UNIVERSIDADE FEDERAL DO ESPÍRITO SANTO
CENTRO TECNOLÓGICO
PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

RAFAEL DUARTE VENCIONECK

Uma proposta de Rede para Datacenter Centrado em Servidores, Definida por Software e de Baixa Latência

VITÓRIA
2015

Uma proposta de Rede para Datacenter Centrado em Servidores, Definida por Software e de Baixa Latência - 2015

RAFAEL DUARTE VENCIONECK

Uma proposta de Rede para Datacenter Centrado em Servidores, Definida por Software e de Baixa Latência

Dissertação apresentada ao Programa de Pós-Graduação em Informática do Centro Tecnológico da Universidade Federal do Espírito Santo, como requisito parcial para obtenção do Grau de Mestre em Informática.
Orientador: Prof. Dr. Magnos Martinello.
Co-Orientador: Gilmar Luiz Vassoler

VITÓRIA
2015

Para fazer a ficha catalográfica, que ficará no lugar desta página, na versão final, ligue ou escreva para Arlete Franco
Bibliotecária/BC/UFES
franco@bc.ufes.br
27 4009-2405

Há um formulário específico para solicitar tal ficha, que é enviado à bibliotecária via e-mail (ver www.ele.ufes.br/paginappgee.html)

RAFAEL DUARTE VENCIONECK

Uma proposta de Rede para Datacenter Centrado em Servidores, Definida por Software e de Baixa Latência

Dissertação submetida ao programa de Pós-Graduação em Informática do Centro Tecnológico da Universidade Federal do Espírito Santo, como requisito parcial para a obtenção do Grau de Mestre em Informática.

Aprovada em 04 de setembro de 2015.

COMISSÃO EXAMINADORA

Prof. Dr. Magnos Martinello
Universidade Federal do Espírito Santo
Orientador

Prof. Dr. Gilmar Luiz Vassoler
Universidade Federal do Espírito Santo
Co-orientador

Prof. Dr. Rodolfo da Silva Villaca
Universidade Federal do Espírito Santo

Prof. Dr. Leobino Nascimento Sampaio
Universidade Federal da Bahia

Dedicado à minha amada esposa Dani

Agradecimentos

Agradeço à Deus por permitir que eu terminasse esta fase da minha vida. Tantas vezes só Ele foi capaz de me ajudar, renovando a esperança que muitas vezes havia acabado.

Agradeço à minha esposa Dani, por toda paciência e compreensão pelas horas de ausência, pelo incentivo mesmo quando eu não via mais saída nos desafios, por nunca me deixar desesperar, pelo apoio constante. Agradeço à minha mãe Laudicéia, por sempre estar rezando pelo meu sucesso.

Agradeço ao meu orientador Magnos Martinello pela oportunidade que me fez crescer imensamente, pelas indicações, correções e incentivo às ideias que eu apresentava, mesmo que malucas. Agradeço ao Gilmar Vassoler, co-orientador, por ter acreditado em mim como desbravador do Open Vswitch, pelas explicações e dicas importantíssimas. Agradeço ao prof. Moisés Ribeiro pela grande ajuda, pelas ideias e esclarecimentos, por compartilhar experiências, pelas indicações de artigos.

Agradeço à equipe do Laboratório NERDS pela disponibilidade e pela certeza de poder contar com todos sempre que precisasse, ao Alextian Liberato, sempre disposto a ajudar com o que fosse necessário e por compartilhar as ideias.

Agradeço à equipe do PoP-ES, especialmente Luis Filipe Pinotti, pela paciência de esperar tarefas atrasada serem concluídas, por pararem o que estiverem fazendo para ajudar em algum problema específico, por ajudarem desde a ligar cabos ou trocar memórias até a pensar em resoluções de problemas conceituais.

Agradeço ao Rafael Emerick, por explicar os detalhes do KeyFlow, por fornecer o código de cálculo do valor da chave e por tantas ajudas pequenas que foram fundamentais. Agradeço ao André Oshiro, por disponibilizar os servidores para testes e por sempre ajudar com o que estivesse ao alcance.

Muito Obrigado!

Sumário

Lista de Figuras

Lista de Abreviaturas

ACL	Access Control List
ARP	*Address Resolution Protocol*
ASIC	*Application Specific Integrated Circuits*
CLI	*Command-Line Interface*
CPU	*Central Processing Unit*
DPDK	*Data Plane Development Kit*
FIB	*Forwarding Information Base*
IaaS	*Infrastructure as a Service*
MIB	*Management Information Base*
MPLS	*Multiprotocol Label Switching*
NETCONF	*Network Configuration Protocol*
NFV	*Network Functions Virtualization*
ONF	*Open Networking Fundation*
OVS	*Open vSwitch*
P2P	*Peer to Peer*
PaaS	*Platform as a Service*
QoS	*Quality of Service*
RFC	*Request For Comments*
RIB	*Routing Information Base*
SaaS	*Software as a Service*

SDN	*Software Defined Networking*
SLA	*Service-Level Agreement*
SSH	*Secure Shell*
TCAM	*Ternary Content-Addressable Memory*
TI	*Tecnologia da Informação*
VLAN	*Virtual Local Area Network*
VM	*Virtual Machine*
VPN	*Virtual Private Network*
VTEP	*Virtual Tunnel Endpoint*

Resumo

O principal benefício das Redes Definidas por *Software* (SDN) e da Virtualização das Funções de Rede (NFV) é habilitar inovação a longo prazo. Entretanto, no *Datacenter* estas tecnologias são, na maioria das vezes, colocadas em prática de maneira que adiciona latência na comunicação, tornando impraticável a implantação de algumas aplicações. Isto se dá devido ao modo como a rede foi pensada no passado, não prevendo os requisitos atuais de automação, agilidade e flexibilidade. A fim de resolver este problema, o presente estudo identifica os gargalos de latência na rede do *Datacenter* definido por *software* atual, para então propor uma nova arquitetura que aperfeiçoa as redes centradas em servidores ao eliminar completamente equipamentos de rede e utilizar um método de encaminhamento inovador, além de um acelerador no tratamento de pacotes. O conceito de borda e núcleo da Internet é levado para o *Datacenter*, transformando o plano de dados, completamente virtual, em núcleo, que obtém como única função o processamento pacotes o mais rápido possível, enquanto a borda, isto é, as aplicações e serviços executados em máquinas virtuais ou *containers*, recebem informações do controlador a respeito de rotas pré-calculadas. Como prova de conceito, um protótipo da rede proposta é validado em ambiente físico e virtual. Os resultados mostram que na rede proposta a latência cai pela metade em comparação com a aceleração de encaminhamento definida pelo acelerador de pacotes DPDK, e por volta de 6 vezes menor em comparação com ambiente de virtualização tradicional.

Abstract

The main benefit of Software Defined Networking (SDN) and Network Functions Virtualization (NFV) is to enable long-term innovation. However, in the Datacenter these technologies are mostly put into practice in a manner that adds communication latency, becoming impractical to implement some applications. This is due to the way the network was thought in the past, not anticipating current requirements for automation, agility and flexibility. In order to solve this problem, the present study indentifies latency bottlenecks in current software defined Datacenter network, and then proposes a new architecture that improves server-centric networks by completely eliminating networking equipments and using an innovative forwarding method, in addition to an packet handling accelerator. Internet's concept of core and edge is brought to the Datacenter, transforming in core the entirely virtual data plane, getting the only function of forward as fast as possible, while the border, namely aplications and services running in virtual machines and containers, receives controller information regarding to precalculated routes. As proof of concept, a prototype of the proposed network is validated in a physical and virtual environment. Result shows that in the proposed network, latency is halved in comparison to the forwarding acceleration defined by DPDK accelerator, and it is 6 times lower than traditional Linux virtualization environment.

1 Introdução

As mudanças em TI impactam todo o panorama de negócios, modificando como a tecnologia é construída, consumida e distribuída. Uma destas transformações foi a virtualização de servidores, cujo início se deu com engenheiros que precisavam executar outro sistema operacional além do instalado em seus computadores para algum uso específico. Assim que terminavam, poderiam encerrar a execução como se fosse um *software* qualquer e voltar a trabalhar em seu sistema primário. Embora possa parecer simples, a ideia transformou-se num modelo fundamental para acompanhar a evolução da demanda de serviços e aplicações, mudando o modo como desenvolvedores e engenheiros constroem toda sua arquitetura de trabalho, além de criar novos tipos de profissões. Com a virtualização, foi possível aumentar a agilidade na criação de aplicações, criar maior independência em relação ao *hardware*, economizar recursos e reagir melhor a falhas.

Atualmente, o *Datacenter* é pensado com foco na virtualização. Com as novas demandas, o poder computacional, a capacidade de memória, o armazenamento e a banda de rede cresceram notoriamente, permitindo que um servidor seja capaz de executar um série de máquinas virtuais (VM) em paralelo e utilizar melhor seus recursos. Processamento e armazenamento são recursos vistos como onipresentes e acessíveis, embora estejam propositalmente separados. Essa evolução, gerou uma nova situação: ao planejar a compra dos equipamentos para um *Datacenter*, notou-se que os recursos eram projetados com base no uso futuro, ou seja, apenas em alguns anos o poder computacional adquirido seria utilizado próximo de seu limite, deixando-o em grande parte ocioso durante o tempo em que não atingia o crescimento planejado.

Tal estado de coisas criou um *Datacenter* com novos requisitos, onde, para que sua utilização seja próxima do limite, em paralelo com outras motivações como economia de energia e espaço, surge um modelo de negócios no qual os recursos ociosos são comercializados. Para isto, é preciso que o *Datacenter* tenha características de multi-inquilino (*multi-tentant*), ou seja, que permita "o uso dos mesmos recursos ou aplicações por múltiplos consumidores que podem ou não pertencer à mesma organização" (SAMANI, Raj and Reavis, Jim and Honan, Brian, 2011). "Em um *Datacenter multitenant*, processamento, armazenamento e recursos de rede

podem ser ofertados em porções que são independentes e isoladas umas da outras" (NADEAU; GRAY, 2013).

