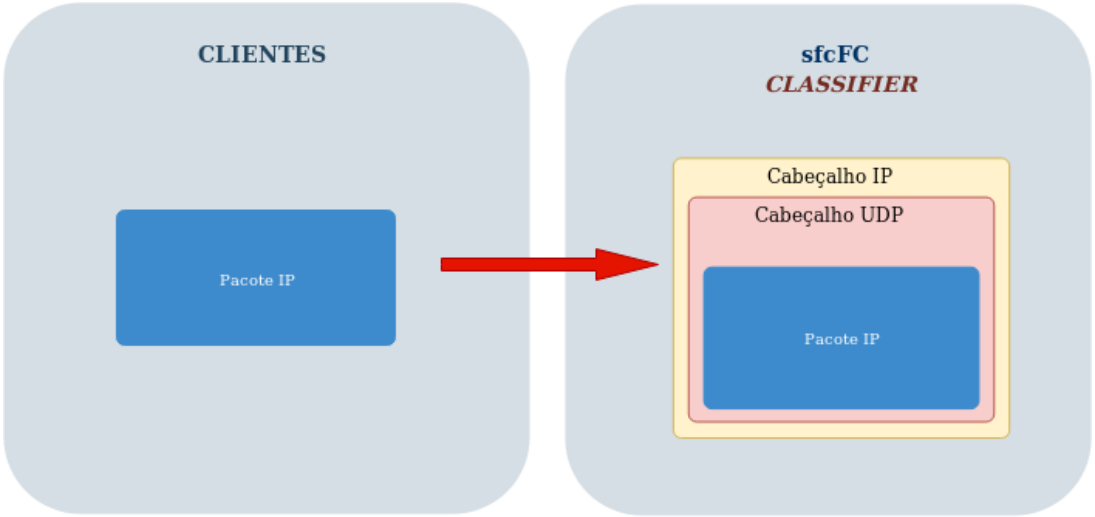


Figura 4 – Encapsulamento do cabeçalho sfcFC ao pacote IP

4.1.1 sfcFC-Classifier

O sfcFC-Classifier deve estar pronto para receber pacotes e, com base em cadeias pré definidas, classificar e encapsular o pacote IP recebido com o cabeçalho sfcFC ao pacote que se encaixar nas regras definidas pelo ambiente SFC e em seguida encaminhar ao forwarder. O diagrama de atividades da Figura 5 ilustra o ciclo do pacote durante a

sua passagem pelo sfcFC-Classifier.

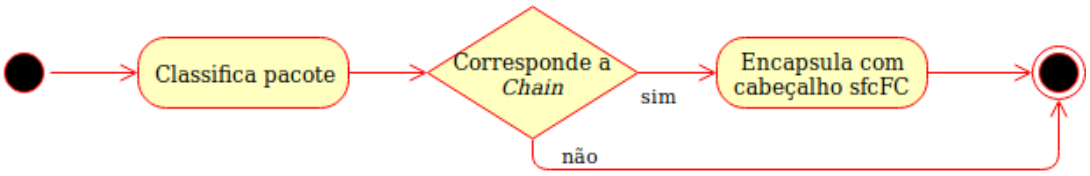


Figura 5 – Diagrama de Atividades correspondente ao sfcFC-Classifier

4.1.2 Forwarder

Ao receber um pacote, o forwarder analisa as informações contidas no cabeçalho sfcFC e determina a SF apropriada para onde o tráfego deve ser encaminhado. Após o processamento pela última SF, o forwarder deverá receber o pacote, desencapsular o cabeçalho sfcFC e enviar ao destino final. Este processo é apresentado através do diagrama da Figura 6.

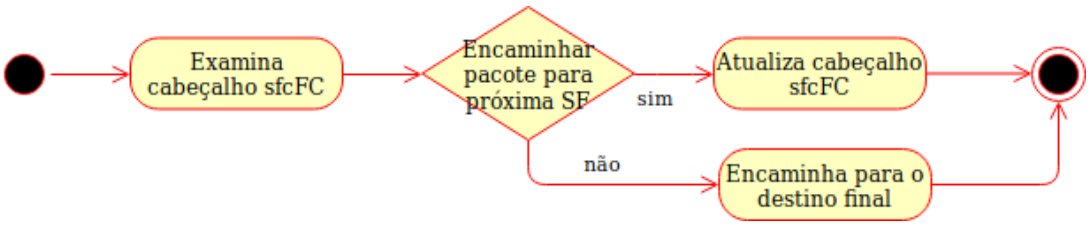


Figura 6 – Diagrama de Atividades correspondente ao SFF

4.1.3 *Service Functions*

As SFs propostas neste trabalho possuem conhecimento da existência do cabeçalho sfcFC e devem, então, processar o cabeçalho e realizar operações. Assim, de acordo com as etapas do diagrama da Figura 7, uma SF deve desencapsular o pacote sfcFC, realizar determinada operação, e então encapsular o pacote com o cabeçalho sfcFC com os dados atualizados. Após isto o pacote retorna ao SFF.

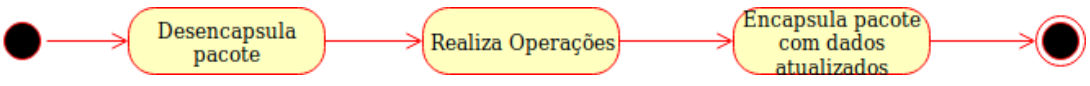


Figura 7 – Diagrama de Atividades correspondente aos SFs

As SFs propostas neste trabalho possuem conhecimento da existência do cabeçalho sfcFC e devem, então, processar o cabeçalho e realizar operações. Assim, de acordo com as etapas do diagrama da Figura 7, uma SF deve desencapsular o pacote sfcFC, realizar determinada operação, e então encapsular o pacote com o cabeçalho sfcFC com os dados atualizados. Após

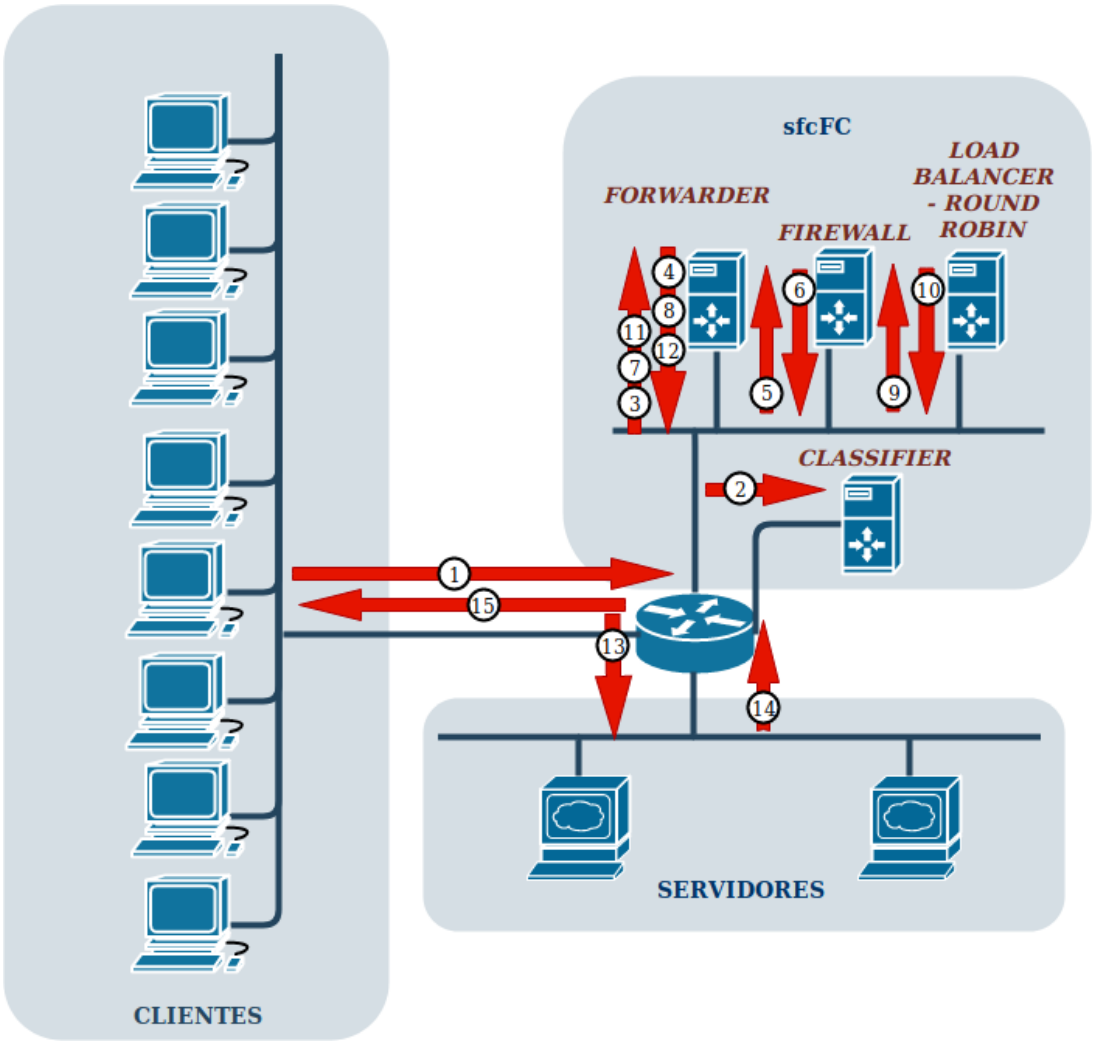


Figura 8 – Representação do Primeiro SFP

4.2 Elaboração da Execução do sfcFC

Na Seção 4.1 foram citados os componentes do SFC que este trabalho irá utilizar para a entrega do SFC. Tais componentes foram utilizados a fim de testar alguns casos de uso. No primeiro caso, para o primeiro SFP, foi utilizada a topologia já mencionada na Figura 3. Os clientes acessam um serviço web que está disponível nos servidores. Entretanto, ao realizar a requisição o pacote chega ao `sfcFC-Classifier` que classifica o pacote como uma requisição que precisa passar pelo primeiro SFP, este que será descrito a seguir. Assim, o `sfcFC-Classifier` encapsula o pacote com o cabeçalho `sfcFC` e imediatamente o encaminha ao SFF. Este, por sua vez, identifica o pacote como sendo do primeiro SFP e o faz percorrer as SFs definidas. Para este SFP foram especificadas as SFs *firewall* e balanceador de carga *Round Robin*.

Neste exemplo o *firewall* tem como regra apenas aceitar conexões vindas na porta 80(http) ou 443(https). Já o balanceador de carga *Round Robin* alterna os destinatários da requisição como se os servidores estivessem numa fila circular. Ou seja, uma requisição

para cada servidor por vez, sequencialmente, sendo que o servidor que recebeu a última requisição (mais recente) vai para o final da fila de atendimento a novos clientes.

A Figura 8 mostra o passo-a-passo da requisição: em 1 o cliente faz a requisição, que é direcionada ao `sfcFC-Classifier` em 2 assim que o pacote alcança o *gateway* da rede. Em 3 o `sfcFC-Classifier` classifica e encaminha o pacote já com o cabeçalho `sfcFC` para o SFF. Em 4 e 5, o SFF identifica o SFP e envia o pacote para o *firewall*, que processa o pacote e devolve para o SFF nos passos 6 e 7. A seguir, em 8 e 9, o SFF encaminha o pacote para o balanceador de carga *Round Robin*, que por sua vez, escolhe o servidor de destino e devolve para o SFF. Por fim, em 12 e 13, o SFF envia a requisição para o servidor, que a processa e devolve para o cliente em 14 e 15.

No segundo caso, ilustrado pela Figura 9, o SFP exerce papel semelhante a do primeiro SFP. No entanto, o balanceador de carga Round Robin é substituído pelo balanceador de carga *Source* IP. Este balanceador realiza um mapeamento baseado no IP de origem da requisição. Assim, o tráfego é sempre direcionado para o mesmo destino em requisições cujo o endereço IP já está mapeado.

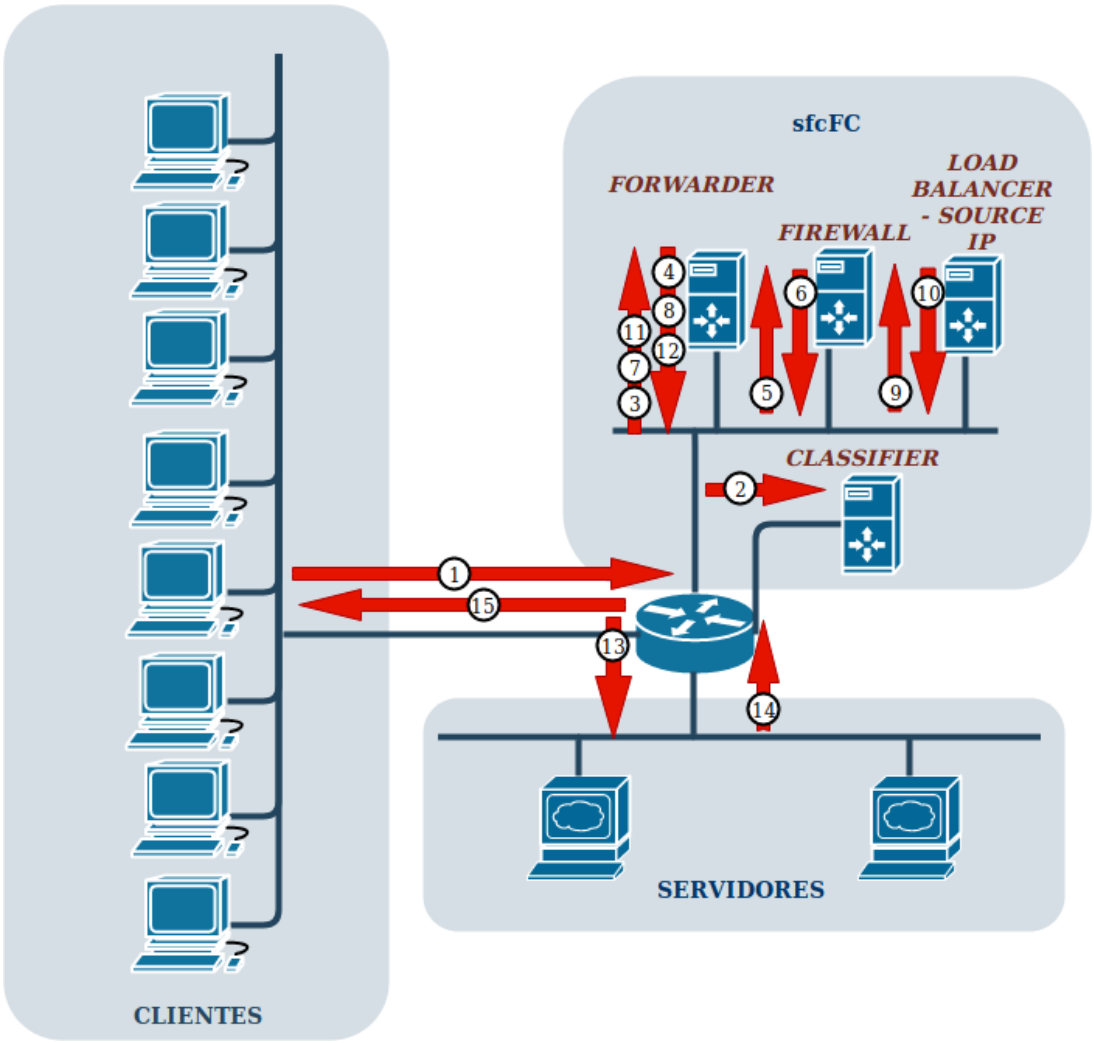


Figura 9 – Representação do Segundo SFP

O terceiro exemplo deste trabalho é também similar aos anteriores. No entanto, neste modelo não há nenhuma SF realizando balanceamento de carga, apenas a SF que executa o *firewall*, que assim como nos exemplos anteriores, somente aceita conexões nas porta 80 e 443.

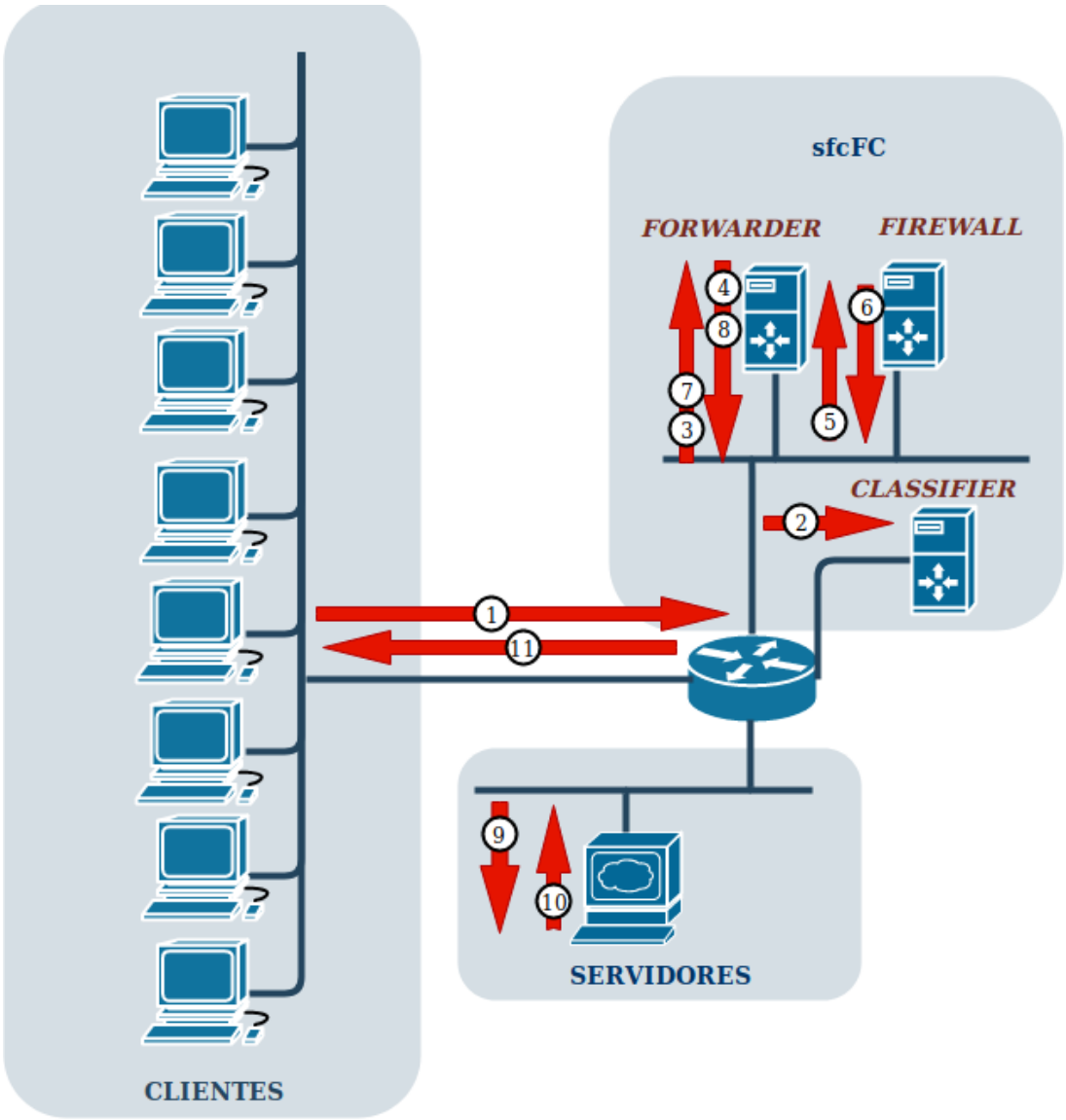


Figura 10 – Representação do Terceiro SFP

A representação do terceiro e quarto SFP é a mesma, como mostra a Figura 10.