O provisionamento manual de recursos solicitados por um inquilino pode durar um tempo impraticável, não sendo escalável. A automatização é outra prática dos *Datacenters* modernos, onde máquinas virtuais, serviços e aplicações são criados sem interferência de pessoas, de maneira ágil. Para atingir essa automaticidade, várias ferramentas foram e estão sendo desenvolvidas, aproveitando-se da evolução do poder computacional, da memória, da rede e do armazenamento. Entretanto, segundo MacVittie (MACVITTIE, 2014), "a rede é o gargalo do *Datacenter*", no sentido de automação, visto que houveram poucas inovações na área.

Essa necessidade de automação exige também que haja inovação, visto que os equipamentos de rede entraram em estado de engessamento, onde os produtos oferecidos são fechados de maneira que impedem sua evolução senão pelo próprio fabricante. A fim de aumentar a capacidade de inovação em redes de computadores, foi proposto um novo conceito chamado de Redes Definidas por *Software* (SDN), no qual basicamente o equipamento de rede é controlado via software externo, partindo do pressuposto da dificuldade de se modificar um *hardware*. De forma conjunta, a Virtualização das Funções de Rede (NFV) tem como objetivo desassociar completamente os serviços de rede do *hardware* através da virtualização. Com o NFV, os serviços de rede passam a ser definitivamente independentes dos equipamentos e seus respectivos fabricantes.

Apesar de todas as mudanças relatadas causarem grande impacto no projeto de *Datacenters*, no ambiente empresarial, nas carreiras individuais e nos modelos de negócio, um dos objetivos de grande importância no *Datacenter* permaneceu o mesmo, isto é, prover conectividade da melhor maneira possível às aplicações e aos serviços computacionais necessários para suprir as demandas dos modelos de negócio existentes.

1.1 Motivação e Justificativa do Trabalho

Para uma automação completa do *Datacenter*, é necessário que a rede deixe de ser o gargalo e esteja equivalente aos avanços do processamento e do armazenamento. É preciso inserir processos automáticos que não necessitem de configuração manual do operador de redes através da interface de linha de comandos (CLI).

O modelo atual de padronização de funções e protocolos de rede, feito através de RFCs (*Request For Comments*) resulta em adoção lenta, pois exige documentação detalhada, testes intensivos, tradução para equipamentos (normalmente através de implementação em *hardware*,

por parte dos fabricantes) e aquisição dos clientes, que tem uma curva de aprendizado e um impacto na implantação da nova solução. Os consumidores da rede, pelo contrário, demandam soluções mais ágeis, que não necessitem de aquisição de um novo equipamento.

Certamente, para um modelo baseado em *hardware*, onde o custo para se produzir circuitos integrados de aplicação especifica (ASIC) é alto e um erro pode ser crucial, é importante que o modelo de padronização seja aplicado com meticulosidade. Porém, para a demanda atual de dinamicidade, uma solução focada no *software* pode adicionar benefícios, como facilidade criação e atualização, onde código é implantado e corrigido em tempo real.

Equipamentos de rede são generalistas, sendo obrigados a suportar um grande número de funções e protocolos, dos quais a maioria não são sequer utilizados. Apesar de suportar essa gama de funções, os equipamentos terminam apresentando rendimento mediano em todas as suas funções e nenhum rendimento excepcional em funções específicas. A criação de *White boxes*, isto é, *hardwares* oferecidos de maneira que passa a ser livre a escolha e instalação de um sistema operacional, normalmente sem plano de controle, objetivam resolver este problema.

Os *Datacenters* centrados no servidor são modelos que percebem esse problema e buscam suprimir os equipamentos de rede, levando os servidores à executar suas funções. O crescente poder de processamento torna possível o tratamento de grande número de pacotes na CPU. Um dos motivos da maior velocidade do *hardware* para encaminhamento de pacotes é sua memória rápida, porém cara. Para que modelos de encaminhamento em *software* sejam comparáveis ou equivalentes aos de *hardware*, permanecendo mais baratos, é necessário que o armazenamento e acesso à memória seja mínimo e que a utilização de CPU seja máxima. Assim, a rede centrada no servidor pode ter maior possibilidade de ser implantada em ambientes de produção.

Na tentativa de eliminar a dependência de *hardware* e de configuração manual, são propostas várias soluções híbridas entre *software* e *hardware*, utilizando redes sobrepostas (*overlay*) à rede efetiva *underlay*, redes virtuais executadas puramente em *software*. Porém, é importante atentar ao fato de que o recurso real é finito, e uma rede sobreposta sem integração com a rede *underlay* resultará em contenção de recursos. Faz-se, então, imprescindível que qualquer tipo de *overlay* tenha integração direta com o recurso real. Além disso, quanto mais camadas de abstração, mais latência é adicionada na comunicação, visto que mais dispositivos, sejam virtuais ou físicos, precisam processar o mesmo pacote a fim de realizar o encaminhamento necessário.

Pode-se afirmar que essas soluções possuem como foco a resolução de problemas como automação, provisionamento e inovação. A latência, entretanto, é negligenciada. Quando o ponto em destaque é a latência, os gargalos atacados são, por exemplo, filas de equipamentos ou protocolos de transporte (BRISCOE et al., 2014). Apesar de serem importantes estudos, o

atraso causado pelas camadas de abstração criadas por *overlays* obscurece seus resultados. Uma arquitetura de *Datacenter* que alcance os objetivos tanto de inovação e automatização, quanto de latência faz-se essencial.

1.2 Objetivo Geral

Este trabalho tem por objetivo propor uma arquitetura de rede de baixa latência para *Datacenters* definidos por *software*, que permita inovação de rede a longo prazo, disponibilize um elemento de rede virtual que realize encaminhamento com velocidade comparável ao *hardware* inflexível e elimine equipamentos físicos de rede, tornando-o centrado em servidores e altamente virtualizado, com processamento de pacotes na CPU dos servidores.

1.3 Objetivos Específicos

Analisar o modo atual como a rede do *Datacenter* é normalmente colocada em prática, a fim de identificar os pontos de maior lentidão e impedimentos para inovação, para então propor uma modificação à estrutura tradicional do *Datacenter* a fim de torná-la centrada no servidor, virtualizando os elementos de rede para que eles possam ser automatizados. Sendo o elemento de rede virtual e flexível, o encaminhamento deve ser executado principalmente na CPU, a fim de evitar a latência causada pelo acesso à memória típica de servidores, sendo o método de encaminhamento independente de topologia.

A consulta em tabela de fluxos para encaminhar, a quantidade de memória necessária em cada elemento de rede, além do número de consultas ao controlador, problemas comuns nas Redes Definidas por *Software*, devem ser eliminados ou reduzidos. Os gargalos de virtualização e da utilização de sistemas operacionais genéricos nos servidores devem ser eliminados, como por exemplo o tratamento de pacotes pelo kernel, os drivers de rede, a camada de hipervisor e as cópias desnecessárias de pacote.

A rede deve ser pensada tendo como base o conceito das redes de núcleo e borda da Internet, onde o elemento virtual de rede deve atuar como o núcleo, executando apenas tarefas simples com os pacotes para encaminhá-los da maneira mais rápida possível, enquanto as aplicações e os serviços devem atuar como borda, executando algum tipo de roteamento na origem. As aplicações, rodando em máquinas virtuais ou *containers*, devem ser o foco do *Datacenter*.

A camada de controle deve possuir informação de toda infraestrutura, inclusive máquinas virtuais existentes, portas virtuais e reais, além do elemento acessível em cada uma destas por-

tas. O controlador deve atuar também como orquestrador, sendo também responsável por criar e remover máquinas virtuais, executar migração em tempo real e gerenciar os recursos disponíveis.

1.4 Metodologia de Desenvolvimento

A metodologia do trabalho consistiu no estudo do problema, criação e implementação de uma solução, que constitui de uma versão evoluída de switch virtual (FlexForward) e um *software* para execução na VM. Todo o código resultado do trabalho está disponível *online*[1]. Um protótipo anterior do FlexForward foi publicado recentemente em (VENCIONECK et al., 2014).

O protótipo desenvolvido foi executado em ambiente composto por duas máquinas físicas, onde a primeira executava uma máquina virtual e ambas o método de encaminhamento. Um substituto do controlador foi igualmente executado na segunda máquina. Foram realizados testes de performance, que consistiram em medições de latência e *jitter*, comparados em seguida com os mesmos testes executados em ambientes de rede tradicionais.

O trabalho está organizado de forma que, no Cap. 2 toda fundamentação teórica é exposta, juntamente com os problemas do *Datacenter* atual e modelos utilizados de topologia. Também é apresentada a abordagem de Redes Definidas por *Software* e os problemas que causam dificuldades em sua adoção. Em sequência, o Cap. 3 expõe os trabalhos que se relacionam com esta proposta. No Cap. 4, o modelo conceitual deste trabalho é apresentado, seguido do protótipo implementado e resultados obtidos no Cap. 5. Por fim, uma conclusão, juntamente com os trabalhos futuros são apresentados no Cap. 6.

[1] http://git.lprm.inf.ufes.br

2 *Fundamentação Teórica*

2.1 Introdução

O termo "computação em nuvem" ou *cloud computing* refere-se aos serviços utilizados por empresas e indivíduos, sendo executados em ambientes cuja gerência não é diretamente realizada pelos seus usuários. Esses serviços podem constituir desde uma aplicação, passando por máquinas virtuais, até toda uma infraestrutura equivalente a um completo *Datacenter* sendo remoto o acesso à estes serviços, através da Internet.

Segundo (MELL; GRANCE, 2011), "Computação em nuvem é um modelo que permite acesso sob demanda através da rede de computadores à um conjunto de recursos configuráveis de computação como redes, servidores, armazenamento, aplicações e serviços." A diferença entre nuvens privadas, híbridas e públicas institui-se basicamente pelo nível de gerência de recursos que é permitida ao usuário, sendo maior no primeiro caso.

A grande atração da nuvem é poder dispensar a operação de uma infraestrutura própria. Eliminar a preocupação relacionada com a posse e operação de computadores, conjuntos de *storages*, redes, sistemas operacionais, servidores, *backups*, atualizações e segurança elimina também gastos, tempo e muda o foco para o que realmente importa, mesmo que o fato de terceirizar a operação do *Datacenter* para uma empresa de *cloud* traga consigo novas preocupações.