Os códigos implementados no `sfcFC-Classifier`, SFF e SFs estão disponibilizados nos Apêndices B, C, D, E se F. Apesar do Fastclick trabalhar tanto com DPDK como com Netmap, este trabalho faz uso apenas do Fastclick com o DPDK. A escolha foi feita apenas por questão de comodidade. O DPDK tende a ser o mais rápido em termos de taxa de transferência de entrada e saída, enquanto o Netmap oferece controle mais refinado do RX e TX (BARBETTE; SOLDANI; MATHY, 2015). Dos códigos, vale se destacar os seguintes elementos do Click:

- IPClassifier - Elemento que classifica pacotes IP de acordo com regras pré-estabelecidas. O padrão para a classificação dos pacotes segue o estilo usadno no `tcpdump`. É possível, assim, classificar pacotes pelo endereço de origem e destino, por exemplo. `sfcFC-Classifier`, SFF e SFs utilizam este elemento para classificar os pacotes, tanto por endereço como por porta de origem ou destino;

- UDPIPEncap - É o elemento base para este trabalho. O mesmo encapsula o pacote com um cabeçalho IP/UDP, onde se pode indicar endereços e portas de origem e destino;

- CheckIPHeader - Checa o cabeçalho do pacote IP e valida seus respectivos atributos;

- CheckLength - Recebe como parâmetro o tamanho do pacote, ou MTU. Caso o pacote passe pelo elemento com um MTU maior que o especificado, o pacote é enviado para a saída 1. Caso contrário, prossegue pela saída 0;

- IPFragmenter - Fragmenta os pacotes que possuem um MTU maior que o designado nos parâmetros. Para tal, o pacote precisa permitir isto. Caso não permita, o pacote é enviado para a saída 1. Pacotes fragmentados são enviados para a saída 0;

- SetTCPChecksum - Calcula o *checksum* e atualiza o respectivo campo no dado cabeçalho do pacote;

- StripIPHeader - Remove o cabeçalho IP mais externo do pacote;

- Strip - Remove uma quantidade de Bytes do pacote especificada por parâmetro. É utilizado na maior parte das vezes para remover cabeçalhos dos pacotes;

- SetIPAddress - Determina o IP destino do pacote;

- IPFilter - É a base para a SF *Firewall*, este elemento filtra os pacotes de acordo com regras estabelecidas em seus parâmetros. Utiliza a palavra chave *allow* para permitir e *drop* para bloquea;

- IPRewriter - Este elemento é utilizado nas SFs de balanceamento de carga. O mesmo reescreve origem e destino tanto do endereço como da porta do pacote. Este elemento funciona como um tradutor de endereços, ou seja, realiza um NAT (*Network Address Translation*);

- RoundRobinIPMapper - Em conjunto com o IPRewriter, reescreve o endereço do pacote. Funciona em aplicações de balanceamento de carga. Este elemento distribui as requisições em um mapeamento que utiliza uma fila circular(Round-Robin). É utilizado na SF balanceador de carga *Round Robin*.

- SourceIPHashMapper - Este elemento também atua em conjunto com o IPRewri-

ter reescrevendo os endereços dos pacotes. Porém, a reescrita é determinada de acordo com o IP de origem do pacote. Assim, um dado IP sempre será associado a mesma regra, e, consequentemente, a mesma saída. Este elemento é utilizado na SF balanceador de carga *source* IP.

5 Avaliação do sfcFC

Neste capítulo estão descritos os testes realizados com intuito de avaliar a proposta sfcFC. No Capítulo 4 foi descrito um cenário de exemplo, através da Figura 3. Este cenário foi replicado no Openstack e é ilustrado através da Figura 11. Assim, pode-se perceber em ordem, da esquerda para a direita: (i) a rede sfcFC, (ii) a rede dos servidores e (iii) a rede dos clientes. Já a Tabela 3 serve de apoio para o entendimento do cenário, identificando cada elemento da topologia através do seu respectivo número dado pela Figura 11. É importante salientar que neste capítulo todos os elementos também serão citados se referindo a este número.

De acordo com a Tabela 3, cada SFP possui um IP de destino. Isto implica dizer que o sfcFC-Classifier implementado neste trabalho realiza a classificação de cada SFP com base no IP de destino. Contudo, outras formas de classificação também podem ser implementadas.

Vale destacar que para fazer o uso do Fastclick com DPDK no Openstack foram seguidos os passos descritos no Apêndice A. A configuração do ambiente de teste envolvia um servidor Dell PowerEdge T430, com 16 GB de memória RAM, processador de 6 núcleos 2.4GHz e 1TB de disco. Cada módulo do sfcFC foi implementado com a mesma configuração em uma máquina virtual no Openstack possuindo um processador com um núcleo, 1 GB de RAM e executando o sistema operacional Ubuntu.

Os testes se dividem em duas categorias: qualitativo e quantitativo. O primeiro leva em consideração o funcionamento do do mecanismo implementado no sfcFC, e está representado pelas Seções 5.1, 5.2 e **??**. O segundo, demonstrado pela Seção 5.4, avalia o desempenho da solução frente a não utilização de SFC no encaminhamento de pacotes.

5.1 Teste para o primeiro SFP

O primeiro teste qualitativo, o SFP 1 realiza o seguinte fluxo: (i) o cliente (6,7) realiza a requisição HTTP, (ii) que é classificada a cada pacote pelo sfcFC-Classifier (1), e (iii) encaminhada para o forwarder (5), que, por sua vez, (iv) faz o fluxo passar pelas VNFs *firewall* (4) e balanceador de carga *Round Robin* (3) até chegar ao (v) servidor (8,9). No retorno o servidor (8,9) encaminha os pacotes ao (vi) sfcFC-Classifier (1), que também é o *gateway* da rede. Como último passo, portanto, estes pacotes retornam ao (vii) cliente (6,7).

Grande parte imagens a seguir foram geradas através do programa Wireshark[1].

[1] https://www.wireshark.org/

Tabela 3 – Tabela de IPs utilizados no Openstack durante os Testes com base na Figura 11

REDE CLIENTES	10.0.5.0/24
REDE SFC	10.0.3.0/24
REDE SERVIDORES	10.0.1.0/24
SFP 1	10.0.5.3
SFP 2	10.0.5.22
	10.0.5.24
SFP 3	10.0.5.21
SFP 4	10.0.5.22
CLIENTE 1 (7)	10.0.5.5
CLIENTE 2 (6)	10.0.5.12
CLASSIFIER (1)	10.0.5.3
	10.0.3.108
	10.0.1.121
FORWARDER (5)	10.0.3.106
FIREWALL (4)	10.0.3.128
BALANCEADOR DE CARGA - *ROUND ROBIN* (3)	10.0.3.107
BALANCEADOR DE CARGA - *SOURCE* IP (2)	10.0.3.102
SERVIDOR 1 (9)	10.0.1.102
SERVIDOR 2 (8)	10.0.1.106

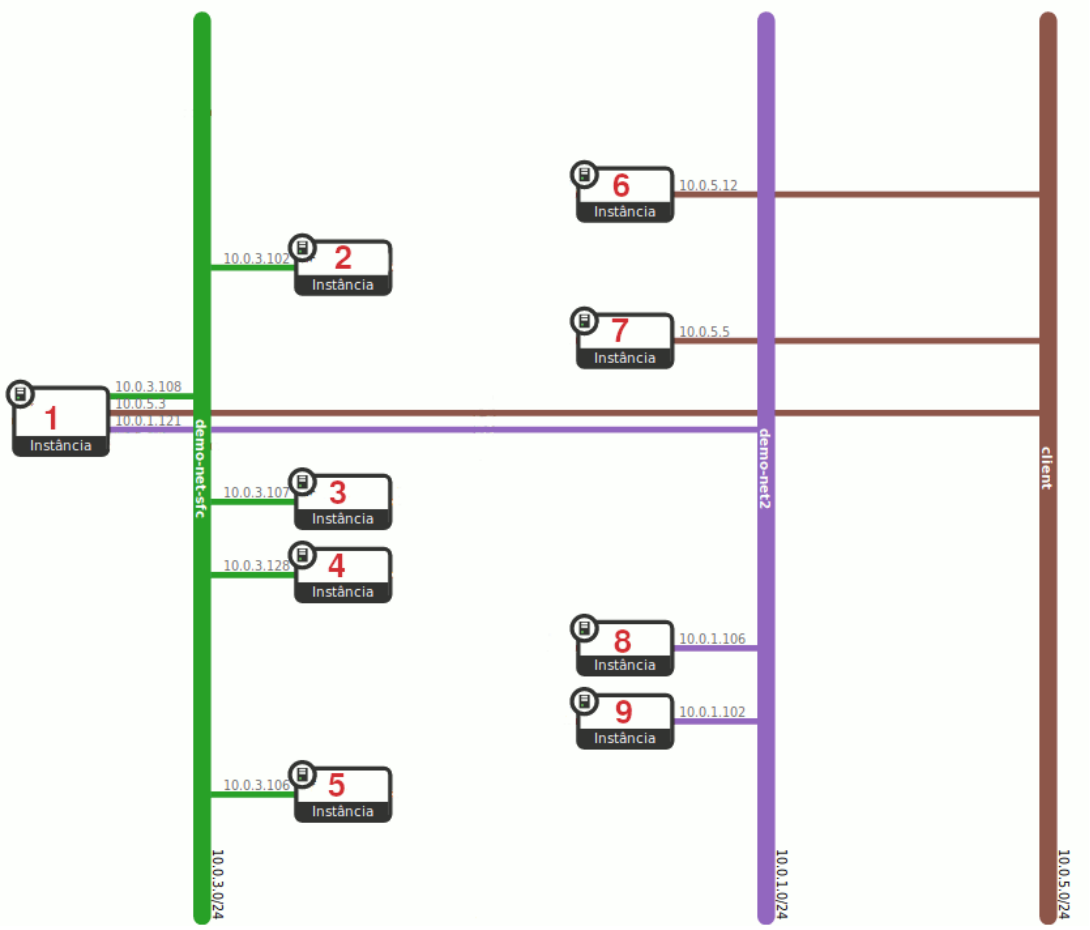


Figura 11 – Topologia no Openstack

Para estas imagens é importante apenas identificar os campo *Source*, que indica o IP de Origem, *Destination* que aponta o IP de Origem e o campo *Info*, que indica a porta de origem do cabeçalho UDP, seguido de uma seta e a porta de destino do cabeçalho UDP. As informações deste campo que vêm após estas informações mencionadas podem ser desconsideradas neste trabalho.

No.	Time	Source	Destination	Protocol	Length	Info
40	517.664685	10.0.3.108	10.0.3.106	UDP	102	1 → 0 Len=60
47	517.681439	10.0.5.3	10.0.1.106	TCP	74	54712 → 80 [SYN] Seq=0
48	517.684190	10.0.3.108	10.0.3.106	UDP	94	1 → 0 Len=52
49	517.684399	10.0.3.108	10.0.3.106	UDP	166	1 → 0 Len=124
50	517.697138	10.0.5.3	10.0.1.106	TCP	66	54712 → 80 [ACK] Seq=1
51	517.697566	10.0.5.3	10.0.1.106	HTTP	138	GET / HTTP/1.1
52	517.699627	10.0.3.108	10.0.3.106	UDP	94	1 → 0 Len=52
53	517.707856	10.0.5.3	10.0.1.106	TCP	66	54712 → 80 [ACK] Seq=7
54	518.699758	10.0.3.108	10.0.3.106	UDP	94	1 → 0 Len=52
55	518.703709	10.0.5.3	10.0.1.106	TCP	66	54712 → 80 [FIN, ACK]

Figura 12 – SFP 1 - Exemplo de requisição capturada pelo Wireshark na interface do `sfcFC-Classifier` (1) com a rede SFC

O `sfcFC-Classifier` (1) possui três interfaces de rede que interagem: (i) com os clientes, (ii) com o sfc e (iii) com os servidores. Através da Figura 12, nota-se o envio de pacotes com o encapsulamento `sfcFC` pelo `sfcFC-Classifier` (1) ao `forwarder` (5). O

cabeçalho contém SPI 1 e SI 0, como pode-se verificar no campo *Info* ao indicar as portas de origem e destino do cabeçalho UDP. É possível visualizar também pacotes oriundos do `forwarder` (5) para o servidor (9). Também é constatado que o pacote foi modificado em seu campo IP de origem para o IP do SFP 1. Isso porque o `sfcFC-Classifier` (1) realiza um NAT a fim de que o pacote seja reconhecido no ambiente `sfcFC`. O NAT é realizado de acordo com as seguintes etapas:

- No início da requisição, troca-se o IP de origem do pacote para o IP do SFP 1. Deste modo o cabeçalho IP do pacote ficará com o mesmo endereço IP de origem e destino;
- O pacote seguirá o caminho estabelecido até chegar ao balanceador de carga *Round Robin* (3). Neste ponto o endereço IP de destino será trocado por um dos servidores (8,9).
- Ao atingir um dos servidores, estes irão processar o pacote e devolvê-lo ao `sfcFC-Classifier` (1) com o endereço IP de origem do servidor.
- Deste modo, o pacote é novamente reescrito para que retorne ao elemento NAT do Fastclick com os mesmos endereços IP de quando deixou o `sfcFC-Classifier` (1). Ou seja, com o mesmo endereço IP de origem e destino. Assim, o NAT é capaz de identificar o mapeamento de endereços e devolver o pacote aos clientes (6,7).

O código responsável pelos passos seguidos pelo `sfcFC-Classifier` (1) estão na Listagem 5.1, onde das Linhas 1 a 7 são definidos os endereços utilizados, através do elemento AddressInfo. Das Linhas 9 a 16 os endereços IP são classificados pelo elemento IPClassifier. No fluxo de ida, os pacotes entram pela porta 0 e em seguida são reescritos pelo elemento IPRewriter e continuam por sua porta 0 na linha 20. Antes de deixar o `sfcFC-Classifier` (1) os pacotes têm seu MTU verificados a fim de tratar eventuais requisitos de fragmentação. Na volta, o pacote chega ao IPClassifier e em seguida é encaminhado ao segundo IPRewriter, que o devolve ao primeiro para uma última reescrita nos endereços do cabeçalho do pacote. Finalmente, o pacote sai do `sfcFC-Classifier` (1) com destino ao Cliente.

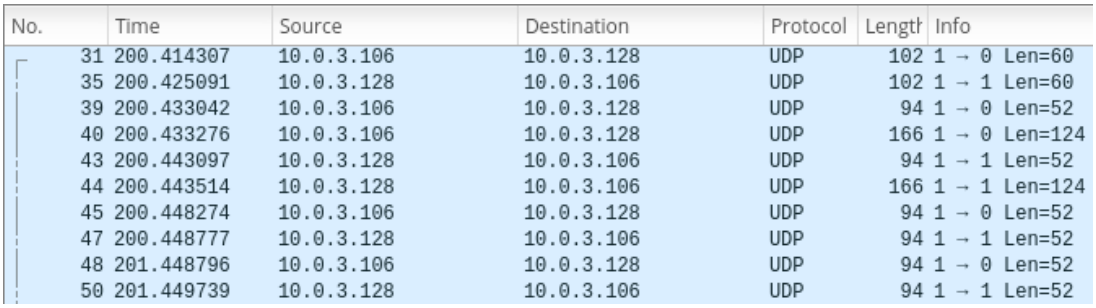
Listagem 5.1 – Trecho de código referente ao SFP 1 adaptado do `sfcFC-Classifier`(1).

```
1  AddressInfo(
2              sfp1   10.0.5.3   10.0.5.0/24       FA:16:3E:77:A3:34,
3              sfc      10.0.3.108   10.0.3.0/24     FA:16:3E:4C:AA:C7,
4              sff      10.0.3.106,
5              ws1     10.0.1.102,
6              ws2     10.0.1.106
7  );
8
9  sfcclassifier :: IPClassifier(
10              src net != net1 && dst host sfp1:ip,
```

```
11              (src host ws1 || src host ws2) && dst host sfp1:ip,
12              -);
13
14 sfcclassifier[0] -> [0]rewriterSfp1IN;
15 sfcclassifier[1] -> [0]rewriterSfp1OUT;
16 sfcclassifier[2] -> Discard;
17
18 rewriterSfp1IN :: IPRewriter(pattern sfp1:ip - - - 0 1);
19
20 rewriterSfp1IN[0] -> UDPIPEncap(sfc:ip,1,sff,0) -> checklen1;
21 rewriterSfp1IN[1] -> SetTCPChecksum() -> checklen2;
22
23 checklen1 :: CheckLength(1400);
24 checklen2 :: CheckLength(1400);
25
26 checklen1[0] -> [0]arpq1;
27 checklen1[1] -> IPFragmenter(1400) -> [0]arpq1;c
28
29 checklen2[0] -> [0]arpq2;
30 checklen2[1] -> IPFragmenter(1400) -> [0]arpq2;
31
32 rewriterSfp1OUT :: IPRewriter(pattern sfp1:ip - - - 0 1);
33
34 rewriterSfp1OUT[0] -> [0]rewriterSfp1INv;
35 rewriterSfp1OUT[1] -> Discard;
```

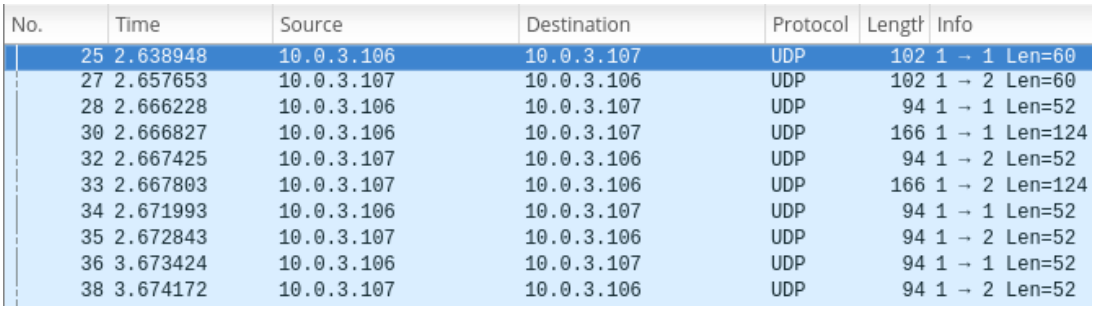
No.	Time	Source	Destination	Protocol	Length	Info
76	270.153422	10.0.3.108	10.0.3.106	UDP	102	1 → 0 Len=60
80	270.154670	10.0.3.106	10.0.3.128	UDP	102	1 → 0 Len=60
83	270.165705	10.0.3.128	10.0.3.106	UDP	102	1 → 1 Len=60
88	270.167160	10.0.3.106	10.0.3.107	UDP	102	1 → 1 Len=60
92	270.168770	10.0.3.107	10.0.3.106	UDP	102	1 → 2 Len=60
96	270.169975	10.0.5.3	10.0.1.106	TCP	74	54712 → 80 [SYN] Seq=
97	270.172844	10.0.3.108	10.0.3.106	UDP	94	1 → 0 Len=52
99	270.173049	10.0.3.108	10.0.3.106	UDP	166	1 → 0 Len=124
101	270.173495	10.0.3.106	10.0.3.128	UDP	94	1 → 0 Len=52
102	270.173731	10.0.3.106	10.0.3.128	UDP	166	1 → 0 Len=124
103	270.183603	10.0.3.128	10.0.3.106	UDP	94	1 → 1 Len=52
105	270.184036	10.0.3.128	10.0.3.106	UDP	166	1 → 1 Len=124
107	270.184303	10.0.3.106	10.0.3.107	UDP	94	1 → 1 Len=52
108	270.184484	10.0.3.106	10.0.3.107	UDP	166	1 → 1 Len=124
111	270.185325	10.0.3.107	10.0.3.106	UDP	94	1 → 2 Len=52
113	270.185729	10.0.5.3	10.0.1.106	TCP	66	54712 → 80 [ACK] Seq=
114	270.185764	10.0.3.107	10.0.3.106	UDP	166	1 → 2 Len=124
116	270.186161	10.0.5.3	10.0.1.106	HTTP	138	GET / HTTP/1.1
117	270.188286	10.0.3.108	10.0.3.106	UDP	94	1 → 0 Len=52
119	270.188729	10.0.3.106	10.0.3.128	UDP	94	1 → 0 Len=52
120	270.189278	10.0.3.128	10.0.3.106	UDP	94	1 → 1 Len=52
122	270.189654	10.0.3.106	10.0.3.107	UDP	94	1 → 1 Len=52
124	270.195985	10.0.3.107	10.0.3.106	UDP	94	1 → 2 Len=52
126	270.196449	10.0.5.3	10.0.1.106	TCP	66	54712 → 80 [ACK] Seq=
129	271.188413	10.0.3.108	10.0.3.106	UDP	94	1 → 0 Len=52
131	271.189227	10.0.3.106	10.0.3.128	UDP	94	1 → 0 Len=52
132	271.190258	10.0.3.128	10.0.3.106	UDP	94	1 → 1 Len=52
134	271.190903	10.0.3.106	10.0.3.107	UDP	94	1 → 1 Len=52
136	271.191719	10.0.3.107	10.0.3.106	UDP	94	1 → 2 Len=52
138	271.192293	10.0.5.3	10.0.1.106	TCP	66	54712 → 80 [FIN, ACK]

Figura 13 – SFP 1 - Exemplo de requisição, capturada pelo Wireshark, passando pela interface do `Forwarder` (5)



No.	Time	Source	Destination	Protocol	Length	Info
31	200.414307	10.0.3.106	10.0.3.128	UDP	102	1 → 0 Len=60
35	200.425091	10.0.3.128	10.0.3.106	UDP	102	1 → 1 Len=60
39	200.433042	10.0.3.106	10.0.3.128	UDP	94	1 → 0 Len=52
40	200.433276	10.0.3.106	10.0.3.128	UDP	166	1 → 0 Len=124
43	200.443097	10.0.3.128	10.0.3.106	UDP	94	1 → 1 Len=52
44	200.443514	10.0.3.128	10.0.3.106	UDP	166	1 → 1 Len=124
45	200.448274	10.0.3.106	10.0.3.128	UDP	94	1 → 0 Len=52
47	200.448777	10.0.3.128	10.0.3.106	UDP	94	1 → 1 Len=52
48	201.448796	10.0.3.106	10.0.3.128	UDP	94	1 → 0 Len=52
50	201.449739	10.0.3.128	10.0.3.106	UDP	94	1 → 1 Len=52

Figura 14 – SFP 1 - Requisição passando pelo *firewall* (4) capturada pelo Wireshark



No.	Time	Source	Destination	Protocol	Length	Info
25	2.638948	10.0.3.106	10.0.3.107	UDP	102	1 → 1 Len=60
27	2.657653	10.0.3.107	10.0.3.106	UDP	102	1 → 2 Len=60
28	2.666228	10.0.3.106	10.0.3.107	UDP	94	1 → 1 Len=52
30	2.666827	10.0.3.106	10.0.3.107	UDP	166	1 → 1 Len=124
32	2.667425	10.0.3.107	10.0.3.106	UDP	94	1 → 2 Len=52
33	2.667803	10.0.3.107	10.0.3.106	UDP	166	1 → 2 Len=124
34	2.671993	10.0.3.106	10.0.3.107	UDP	94	1 → 1 Len=52
35	2.672843	10.0.3.107	10.0.3.106	UDP	94	1 → 2 Len=52
36	3.673424	10.0.3.106	10.0.3.107	UDP	94	1 → 1 Len=52
38	3.674172	10.0.3.107	10.0.3.106	UDP	94	1 → 2 Len=52

Figura 15 – SFP 1 - Pacotes capturados pelo balanceador *Round Robin* (3) capturados pelo Wireshark

Passando para a execução do `forwarder` (5), é possível visualizar o encaminhamento para as duas SFs: *firewall* (4) e balanceador de carga *Round Robin* (3). Nota-se que o primeiro passo é o *firewall* (4) seguido do balanceador, SI 1 e 2, respectivamente no cabeçalho UDP dos pacotes capturados. Há também os pacotes desencapsulados enviados aos servidores já com o campo de IP destino alterado pelo balanceador. A Figura 13 exemplifica uma requisição feita por um dos clientes e o comportamento do balanceador *Round Robin* (3), escolhendo ora servidor 1 (7), ora servidor 2 (6).

Deste modo, na Figura 14, pode-se observar a troca de pacotes entre o *firewall* (4) e o `forwarder` (5). Percebe-se que o `forwarder` (5) incrementa o SI, porta de destino do pacote UDP. Já na Figura 15, pode-se constatar a troca de pacotes entre o balanceador *Round Robin* (3) e o `forwarder` (5) e o incremento do SI do cabeçalho `sfcFC`.

O trecho de código Click responsável pelo comportamento descrito acima está destacado pela Listagem 5.2. Onde no na primeira linha do IPClassifier chega o tráfego provindo do `sfcFC-Classifier` (1), na segunda do *firewall* (4) e na terceira do balanceador de carga *Round Robin* (3).

Listagem 5.2 – Trecho de código referente ao SFP 1 adaptado do `Forwarder`(5).

```
1 sfcclassifier :: IPClassifier(
2             udp && src port 1 && dst port 0, //classifier to firewall
3             udp && src port 1 && dst port 1, //firewall to lb
4             udp && src port 1 && dst port 2, //lb to ws
```

```
5                -);
6
7 sfcclassifier[0] -> StripIPHeader() -> Strip(8) -> CheckIPHeader() -> UDPIPEncap(
      sfc:ip,1,firewall,0) -> [0]arpq;
8 sfcclassifier[1] -> StripIPHeader() -> Strip(8) -> CheckIPHeader() -> UDPIPEncap(
      sfc:ip,1,loadbalancer,1) -> [0]arpq;
9 sfcclassifier[2] -> StripIPHeader() -> Strip(8) -> CheckIPHeader() ->
      SetTCPChecksum() -> SetIPAddress(router) -> [0]arpq;
```

5.2 Teste para o segundo SFP

Este teste qualitativo, o SFP 2, como já mencionado no Capítulo 4, possui similaridade com o SFP 1. A única diferença, porém, está no balanceador de carga. Por este motivo este SFP conta com dois endereços IP. Isto porque o balanceador *source* IP (2) deve escolher um servidor (8,9) baseado no IP de origem. E, como já explicado na Seção 5.1, o funcionamento do NAT faz com que os endereços IP de origem sejam trocados. Isto faria com que o balanceador realizasse o mapeamento sempre para o mesmo destino, caso somente um endereço IP fosse designado para este SFP. Deste modo, espera-se que as requisições para 10.0.5.23 sigam para um servidor, enquanto que para 10.0.5.24 sigam para outro servidor distinto.

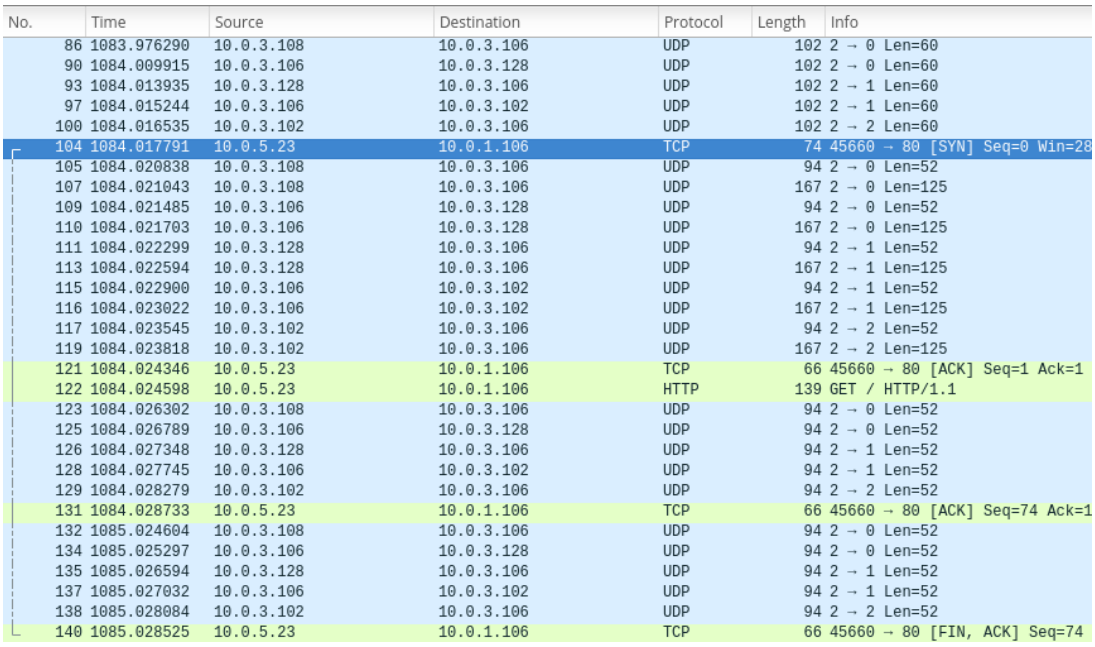
Deste modo, este SFP possui o seguinte fluxo:

1. Cliente (6,7) faz requisição;
2. A requisição é classificada pelo `sfcFC-Classifier` (1);
3. O `sfcFC-Classifier` (1) encaminha para o `forwarder` (5);
4. O tráfego passa pelas VNFs *firewall* (4) e balanceador de carga *source* IP (2);
5. A requisição chega aos servidores (8,9);
6. O `sfcFC-Classifier` (1) encaminha a resposta para os clientes (6,7).

No.	Time	Source	Destination	Protocol	Length	Info
116	1217.850652	10.0.3.108	10.0.3.106	UDP	102	2 → 0 Len=60
117	1217.853374	10.0.5.24	10.0.1.102	TCP	74	53640 → 80 [SYN] Seq=0
118	1217.855990	10.0.3.108	10.0.3.106	UDP	94	2 → 0 Len=52
119	1217.856201	10.0.3.108	10.0.3.106	UDP	167	2 → 0 Len=125
120	1217.859446	10.0.5.24	10.0.1.102	TCP	66	53640 → 80 [ACK] Seq=1
121	1217.859682	10.0.5.24	10.0.1.102	HTTP	139	GET / HTTP/1.1
122	1217.862396	10.0.3.108	10.0.3.106	UDP	94	2 → 0 Len=52
123	1217.865001	10.0.5.24	10.0.1.102	TCP	66	53640 → 80 [ACK] Seq=74
124	1218.859124	10.0.3.108	10.0.3.106	UDP	94	2 → 0 Len=52
125	1218.862974	10.0.5.24	10.0.1.102	TCP	66	53640 → 80 [FIN, ACK] Se

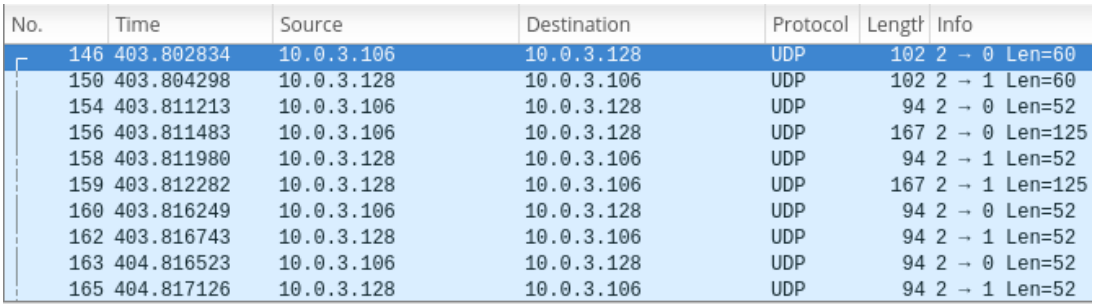
Figura 16 – SFP 2 - Requisições capturadas pelo Wireshark na interface do `sfcFC-Classifier` (1) com a rede SFC

Neste exemplo o `sfcFC-Classifier` (1) possui comportamento idêntico ao descrito no SFP 1. Percebe-se Figura 16, no entanto, o campo SFP igual a dois, detalhado através do campo *Info* do Wireshark.



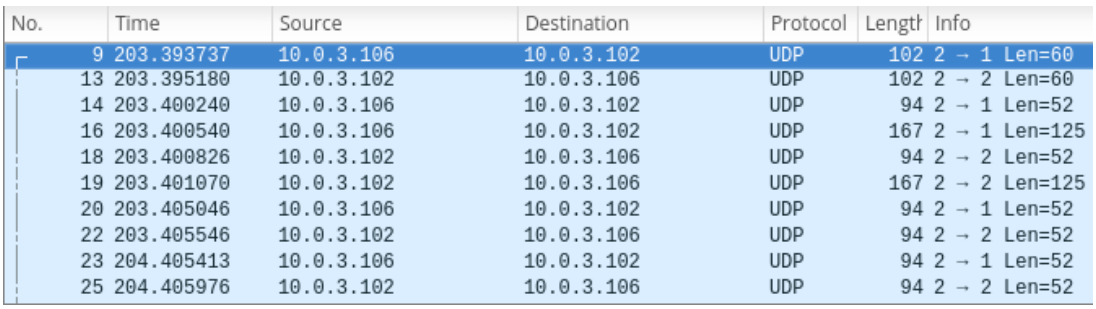
No.	Time	Source	Destination	Protocol	Length	Info
86	1083.976290	10.0.3.108	10.0.3.106	UDP	102	2 → 0 Len=60
90	1084.009915	10.0.3.106	10.0.3.128	UDP	102	2 → 0 Len=60
93	1084.013935	10.0.3.128	10.0.3.106	UDP	102	2 → 1 Len=60
97	1084.015244	10.0.3.106	10.0.3.102	UDP	102	2 → 1 Len=60
100	1084.016535	10.0.3.102	10.0.3.106	UDP	102	2 → 2 Len=60
104	1084.017791	10.0.5.23	10.0.1.106	TCP	74	45660 → 80 [SYN] Seq=0 Win=28
105	1084.020838	10.0.3.108	10.0.3.106	UDP	94	2 → 0 Len=52
107	1084.021043	10.0.3.108	10.0.3.106	UDP	167	2 → 0 Len=125
109	1084.021485	10.0.3.106	10.0.3.128	UDP	94	2 → 0 Len=52
110	1084.021703	10.0.3.106	10.0.3.128	UDP	167	2 → 0 Len=125
111	1084.022299	10.0.3.128	10.0.3.106	UDP	94	2 → 1 Len=52
113	1084.022594	10.0.3.128	10.0.3.106	UDP	167	2 → 1 Len=125
115	1084.022900	10.0.3.106	10.0.3.102	UDP	94	2 → 1 Len=52
116	1084.023022	10.0.3.106	10.0.3.102	UDP	167	2 → 1 Len=125
117	1084.023545	10.0.3.102	10.0.3.106	UDP	94	2 → 2 Len=52
119	1084.023818	10.0.3.102	10.0.3.106	UDP	167	2 → 2 Len=125
121	1084.024346	10.0.5.23	10.0.1.106	TCP	66	45660 → 80 [ACK] Seq=1 Ack=1
122	1084.024598	10.0.5.23	10.0.1.106	HTTP	139	GET / HTTP/1.1
123	1084.026302	10.0.3.108	10.0.3.106	UDP	94	2 → 0 Len=52
125	1084.026789	10.0.3.106	10.0.3.128	UDP	94	2 → 0 Len=52
126	1084.027348	10.0.3.128	10.0.3.106	UDP	94	2 → 1 Len=52
128	1084.027745	10.0.3.106	10.0.3.102	UDP	94	2 → 1 Len=52
129	1084.028279	10.0.3.102	10.0.3.106	UDP	94	2 → 2 Len=52
131	1084.028733	10.0.5.23	10.0.1.106	TCP	66	45660 → 80 [ACK] Seq=74 Ack=1
132	1085.024604	10.0.3.108	10.0.3.106	UDP	94	2 → 0 Len=52
134	1085.025297	10.0.3.106	10.0.3.128	UDP	94	2 → 0 Len=52
135	1085.026594	10.0.3.128	10.0.3.106	UDP	94	2 → 1 Len=52
137	1085.027032	10.0.3.106	10.0.3.102	UDP	94	2 → 1 Len=52
138	1085.028084	10.0.3.102	10.0.3.106	UDP	94	2 → 2 Len=52
140	1085.028525	10.0.5.23	10.0.1.106	TCP	66	45660 → 80 [FIN, ACK] Seq=74