Foi com o surgimento das nuvens que ficou evidente a necessidade de uma maior flexibilização da rede. Enquanto os recursos de computação e armazenamento permitiram ser facilmente fatiados entre os usuários e, juntamente com a virtualização, automatizados para a criação de todo tipo de recurso virtual, as redes continuaram ossificadas em inovação, ampliando apenas a banda disponível. O equipamento de rede é visto como uma "caixa preta" imutável, acessível apenas por algumas interfaces limitadas. Isso dificultou a elasticidade dos *Datacenters* na nuvem. Para tentar contornar o problema, soluções de redes sobrepostas são criadas, juntamente com novos protocolos, como VxLAN (WANGUHGU, 2012), utilizado para contornar o limite

de identificadores que uma VLAN tradicional pode abarcar.

Dentre estas abordagens, destacam-se as Redes Definidas por Software (SDN) e a Virtualização das Funções de Rede (NFV). A primeira foca em separar do *hardware*, onde permanece o plano de dados, o plano de controle, executado em software e que comanda todo o comportamento do *switch*. Já a última objetiva executar serviços de rede em *software* de maneira que seja independente do *hardware*, ampliando o limite da virtualização de redes tradicional.

2.2 Conceitos de Datacenters

Após os Mainframes e antes dos *Datacenters*, a computação, armazenamento e a rede, existiam nos *desktops* de empresas. Com o crescimento dos dados e a necessidade de colaboração, esse objetivo passou a ser alcançado com servidores. Com o passar do tempo, a carga crescente já não era mais suportada por simples servidores, onde houve a necessidade de migrar para um modelo mais centralizado (NADEAU; GRAY, 2013).

O *Datacenter* tornou central os servidores, mas ainda havia grande dependência dos elementos físicos, dificuldade para migração e recuperação de falhas, pelo fato do serviço estar atrelado ao *hardware*. Como resolução à este problema, a virtualização trouxe então um novo paradigma no qual uma grande flexibilidade foi alcançada. O poder de computação pôde crescer ou diminuir dinamicamente de acordo com a necessidade de demanda, sendo este conceito nomeado como computação elástica.

O planejamento de *Datacenters* envolve a previsão de crescimento. A cada período de tempo, os elementos físicos devem ser substituídos para atender os requisitos de computação. Porém, entre o tempo de aquisição e o uso próximo ao total de sua capacidade, grande parte dos recursos fica ociosa. Observando que estes recursos poderiam ser comercializados, criam-se os serviços de computação elástica. Pode-se dizer que existem várias visões diferentes das redes de *Datacenter*, onde cada usuário faz uso de uma "fatia" para ele reservada.

2.2.1 *Multitenancy*

Comercializar recursos ociosos exige que sejam feitas configurações diferenciadas de controle de acesso, gerência de recursos e isolamento entre os usuários externos e internos, aos quais são permitidos certo controle da infraestrutura, requisitos alcançados através do conceito de *Datacenter multi-inquilino*, ou *multitenant*, onde o inquilino é definido primeiramente como um usuário de um subconjunto do ambiente de recursos (NADEAU; GRAY, 2013).

Os problemas que um *Datacenter multitenant* deve enfrentar envolvem principalmente a garantia de acesso privado à um conjunto de fatias de recursos, separando milhares de usuários em potencial, onde os recursos precisam ser propagados arbitrariamente entre diferentes máquinas virtuais em diferentes *Datacenters* físicos, ao mesmo tempo em que disponibiliza o acesso onipresente à rede interna através da Internet.

2.2.2 Virtualização

Virtualização é uma maneira de abstrair aplicações e seus componentes subjacentes do hardware que os suporta, apresentando uma visão lógica ou virtual destes recursos, que pode ser contundentemente diferente da visão física. Ela pode também criar uma visão artificial de que muitos computadores são um único recurso ou que uma única máquina é na verdade muitos computadores individuais. (KUSNETZKY, 2011). A virtualização transformou o modo como os *Datacenters* são vistos, planejados, gerenciados e mantidos. Grandes avanços somente foram possíveis devido a essa tecnologia. Toda infraestrutura pode ser virtualizada, e os avanços na área são grandes. A virtualização de redes é a única que permanece atrás em comparação com os avanços dos outros elementos de computação.

A virtualização de servidores é a mais comum, sendo muitos dos seus conceitos utilizados na virtualização em geral. Para gerenciar os recursos, um elemento chamado Hipervisor é criado. "O hipervisor é a plataforma de máquina virtual. Suas principais funções consistem em agendamento, gerência da memória e manutenção do estado da máquina virtual. Também permite criar partições para as máquinas virtuais, mantendo o isolamento entre as partições" (VERAS, 2011).

2.2.3 Computação em nuvem

A "nuvem" pode ser definida como uma infraestrutura de computação disponível em qualquer lugar do mundo pela rede. Ela complementa a Internet ao proporcionar um ambiente completo de computação de maneira remota. (GENG, 2014)

Uma série de arquiteturas de nuvem foram mapeadas para os tipos comuns de aplicações e serviços utilizados em *Datacenters*. No primeiro, chamado *Infrastructure as a Service* (IaaS) a infraestrutura física, isto é, processamento, rede e armazenamento são compartilhados. O modelo onde apenas uma ambiente de desenvolvimento de aplicação é compartilhado se chama *Platform as a Service* (PaaS). Quando apenas o aplicativo é compartilhado, para uso de vários *tenants*, temos um modelo de *Software as a Service* (SaaS). (NADEAU; GRAY, 2013)

Quando a infraestrutura de nuvem é operada exclusivamente por uma organização, ela é chamada de "nuvem privada". Quando uma organização resolve utilizar serviços prestados remotamente que são abertas ao uso público, a nuvem é chamada de "nuvem pública". Tecnicamente não existe nenhuma diferença entre os dois tipos de nuvem. A pública pode exigir mais segurança, por estar facilmente acessível. As nuvens centralizam a computação e facilitam a escalabilidade para clientes (GENG, 2014).

2.2.4 Orquestração de redes

Como um maestro que pode fazer uma harmonia completa de diferentes instrumentos em uma orquestra, aplicações devem obter um comando ou objetivo e aplicá-lo transversalmente em larga extensão dispositivos muitas vezes heterogêneos, utilizando tipicamente APIs (*Application Programmer Interfaces*) comuns entre eles. (GORANSSON; BLACK, 2014)

Em alto nível, a camada de orquestração provê a capacidade de adicionar ou remover *tenants*, de automatizar o fluxo de trabalho, de suportar um sistema de cobrança após operações de provisionamento serem realizadas, de adicionar e remover máquinas virtuais dos *tenants* e especificar a banda, qualidade de serviço e atributos de segurança na rede de um *tenant*. (NADEAU; GRAY, 2013)

Soluções de orquestração podem então ter certas políticas de alto nível que são por sua vez executados em baixo nível através do dispositivo apropriado, aliviando assim a carga de atualizar a configuração de dispositivos constantemente. Porém, as aplicações atuais, apesar de úteis para tarefas primárias como atualizações de *software* e de *firmware*, não conseguem automatizar tarefas mais complexas. (GORANSSON; BLACK, 2014)

2.2.5 A rede no *Datacenter*

A virtualização e a nuvem impactaram significativamente o *Datacenter*, em particular a infraestrutura de rede. Existem muitas demandas e um número cada vez maior de dispositivos e aplicações. Servidores e *storages* estão sendo virtualizados e a computação em nuvem está criando *pools* de recursos dinâmicos. Ainda assim, algo fica no caminho dessa expansão: a rede. Ela se tornou muito complexa, destacando-se como um ponto de foco. A solução atual de adicionar mais equipamentos para escalar tem um impacto, aumentando cada vez mais a complexidade. (GENG, 2014)

A rede do *Datacenter* ainda não está completamente virtualizada. segundo Geng (GENG, 2014), existem três desafios para as redes. O primeiro reside na habilidade de configurar a rede

facilmente. O segundo é facilitar a operação de rede em um ambiente virtualizado. Por fim, o terceiro desafio envolve as limitações de recursos, especialmente de entrada e saída.

Soluções de redes sobrepostas baseadas em hipervisor estão sendo aplicadas no *Datacenter*, a fim de resolver alguns problemas como automação ou unir redes de *Datacenters* ao mesmo domínio *broadcast*. Sob este conceito, a atual rede física é mantida sem alteração. As bordas interagem com essas redes virtuais, enquanto os detalhes da rede física ficam omitidos. O tráfego passa pelos dispositivos físicos, mas os nós das extremidades não possuem informações da topologia, do modo como o roteamento acontece ou outras funções básicas. Essas redes virtuais são controladas pelos elementos da borda, que nos *Datacenters*, são tipicamente os hipervisores das VMs que estão rodando em cada servidor.

O mecanismo que torna isto possível é chamado de tunelamento, que utiliza encapsulamento. Quando um pacote entra na borda de uma rede virtual na origem, o equipamento de rede (normalmente o hipervisor) pegará o pacote como um todo e encapsulará com outro quadro, enviando à outra borda em seguida. A borda de uma rede virtual é chamada de VTEP (*Virtual Tunnel Endpoint*) . Este mecanismo de tunelamento é chamado de MAC-in-IP, devido ao fato do quadro inteiro, do endereço MAC para dentro, é encapsulado em um pacote IP *unicast* (GORANSSON; BLACK, 2014). Esta abordagem é utilizada por diferentes fabricantes com pequenas diferenças. Entre as implementações existentes, podemos citar como exemplo a VxLAN (WANGUHGU, 2012), NVGRE (GARG; WANG, 2015) e STT (DAVIE; GROSS, 2014).

2.2.6 Arquitetura Centrada em Servidores

A arquitetura centrada em servidores foi pensada a fim de eliminar os equipamentos de rede, fazendo com que o processamento de pacotes seja realizado pelo próprio servidor. O Hipercubo (SAAD; SCHULTZ, 1989) é um exemplo desta abordagem. Há porém uma dificuldade em eliminar completamente esses elementos, gerando a criação de modelos híbridos, como DCell (GUO et al., 2008) e BCube (GUO, C. et al., 2009). Nesta abordagem os equipamentos de rede não são eliminados, sendo utilizados *hardwares* de baixo custo para prover algum tipo de conexão entre os servidores, que tem como responsabilidade o trabalho maior de processamento de pacotes. No projeto de redes centrada em servidores, o modo como é feito o roteamento interno é muitas vezes diferenciado, envolvendo níveis de hierarquia ou modelos que ajudem a criar uma carga mais leve.