Figura 17 – SFP 2 - Exemplo de requisição capturada pelo Wireshark, passando pela interface do `forwarder` (5)



No.	Time	Source	Destination	Protocol	Length	Info
146	403.802834	10.0.3.106	10.0.3.128	UDP	102	2 → 0 Len=60
150	403.804298	10.0.3.128	10.0.3.106	UDP	102	2 → 1 Len=60
154	403.811213	10.0.3.106	10.0.3.128	UDP	94	2 → 0 Len=52
156	403.811483	10.0.3.106	10.0.3.128	UDP	167	2 → 0 Len=125
158	403.811980	10.0.3.128	10.0.3.106	UDP	94	2 → 1 Len=52
159	403.812282	10.0.3.128	10.0.3.106	UDP	167	2 → 1 Len=125
160	403.816249	10.0.3.106	10.0.3.128	UDP	94	2 → 0 Len=52
162	403.816743	10.0.3.128	10.0.3.106	UDP	94	2 → 1 Len=52
163	404.816523	10.0.3.106	10.0.3.128	UDP	94	2 → 0 Len=52
165	404.817126	10.0.3.128	10.0.3.106	UDP	94	2 → 1 Len=52

Figura 18 – SFP 2 - Requisição passando pelo *firewall* (4) capturada pelo Wireshark



No.	Time	Source	Destination	Protocol	Length	Info
9	203.393737	10.0.3.106	10.0.3.102	UDP	102	2 → 1 Len=60
13	203.395180	10.0.3.102	10.0.3.106	UDP	102	2 → 2 Len=60
14	203.400240	10.0.3.106	10.0.3.102	UDP	94	2 → 1 Len=52
16	203.400540	10.0.3.106	10.0.3.102	UDP	167	2 → 1 Len=125
18	203.400826	10.0.3.102	10.0.3.106	UDP	94	2 → 2 Len=52
19	203.401070	10.0.3.102	10.0.3.106	UDP	167	2 → 2 Len=125
20	203.405046	10.0.3.106	10.0.3.102	UDP	94	2 → 1 Len=52
22	203.405546	10.0.3.102	10.0.3.106	UDP	94	2 → 2 Len=52
23	204.405413	10.0.3.106	10.0.3.102	UDP	94	2 → 1 Len=52
25	204.405976	10.0.3.102	10.0.3.106	UDP	94	2 → 2 Len=52

Figura 19 – SFP 2 - Requisição passando pelo balanceador *Source* IP (2) capturada pelo Wireshark

A Figura 17 exibe as requisições dos clientes vindas do `sfcFC-Classifier` (1) para o `forwarder` (5), e o comportamento do balanceador *source* IP (2). Como já mencionado, diferentemente do balanceador *Round Robin* (3), este balanceador define o destino baseado no endereço IP de origem. Assim, todas as requisições feitas ao IP 10.0.5.23 serão enviadas ao servidor (8), como é exemplificado na Figura 17 na única linha onde o campo *Protocol* é igual a HTTP.

Por fim, após passar pelas SFs, como é mostrado nas Figuras 18 e 19, a requisição alcança o seu destino final.

5.3 Teste para o terceiro SFP

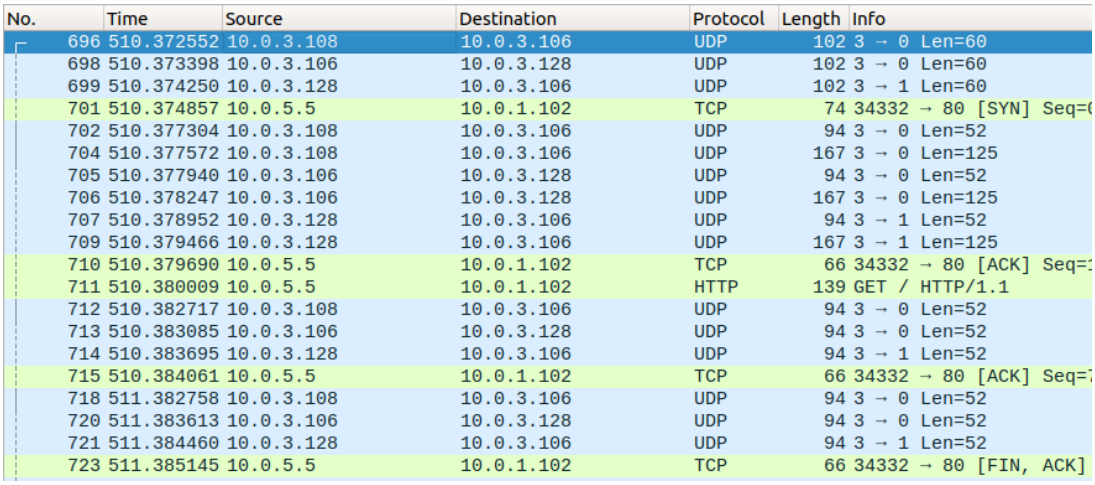
Através do IP de destino 10.0.5.21, o teste qualitativo SFP 3 percorre o seguinte caminho:

1. Cliente (7) faz uma requisição HTTP;
2. O `sfcFC-Classifier` (1) classifica o SFP e encaminha ao `forwarder` (5);
3. O tráfego passa somente pela VNFs *firewall* (4);
4. A requisição chega ao servidor (9) que encaminha resposta ao `sfcFC-Classifier` *gateway*;
5. O `sfcFC-Classifier` (1) encaminha a resposta HTTP para o cliente (7).

Diferentemente dos SFPs anteriores, este SFP não necessita de um NAT adicional no `sfcFC-Classifier` (1), uma vez que não possui SF balanceador de carga em seu caminho.

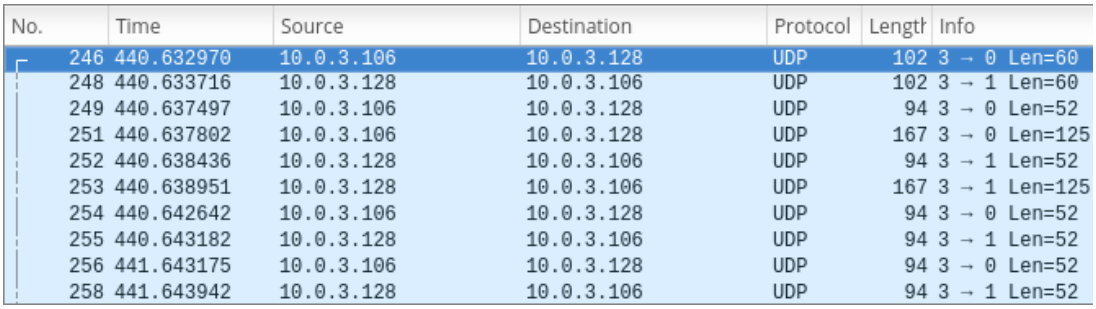
No.	Time	Source	Destination	Protocol	Length	Info
189	757.883871	10.0.3.108	10.0.3.106	UDP	102	3 → 0 Len=60
190	757.886278	10.0.5.5	10.0.1.102	TCP	74	34332 → 80 [SYN] Seq=
191	757.888644	10.0.3.108	10.0.3.106	UDP	94	3 → 0 Len=52
192	757.888913	10.0.3.108	10.0.3.106	UDP	167	3 → 0 Len=125
193	757.891105	10.0.5.5	10.0.1.102	TCP	66	34332 → 80 [ACK] Seq=
194	757.891418	10.0.5.5	10.0.1.102	HTTP	139	GET / HTTP/1.1
195	757.894063	10.0.3.108	10.0.3.106	UDP	94	3 → 0 Len=52
196	757.895469	10.0.5.5	10.0.1.102	TCP	66	34332 → 80 [ACK] Seq=
197	758.894094	10.0.3.108	10.0.3.106	UDP	94	3 → 0 Len=52
198	758.896559	10.0.5.5	10.0.1.102	TCP	66	34332 → 80 [FIN, ACK]

Figura 20 – SFP 3 - Requisições capturadas pelo Wireshark na interface do `sfcFC-Classifier` (1) com a rede SFC



No.	Time	Source	Destination	Protocol	Length	Info
696	510.372552	10.0.3.108	10.0.3.106	UDP	102	3 → 0 Len=60
698	510.373398	10.0.3.106	10.0.3.128	UDP	102	3 → 0 Len=60
699	510.374250	10.0.3.128	10.0.3.106	UDP	102	3 → 1 Len=60
701	510.374857	10.0.5.5	10.0.1.102	TCP	74	34332 → 80 [SYN] Seq=(
702	510.377304	10.0.3.108	10.0.3.106	UDP	94	3 → 0 Len=52
704	510.377572	10.0.3.108	10.0.3.106	UDP	167	3 → 0 Len=125
705	510.377940	10.0.3.106	10.0.3.128	UDP	94	3 → 0 Len=52
706	510.378247	10.0.3.106	10.0.3.128	UDP	167	3 → 0 Len=125
707	510.378952	10.0.3.128	10.0.3.106	UDP	94	3 → 1 Len=52
709	510.379466	10.0.3.128	10.0.3.106	UDP	167	3 → 1 Len=125
710	510.379690	10.0.5.5	10.0.1.102	TCP	66	34332 → 80 [ACK] Seq=1
711	510.380009	10.0.5.5	10.0.1.102	HTTP	139	GET / HTTP/1.1
712	510.382717	10.0.3.108	10.0.3.106	UDP	94	3 → 0 Len=52
713	510.383085	10.0.3.106	10.0.3.128	UDP	94	3 → 0 Len=52
714	510.383695	10.0.3.128	10.0.3.106	UDP	94	3 → 1 Len=52
715	510.384061	10.0.5.5	10.0.1.102	TCP	66	34332 → 80 [ACK] Seq=7
718	511.382758	10.0.3.108	10.0.3.106	UDP	94	3 → 0 Len=52
720	511.383613	10.0.3.106	10.0.3.128	UDP	94	3 → 0 Len=52
721	511.384460	10.0.3.128	10.0.3.106	UDP	94	3 → 1 Len=52
723	511.385145	10.0.5.5	10.0.1.102	TCP	66	34332 → 80 [FIN, ACK]

Figura 21 – SFP 3 - Exemplo de requisição, capturada pelo Wireshark, passando pela interface do `forwarder` (5)



No.	Time	Source	Destination	Protocol	Lengtl	Info
246	440.632970	10.0.3.106	10.0.3.128	UDP	102	3 → 0 Len=60
248	440.633716	10.0.3.128	10.0.3.106	UDP	102	3 → 1 Len=60
249	440.637497	10.0.3.106	10.0.3.128	UDP	94	3 → 0 Len=52
251	440.637802	10.0.3.106	10.0.3.128	UDP	167	3 → 0 Len=125
252	440.638436	10.0.3.128	10.0.3.106	UDP	94	3 → 1 Len=52
253	440.638951	10.0.3.128	10.0.3.106	UDP	167	3 → 1 Len=125
254	440.642642	10.0.3.106	10.0.3.128	UDP	94	3 → 0 Len=52
255	440.643182	10.0.3.128	10.0.3.106	UDP	94	3 → 1 Len=52
256	441.643175	10.0.3.106	10.0.3.128	UDP	94	3 → 0 Len=52
258	441.643942	10.0.3.128	10.0.3.106	UDP	94	3 → 1 Len=52

Figura 22 – SFP 3 - Requisição passando pelo *firewall* (4) capturada pelo Wireshark

Para esta SFP, o `sfcFC-Classifier` (1) apenas reescreve o endereço de destino para o IP do servidor (9). É importante ressaltar também a identificação do SFP através do campo *info* da Figura 20, onde os pacotes estão sendo classificados e enviados ao `forwarder`. Este, por sua vez, encaminha as requisições para o *firewall*, como pode ser observado nas Figuras 21 e 22.

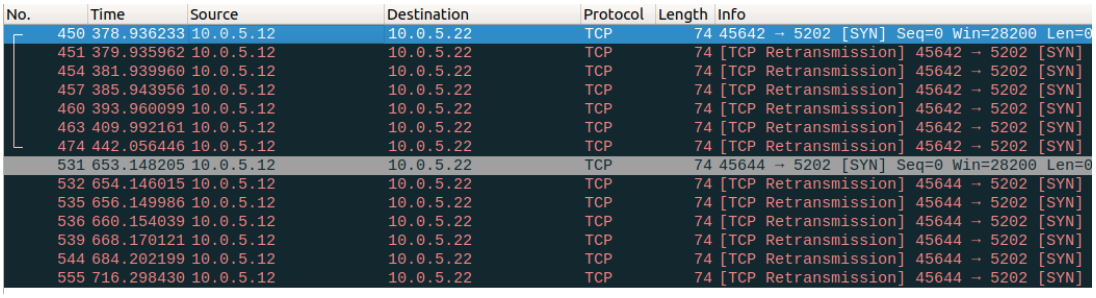
5.4 Teste para o quarto SFP

No quarto SFP o teste quantitativo foi feito utilizando o iperf3[2]. Neste programa um servidor recebe requisições na porta 5201 e um cliente envia uma rajada de pacote com determinada taxa de transferência. A SF *firewall* para este caso apenas permite requisições na porta onde o servidor iperf3 opera. Semelhante ao SFP anterior, o SFP 4 segue o mesmo fluxo utilizando o IP de destino 10.0.5.22. Desta forma, o cliente executa o comando iperf3 como cliente, enviando pacotes pelo comando: `iperf3 -c 10.0.5.22 -l 1300 -u`. As requisições chegam no servidor (8) que, por sua vez, executa o comando iperf3 -s. Um detalhe a ser

[2] https://iperf.fr/

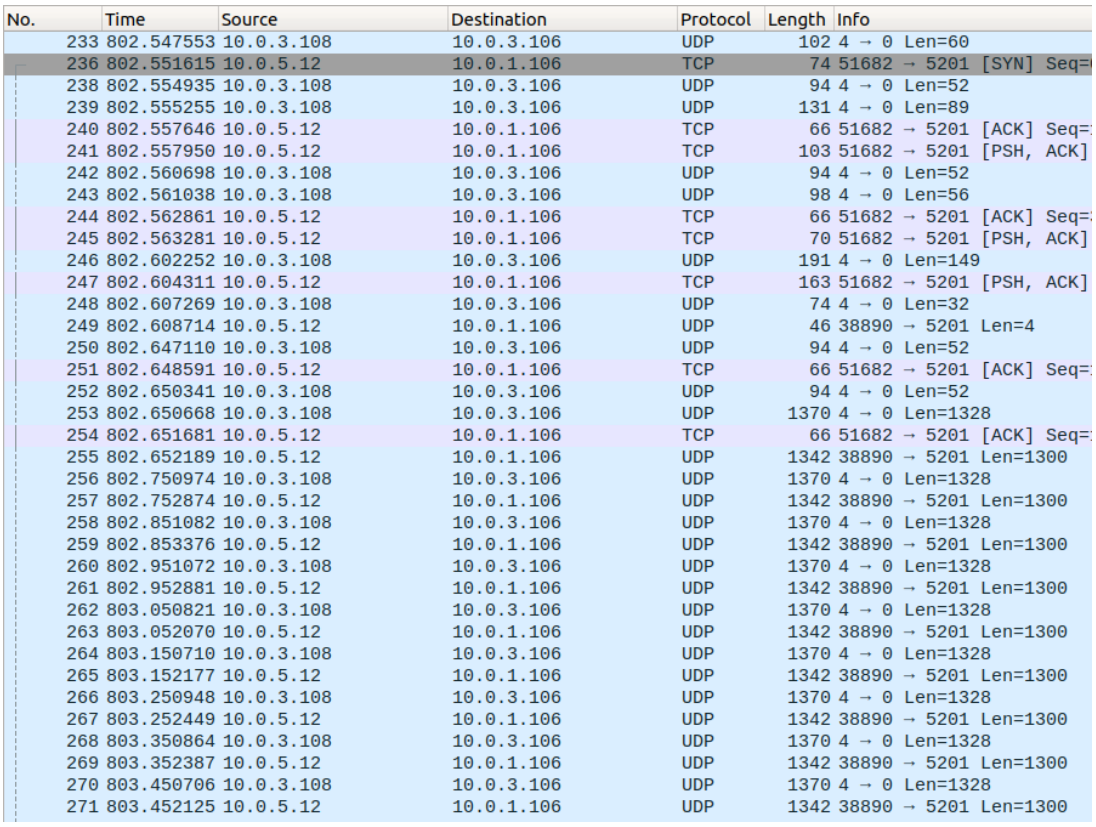
evidenciado é que para este teste foi necessário configurar o MTU dos pacotes para 1300, através da opção –l 1300 do comando.

Neste teste foram realizados testes a fim de averiguar o comportamento do `firewall`. A requisição foi feita na porta 5202, no entanto para este teste o `firewall` está configurado para permitir apenas pacotes com destino a porta 5201. A Figura 23 mostra que os pacotes foram barrados dada a retransmissão TCP.



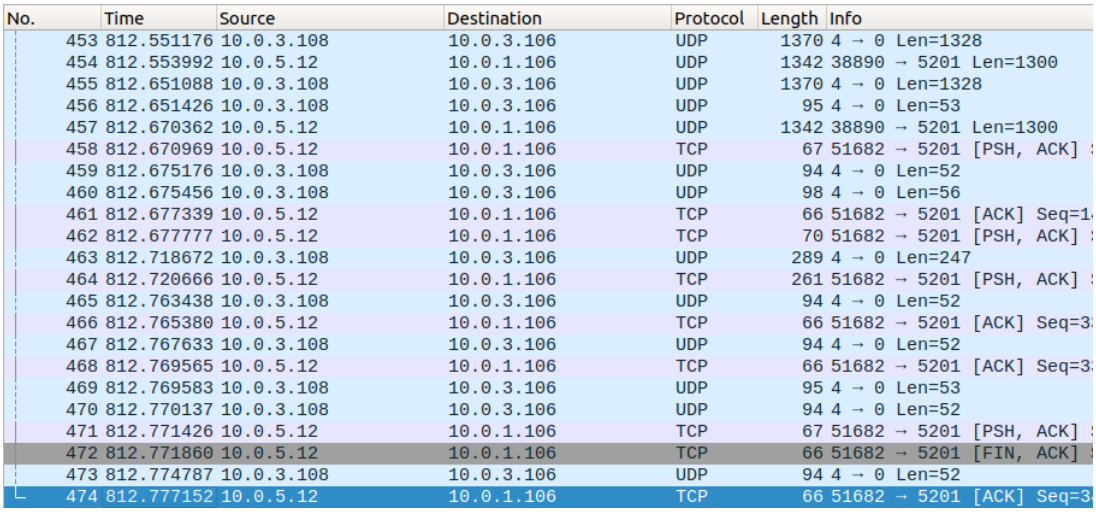
No.	Time	Source	Destination	Protocol	Length	Info
450	378.936233	10.0.5.12	10.0.5.22	TCP	74	45642 → 5202 [SYN] Seq=0 Win=28200 Len=0
451	379.935962	10.0.5.12	10.0.5.22	TCP	74	[TCP Retransmission] 45642 → 5202 [SYN]
454	381.939960	10.0.5.12	10.0.5.22	TCP	74	[TCP Retransmission] 45642 → 5202 [SYN]
457	385.943956	10.0.5.12	10.0.5.22	TCP	74	[TCP Retransmission] 45642 → 5202 [SYN]
460	393.960099	10.0.5.12	10.0.5.22	TCP	74	[TCP Retransmission] 45642 → 5202 [SYN]
463	409.992161	10.0.5.12	10.0.5.22	TCP	74	[TCP Retransmission] 45642 → 5202 [SYN]
474	442.056446	10.0.5.12	10.0.5.22	TCP	74	[TCP Retransmission] 45642 → 5202 [SYN]
531	653.148205	10.0.5.12	10.0.5.22	TCP	74	45644 → 5202 [SYN] Seq=0 Win=28200 Len=0
532	654.146015	10.0.5.12	10.0.5.22	TCP	74	[TCP Retransmission] 45644 → 5202 [SYN]
535	656.149986	10.0.5.12	10.0.5.22	TCP	74	[TCP Retransmission] 45644 → 5202 [SYN]
536	660.154039	10.0.5.12	10.0.5.22	TCP	74	[TCP Retransmission] 45644 → 5202 [SYN]
539	668.170121	10.0.5.12	10.0.5.22	TCP	74	[TCP Retransmission] 45644 → 5202 [SYN]
544	684.202199	10.0.5.12	10.0.5.22	TCP	74	[TCP Retransmission] 45644 → 5202 [SYN]
555	716.298430	10.0.5.12	10.0.5.22	TCP	74	[TCP Retransmission] 45644 → 5202 [SYN]

Figura 23 – Requisições do Cliente para o SFP 4 capturadas pelo Wireshark negadas pelo *firewall* (4)



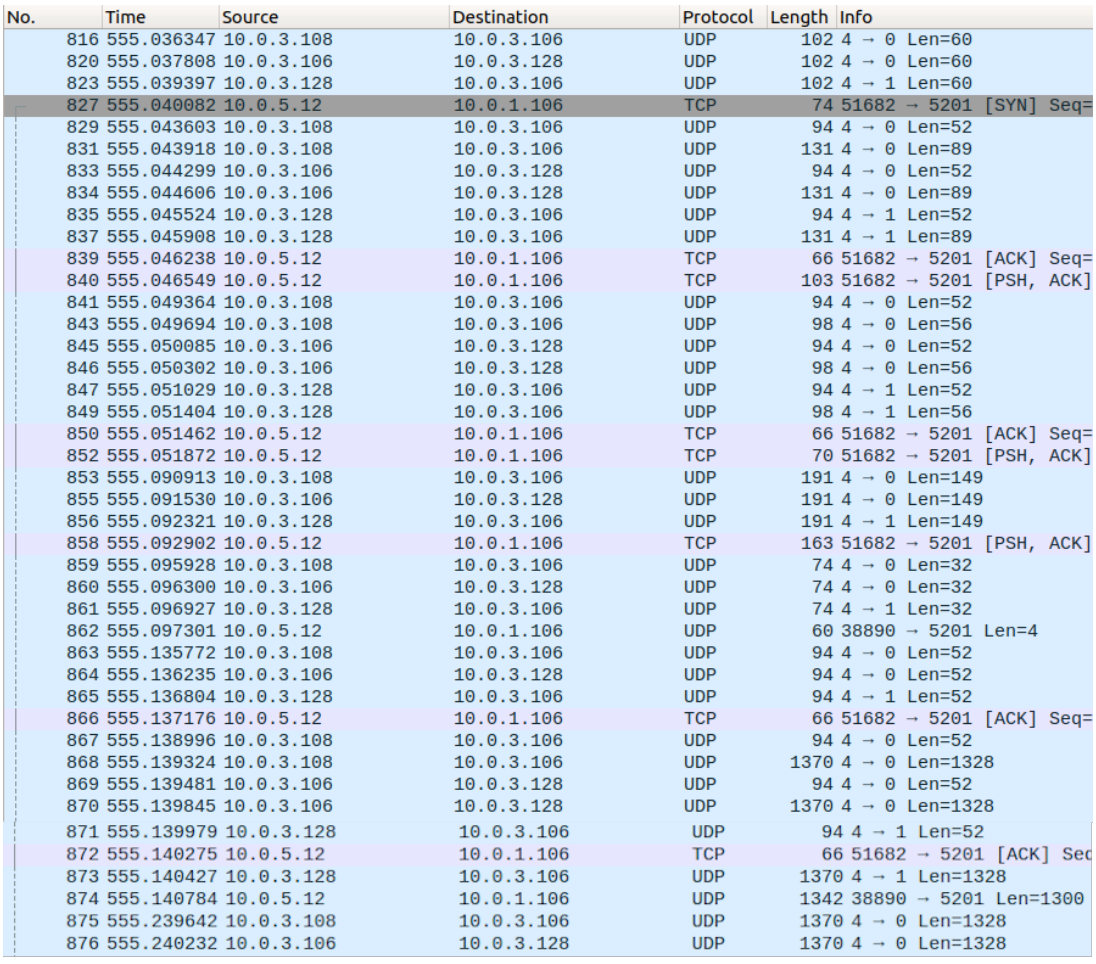
No.	Time	Source	Destination	Protocol	Length	Info
233	802.547553	10.0.3.108	10.0.3.106	UDP	102	4 → 0 Len=60
236	802.551615	10.0.5.12	10.0.1.106	TCP	74	51682 → 5201 [SYN] Seq=
238	802.554935	10.0.3.108	10.0.3.106	UDP	94	4 → 0 Len=52
239	802.555255	10.0.3.108	10.0.3.106	UDP	131	4 → 0 Len=89
240	802.557646	10.0.5.12	10.0.1.106	TCP	66	51682 → 5201 [ACK] Seq=
241	802.557950	10.0.5.12	10.0.1.106	TCP	103	51682 → 5201 [PSH, ACK]
242	802.560698	10.0.3.108	10.0.3.106	UDP	94	4 → 0 Len=52
243	802.561038	10.0.3.108	10.0.3.106	UDP	98	4 → 0 Len=56
244	802.562861	10.0.5.12	10.0.1.106	TCP	66	51682 → 5201 [ACK] Seq=
245	802.563281	10.0.5.12	10.0.1.106	TCP	70	51682 → 5201 [PSH, ACK]
246	802.602252	10.0.3.108	10.0.3.106	UDP	191	4 → 0 Len=149
247	802.604311	10.0.5.12	10.0.1.106	TCP	163	51682 → 5201 [PSH, ACK]
248	802.607269	10.0.3.108	10.0.3.106	UDP	74	4 → 0 Len=32
249	802.608714	10.0.5.12	10.0.1.106	UDP	46	38890 → 5201 Len=4
250	802.647110	10.0.3.108	10.0.3.106	UDP	94	4 → 0 Len=52
251	802.648591	10.0.5.12	10.0.1.106	TCP	66	51682 → 5201 [ACK] Seq=
252	802.650341	10.0.3.108	10.0.3.106	UDP	94	4 → 0 Len=52
253	802.650668	10.0.3.108	10.0.3.106	UDP	1370	4 → 0 Len=1328
254	802.651681	10.0.5.12	10.0.1.106	TCP	66	51682 → 5201 [ACK] Seq=
255	802.652189	10.0.5.12	10.0.1.106	UDP	1342	38890 → 5201 Len=1300
256	802.750974	10.0.3.108	10.0.3.106	UDP	1370	4 → 0 Len=1328
257	802.752874	10.0.5.12	10.0.1.106	UDP	1342	38890 → 5201 Len=1300
258	802.851082	10.0.3.108	10.0.3.106	UDP	1370	4 → 0 Len=1328
259	802.853376	10.0.5.12	10.0.1.106	UDP	1342	38890 → 5201 Len=1300
260	802.951072	10.0.3.108	10.0.3.106	UDP	1370	4 → 0 Len=1328
261	802.952881	10.0.5.12	10.0.1.106	UDP	1342	38890 → 5201 Len=1300
262	803.050821	10.0.3.108	10.0.3.106	UDP	1370	4 → 0 Len=1328
263	803.052070	10.0.5.12	10.0.1.106	UDP	1342	38890 → 5201 Len=1300
264	803.150710	10.0.3.108	10.0.3.106	UDP	1370	4 → 0 Len=1328
265	803.152177	10.0.5.12	10.0.1.106	UDP	1342	38890 → 5201 Len=1300
266	803.250948	10.0.3.108	10.0.3.106	UDP	1370	4 → 0 Len=1328
267	803.252449	10.0.5.12	10.0.1.106	UDP	1342	38890 → 5201 Len=1300
268	803.350864	10.0.3.108	10.0.3.106	UDP	1370	4 → 0 Len=1328
269	803.352387	10.0.5.12	10.0.1.106	UDP	1342	38890 → 5201 Len=1300
270	803.450706	10.0.3.108	10.0.3.106	UDP	1370	4 → 0 Len=1328
271	803.452125	10.0.5.12	10.0.1.106	UDP	1342	38890 → 5201 Len=1300

Figura 24 – SFP 4 - Início da requisição capturada pelo Wireshark na interface do `sfcFC-Classifier` (1) com a rede SFC



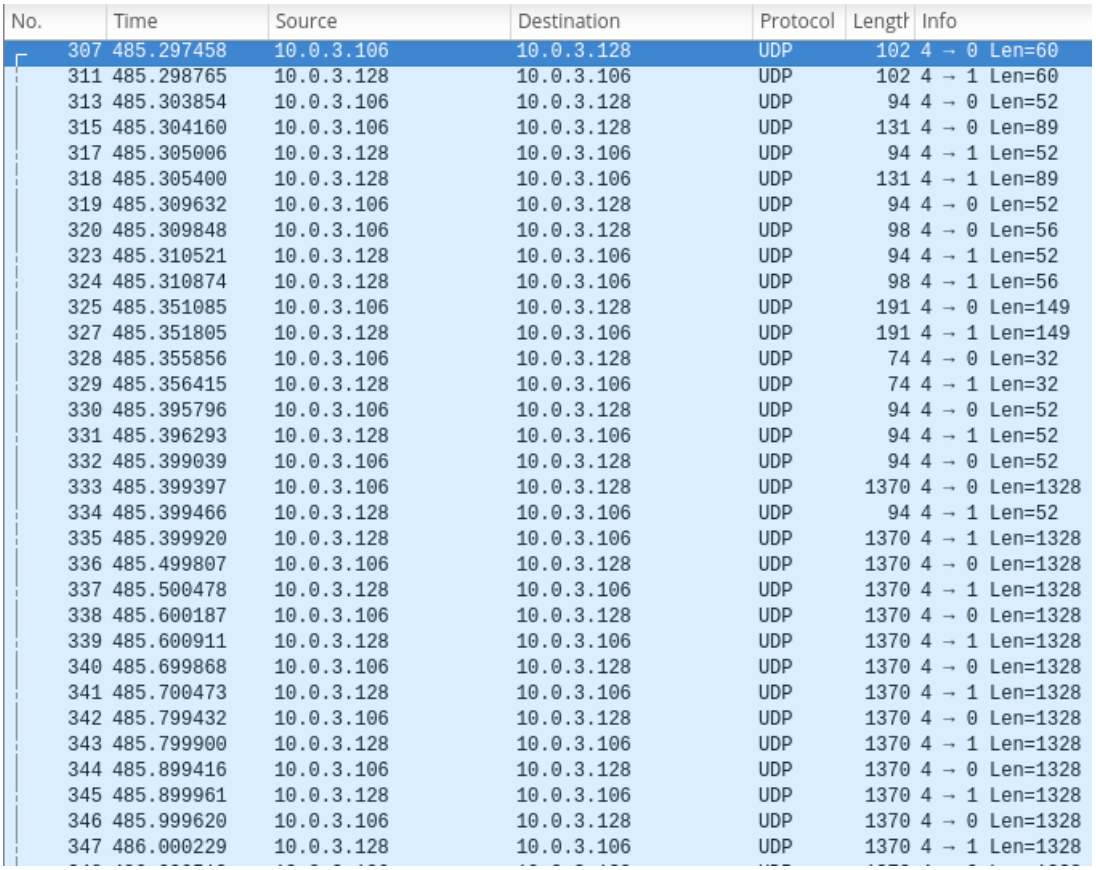
No.	Time	Source	Destination	Protocol	Length	Info
453	812.551176	10.0.3.108	10.0.3.106	UDP	1370	4 → 0 Len=1328
454	812.553992	10.0.5.12	10.0.1.106	UDP	1342	38890 → 5201 Len=1300
455	812.651088	10.0.3.108	10.0.3.106	UDP	1370	4 → 0 Len=1328
456	812.651426	10.0.3.108	10.0.3.106	UDP	95	4 → 0 Len=53
457	812.670362	10.0.5.12	10.0.1.106	UDP	1342	38890 → 5201 Len=1300
458	812.670969	10.0.5.12	10.0.1.106	TCP	67	51682 → 5201 [PSH, ACK]
459	812.675176	10.0.3.108	10.0.3.106	UDP	94	4 → 0 Len=52
460	812.675456	10.0.3.108	10.0.3.106	UDP	98	4 → 0 Len=56
461	812.677339	10.0.5.12	10.0.1.106	TCP	66	51682 → 5201 [ACK] Seq=1
462	812.677777	10.0.5.12	10.0.1.106	TCP	70	51682 → 5201 [PSH, ACK]
463	812.718672	10.0.3.108	10.0.3.106	UDP	289	4 → 0 Len=247
464	812.720666	10.0.5.12	10.0.1.106	TCP	261	51682 → 5201 [PSH, ACK]
465	812.763438	10.0.3.108	10.0.3.106	UDP	94	4 → 0 Len=52
466	812.765380	10.0.5.12	10.0.1.106	TCP	66	51682 → 5201 [ACK] Seq=3
467	812.767633	10.0.3.108	10.0.3.106	UDP	94	4 → 0 Len=52
468	812.769565	10.0.5.12	10.0.1.106	TCP	66	51682 → 5201 [ACK] Seq=3
469	812.769583	10.0.3.108	10.0.3.106	UDP	95	4 → 0 Len=53
470	812.770137	10.0.3.108	10.0.3.106	UDP	94	4 → 0 Len=52
471	812.771426	10.0.5.12	10.0.1.106	TCP	67	51682 → 5201 [PSH, ACK]
472	812.771860	10.0.5.12	10.0.1.106	TCP	66	51682 → 5201 [FIN, ACK]
473	812.774787	10.0.3.108	10.0.3.106	UDP	94	4 → 0 Len=52
474	812.777152	10.0.5.12	10.0.1.106	TCP	66	51682 → 5201 [ACK] Seq=3

Figura 25 – SFP 4 - Final da requisição capturada pelo Wireshark na interface do `sfcFC-Classifier` (1) com a rede SFC



No.	Time	Source	Destination	Protocol	Length	Info
816	555.036347	10.0.3.108	10.0.3.106	UDP	102	4 → 0 Len=60
820	555.037808	10.0.3.106	10.0.3.128	UDP	102	4 → 0 Len=60
823	555.039397	10.0.3.128	10.0.3.106	UDP	102	4 → 1 Len=60
827	555.040082	10.0.5.12	10.0.1.106	TCP	74	51682 → 5201 [SYN] Seq=
829	555.043603	10.0.3.108	10.0.3.106	UDP	94	4 → 0 Len=52
831	555.043918	10.0.3.108	10.0.3.106	UDP	131	4 → 0 Len=89
833	555.044299	10.0.3.106	10.0.3.128	UDP	94	4 → 0 Len=52
834	555.044606	10.0.3.106	10.0.3.128	UDP	131	4 → 0 Len=89
835	555.045524	10.0.3.128	10.0.3.106	UDP	94	4 → 1 Len=52
837	555.045908	10.0.3.128	10.0.3.106	UDP	131	4 → 1 Len=89
839	555.046238	10.0.5.12	10.0.1.106	TCP	66	51682 → 5201 [ACK] Seq=
840	555.046549	10.0.5.12	10.0.1.106	TCP	103	51682 → 5201 [PSH, ACK]
841	555.049364	10.0.3.108	10.0.3.106	UDP	94	4 → 0 Len=52
843	555.049694	10.0.3.108	10.0.3.106	UDP	98	4 → 0 Len=56
845	555.050085	10.0.3.106	10.0.3.128	UDP	94	4 → 0 Len=52
846	555.050302	10.0.3.106	10.0.3.128	UDP	98	4 → 0 Len=56
847	555.051029	10.0.3.128	10.0.3.106	UDP	94	4 → 1 Len=52
849	555.051404	10.0.3.128	10.0.3.106	UDP	98	4 → 1 Len=56
850	555.051462	10.0.5.12	10.0.1.106	TCP	66	51682 → 5201 [ACK] Seq=
852	555.051872	10.0.5.12	10.0.1.106	TCP	70	51682 → 5201 [PSH, ACK]
853	555.090913	10.0.3.108	10.0.3.106	UDP	191	4 → 0 Len=149
855	555.091530	10.0.3.106	10.0.3.128	UDP	191	4 → 0 Len=149
856	555.092321	10.0.3.128	10.0.3.106	UDP	191	4 → 1 Len=149
858	555.092902	10.0.5.12	10.0.1.106	TCP	163	51682 → 5201 [PSH, ACK]
859	555.095928	10.0.3.108	10.0.3.106	UDP	74	4 → 0 Len=32
860	555.096300	10.0.3.106	10.0.3.128	UDP	74	4 → 0 Len=32
861	555.096927	10.0.3.128	10.0.3.106	UDP	74	4 → 1 Len=32
862	555.097301	10.0.5.12	10.0.1.106	UDP	60	38890 → 5201 Len=4
863	555.135772	10.0.3.108	10.0.3.106	UDP	94	4 → 0 Len=52
864	555.136235	10.0.3.106	10.0.3.128	UDP	94	4 → 0 Len=52
865	555.136804	10.0.3.128	10.0.3.106	UDP	94	4 → 1 Len=52
866	555.137176	10.0.5.12	10.0.1.106	TCP	66	51682 → 5201 [ACK] Seq=
867	555.138996	10.0.3.108	10.0.3.106	UDP	94	4 → 0 Len=52
868	555.139324	10.0.3.108	10.0.3.106	UDP	1370	4 → 0 Len=1328
869	555.139481	10.0.3.106	10.0.3.128	UDP	94	4 → 0 Len=52
870	555.139845	10.0.3.106	10.0.3.128	UDP	1370	4 → 0 Len=1328
871	555.139979	10.0.3.128	10.0.3.106	UDP	94	4 → 1 Len=52
872	555.140275	10.0.5.12	10.0.1.106	TCP	66	51682 → 5201 [ACK] Seq
873	555.140427	10.0.3.128	10.0.3.106	UDP	1370	4 → 1 Len=1328
874	555.140784	10.0.5.12	10.0.1.106	UDP	1342	38890 → 5201 Len=1300
875	555.239642	10.0.3.108	10.0.3.106	UDP	1370	4 → 0 Len=1328
876	555.240232	10.0.3.106	10.0.3.128	UDP	1370	4 → 0 Len=1328