Isto acontece devido à desvantagem dos servidores não possuírem ASICs e memórias rápidas como a TCAM (*Ternary Content-Addressable Memory*) . Essas memórias, presentes nos

equipamentos de rede, são capazes de encontrar resultados nas tabelas de maneira muito mais rápida, em comparação com a memória geralmente presente nos servidores. Outra grande desvantagem desta arquitetura é que o poder de processamento deixa de ter exclusividade para as aplicações e passa a se preocupar com a rede, função antes tratada de maneira básica. (VASSOLER, 2015)

2.2.7 Arquitetura a Nível de Rack

Ainda que um conceito recente, a arquitetura a nível de Rack é importante por ser uma alternativa futura ao modelo atual. Sua base é enraizada no conceito de desagregação do rack, que se refere a separação dos recursos de processamento, armazenamento, rede e distribuição de energia existentes nele, substituindo-os por módulos (INTEL, 2013a). Assim, atualizações físicas tornam-se muito mais fáceis, a flexibilidade é aumentada drasticamente e os custos são diminuídos quando comparada ao modelo atual, onde cada servidor no rack tem seu próprio grupo de recursos.

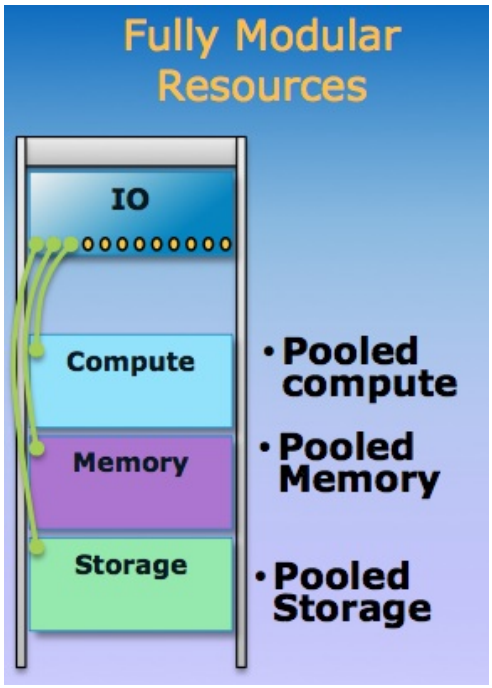

Figura 2.1: A abordagem *Rack Scale Architecture*. fonte: (http://intel.com)

Existem duas abordagens para tornar este modelo real. A primeira é um modelo de transição, onde os recursos continuam presentes de forma agregada, mas a gerência se dá de forma centralizada. Na segunda abordagem existem prateleiras de recursos, como por exemplo, prateleiras somente para processamento, disponibilizando uma gama de CPUs. Esses recursos são

gerenciados centralmente e alocados conforme a necessidade, sem dependência de um servidor físico detentor deles.

A separação dos elementos críticos torna possível a atualização de recursos físicos independentemente dos outros existentes. Assim, a vida útil dos equipamentos aumenta e cada recurso pode ser substituído, ao invés de todo o sistema. A Intel[1] é pioneira nos estudos desta arquitetura. A figura 2.1 mostra a abordagem de prateleira de recursos, tendo em vista uma comunicação fotônica, explicada a seguir, e os recursos agregados de maneira que o conceito de servidor se dilui.

Para comunicação entre as prateleiras e funcionamento dos recursos mesmo que eles estejam fisicamente distantes, são utilizados fotônicos de silício que utilizam menos energia, são mais baratos, menores e mais rápidos que os subconjuntos ópticos ou elementos de sinais elétricos. Após mais de uma década de pesquisa, o protótipo de fotônico de silício da Intel pode mover dados com uma taxa de até 100Gbps (INTEL, 2013b)

2.3 Virtualização de redes

"Um ambiente de rede suporta virtualização de redes se ele permite a coexistência de múltiplas redes virtuais no mesmo substrato físico" (CHOWDHURY; BOUTABA, 2010). Antes das Redes Definidas por *Software* se consolidarem como certo tipo de padrão para inovação, as redes já eram virtualizadas. A VLAN é um exemplo clássico. Também podem ser citadas a VPN (*Virtual Private Network*) e as redes *Peer to Peer* (P2P).

Já era sabido que a virtualização de redes seria ponto de apoio para a inovação nesta área (ANDERSON et al., 2005), sendo as SDN totalmente compatíveis com o conceito de múltiplas redes, mesmo que para isto não seja necessária a separação dos planos de controle e de dados. Contudo, a união desses modelos permite a construção novos paradigmas de inovação, habilitando novas funcionalidades em diversas áreas de pesquisa, definindo redes flexíveis que buscam resolver problemas atuais.

Para as Redes Definidas por *Software*, o benefício da virtualização de enlaces complementa a sua capacidade de generalização dos elementos de rede. A criação de segmentos lógicos separados promove um maior isolamento dos fluxos, permitindo um tratamento mais refinado no compartilhamento dos recursos de computação disponíveis na rede programável.

[1] http://intel.com

2.3.1 Redes Sobrepostas

Uma Rede Sobreposta, ou rede *Overlay*, é uma rede virtual criada no topo de uma outra rede, chamada *underlay*, podendo ela ser também virtual ou física. Elas podem ser consideradas dentro do contexto de virtualização de redes(CHOWDHURY; BOUTABA, 2010). Redes *overlay* podem ser usadas para "garantir *performance*, disponibilidade de roteamento, habilitar *multicast*, prover Qualidade de serviço (QoS), proteger de ataques e distribuição de conteúdo" (CHOWDHURY; BOUTABA, 2010).

As redes *Overlay* são constantemente utilizadas em ambientes experimentais, para projetar e avaliar novas arquiteturas. Sua implantação não causa nem requer nenhuma alteração na rede *underlay*. Por isto, são usadas com relativa facilidade e baixo custo para testar novas funções e correções para a Internet.

Entretanto, as redes sobrepostas muitas vezes ignoram ou abstraem a rede subjacente, podendo considerar ilimitados os recursos disponíveis, causando contenção. A fim de evitar isso, é importante que haja uma integração entre as redes, para que a capacidade da rede seja aproveitada da melhor maneira possível.

2.4 Programabilidade da rede

A programabilidade da rede é um princípio chave das Redes Definidas por *Software*. Embora este não seja um conceito novo, a diferença está no modo como ele é praticado nas SDN, isto é, os equipamentos de rede não são apenas gerenciados, mas há também uma interação com eles (NADEAU; GRAY, 2013). Há uma comunicação bidirecional entre os planos de dados e controle, modelo este bem diferente do método tradicional de gerência de redes, onde não há *feedback* do ativo. A programabilidade está ligada ao ato de modificar direta ou indiretamente as tabelas de rotas do elemento de rede. Indo mais além, na metodologia proposta pelas Redes Definidas por *Software*, a programabilidade não está atrelada ao limite de funcionalidades habilitadas para o *hardware* em específico, onde novos conceitos de comunicação e transmissão de pacotes podem ser implementados sem que o plano de dados seja modificado. Entre os métodos mais antigos que podem ser considerados como programabilidade, visto a função em comum entre eles de configurar um equipamento, podem ser citados a linha de comando (CLI), SNMP e NETCONF.

2.4.1 CLI

Sendo a forma mais comum de interagir com os equipamentos de rede, na CLI os comandos são digitados, influenciando no modo como o equipamento se comporta. É um método manual e lento, diferente para cada fabricante, que pressupõe as funções já instaladas no dispositivo. Normalmente a CLI é acessada remotamente, através do protocolo telnet ou SSH (*Secure Shell*). A fim de automatizar a inserção de comandos, permitindo certa agilidade, algumas ferramentas foram criadas, como por exemplo o Puppet[2].

2.4.2 SNMP

É o método mais popular de gerência de rede, possibilita também comandos de configuração, embora a abordagem inicial era apenas de consulta de informação. Portanto, a adoção para essa funcionalidade é baixa (NADEAU; GRAY, 2013). O protocolo carrega o conceito de agente, elemento a ser gerenciado, e gerente. O primeiro é responsável por manter sua informação SNMP e enviá-la para o segundo. A informação é manuseada pelas definições de sua MIB (*Management Information Base*), uma base com as variáveis a serem gerenciadas, relativas ao equipamento. A comunicação pode se dar através de *poll*, onde o gerente consulta o cliente de tempos em tempos para obter uma resposta, ou em *trap*, onde o agente envia informações para o gerente quando um evento acontece.

2.4.3 NETCONF

O NETCONF (*Network Configuration Protocol*) foi padronizado pela IETF em 2006[3]. Ao contrário do SNMP, ele foi pensado para configurar equipamentos. Com ele, operadores de rede podem escrever *scripts* para configurar os equipamentos da rede. "Em resumo, NETCONF provê mecanismos para instalação, manipulação e deleção de configuração de equipamentos de rede" (NADEAU; GRAY, 2013).

As operações básicas do NETCONF são: *get, get-config, editconfig, copy-config, delete-config, lock, unlock, close-session, and killsession*. O protocolo utiliza *Extensible Markup Language* (XML) como padrão de troca de mensagens, oferecendo também a possibilidade de receber, assincronamente, notificações de eventos do equipamento.

[2]http://puppetlabs.com
[3]http://datatracker.ietf.org/doc/rfc6241/

2.5 Software Defined Networking

O modelo convencional de redes impõe limitações aos seus administradores. Qualquer mudança de configuração avançada do equipamento, na especialização da lógica de controle e tratamento dos pacotes ou ainda, inserção de novas funcionalidades estão sujeitas a ciclos de desenvolvimento e de testes restritos ao fabricante do equipamento, resultando em um processo demorado e custoso (HAMILTON, 2011).

A partir do problema da falta de inovação nas redes de computadores, surgiram diversas propostas que visavam a centralização da inteligência de controle da rede, tais como redes ativas (TENNENHOUSE; WETHERALL, 1996), o modelo 4D (GREENBERG et al., 2005) e o Ethane (CASADO et al., 2007). Existe a possibilidade da a arquitetura de Redes Definidas por Software ser inspirada por estas. Inicialmente, o conceito de SDN estava extremamente atrelado ao protocolo que permitiu a implantação inicial de SDNs, chamado OpenFlow (MCKEOWN et al., 2008). Tal protocolo surgiu como um habilitador de inovação em redes de campus, com objetivo de permitir a criação de novos protocolos internet.