Figura 26 – SFP 4 - Exemplo de início de uma requisição, capturada pelo Wireshark, passando pela interface do `forwarder` (5)



No.	Time	Source	Destination	Protocol	Length	Info
307	485.297458	10.0.3.106	10.0.3.128	UDP	102	4 → 0 Len=60
311	485.298765	10.0.3.128	10.0.3.106	UDP	102	4 → 1 Len=60
313	485.303854	10.0.3.106	10.0.3.128	UDP	94	4 → 0 Len=52
315	485.304160	10.0.3.106	10.0.3.128	UDP	131	4 → 0 Len=89
317	485.305006	10.0.3.128	10.0.3.106	UDP	94	4 → 1 Len=52
318	485.305400	10.0.3.128	10.0.3.106	UDP	131	4 → 1 Len=89
319	485.309632	10.0.3.106	10.0.3.128	UDP	94	4 → 0 Len=52
320	485.309848	10.0.3.106	10.0.3.128	UDP	98	4 → 0 Len=56
323	485.310521	10.0.3.128	10.0.3.106	UDP	94	4 → 1 Len=52
324	485.310874	10.0.3.128	10.0.3.106	UDP	98	4 → 1 Len=56
325	485.351085	10.0.3.106	10.0.3.128	UDP	191	4 → 0 Len=149
327	485.351805	10.0.3.128	10.0.3.106	UDP	191	4 → 1 Len=149
328	485.355856	10.0.3.106	10.0.3.128	UDP	74	4 → 0 Len=32
329	485.356415	10.0.3.128	10.0.3.106	UDP	74	4 → 1 Len=32
330	485.395796	10.0.3.106	10.0.3.128	UDP	94	4 → 0 Len=52
331	485.396293	10.0.3.128	10.0.3.106	UDP	94	4 → 1 Len=52
332	485.399039	10.0.3.106	10.0.3.128	UDP	94	4 → 0 Len=52
333	485.399397	10.0.3.106	10.0.3.128	UDP	1370	4 → 0 Len=1328
334	485.399466	10.0.3.128	10.0.3.106	UDP	94	4 → 1 Len=52
335	485.399920	10.0.3.128	10.0.3.106	UDP	1370	4 → 1 Len=1328
336	485.499807	10.0.3.106	10.0.3.128	UDP	1370	4 → 0 Len=1328
337	485.500478	10.0.3.128	10.0.3.106	UDP	1370	4 → 1 Len=1328
338	485.600187	10.0.3.106	10.0.3.128	UDP	1370	4 → 0 Len=1328
339	485.600911	10.0.3.128	10.0.3.106	UDP	1370	4 → 1 Len=1328
340	485.699868	10.0.3.106	10.0.3.128	UDP	1370	4 → 0 Len=1328
341	485.700473	10.0.3.128	10.0.3.106	UDP	1370	4 → 1 Len=1328
342	485.799432	10.0.3.106	10.0.3.128	UDP	1370	4 → 0 Len=1328
343	485.799900	10.0.3.128	10.0.3.106	UDP	1370	4 → 1 Len=1328
344	485.899416	10.0.3.106	10.0.3.128	UDP	1370	4 → 0 Len=1328
345	485.899961	10.0.3.128	10.0.3.106	UDP	1370	4 → 1 Len=1328
346	485.999620	10.0.3.106	10.0.3.128	UDP	1370	4 → 0 Len=1328
347	486.000229	10.0.3.128	10.0.3.106	UDP	1370	4 → 1 Len=1328

Figura 27 – SFP 4 - Início da requisição passando pelo *firewall* (4) capturada pelo Wireshark

Das Figuras 24 a 27 o pacote percorre o caminho dentro do ambiente SFC: `sfcFC-Classifier` (1), `forwarder` (5), *firewall* (4). Nestas imagens é possível perceber através do campo *info* a dinâmica dos campos SFP e SI.

5.5 Avaliação de Desempenho do `sfcFC`

Os testes para os quatro SFPs cumpriram o esperado. Todos os caminhos foram seguidos e o SFC seguido de acordo com as regras pré-estabelecidas, o que comprova a eficácia do `sfcFC` em entregar o que foi proposto.

A fim de analisar o impacto do uso de SFC na rede, também foram realizados testes sem o processamento SFC. Ou seja, passando apenas pelo *gateway* da rede. Para todos os testes foram coletados resultados do iperf3 utilizando rajadas de 10MB/s a 256MB/s dentro da mesma topologia criada para o SFP 4. É importante ressaltar que o comando utilizado foi semelhante ao da seção 5.4, adicionando apenas a opção `-l` para definir a largura de banda. Para cada amostra da largura de banda foi extraída a média de três execuções.

5.5.1 Teste comparativo avaliando a perda de pacotes

Com o iperf3, pode-se extrair uma medidas de suma importância para a avaliação de desempenho de redes, a taxa de perda de pacotes. A Figura 28 ilustra o impacto da perda de pacotes quando se utiliza a abordagem com uso do SFC implementada neste trabalho. Em contraponto, a Figura 29 mostra os dados obtidos sem a implementação do SFC. Para este teste, o fluxo passa apenas pela máquina virtual do `sfcFC-Classifier`, que também é o *gateway* da rede.

Figura 28 – Taxa de perda(%) de pacotes na abordagem com o uso de SFC

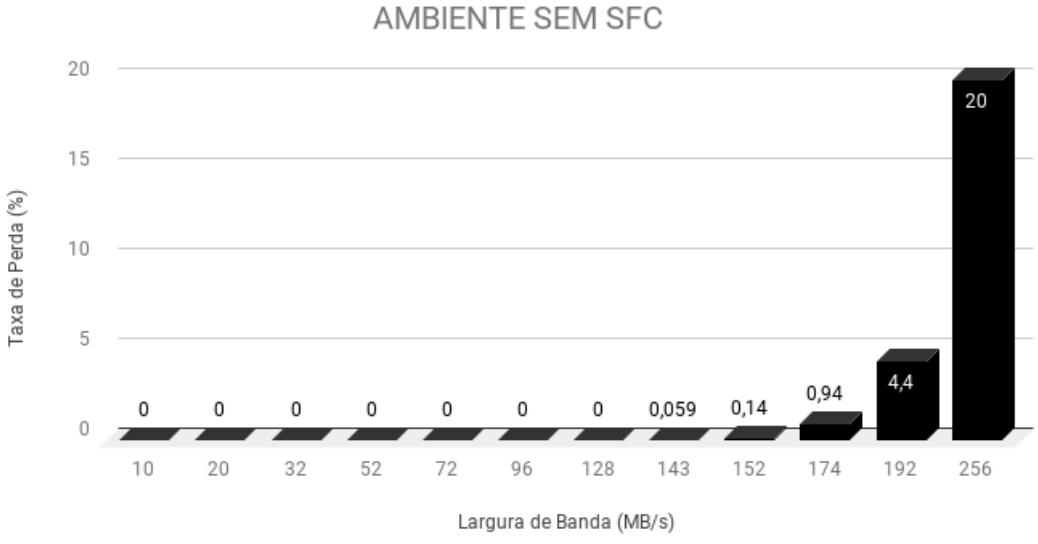


Figura 29 – Taxa de perda(%) de pacotes na abordagem sem o uso de SFC

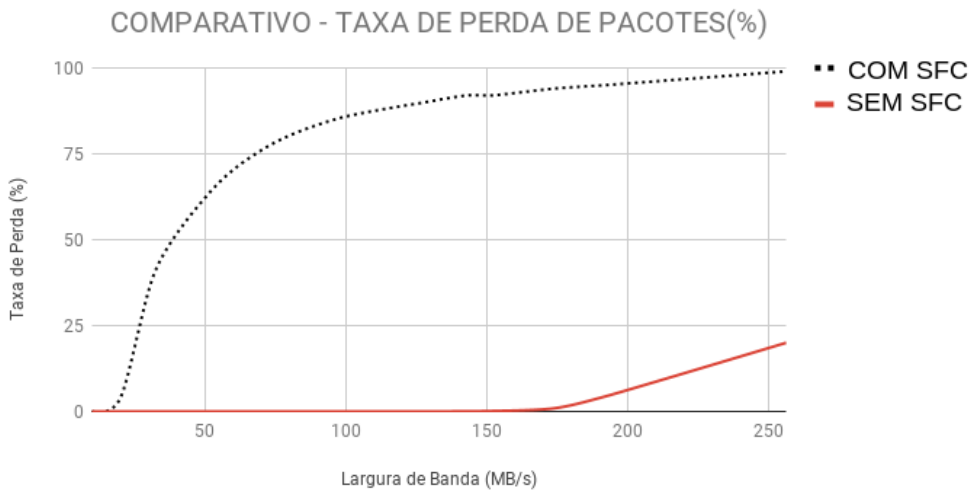


Figura 30 – Comparativo da Taxa de Perda(%) de pacotes nos ambientes com e sem uso do SFC

Através da Figura 30 pode-se notar a grande perda de pacotes da abordagem com uso do SFC na medida em que a largura de banda aumenta. O gráfico comparativo entre as abordagens com e sem SFC foi gerado com base nos gráficos das Figuras 28 e 28. De fato, a perda de pacotes pode vir a ser um gargalo em ambientes SFC. Sobretudo nas abordagens que se inspiram no NSH e adicionam um cabeçalho ao pacote (KULKARNI et al., 2017).

6 Conclusão

Este capitulo apresenta uma avaliação final e elenca os prós e contras do `sfcFC`, além de apresentar sugestões de trabalhos futuros na segunda seção.

6.1 Considerações Finais

O `sfcFC` tem como proposta implementar elementos do SFC utilizando Fastclick de forma que o encadeamento dinâmico seja garantido de maneira programável e aberta. A simplicidade de construir VNFs com Click, arquitetura de software de código aberto que serve de base para o Fastclick, é um dos pontos positivos deste trabalho. Uma vez que se tem domínio de alguns elementos da arquitetura, é possível construir VNFs robustas com poucas linhas de programação. O único requisito que o programador deve ter é ter um mínimo conhecimento de redes e da sintaxe da linguagem, que se mostra bastante intuitiva. É importante ressaltar que caso não haja elementos com a funcionalidade desejada, é necessário construir um através da linguagem de programação C++. E isto demanda um alto conhecimento da linguagem. Além deste ponto a ferramenta conseguiu cobrir estes requisitos desejados em um ambiente de nuvem, o Openstack. Independente de plataforma de nuvem, o `sfcFC` se utiliza da vantagem do Fastclick poder ser utilizado em máquinas de arquitetura x86, fazendo com que seja possível aplicar os elementos SFC construídos neste trabalho em todos os ambientes de nuvem disponíveis atualmente.

O `sfcFC` é facilmente escalável. Como os elementos do SFC estão associados a instâncias virtuais, é possível construir vários elementos e distribuí-los na rede de acordo com a demanda. Além disso, as VNFs podem ser instanciadas com poucos recursos computacionais, como citado no Capítulo 5.

Apesar destes pontos positivos, o desempenho do `sfcFC` não foi satisfatório. Embora hajam benefícios notórios com a adoção de um cabeçalho, como proposto pelo NSH, vale a pena notar que o NSH aumenta o tamanho do pacote e gera uma sobrecarga na comunicação devido ao encapsulamento necessário para seu cabeçalho. Portanto, este protocolo adiciona uma complexidade considerável nas implantações do SFC (HANTOUTI; BENAMAR; TALEB, 2018). O teste quantitativo da Seção 5.5.1 que avalia a performance do `sfcFC`, traz luz ao desafio que o `sfcFC` precisa lidar no que diz respeito a adição de um cabeçalho ao processamento de pacotes (KULKARNI et al., 2017).

Como avaliação geral, apesar do desempenho ruim, o `sfcFC` realizou SFC com Fastclick e, junto ao PhantomSFC(CASTANHO et al., 2018), é mais uma ferramenta elaborada pelo laboratório NERDS (Núcleo de Estudo em Redes Definidas por Software)

que ajuda a contribuir na compreensão dos desafios do SFC. O código do `sfcFC` está disponível no `Github`[1].

6.2 Trabalhos Futuros

Algumas melhorias precisam ser feitas ao `sfcFC`, principalmente no que tange ao seu desempenho. A fim de reduzir o impacto, deve-se primeiramente avaliar o desempenho de máquinas virtuais frente aos contêineres Linux. Felter et al. (2015) aponta que apesar de oferecer mais segurança, máquinas virtuais tem uma performance inferior aos contêineres. A configuração de teste se portou bem para casos simples, mas para lidar com cenários mais reais é necessário que estas instâncias possuam mais poder de processamento.

Embora este trabalho não tenha como finalidade entregar desempenho, alguns ajustes precisam ser realizados. Para uma avaliação de desempenho mais detalhada, primeiramente é necessário desafogar o ambiente de Nuvem, tirando os clientes do cenário. Escalar elementos principais como o `sfcFC-Classifier` e `forwarder`, verticalmente ou horizontalmente, ou seja, alocar mais instâncias ou reservar mais recursos para tais também se mostra necessário.

Torna-se necessário, na medida em que a rede escala, a construção de um controlador `sfcFC` a fim de gerenciar e dar mais agilidade ao procedimento do SFC. Por fim, único elemento não desenvolvido no `sfcFC`, um *proxy* também deve ser desenvolvido a fim de garantir o suporte a SFs que não tem o conhecimento do ambiente SFC (aplicações legadas).

Solucionar estes desafios tende a colocar o `sfcFC` como uma ferramenta competitiva para atuar em cenários reais.

[1] https://github.com/orenanft/masterthesis

Referências

BARBETTE, T.; SOLDANI, C.; MATHY, L. Fast userspace packet processing. In: IEEE. *2015 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. [S.l.], 2015. p. 5–16. Citado 3 vezes nas páginas 23, 33 e 46.

BHAMARE, D. et al. A survey on service function chaining. *Journal of Network and Computer Applications*, Elsevier, v. 75, p. 138–155, 2016. Citado 2 vezes nas páginas 24 e 35.

CASTANHO, M. S. et al. Phantomsfc: a fully virtualized and agnostic service function chaining architecture. In: IEEE. *2018 IEEE Symposium on Computers and Communications (ISCC)*. [S.l.], 2018. p. 354–359. Citado 2 vezes nas páginas 38 e 65.

CERRATO, I.; ANNARUMMA, M.; RISSO, F. Supporting fine-grained network functions through intel dpdk. In: *EWSDN*. [S.l.: s.n.], 2014. p. 1–6. Citado na página 33.

ETSI. Etsi gs nfv 002 v1.1.1 (2013-10). 2013. Citado 2 vezes nas páginas 15 e 29.

FELTER, W. et al. An updated performance comparison of virtual machines and linux containers. In: IEEE. *2015 IEEE international symposium on performance analysis of systems and software (ISPASS)*. [S.l.], 2015. p. 171–172. Citado na página 66.

HALPERN, J.; PIGNATARO, C. *Service Function Chaining (SFC) Architecture*. [S.l.], 2015. Citado 3 vezes nas páginas 23, 35 e 37.

HAN, B. et al. Network function virtualization: Challenges and opportunities for innovations. *IEEE Communications Magazine*, IEEE, v. 53, n. 2, p. 90–97, 2015. Citado 2 vezes nas páginas 28 e 29.

HANTOUTI, H.; BENAMAR, N.; TALEB, T. A novel compact header for traffic steering in service function chaining. In: IEEE. *2018 IEEE International Conference on Communications (ICC)*. [S.l.], 2018. p. 1–6. Citado 2 vezes nas páginas 37 e 65.

JOHN, W. et al. Research directions in network service chaining. In: *2013 IEEE SDN for Future Networks and Services (SDN4FNS)*. [S.l.: s.n.], 2013. Citado na página 23.

KOHLER, E. et al. The click modular router. *ACM Transactions on Computer Systems (TOCS)*, ACM, v. 18, n. 3, p. 263–297, 2000. Citado 6 vezes nas páginas 15, 17, 23, 30, 31 e 32.

KULKARNI, S. et al. Neo-nsh: Towards scalable and efficient dynamic service function chaining of elastic network functions. In: IEEE. *2017 20th Conference on Innovations in Clouds, Internet and Networks (ICIN)*. [S.l.], 2017. p. 308–312. Citado 3 vezes nas páginas 38, 63 e 65.

MEDHAT, A. M. et al. Service function chaining in next generation networks: State of the art and research challenges. *IEEE Communications Magazine*, IEEE, v. 55, n. 2, p. 216–223, 2016. Citado 3 vezes nas páginas 23, 34 e 35.

MELL, P.; GRANCE, T. et al. The NIST definition of cloud computing. Computer Security Division, Information Technology Laboratory, National ..., 2011. Citado na página 27.

MIJUMBI, R. et al. Network function virtualization: State-of-the-art and research challenges. *IEEE Communications Surveys & Tutorials*, IEEE, v. 18, n. 1, p. 236–262, 2015. Citado 3 vezes nas páginas 23, 28 e 29.

MIRJALILY, G.; ZHIQUAN, L. Optimal network function virtualization and service function chaining: A survey. *Chinese Journal of Electronics*, IET, v. 27, n. 4, p. 704–717, 2018. Citado na página 24.

OPENSTACK. 2019. Disponível em: <https://docs.openstack.org>. Citado 3 vezes nas páginas 17, 27 e 28.

PERKINS, C. *IP Encapsulation within IP*. [S.l.], 1996. Citado na página 41.

QUINN, P.; ELZUR, U.; PIGNATARO, C. *Network Service Header (NSH)*. [S.l.], 2018. Citado 3 vezes nas páginas 24, 36 e 37.

RIZZO, L. Netmap: a novel framework for fast packet i/o. In: *21st USENIX Security Symposium (USENIX Security 12)*. [S.l.: s.n.], 2012. p. 101–112. Citado na página 33.