Cada fabricante possui sua visão de SDN, baseada no tipo de solução de programabilidade que oferece. Isso se deve ao fato de ser uma tecnologia em evolução, ainda em fase de amadurecimento. Como princípio base, é possível afirmar que nas SDNs sempre existem *softwares* controlando a rede, ao invés dos métodos tradicionais de controle. Há também uma separação entre os planos de dados e de controle, mesmo que incompleta.

A Cisco, por exemplo, dá enfoque maior no *hardware*, que continua possuindo parte do plano de controle. A separação se dá pelas regras de políticas de rede, que são definidas por *software* (CISCO SYSTEMS, 2014). Similarmente, a Cumulus Networks possui seu próprio sistema operacional (SO), que pode ser aplicado em soluções de *whitebox*, isto é, switches que permitem a instalação de qualquer SO. No caso da Cumulus, há a possibilidade de implementar o plano de controle juntamente com o SO, de maneira distribuída (CUMULUS NETWORKS, 2015). Para o NSX, da VMware, SDN é uma rede *overlay* virtual que não interfere nos equipamentos legados, controlada centralmente (VMWARE, 2015).

A definição mais generalista, porém é a da ONF, que diz que a arquitetura SDN tem por principais características o desacoplamento do plano controle do plano de dados, e a capacidade de ser programável por uma interface logicamente centralizada. A inteligência da rede é, assim, logicamente centralizada em controladores baseados em software que comandam e interagem diretamente com o plano de dados. (ONF - Open Network Foundation, 2015)

2.5.1 Arquitetura SDN

Os planos de controle e de dados são alicerce da arquitetura em camadas que as redes de computadores estão envolvidas hoje (SALISBURY, Brent, 2015). Dado que a arquitetura SDN separa estes dois planos, torna-se importante defini-los, numa visão baseada em Redes Definidas por *Software*. Além disso, é também adicionado o conceito de dois planos por vezes implícitos: o de gerência e de aplicação.

Plano de dados

O plano de dados manipula datagramas através de uma série de operações a nível do enlace (NADEAU; GRAY, 2013). Sua responsabilidade é analisar os cabeçalhos dos pacotes, através de uma ASIC de alta velocidade, manipulando desde encapsulamentos até políticas de rede. Ele é muitas vezes referenciado como o "caminho rápido" para processamento de pacotes, devido ao fato de ele não precisar de nenhuma consulta além do destino do pacote, quando sua tabela está programada para aquele fluxo.

Realizar operação de busca em tabelas de *hardware*, em memórias velozes, como por exemplo TCAMs (*Ternary Content Addressable Memory*), trazem historicamente resultados com uma *performance* muito maior, dominando assim o projeto de elementos de rede. Porém, os avanços em processamento de entrada e saída para processadores genéricos, estimulados pelos avanços na computação em nuvem, estão causando aumento de projetos construídos com um objetivo específico, em *software*.

As ações mais comuns (algumas delas podendo ser combinadas entre elas) resultantes da busca na tabela do plano de dados são encaminhar (em casos especiais como multicast, replicar), descartar, remarcar, contar e enfileirar. Em adição à decisão de encaminhamento, o plano de dados pode implementar alguns serviços, como ACLs (Access Control List) e QoS, que podem, em alguns casos, ter suas próprias tabelas, ou atuarem como extensão às tabelas de encaminhamento.

Plano de controle

Em alto nível, o plano de controle estabelece o conjunto de dados locais usado para criar as entradas da tabela de encaminhamento, que são por sua vez utilizadas pelo plano de dados para encaminhar pacotes (NADEAU; GRAY, 2013). Também é chamado de "caminho lento", devido ao tempo de tratamento de pacotes ser bem maior, em comparação com o plano de dados. Quanto menor for o número de pacotes tratados pelo plano de controle, maior a *performance*

da rede como um todo.

As funções do plano de controle incluem a configuração do sistema, gerência e troca de informação de rotas. Dado que as funções de controle não são realizadas para todos os pacotes, ele não possui informações diretas de velocidade e estatísticas, necessitando consultar o plano de dados para estas informações. (CHAO; LIU, 2007)

O conjunto de dados do plano de controle é chamado de RIB (*Routing Information Base*). Já as entradas da tabela de encaminhamento são chamadas FIB(*Forwarding Information Base*) . A FIB é programada uma vez que a RIB está consistente e estável. Para isso, o plano de controle deve criar uma visão da rede. Em SDNs, essa visão normalmente é completa, ou seja, abrangendo toda a topologia.

Plano de Aplicação

O plano de aplicação consiste nos *softwares* que, explicitamente, diretamente e programaticamente se comunicam com o plano de controle, informando os requisitos da rede e o comportamento que desejam da rede, em alto nível, que será traduzido aos equipamentos pelo controlador. Adicionalmente, ele pode possuir uma visão abstrata da rede, a fim de tomarem suas decisões internas. As aplicações SDN aumentam o nível de abstração das camadas abaixo, eliminando detalhes de implementação e funções particulares, tornando mais fácil a programação da rede (ONF, 2013).

Plano de gerenciamento

O plano de Gerência cobre tarefas estáticas que são normalmente manuseadas fora dos outros planos. Como exemplo, podemos citar a gerência da relação de trabalho entre o provedor e o cliente, atribuindo recursos para clientes, disposição de equipamentos físicos, coordenação de alcançabilidade e credenciais entre entidades lógicas e físicas. (ONF, 2013)

Ele se comunica com todos os outros planos, trocando informações como contratos e SLAs (*Service-Level Agreement*) com o plano de aplicação, políticas com o plano de controle e configuração de elementos com o plano de dados.

Interfaces entre os planos

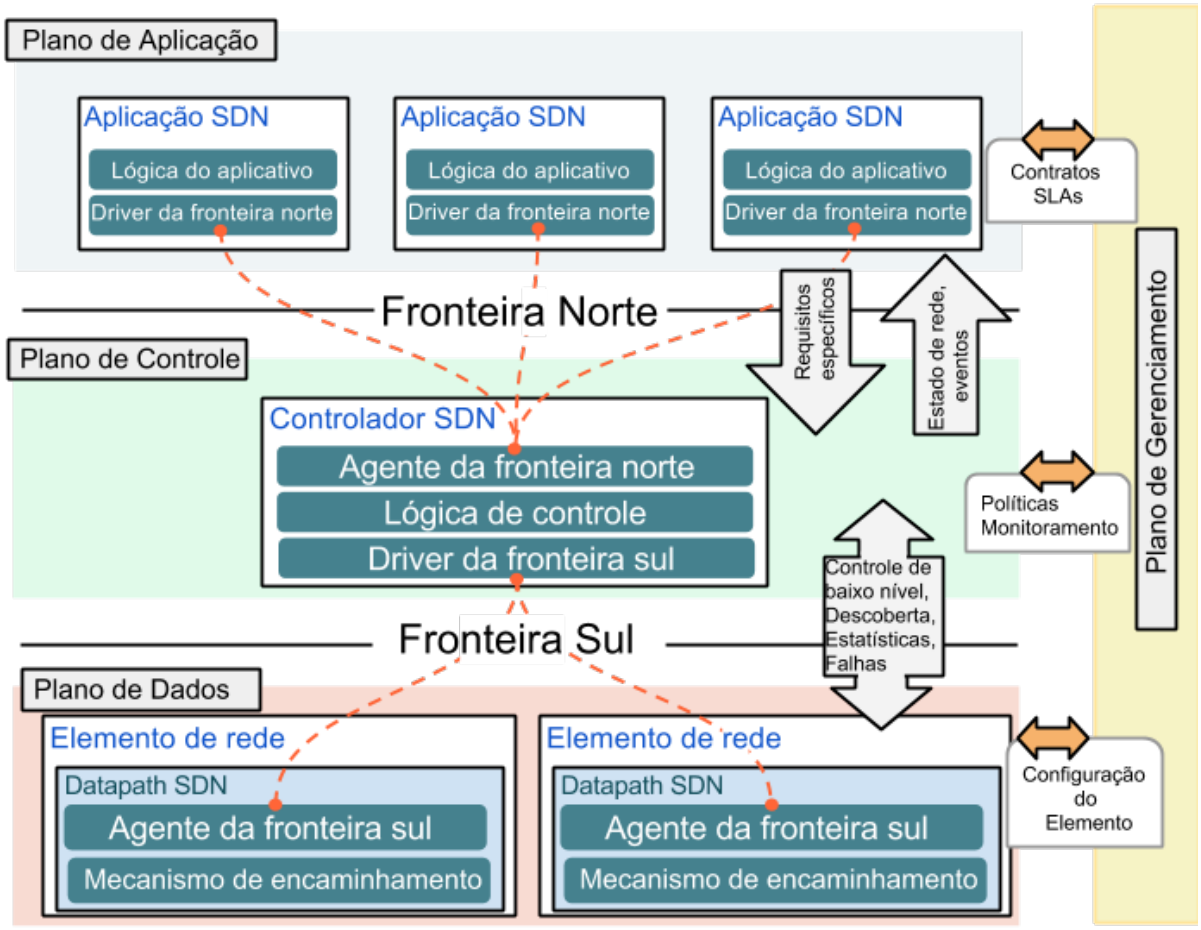

Figura 2.2: Arquitetura de Redes Definidas por *Software*, fonte: (ONF, 2013)

Como mostra a figura 2.2, o plano de dados, localizado na parte inferior, composto pelos elementos de rede, são expostos ao plano de controle, no centro, que traça a topologia e os manipula diretamente. No topo da figura, a camada de aplicação comunica seus requisitos para o plano de controle, que os traduz em comandos de baixo nível para os equipamentos de rede.

A interface entre o plano de controle e as aplicações é chamado de fronteira norte, ou *northbound*. Nesta interface transitam as mensagens no sentido controlador-aplicação, como a abstração da topologia, as estatísticas da rede e formas de autorização na rede. No sentido aplicação-controlador, pode-se citar as mensagens de definição de QoS, de roteamento e de filtros de pacotes.

Já interface entre o controlador e os elementos de rede é chamada de fronteira sul, ou *southbound*. Os comutadores geram mensagens de estatísticas do plano de dados, informações de eventos na rede e são os responsáveis por encaminhar os pacotes na rede. No sentido

controlador-comutador são enviadas as mensagens de comando de comutação, por meio da definição de estado da tabela de encaminhamento do elemento de rede. O OpenFlow é um exemplo dessa interface, pois sua especificação define as características do comutador e todas as mensagens necessárias ao protocolo de comunicação entre ele e o controlador da rede.