Apêndices

APÊNDICE A – Passo-a-passo da instalação do Fastclick e DPDK no Openstack

Openstack

Será necessário uma configuração do openstack com **kvm** habilitado. Para isso, execute o comando `kvm-ok` que deve retornar o seguinte resultado:

```
 INFO: /dev/kvm exists <br/>
KVM acceleration can be used
```

Feito isso, proceda para a etapa de instalação do openstack. Para habilitar a criação de máquinas virtuais com kvm no openstack será necessário editar o arquivo `/etc/nova/nova.conf`, que deve conter a seguinte configuração:

```
 [libvirt]
cpu_mode = host-passthrough
virt_type = kvm
```

Numa instalação com o devstack, adicione as seguintes linhas ao `local.conf`:

```
 ##
#libvirt
##
VIRT_DRIVER=libvirt
# To use nested KVM, un-comment the below line
LIBVIRT_TYPE=kvm

[[post-config|/etc/nova/nova.conf]]
[libvirt]
cpu_mode = host-passthrough
```

Com o openstack instalado, acesse o horizon e crie uma imagem da distro que preferir com o seguinte metadado: `hw_vif_model e1000`. Para isso: 1. Clique em **Projeto > Imagens > Criar Imagem > Metadado > libvirt Driver Options for Images > Virtual Network Interface**. 2. Selecione **e1000** em **hw_vif_model**. 3. Crie a imagem e crie uma VM com a imagem criada.

Máquina Virtual

Acesse a máquina virtual, em nosso caso estamos utilizando uma cloud image do ubuntu 16.04.5. Para isso, é necessário ter configurado no mínimo 3 redes (publica, tenant e management), e um roteador no Openstack. O acesso a uma VM no openstack se dá através dos namespaces do Linux, por exemplo:

```
 sudo ip netns exec qrouter-84c5e6ed-f48e-4c2e-a3b5-7076031ea835 \
ssh -i key.pem ubuntu@192.168.120.7
```

, onde q-router* é o id do roteador openstack, key.pem é a chave privada criada no openstack, e 192.168.120.7 é o endereço da VM na rede management.

Instalando o DPDK

1. Atualize a máquina e baixe os pacotes necessários: `sudo apt update && sudo apt install python python3 python3-pip libcap-dev libnuma-dev libarchive-dev linux-headers-$(uname -r) liblz4-dev liblz4-tool`
2. Baixe o código do DPDK compatível com o Fastclick (<=17.05)
3. Descompacte o arquivo: `tar xfv dpdk-16.11.8.tar.xz` e renomeie a pasta descompactada para dpdk
4. O dpdk precisa de duas variáveis de ambiente configuradas. Para isso, crie um arquivo com o seguinte conteúdo:

```
export RTE_SDK=$HOME/dpdk
export RTE_TARGET=x86_64-native-linuxapp-gcc
```

5. Execute o comando **source** no arquivo para carregar as variáveis de ambiente.
6. Dentro da pasta do dpdk há um assitente de instalação em **usertools/dpdk-setup.sh**. Execute-o.
    - Para compilar o dpdk selecione o número relacionado a opção **x86_64-native-linuxapp-gcc**
    - Saia do assistente e entre na pasta criada(x86_64-native-linuxapp-gcc). Execute o comando `make`
    - Execute novamente o assistente e selecione o número relacionado a opção **dnsert IGB UIO module**
    - Selecione o número relacionado a opção **Bind Ethernet/Crypto device to IGB UIO module**. Selecione a interface de rede. Atenção, pois a máquina virtual deve conter duas interfaces de rede. Uma management e uma tenant que deve ficar sem IP para que o fastclick possa operar.
    - Selecione o número relacionado a opção **Setup hugepage mappings for NUMA systems**. E coloque o número de hugepages necessário.

Instalando o Fastclick

Com o dpdk instalado e configurado, é o momento de configurar o fastclick. 1. [Clone](git clone https://github.com/tbarbette/fastclick/) o repositório do fastclick 2. Instale a biblioteca zlib1g-dev: `apt install zlib1g-dev` 3. Entre na pasta e execute-o com o dpdk:

```
./configure --enable-multithread --disable-linuxmodule --enable-intel-cpu \
--enable-user-multithread --verbose CFLAGS="-g -O3" CXXFLAGS="-g -std=gnu++11 -O3" \
--disable-dynamic-linking --enable-poll --enable-bound-port-transfer --enable-dpdk \
--enable-batch --with-netmap=no --enable-zerocopy --enable-dpdk-pool \
--disable-dpdk-packet
```

3. Crie o seu arquivo (.click) e execute: `sudo bin/click --dpdk -- /caminho/para/arquivo.click`

APÊNDICE B – Código Click do sfcFC-Classifier

```
1 // SFC Classifier: A sfc packet classifier based on some rules. This also acts
2 //as a gateway between 10.0.0.0/24, 10.0.3.0/24 and 10.0.1.0/24.
3 // Author: Renan Freire Tavares
4
5 //define($IFNET0 3); //pci nic
6 //define($IFSFC 3); //pci nic
7 //define($IFNET1 3); //pci nic
8 //define($IFCLIENT 3); //pci nic
9
10 // IPs, networks e MACs.
11 //          name      ip              ipnet               mac
12 AddressInfo(net0     10.0.0.5     10.0.0.0/24     FA:16:3E:2C:5C:46,
13              sfc      10.0.3.108   10.0.3.0/24     FA:16:3E:4C:AA:C7,
14              net1     10.0.1.121   10.0.1.0/24     FA:16:3E:87:17:6B,
15              client   10.0.5.3   10.0.5.0/24       FA:16:3E:77:A3:34,
16              cchain2  10.0.5.20,
17              cchain3  10.0.5.21,
18              cchain4  10.0.5.22,
19              sff      10.0.3.106,
20              public   10.0.2.156   10.0.2.0/24,
21              ws1     10.0.1.102,
22              ws2     10.0.1.106
23 );
24
25 //incoming packets
26 src1 :: FromDPDKDevice($IFNET0);
27 src2 :: FromDPDKDevice($IFSFC);
28 src3 :: FromDPDKDevice($IFNET1);
29 src4 :: FromDPDKDevice($IFCLIENT);
30
31 //outcoming packets
32 sink1    :: ARPPrint() -> ToDPDKDevice($IFNET0);
33 sink2    :: ARPPrint() -> ToDPDKDevice($IFSFC);
34 sink3    :: ARPPrint() -> ToDPDKDevice($IFNET1);
35 sink4    :: ARPPrint() -> ToDPDKDevice($IFCLIENT);
36
37 // click router packet classifiers
38 c1,c2,c3,c4 :: Classifier(
39     12/0806 20/0001, // ARP Requests/queries out 0
40     12/0806 20/0002, // ARP Replies out 1
41     12/0800, // IP Packets out 2
42     -); // other packets - out 3
43
44 // For both: Incoming packets from interfaces are going to layer 2 classifiers
45 //input 0.
46 src1 -> [0]c1; // First network port for net0
47 src2 -> [0]c2; // for netsfc;
48 src3 -> [0]c3; // for net1;
```

```
49 src4 -> [0]c4; // for client;
50
51 // ARP Responder definitions. It's useful for host visibility by others in
52 //networks. It going to answer ARP queries with MAC address based on IP-matched.
53 //It could contain more than one entry, which means ARP Responders could answer
54 //queries about another machines and networks. ProxyARP is an application of
55 //this.
56 // Connecting queries from classifier to ARPResponder after this, to outside
57 //world through queues.
58
59 // ARP REQUESTS
60 //ARP queries or requests
61 arpq1 :: ARPQuerier(net0) -> sink1;
62 arpq2 :: ARPQuerier(sfc) -> sink2;
63 arpq3 :: ARPQuerier(net1) -> sink3;
64 arpq4 :: ARPQuerier(client) -> sink4;
65
66 //ARP replies
67 arpr1 :: ARPResponder(net0) -> sink1;
68 arpr2 :: ARPResponder(sfc) -> sink2;
69 arpr3 :: ARPResponder(net1) -> sink3;
70 arpr4 :: ARPResponder(client) -> sink4;
71
72 c1[0] -> arpr1;
73 c2[0] -> arpr2;
74 c3[0] -> arpr3;
75 c4[0] -> arpr4;
76
77 // Delivering ARP responses to the ARP queriers.
78 c1[1] -> [1]arpq1;
79 c2[1] -> [1]arpq2;
80 c3[1] -> [1]arpq3;
81 c4[1] -> [1]arpq4;
82
83 // Other protocol types inside ethernet frames. They are dropped/discarded.
84 c1[3] -> Discard;
85 c2[3] -> Discard;
86 c3[3] -> Discard;
87 c4[3] -> Discard;
88
89 //IP PACKETS
90
91 //Element with n outputs to classify IP packets.
92 //Output 0 is for incoming chain1 packets
93 //Output 1 is for incoming chain2 packets
94 //Output 2 is for incoming chain3 packets
95 //Output 3 is for incoming chain4 packets
96 //Output 4 is for net0 packets
97 //Output 5 is for sfc packets
98 //Output 6 is for net1 packets.
99 //Output 7 is for outcoming chain1 packets.
100 //Output 8 is for outcoming chain2 packets.
101 //Output 9 is for outcoming chain3 packets.
102 //Output 10 is for outcoming chain4 packets.
103 //Output 11 is for all other IP packets.
104 sfcclassifier :: IPClassifier(
105             src net != net1 && dst host client:ip,
```

```
106             src net != net1 && dst host cchain2:ip,
107             src net != net1 && dst host cchain3:ip,
108             src net != net1 && dst host cchain4:ip,
109             dst net net0,
110             dst net sfc,
111             dst net net1,
112             (src host ws1 || src host ws2) && dst host client:ip,
113             (src host ws1 || src host ws2) && dst host cchain2:ip,
114             src host ws1 && dst net client,
115             src host ws2 && dst net client,
116             -);
117
118 //traffic from net0, sfc, net1 and client will go to sfcclassifier
119 c1[2] -> Strip(14) -> CheckIPHeader() -> [0]sfcclassifier;
120 c2[2] -> Strip(14) -> CheckIPHeader() -> [0]sfcclassifier;
121 c3[2] -> Strip(14) -> CheckIPHeader() -> [0]sfcclassifier;
122 c4[2] -> Strip(14) -> CheckIPHeader() -> [0]sfcclassifier;
123
124 //redirecting/routing traffic from net0
125 //now the traffic from net0 to net1 will follow the sfc rules
126 //Thus, net1 traffic will be routed to sfc network
127 //The ip packet will be encapsulated and the header (source: classifier ip, dest:
      sff ip)
128 //Finally the paint annotation will indicate the chain and the step(third element
      is unused)
129
130 rewriterChain1IN :: IPRewriter(pattern client:ip - - - 0 1);
131 rewriterChain2IN :: IPRewriter(pattern cchain2:ip - - - 0 1);
132 rewriterChain3 :: IPRewriter(pattern - - ws1 - 0 1);
133 rewriterChain4 :: IPRewriter(pattern - - ws2 - 0 1);
134
135 rewriterChain1OUT :: IPRewriter(pattern client:ip - - - 0 1);
136 rewriterChain2OUT :: IPRewriter(pattern cchain2:ip - - - 0 1);
137
138 checklen1 :: CheckLength(1400);
139 checklen2 :: CheckLength(1400);
140 checklen3 :: CheckLength(1400);
141 checklen4 :: CheckLength(1400);
142
143 checklen1[0] -> [0]arpq1;
144 checklen1[1] -> IPFragmenter(1400) -> [0]arpq1;
145
146 checklen2[0] -> [0]arpq2;
147 checklen2[1] -> IPFragmenter(1400) -> [0]arpq2;
148
149 checklen3[0] -> [0]arpq3;
150 checklen3[1] -> IPFragmenter(1400) -> [0]arpq3;
151
152 checklen4[0] -> [0]arpq4;
153 checklen4[1] -> IPFragmenter(1400) -> [0]arpq4;
154
155 sfcclassifier[0] -> [0]rewriterChain1IN;
156 sfcclassifier[1] -> [0]rewriterChain2IN;
157 sfcclassifier[2] -> [0]rewriterChain3;
158 sfcclassifier[3] -> [0]rewriterChain4;
159 sfcclassifier[4] -> checklen1;
160 sfcclassifier[5] -> checklen2;
```

```
161 sfcclassifier[6] -> checklen3;
162 sfcclassifier[7] -> [0]rewriterChain1OUT;
163 sfcclassifier[8] -> [0]rewriterChain2OUT;
164 sfcclassifier[9] -> [0]rewriterChain3;
165 sfcclassifier[10] -> [0]rewriterChain4;
166 sfcclassifier[11] -> Discard;
167
168 rewriterChain1IN[0] -> UDPIPEncap(sfc:ip,1,sff,0) -> checklen2;
169 rewriterChain1IN[1] -> SetTCPChecksum() -> checklen4;
170
171 rewriterChain2IN[0] -> UDPIPEncap(sfc:ip,2,sff,0) -> checklen2;
172 rewriterChain2IN[1] -> SetTCPChecksum() -> checklen4;
173
174 rewriterChain3[0] -> UDPIPEncap(sfc:ip,3,sff,0) -> checklen2;
175 rewriterChain3[1] -> SetTCPChecksum() -> checklen4;
176
177 rewriterChain4[0] -> UDPIPEncap(sfc:ip,4,sff,0) -> checklen2;
178 rewriterChain4[1] -> SetTCPChecksum() -> checklen4;
179
180 rewriterChain1OUT[0] -> [0]rewriterChain1IN;
181 rewriterChain1OUT[1] -> Discard;
182
183 rewriterChain2OUT[0] -> [0]rewriterChain2IN;
184 rewriterChain2OUT[1] -> Discard;
```

APÊNDICE C – Código Click do Forwarder

```
1 // SFC Forwarder: A firewall to a web based network containing it, a NAT with
      Load
2 //Balancer, and some Web Servers and Hosts in a private network. This also acts
3 //as a gateway between 198.51.100.0/24 and 192.0.2.0/24 IETF defined test
4 //networks, where the entire "public network" are set.
5 // Author: Renan Freire Tavares
6
7 //define($IFSFC 3); //pci nic
8
9 // IPs, networks e MACs.
10 //          name     ip              ipnet               mac
11 AddressInfo(sfc         10.0.3.106   10.0.3.0/24    FA:16:3E:C4:00:94,
12             firewall    10.0.3.128,
13             loadbalancer     10.0.3.107,
14             lbsrcip      10.0.3.102,
15             router      10.0.3.108
16 );
17
18 //incoming packets
19 src :: FromDPDKDevice($IFSFC);
20
21 //outcoming packets
22 sink :: ARPPrint() -> ToDPDKDevice($IFSFC);
23
24 // click router packet classifier
25 c :: Classifier(
26     12/0806 20/0001, // ARP Requests/queries out 0
27     12/0806 20/0002, // ARP Replies out 1
28     12/0800, // IP Packets out 2
29     -); // other packets - out 3
30
31 // Incoming packets from interfaces are going to layer 2 classifier
32 //input 0.
33 src -> [0]c; // network port for net0
34
35 // ARP Responder definitions. It's useful for host visibility by others in
36 //networks. It going to answer ARP queries with MAC address based on IP-matched.
37 //It could contain more than one entry, which means ARP Responders could answer
38 //queries about another machines and networks. ProxyARP is an application of
39 //this.
40 // Connecting queries from classifier to ARPResponder after this, to outside
41 //world through queues.
42
43 // ARP REQUESTS
44 //ARP queries or requests
45 arpq :: ARPQuerier(sfc) -> sink;
46
47 //ARP replies
```

```
48 arpr :: ARPResponder(sfc) -> sink;
49
50 c[0] -> arpr;
51
52 // Delivering ARP responses to the ARP queriers.
53 c[1] -> [1]arpq;
54
55 // Other protocol types inside ethernet frames. They are dropped/discarded.
56 c[3] -> Discard;
57
58
59 //IP PACKETS
60
61 //checkpaints from classifier and for each SF
62 //checkchain :: CheckPaint(1,CHAIN);
63 //checkclassifier :: CheckPaint(0,STEP);
64 //checkfirewall :: CheckPaint(1,STEP);
65 //checklb :: CheckPaint(2,STEP);
66
67 sfcclassifier :: IPClassifier(
68             udp && src port 1 && dst port 0, //classifier to firewall
69             udp && src port 1 && dst port 1, //firewall to lb
70             udp && src port 1 && dst port 2, //lb to ws
71             udp && src port 2 && dst port 0, //classifier to firewall
72             udp && src port 2 && dst port 1, //firewall to lb-srcip
73             udp && src port 2 && dst port 2, //lb-srcip to ws
74             udp && src port 3 && dst port 0, ///classifier to firewall
75             udp && src port 3 && dst port 1, //firewall to server
76             udp && src port 4 && dst port 0, //classifier para o firewall
77             udp && src port 4 && dst port 1, //firewall to server
78             -);
79
80
81 //ip traffic is stripped, checked and sent to check it has classifier paint
82 c[2] -> Strip(14) -> CheckIPHeader() -> sfcclassifier;
83
84 sfcclassifier[0] -> Print("C1 - IN") -> StripIPHeader() -> Strip(8) ->
       CheckIPHeader() -> UDPIPEncap(sfc:ip,1,firewall,0) -> [0]arpq;
85 sfcclassifier[1] -> StripIPHeader() -> Strip(8) -> CheckIPHeader() -> UDPIPEncap(
       sfc:ip,1,loadbalancer,1) -> [0]arpq;
86 sfcclassifier[2] -> StripIPHeader() -> Strip(8) -> CheckIPHeader() ->
       SetTCPChecksum() -> SetIPAddress(router) -> Print("C1- OUT") -> [0]arpq;
87
88 sfcclassifier[3] -> Print("C2 - IN") -> StripIPHeader() -> Strip(8) ->
       CheckIPHeader() -> UDPIPEncap(sfc:ip,2,firewall,0) -> [0]arpq;
89 sfcclassifier[4] -> StripIPHeader() -> Strip(8) -> CheckIPHeader() -> UDPIPEncap(
       sfc:ip,2,lbsrcip,1) -> [0]arpq;
90 sfcclassifier[5] -> StripIPHeader() -> Strip(8) -> CheckIPHeader() ->
       SetTCPChecksum() -> SetIPAddress(router) -> Print("C2- OUT") -> [0]arpq;
91
92 sfcclassifier[6] -> Print("C3 - IN") -> StripIPHeader() -> Strip(8) ->
       CheckIPHeader() -> UDPIPEncap(sfc:ip,3,firewall,0) -> [0]arpq;
93 sfcclassifier[7] -> StripIPHeader() -> Strip(8) -> CheckIPHeader() ->
       SetTCPChecksum() -> SetIPAddress(router) -> Print("C3- OUT") -> [0]arpq;
94
95 sfcclassifier[8] -> Print("C4 - IN") -> StripIPHeader() -> Strip(8) ->
       CheckIPHeader() -> UDPIPEncap(sfc:ip,4,firewall,0) -> [0]arpq;
```

```
96 sfcclassifier[9] -> StripIPHeader() -> Strip(8) -> CheckIPHeader() ->
       SetTCPChecksum() -> SetIPAddress(router) -> Print("C4- OUT") -> [0]arpq;
97
98 sfcclassifier[10] -> Discard;
99
100 //checkchain[0] -> Print("CHAIN1") -> checkclassifier;
101 //checkchain[1] -> Print("DISCARD") -> Discard;
102
103 //if the packet has a classifier paint it is redirect to 0 output,
104 //encapsulated (source:sff,dest:firewall), painted and encapsulated by ARP
       querier based
105 //on its destination address;
106 //else to 1 output and to firewall paint check
107 //checkclassifier[0] -> Print("STEP1") -> StripIPHeader() -> CheckIPHeader() ->
       IPEncap(4, sfc:ip, firewall) -> Paint(1,CHAIN) -> Paint(0,STEP) ->
       IPFragmenter(1436) -> [0]arpq;
108 //checkclassifier[1] -> [0]checkfirewall;
109
110 //if the packet has a firewall paint it is redirect to 0 output,
111 //encapsulated (source:sff,dest:loadbalancer), painted and encapsulated by ARP
       querier based
112 //on its destination address;
113 //else to 1 output and to loadbalancer paint check
114 //checkfirewall[0] -> Print("STEP2") -> StripIPHeader() -> CheckIPHeader() ->
       IPEncap(4, sfc:ip, loadbalancer) -> Paint(1,CHAIN) -> Paint(1,STEP) ->
       IPFragmenter(1436) -> [0]arpq;
115 //checkfirewall[1] -> IPFragmenter(1450) -> [0]checklb;
116
117 //if the packet has a loadbalancer paint it is redirect to 0 output,
118 //is stripped, checked and encapsulated by ARP querier based
119 //on its destination address;
120 //else to 1 output and encapsulated by ARP querier based
121 //on its destination address.
122 //checklb[0] ->  Print("OUT") -> StripIPHeader() -> CheckIPHeader() ->
       IPFragmenter(1436) -> [0]arpq;
123 //checklb[1] -> Print("OUTDISCARD") -> Discard;
```