2.5.2 SD-Wan

A adoção das Redes Definidas por *Software* nos *Datacenters* é lenta ou em muitos casos inexistente. Uma pesquisa recente realizada em uma conferência (LERNERM Andrew, 2014) revela que a grande maioria das organizações ainda está avaliando a ideia de implantar SDN em ambiente de produção, enquanto uma parte considerável ainda não entendeu o conceito da tecnologia. Isso se dá devido ao alto custo das soluções prontas, ou da radicalidade exigida por soluções como OpenFlow, que exigem a troca completa de todos os equipamentos de rede. Faltam também aplicações de controlador padronizadas e com usos específicos, testados e prontos para serem implementados.

Nesse contexto, empresas como a Viptela[4] e Cloudgenix[5] atuam implementando uma estratégia com menor impacto e grande vantagem para os usuários, a *Software Defined Wan*. O conceito é o mesmo que as SDNs, porém o controlador não foca na rede interna do *Datacenter*, e sim na saída para a Internet. A ideia é concentrar as políticas para as aplicações na nuvem e nas comunicações externas, sendo capaz de escolher o melhor enlace para uma determinada situação, como por exemplo VPN, MPLS (*Multiprotocol Label Switching*), 4G/LTE, etc. Essa solução é vantajosa em relação aos protocolos de roteamento antigos, que escolhiam o melhor caminho (apenas um) baseado em premissas como largura de banda e métrica. A SD-Wan é vista como uma porta de entrada para as SDNs no *Datacenter* empresarial, visto que não exigem alteração nenhuma na rede existente.

2.5.3 OpenFlow

Protocolo da fronteira sul, o OpenFlow foi a primeira implementação que habilitou redes reais definidas por *software*. Emergido do meio acadêmico, motivou a discussão inicial sobre as SDNs. Hoje é mantido pela *Open Networking Fundation* (ONF), uma conglomeração de grandes empresas provedoras de conteúdo, de telecomunicações e de computação em nuvem, com o objetivo de organizar o desenvolvimento de padrões abertos de Redes Definidas por *Software*.

[4]http://viptela.com
[5]http://www.cloudgenix.com

As especificações do OpenFlow definem os comutadores e o protocolo de comunicação entre o plano de dados e o de controle.

OpenFlow usa o conceito de fluxos para identificar o tráfego com base em regras ou ações, que definem a forma como os pacotes devem fluir através dos dispositivos de rede, proporcionando um controle extremamente granular. As regras são instaladas no elemento de encaminhamento pelo controlador externo. No momento de escrita deste trabalho, ele se encontra na versão 1.4.

Segundo as especificações do protocolo (ONF, 2014), três tipos de mensagem são suportadas: controlador-para-*switch*, assíncrona e simétrica, cada uma com múltiplos subtipos. As mensagens controlador-para-*switch* são iniciadas pelo controlador e usadas para gerenciar e inspecionar o estado do plano de dados. As mensagens assíncronas são iniciadas pelo *switch* e usadas para atualizar o controlador sobre eventos e mudanças no estado da rede. As mensagens simétricas podem ser iniciadas tanto pelo controlador quanto pelo *switch* e são envidas sem solicitação, como mensagens de início de sessão ou de *keep alive*.

Workflow de pacotes

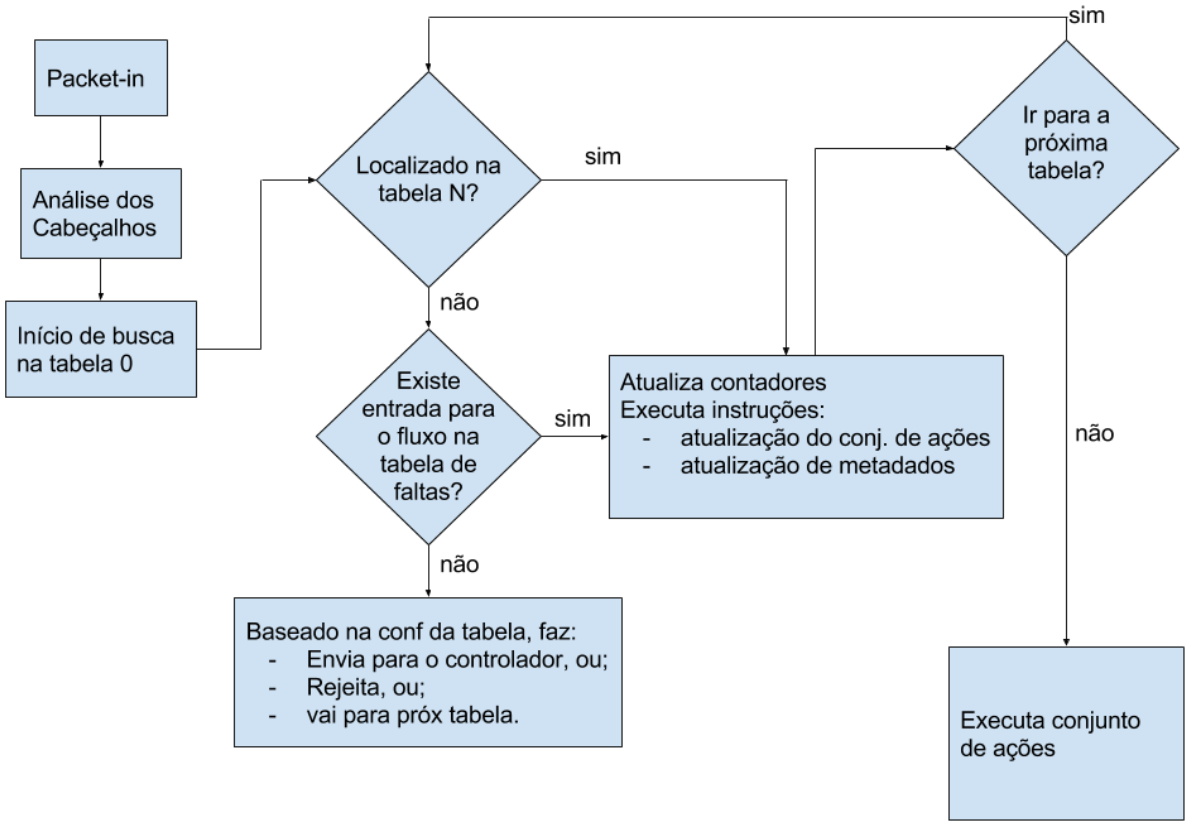

Figura 2.3: *Workflow* de pacotes do *switch* OpenFlow, em sua versão mais recente

A figura 2.3 mostra como os pacotes são tratados em um *switch* OpenFlow. Após a entrada do pacote no comutador, são realizadas buscas seguidas nas tabelas de fluxos existentes, e se nenhuma regra ser encontrada, o pacote é enviado para o controlador. Caso contrario, as ações relacionadas são diretamente aplicadas. Esse processo acontece em múltiplas tabelas com as ações sendo aplicadas em conjunto no final da busca. O pacote ao ser analisado e ter seu cabeçalho identificado, origina uma busca por regra na primeira tabela.

Se essa regra correspondente for localizada, os contadores são atualizados e são executadas três atualizações: do conjunto de ações, dos conjuntos de campos do pacote e dos metadados - utilizados para passar informação entre as tabelas. Se a regra não for localizada, dependendo da configuração da respectiva tabela, podem ocorrer um de três eventos: o pacote ser enviado ao controlador, ser rejeitado ou a busca continuar na próxima tabela. A partir da versão 1.3, foi adicionado o conceito de tabela de falhas. As regras definidas nessa tabela determinam como os pacotes não localizados nas tabelas de fluxo serão tratados. Eles podem, por exemplo, ser enviados para o controlador, rejeitados ou direcionados para a próxima tabela de fluxo, ou para uma porta conectada em uma rede tradicional. (ONF, 2014)

2.5.4 TRIIIAD

O TRIIIAD, acrônimo para *TRIple-Layered Intelligent and Integrated Architecture for Datacenters*, ou arquitetura para *Datacenter* em três camadas, inteligente e integrada, é um modelo de *Datacenter* centrado em servidores, dividido nas camadas de virtualização, encaminhamento e reconfiguração (VASSOLER, 2015). A camada de virtualização é responsável por gerenciar as VMs e redes virtuais entre elas. Nela, está presente de maneira integrada um controlador de rede um gerenciador de recursos de infraestrutura. A camada de encaminhamento é composta por *switches* virtuais controlados pelos elementos do plano de virtualização e por fim, a camada de reconfiguração são elementos que podem receber comandos de modificação das conexões físicas com intuito de reduzir o número médio de saltos da rede. Os elementos de encaminhamento são definidos por *software*, sendo o plano de controle ampliado, de maneira a controlar não apenas a rede, mas também as máquinas virtuais e recursos físicos alocáveis, sendo esta uma das grandes contribuições do TRIIIAD.

A reconfiguração das conexões físicas baseiam-se na topologia de Hipercubo, embora a ideia do TRIIIAD possa ser aplicada em outras topologias que precisam ter a estrutura pré definida e algumas vezes hierárquica para funcionar bem, havendo uma dependência muito grande de como os nós estão organizados, muitas vezes necessitando de um número ideal de elementos para o melhor caso. O modo como o TRIIIAD foi implantado enfrenta um problema

de latência, causado por consequência das muitas camadas de complexidade para que o pacote chegue ao seu destino. É preciso atravessar pelo menos 5 interfaces virtuais de rede e uma física apenas para sair do primeiro servidor físico, como mostra a figura 2.4. Cada mudança de interface é uma cópia completa do pacote, fato que ajuda a aumentar a latência.

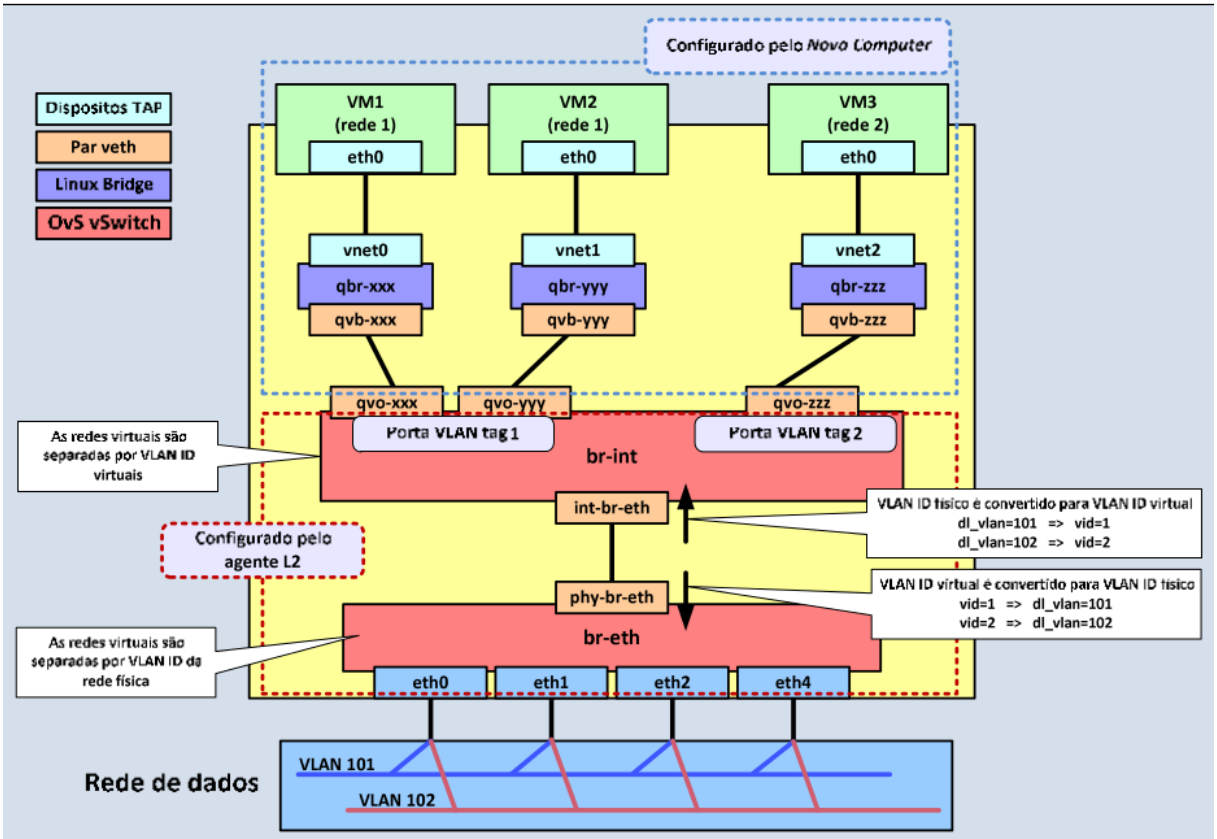

Figura 2.4: Fluxo de pacotes na arquitetura TRIIIAD fonte: (VASSOLER, 2015)

A proposta do presente trabalho diferencia-se do TRIIIAD por não depender de topologia, inserir o conceito de núcleo e borda no *Datacenter* e eliminar camadas de complexidade, como interfaces de rede, cópias de pacotes e chamadas de sistema. Apesar disto, a bordagem utilizada no plano de controle do TRIIIAD é aproveitada neste trabalho, sendo um elemento auxiliador na orquestração da rede do *Datacenter*.

2.5.5 Dificuldades na adoção

Os motivos das dificuldades na adoção das Redes Definidas por *Software* estão em discussão, visto que as vantagens parecem abundantes. Várias especulações são feitas, embora a tecnologia ainda esteja em fase de amadurecimento.

Com SDN, por algum motivo, a maioria das pessoas parecem convencidas de que a sua

adoção está intimamente relacionada com programar, ou seja, engenheiros de rede serão substituídos por programadores (HERBERT, John, 2014). De fato, SDN aumenta a necessidade de programação, porém o ideal é que as soluções sejam apresentadas pelos fabricantes de forma completa, onde os operadores sejam responsáveis por operações específicas, como por exemplo, identificar e solucionar problemas que podem eventualmente acontecer, ou cuidar de implementações específicas de políticas. Muitas vezes é inviável ter programadores de rede para desenvolver funções já abordadas em arquiteturas prontas, tendo ainda o aumento do risco de *bugs* em códigos, ou perda de foco na operação do dia-a-dia. No mercado de tecnologia da informação, mudanças acontecem todo o tempo, sendo SDN apenas mais uma delas.

Um outro motivo que pode ser citado, é a mudança radical exigida por modelos como OpenFlow, que requerem uma substituição completa dos equipamentos de rede existentes. Os novos *switches* acabam sendo extremamente básicos, dependendo completamente do controlador. O que está sendo colocado na prática, entretanto, são equipamentos híbridos, que embarcam uma inteligência suficiente para continuar em operação mesmo sem um controlador (HERBERT, John, 2014).

Há uma divergência ainda entre os técnicos, que querem sempre estar atualizados com a última tecnologia em evidência, apenas por ser interessante, e os seus superiores, que precisam de um *case* de trabalho para investigar os riscos, os resultados, a estabilidade e as vantagens do novo modelo, a fim de avaliar a necessidade, economia ou lucros da implantação da tecnologia.

Os principais problemas do OpenFLow são de latência e de tamanho da tabela de fluxos. A latência é aumentada pela comunicação entre o comutador e o controlador, que pode levar um tempo não negligenciável. Com relação a tabela de fluxos, se ela for muito pequena, o *switch* é forçado a enviar mais mensagens ao controlador. Tabelas grandes exigem mais TCAM, aumentando substancialmente o custo do equipamento. Soluções alternativas ao OpenFlow que evitem comunicação com o controlador e utilizem menor quantidade de memória são necessárias (VENCIONECK et al., 2014).

A maneira que as vantagens são anunciadas parece ser outro problema. A propaganda gira em torno da agilidade que as Redes Definidas por *Software* trazem principalmente no ambiente de nuvem, ou seja, o provisionamento de uma rede para cada *tenant* orquestrada e automaticamente é inexistente nas redes atuais, havendo necessidade de ser feito por um operador, de maneira lenta. SDNs realmente fazem esse trabalho, porém existem ferramentas de automação que realizam essa tarefa de maneira tão simples quanto seria se definido por *software*. Isso pode ser comprovado pelo fato de que as nuvens existem e oferecem soluções atingíveis sem utilizar SDNs. Deste modo a adoção em massa somente acontecerá quando houver uma necessidade

inadiável e essencial.

A grande vantagem, portanto, talvez esteja principalmente no fato das Redes Definidas por *Software* proporcionarem inovação a longo prazo na área de redes de computadores. Com o desacoplamento dos planos de dados e de controle, é possível inovar independentemente, tanto do *hardware*, quanto do *software*. Desta maneira, a inovação não fica presa ao *hardware* e vice-versa.

2.6 Network Functions Virtualization

Consiste na utilização da tecnologia de virtualização para implementação de funções de rede em *software* que pode ser executado em uma gama de servidores de padrão industrial, podendo ser movido para, ou instanciado em várias localizações na rede, como for requerido, sem a necessidade de instalação de um novo equipamento. (CHIOSI, Margaret et al., 2013)

Suas potenciais aplicações incluem: redução nos custos com equipamentos e energia ao utilizar virtualização, aumento de velocidade de provisionamento, disponibilidade de equipamento de rede com *multitenancy*, rápida escalabilidade dos serviços, visibilidade, automação, orquestração e facilidade de gerência. (CHIOSI, Margaret et al., 2013)

Com as tecnologias de NFV e SDN juntas, é possível também implantar facilmente o encadeamento de serviços, ou seja, que o fim de um serviço seja anexado na fronteira de um outro. Por exemplo, anexar uma política de segurança na fronteira da rede, ou inserir um *load balancer* em outra. Essa fronteira pode ser a divisão entre dois *tenants*, entre um *tenant* e uma rede externa, ou ainda entre redes do mesmo *tenant*. (NADEAU; GRAY, 2013)

No início da criação da ideia, não havia capacidade para torná-la real. Atualmente, com os avanços nas Redes Definidas por *Software*, na orquestração do *Datacenter* e nos avanços na virtualização, implantar ambientes com NFV está se tornando factível. Somente a virtualização não resolve todos os problemas na implantação de todos os serviços, trazendo na verdade novos problemas de confiabilidade que um sistema ou arquitetura de orquestração deve ser capaz de mitigar. (NADEAU; GRAY, 2013)

Um dos métodos de inovação para implantação do NFV, são os aceleradores de encaminhamento. Eles oferecem modelos de processamento de pacotes que ignoram algumas camadas de virtualização e kernel, tornando o *troughput*, a latência e a taxa de processamento muito mais altas. Como exemplo, citamos o DPDK e o Netmap.

2.6.1 DPDK

O Kit de desenvolvimento de plano de dados da Intel (DPDK, *Data Plane Development Kit*) é um conjunto de bibliotecas e *drivers* para processamento rápido de pacotes. Ele foi projetado para executar em qualquer processador que conhece a arquitetura x86. Existem portabilidades em progresso para outras arquiteturas, como IBM power 8. O DPDK roda principalmente em linux, com um subconjunto de funcionalidades disponíveis para FreeBSD. (INTEL, 2015b)

O DPDK tem como estratégia de resolução do problema da cópia de memória a criação de uma memória em forma de anel, que não possui interrupção para ser lida ou escrita. Além disso, existe uma memória compartilhada entre a máquina física e a virtual, permitindo que pacotes sejam enviados de uma para a outra sem cópia, pois assim que um pacote está na região compartilhada, ele está ao mesmo tempo nas duas.

Com o DPDK é possível criar qualquer pilha de protocolo, sendo possível atingir velocidades muito superiores à uma implantação comum do kernel do linux, por exemplo. Uma comparação recente (INTEL, 2015a) revela que, para pacotes de 64 *bytes*, enquanto a pilha de protocolos linux trata 12.2 milhões de pacotes por segundo (Mpps) com dois núcleos de um processador, o DPDK alcança 35.2 Mpps com apenas um. O mesmo estudo ainda revela que, com um processador de próxima geração é possível alcançar 80Mpps.