APÊNDICE D – Código Click do Firewall

```
1 // Firewall on SFC: A firewall to a web based network containing it.
2 // Author: Renan Freire Tavares
3 //Based on Felipe Belsholff work available in https://github.com/belsholff/
      undergraduate-thesis/blob/master/clickOS/LoadBalancer;
4
5 //define(sIFFRW 3); //pci nic
6
7 // Organizing IPs, networks and MACs from this MicroVM. Or tagging known hosts.
8 //          name              ip            ipnet                mac
9 AddressInfo(sfc     10.0.3.128     10.0.3.0/24     FA:16:3E:73:B2:BC,
10             sff     10.0.3.106,
11             floatingip  10.0.2.156
12 );
13
14 //incoming packets
15 src :: FromDPDKDevice($IFFRW);
16
17 //outcoming packets
18 sink :: ARPPrint() -> ToDPDKDevice($IFFRW);
19
20 // click router packet classifier
21 c :: Classifier(
22     12/0806 20/0001, // ARP Requests/queries out 0
23     12/0806 20/0002, // ARP Replies out 1
24     12/0800, // IP Packets out 2
25     -); // other packets - out 3
26
27 // Incoming packets from interfaces are going to layer 2 classifier
28 //input 0.
29 src -> [0]c; // network port for net0
30
31 // ARP Responder definitions. It's useful for host visibility by others in
32 //networks. It going to answer ARP queries with MAC address based on IP-matched.
33 //It could contain more than one entry, which means ARP Responders could answer
34 //queries about another machines and networks. ProxyARP is an application of
35 //this.
36 // Connecting queries from classifier to ARPResponder after this, to outside
37 //world through queues.
38
39 // ARP REQUESTS
40 //ARP queries or requests
41 arpq :: ARPQuerier(sfc) -> sink;
42
43 //ARP replies
44 arpr :: ARPResponder(sfc) -> sink;
45
46 c[0] -> arpr;
47
48 // Delivering ARP responses to the ARP queriers.
49 c[1] -> [1]arpq;
50
```

```
51 // Other protocol types inside ethernet frames. They are dropped/discarded.
52 c[3] -> Discard;
53
54 // Firewall application accepting only http/https requests to floatingip.
55 sfcFilterChain1 :: IPFilter(allow icmp,
56                      //allow dst floatingip && dst port 80 && dst port 443,
57                      allow dst port 80 || dst port 443,
58                      drop all)
59
60 sfcFilterChain2 :: IPFilter(allow icmp,
61                      //allow dst floatingip && dst port 80 && dst port 443,
62                      allow dst port 80 || dst port 443,
63                      drop all)
64
65 sfcFilterChain3 :: IPFilter(allow icmp,
66                      //allow dst floatingip && dst port 80 && dst port 443,
67                      allow dst port 80 || dst port 443,
68                      drop all)
69
70 sfcFilterChain4 :: IPFilter(allow icmp,
71                      //allow dst floatingip && dst port 80 && dst port 443,
72                      allow dst port 5201,
73                      drop all)
74
75
76 //IP PACKETS
77
78 //checkpaint from classifier
79 //checkchain :: CheckPaint(1,CHAIN);
80 //checksfc :: CheckPaint(0,STEP);
81
82 // For classifier:
83
84 sfcclassifier :: IPClassifier(
85             udp && src port 1, //chain1
86             udp && src port 2, //chain2
87             udp && src port 3, //chain3
88             udp && src port 4, //chain4
89             -);
90
91 //if the packet has a classifier paint it is redirect to 0 output,
92 // Ethernet packets are stripped and comes to IP packets, that has its headers
93 //checked, filtered by incomming firewall; encapsulated (source:sff,dest:
       loadbalancer),
94 //painted and encapsulated by ARP querier based on its destination address;
95 //else to 1 output and to loadbalancer paint check and encapsulated by ARP
       querier based
96 //on its destination address.
97 c[2] -> Print("IN") -> Strip(14) -> CheckIPHeader() -> sfcclassifier;
98
99 sfcclassifier[0] -> StripIPHeader() -> Strip(8)-> CheckIPHeader()
100   -> sfcFilterChain1
101   -> UDPIPEncap(sfc:ip,1, sff,1) -> Print("OUT") -> [0]arpq;
102
103 sfcclassifier[1] -> StripIPHeader() -> Strip(8)-> CheckIPHeader()
104     -> sfcFilterChain2
105     -> UDPIPEncap(sfc:ip,2, sff,1) -> Print("OUT") -> [0]arpq;
```

```
106
107 sfcclassifier[2] -> StripIPHeader() -> Strip(8)-> CheckIPHeader()
108     -> sfcFilterChain3
109     -> UDPIPEncap(sfc:ip,3, sff,1) -> Print("OUT") -> [0]arpq;
110
111 sfcclassifier[3] -> StripIPHeader() -> Strip(8)-> CheckIPHeader()
112     -> sfcFilterChain4
113     -> UDPIPEncap(sfc:ip,4, sff,1) -> Print("OUT") -> [0]arpq;
114
115 sfcclassifier[4] -> Discard;
```

APÊNDICE E – Código Click do Balanceador *Source* IP

```
1 // Load Balancer on SFC: A Load Balancer to a web based network containing it, a
    NAT with Load
2 //Balancer, and some Web Servers and Hosts in a private network.
3 // Author: Renan Freire Tavares
4 //Based on Felipe Belsholff work available in https://github.com/belsholff/
    undergraduate-thesis/blob/master/clickOS/LoadBalancer
5
6 //define($IFLB 3); //pci nic
7
8
9 // Organizing IPs, networks and MACs from this MicroVM. Or tagging known hosts.
10 //          name      ip              ipnet                  mac
11 AddressInfo(sfc          10.0.3.102     10.0.3.0/24     FA:16:3E:D7:EF:B2,
12             ws1     10.0.1.102,
13             ws2     10.0.1.106,
14             sff     10.0.3.106,
15             floatingip 10.0.2.156
16 );
17
18 //incoming packets
19 src :: FromDPDKDevice($IFLB);
20
21 //outcoming packets
22 sink :: ARPPrint() -> ToDPDKDevice($IFLB);
23
24 // click router packet classifier
25 c :: Classifier(
26     12/0806 20/0001, // ARP Requests/queries out 0
27     12/0806 20/0002, // ARP Replies out 1
28     12/0800, // IP Packets out 2
29     -); // other packets - out 3
30
31 // Incoming packets from interfaces are going to layer 2 classifier
32 //input 0.
33 src -> [0]c; // network port for net0
34
35 // ARP Responder definitions. It's useful for host visibility by others in
36 //networks. It going to answer ARP queries with MAC address based on IP-matched.
37 //It could contain more than one entry, which means ARP Responders could answer
38 //queries about another machines and networks. ProxyARP is an application of
39 //this.
40 // Connecting queries from classifier to ARPResponder after this, to outside
41 //world through queues.
42
43 // ARP REQUESTS
44 //ARP queries or requests
45 arpq :: ARPQuerier(sfc) -> sink;
46
```

```
47 //ARP replies
48 arpr :: ARPResponder(sfc) -> sink;
49
50 c[0] -> arpr;
51
52 // Delivering ARP responses to the ARP queriers.
53 c[1] -> [1]arpq;
54
55 // Other protocol types inside ethernet frames. They are dropped/discarded.
56 c[3] -> Discard;
57
58 // For both: Incoming packets from interfaces are going to layer 2 classifiers
59 //input 0.
60
61
62 // Mapping used to do load balancing based on quintuple SIP, SPort, DIP, DPort
63 //and Protocol. It consists of a hash table with fixed size and seed, in
64 //addition to maintaining consistency, which means always requests are mapped
65 //from a SIP to the same cluster node. It helps in the use of TCP connections.
66 // The last entry in each rule means an ID.
67 // This mapping is used inside the IPRewriter element below.
68 // More detailed documentation about rules and this integration here:
69 //https://github.com/kohler/click/wiki/IPRewriter
70 ws_mapper :: SourceIPHashMapper(129 0xbadbeef,
71                                 - - ws1 - 0 1 4055,
72                                 - - ws2 - 0 1 80147
73 );
74
75 // Simple NAT function. Rewrite packets that cames on it's input ports based on
76 //some rules in a table. This table receives entries by handlers or general
77 //lines hardcoded in arguments function. Rules are set twice at time: first one
78 //about incomming flow and later about outgoing one. Each flow goes out by an
79 //output port previously defined by rule. Hardcode lines sets inputs and outputs
80 //ports numbering them by its order in arguments, and are used just when there's
81 //no rule matched in table.
82 //IPRewriter hardcoded behaviors:
83 //1- pattern
84 //2- drop
85 //3- pass
86 // IPRewriter also could receive previously defined static and dinamic tables
87 //as SourceIPHashMapper (which is our case) or IPRewritterPatterns. More
88 //detailed documentation in link above.
89 rewriter :: IPRewriter(ws_mapper);
90
91 //IP PACKETS
92
93 //checkpaint from classifier
94 //checkchain :: CheckPaint(1,CHAIN);
95 //checksfc :: CheckPaint(1,STEP);
96
97 // For classifier:
98 //if the packet has a classifier paint it is redirect to 0 output,
99 // Ethernet packets are stripped and comes to IP packets, that has its headers
100 //checked, and send to NAT/LB elements.
101 //encapsulated (source:sff,dest:loadbalancer),
102 //painted and encapsulated by ARP querier based on its destination address;
103 //else to 1 output and to loadbalancer paint check and encapsulated by ARP
```

```
    querier based
//on its destination address.
sfcclassifier :: IPClassifier(
            udp && src port 2, //chain2
            -);

c[2] -> Print("IN") -> Strip(14) -> CheckIPHeader() -> sfcclassifier;

sfcclassifier[0] -> StripIPHeader() -> Strip(8) -> CheckIPHeader()
  -> [0]rewriter;

sfcclassifier[1] -> Discard;
//checkchain[0] -> checksfc;
//checkchain[1] -> Discard;

//checksfc[0] -> StripIPHeader()
//            -> CheckIPHeader()
//            -> [0]rewriter;

//checksfc[1] -> IPFragmenter(1436) -> [0]arpq;

//For both rewriters:
// As I sad above, here are incomming and outgoing NAT-ed packets. They have
//their checksum recalculated (TCP in this case, others below) and come out
//ready to be send to its destination through ARPQuerier;


rewriter[0] -> SetTCPChecksum()
      -> UDPIPEncap(sfc:ip,2, sff,2) -> Print("OUT")
            -> [0]arpq;

rewriter[1] -> Discard;
//SetTCPChecksum()
//            -> SetIPAddress(sfc:ip)
//            -> IPFragmenter(1436) -> [0]arpq;
```

APÊNDICE F – Código Click do Balanceador *Round Robin*

```
1 // Load Balancer on SFC: A Load Balancer to a web based network containing it, a
    NAT with Load
2 //Balancer, and some Web Servers and Hosts in a private network.
3 // Author: Renan Freire Tavares
4 //Based on Felipe Belsholff work available in https://github.com/belsholff/
    undergraduate-thesis/blob/master/clickOS/LoadBalancer
5
6 //define($IFLB 3); //pci nic
7
8 // Organizing IPs, networks and MACs from this MicroVM. Or tagging known hosts.
9 //          name     ip              ipnet               mac
10 AddressInfo(sfc     10.0.3.107     10.0.3.0/24     FA:16:3E:F9:A4:9C,
11             ws1     10.0.1.102,
12             ws2     10.0.1.106,
13             sff     10.0.3.106,
14             floatingip 10.0.2.156
15 );
16
17 //incoming packets
18 src :: FromDPDKDevice($IFLB);
19
20 //outcoming packets
21 sink :: ARPPrint() -> ToDPDKDevice($IFLB);
22
23 // click router packet classifier
24 c :: Classifier(
25     12/0806 20/0001, // ARP Requests/queries out 0
26     12/0806 20/0002, // ARP Replies out 1
27     12/0800, // IP Packets out 2
28     -); // other packets - out 3
29
30 // Incoming packets from interfaces are going to layer 2 classifier
31 //input 0.
32 src -> [0]c; // network port for net0
33
34 // ARP Responder definitions. It's useful for host visibility by others in
35 //networks. It going to answer ARP queries with MAC address based on IP-matched.
36 //It could contain more than one entry, which means ARP Responders could answer
37 //queries about another machines and networks. ProxyARP is an application of
38 //this.
39 // Connecting queries from classifier to ARPResponder after this, to outside
40 //world through queues.
41
42 // ARP REQUESTS
43 //ARP queries or requests
44 arpq :: ARPQuerier(sfc) -> sink;
45
46 //ARP replies
```

```
47 arpr :: ARPResponder(sfc) -> sink;
48
49 c[0] -> arpr;
50
51 // Delivering ARP responses to the ARP queriers.
52 c[1] -> [1]arpq;
53
54 // Other protocol types inside ethernet frames. They are dropped/discarded.
55 c[3] -> Discard;
56
57 // For both: Incoming packets from interfaces are going to layer 2 classifiers
58 //input 0.
59
60
61 // Mapping used to do load balancing based on Round Robin distribution, with a
62 //quintuple SIP, SPort, DIP, DPort and Protocol, which means that requests with
63 //they, are mapped to the same cluster node. It helps in the use of TCP
64 //connections.
65 // This mapping is used inside the IPRewriter element below.
66 // More detailed documentation about rules and this integration here:
67 //https://github.com/kohler/click/wiki/IPRewriter
68 ws_mapper :: RoundRobinIPMapper(- - ws1 - 0 1,
69                                 - - ws2 - 0 1
70 );
71
72 // Simple NAT function. Rewrite packets that cames on it's input ports based on
73 //some rules in a table. This table receives entries by handlers or general
74 //lines hardcoded in arguments function. Rules are set twice at time: first one
75 //about incomming flow and later about outgoing one. Each flow goes out by an
76 //output port previously defined by rule. Hardcode lines sets inputs and outputs
77 //ports numbering them by its order in arguments, and are used just when there's
78 //no rule matched in table.
79 //IPRewriter hardcoded behaviors:
80 //1- pattern
81 //2- drop
82 //3- pass
83 // IPRewriter also could receive previously defined static and dinamic tables
84 //as SourceIPHashMapper (which is our case) or IPRewritterPatterns. More
85 //detailed documentation in link above.
86 rewriter :: IPRewriter(ws_mapper);
87
88 //IP PACKETS
89
90 //checkpaint from classifier
91 //checkchain :: CheckPaint(1,CHAIN);
92 //checksfc :: CheckPaint(1,STEP);
93
94 // For classifier:
95 //if the packet has a classifier paint it is redirect to 0 output,
96 // Ethernet packets are stripped and comes to IP packets, that has its headers
97 //checked, and send to NAT/LB elements.
98 //encapsulated (source:sff,dest:loadbalancer),
99 //painted and encapsulated by ARP querier based on its destination address;
100 //else to 1 output and to loadbalancer paint check and encapsulated by ARP
      querier based
101 //on its destination address.
102
```

```
103 sfcclassifier :: IPClassifier(
104             udp && src port 1, //chain2
105             -);
106
107 c[2] -> Print("IN") -> Strip(14) -> CheckIPHeader() -> sfcclassifier;
108
109 sfcclassifier[0] -> StripIPHeader() -> Strip(8) -> CheckIPHeader()
110   -> [0]rewriter;
111
112 sfcclassifier[1] -> Discard;
113 //checkchain[0] -> checksfc;
114 //checkchain[1] -> Discard;
115
116 //checksfc[0] -> StripIPHeader()
117 //            -> CheckIPHeader()
118 //            -> [0]rewriter;
119
120 //checksfc[1] -> IPFragmenter(1436) -> [0]arpq;
121
122 //For both rewriters:
123 // As I sad above, here are incomming and outgoing NAT-ed packets. They have
124 //their checksum recalculated (TCP in this case, others below) and come out
125 //ready to be send to its destination through ARPQuerier;
126
127
128 rewriter[0] -> SetTCPChecksum()
129       -> UDPIPEncap(sfc:ip,1, sff,2) -> Print("OUT")
130             -> [0]arpq;
131
132 rewriter[1] -> Discard;
133 //SetTCPChecksum()
134 //            -> SetIPAddress(sfc:ip)
135 //            -> IPFragmenter(1436) -> [0]arpq;
```