Para atingir esses patamares, o DPDK utiliza uma série de componentes que ajudam a tratar os pacotes da melhor forma possível. Dentre eles podemos citar a memória em anel, adaptada para rápidas operações em massa que armazena os pacotes em filas sem nenhum bloqueio, e também o modelo *IVSHMEM* [6], que compartilha memória entre o convidado e o hospedeiro num ambiente de virtualização, para que não haja nenhuma cópia de memória na troca de mensagens entre eles.

2.6.2 Netmap

É um *framework* projetado para alta velocidade de entrada e saída de pacotes, implementado como um módulo do kernel para o FreeBSD ou linux, ele suporta o acesso à placas de rede, pilha de protocolos do hospedeiro, portas virtuais através do *switch* chamado VALE e os chamados "tubos NetMap". Para as portas virtuais, o Netmap alcança 20 Mpps e mais de 100 Mpps nos tubos NetMap.

De forma similar ao DPDK, ele utiliza memória sem bloqueio e em anel, e também pos-

[6] http://dpdk.org/doc/guides/prog_guide/ivshmem_lib.html

sui compartilhamento de memória entre os hospedeiros e os convidados, os chamados Tubos Netmap. São utilizados processos mais genéricos, que em teoria funcionam para qualquer *hardware*. Embora a generalização possa ser vantagem em alguns casos, como por exemplo para alcançar compatibilidade maior, utilizar funções específicas, apenas em *hardware* compatível, como o DPDK, resulta em uma *performance* superior.

2.6.3 Desafios de implantação

A implantação do NFV ainda encontra uma série de desafios, pois camadas adicionais de *software* são adicionadas, aumentando o número de gargalos de *performance*, podendo comprometer as próprias vantagens prometidas pela tecnologia. Como exemplos de gargalos, podemos citar o *kernel* do sistema operacional, o *switch* virtual e a comunicação entre o convidado e o hospedeiro num ambiente virtual. (LEBLANC, 2015)

O DPDK oferece ferramentas para ultrapassar o kernel do linux, deixando que o tráfego seja executado no espaço do usuário. Porém, para garantir flexibilidade, o DPDK é responsável apenas por funções básicas de vazão pacotes, sendo necessária a criação de toda a arquitetura lógica de fluxo de pacotes. Para demonstrar um caso de uso do DPDK, a Intel criou um *pipeline*, juntamente com um switch virtual chamado DPDK-OVS[7], utilizando a pilha TCP/IP. O DPDK-OVS é baseado no Open vSwitch (OVS), mas possui um ganho notável em relação ao mesmo.

Quanto ao encadeamento de serviços, embora um controlador SDN pareça ser uma conclusão óbvia para implementação, ainda há dificuldades relacionadas ao *delay* de comunicação entre o controlador e uma aplicação de borda, tornando-se mais um gargalo de *performance* geral.

2.7 Considerações

O ambiente em nuvem precisa ser virtualizado para alcançar os requisitos de flexibilidade, disponibilidade, automaticidade e confiabilidade, requerendo também uma mudança no paradigma das redes. Equipamentos totalmente atrelados ao *hardware* não atendem aos requisitos de dinamicidade de tais ambientes. Soluções de redes sobrepostas resolvem o problema pontualmente, mas soluções de maior prazo necessitam ser implementadas.

Existe uma relação natural de ganho de flexibilidade com a perda de desempenho, necessária para o processamento adicional de novas funcionalidades. As NFVs precisam de um

[7] https://github.com/01org/dpdk-ovs

ambiente que permita haver *performance* e flexibilidade, sendo ao mesmo tempo definidas por *software*.

Acima de tudo implantar Redes Definidas por *Software* significa habilitar inovação para o futuro, abrir as portas para a resolução de problemas atuais e futuros. Apesar dos desafios, mudanças neste sentido são necessárias.

3 *Trabalhos relacionados*

A demanda por interatividade, que impacta significantemente a experiência do usuário e a receita do provedor, é traduzida em requisitos rigorosos de tempo em redes de *Datacenter*. Os protocolos da internet foram tradicionalmente planejados para otimização da vazão ou utilização do *link*. Entretanto, a qualidade de experiência entregue por muitas aplicações atuais necessitam de um tempo curto de atraso para completar pequenas transferências de dados, ou para conduzir conversações em tempo real, serviços pelos quais a adição de banda faz pequena ou nenhuma diferença (BRISCOE et al., 2014). Em contraste com a largura de banda, que é a taxa na qual os bits podem ser entregues, a latência é o tempo necessário para que um único bit crítico alcance o destino, medido desde a primeira requisição.

Assim, baixa latência é um assunto de grande preocupação tanto para indústria quanto para a academia. As redes definidas por *software*, juntamente com a virtualização das funções de rede, acrescentam ainda mais preocupações neste quesito. Considerável latência é adicionada devido a alguns fatores, como o plano de controle estar remoto, a tabela local de cada *switch* muitas vezes não ter capacidade para armazenar todos os fluxos, a própria consulta na tabela de fluxos e o *overhead* devido à virtualização dos elementos. Para que a experiência de utilização das aplicações que necessitam de baixa latência sejam aceitáveis, técnicas para eliminação dos pontos de obstrução devem ser aplicadas.

Tendo em vista que existe uma grande variedade de técnicas para a redução de latência, um recente trabalho (BRISCOE et al., 2014) buscou organizar e classificá-las de acordo com os problemas que elas procuram resolver, havendo identificação de 5, como mostra a figura 3.1: atrasos estruturais (I), interação entre os nós das extremidades (II), atrasos ao longo do caminho da transmissão (III), atrasos relacionados às capacidades do enlace (IV) e atrasos internos dos hospedeiros finais (V).

Seguindo esta classificação, o presente trabalho se posiciona em subdivisões dos problemas III e V, ou, mais especificamente, os atrasos que podem ser encontrados no caminho entre a origem e o destino, salto a salto, e os atrasos causados internamente pelo sistema operacional,

indicadas com setas vermelhas na figura 3.1. Portanto, os trabalhos apresentados neste capítulo também possuem relação direta com eles.

Figura 3.1: Classificação de técnicas de latência. fonte: (BRISCOE et al., 2014)

Ao invés de, por exemplo, aumentar a velocidade no acesso à memória, este trabalho procura melhorar a latência salto a salto através da redução drástica tanto na quantidade de consultas quanto no armazenamento de memória, embora o acesso veloz seja um fator de grande importância. Também há um outro fator de melhoria por salto, obtido através da simplificação de processamento em cada nó, onde há apenas uma operação matemática cujo resultado permite o encaminhamento. Portanto, este trabalho identifica a consulta em tabela de fluxos como outra causa de atraso. As seções a seguir revelam os trabalhos relacionados, classificados de acordo com os atrasos que eles buscam resolver.

3.1 Atrasos resultantes da operação de consulta em tabela de fluxos

3.1.1 KeyFlow

O KeyFlow(MARTINELLO et al., 2014) é um algoritmo de encaminhamento/roteamento de pacotes para redes de núcleo definidas por *software*, que não necessita de consulta em tabela, utilizado como padrão na rede de *Datacenter* proposta por este trabalho. Basicamente, cada elemento de rede possui um identificador numérico chamado chave local, configurado por um controlador externo. Para um pacote que necessita ser então enviado, um outro identificador chamado chave global é adicionado no pacote, antes de sair da borda da rede, responsável por calcular previamente todo o caminho que deve ser percorrido no núcleo KeyFlow. Assim que o pacote chega em um *switch* KeyFlow, tudo que ele necessita fazer é executar uma operação aritmética de resto da divisão (MOD) entre a chave global e a local, obtendo então o número exato da porta pelo qual aquele pacote deve ser encaminhado. Qualquer tecnologia de rede pode ser aplicada à borda, inclusive OpenFlow, desde que adicione esta chave global ao pacote.

A motivação da proposta provém do fato do modelo de redes definidas por *software* aumentar a complexidade nas redes de núcleo, aumentando consequentemente a latência de encaminhamento. Redes de núcleo foram pensadas para serem simples e rápidas, havendo um impedimento nestes requisitos quando se pensa na possibilidade de implantar SDN nelas. O principal gargalo apresentado é a tabela de fluxos, que depende do tamanho da TCAM. O KeyFlow consegue eliminar a dependência da memória com um modelo barato e rápido no encaminhamento. A abordagem propõe uma substituição de equipamentos tradicionais por outros que entendam KeyFlow, mais simples, baratos e com performance equivalente.

Por trás desta simples operação, porém, há uma A proposta do KeyFlow é baseada em um sistema numérico de restos, permitindo que um número extenso seja totalmente represen-

tado por uma combinação de números pequenos, obtidos através do resto da divisão do número maior com um conjunto de números coprimos. No modelo do KeyFlow, o número maior representa um rótulo com toda uma rota, enquanto os números menores são as portas de saída dos elementos de encaminhamento de uma rota em particular. Por fim, o conjunto de números coprimos são os rótulos locais dos *switches* (MARTINELLO et al., 2014). O controlador tem um momento inicial de criação das chaves, onde há uma análise da topologia, cálculo das chaves locais para cada *switch* e globais para cada caminho na rede. A partir deste momento, o controlador envia para cada elemento sua chave local, chegando por fim à uma rede pronta para transmitir pacotes.

As chaves locais são geradas como uma sequência de números coprimos. Para geração dos rótulos globais, em cada caminho a computação é efetuada a partir de dois vetores. O primeiro é formado pelas chaves locais de todos os *switches* e o segundo pelo número da porta de saída deles. A condição para a criação de chaves válidas é que todos os rótulos locais sejam primos entre si. Após a instalação, é possível definir um conjunto de caminhos de interesse e os rótulos globais para cada um deles. A borda define o caminho que o fluxo percorrerá, através da inserção de um rótulo disponível (MARTINELLO et al., 2014).

De maneira prática, o KeyFlow se assemelha ao OpenFlow no modo como está organizado hierarquicamente, ou seja, existe um elemento chamado controlador, logicamente centralizado, que troca mensagens com os elementos de rede. Porém, no caso do KeyFlow, não há estado da rede no *switch*, ou seja, tudo que ele possui é a chamada chave local. Cada pacote que passa pelo *switch* deve possuir uma chave global. O resultado da operação de resto da divisão entre o rótulo global e o rótulo local determina exatamente a porta de saída de cada elemento no caminho, sem necessidade de mais interações com o controlador. A figura 3.2 mostra este processo.

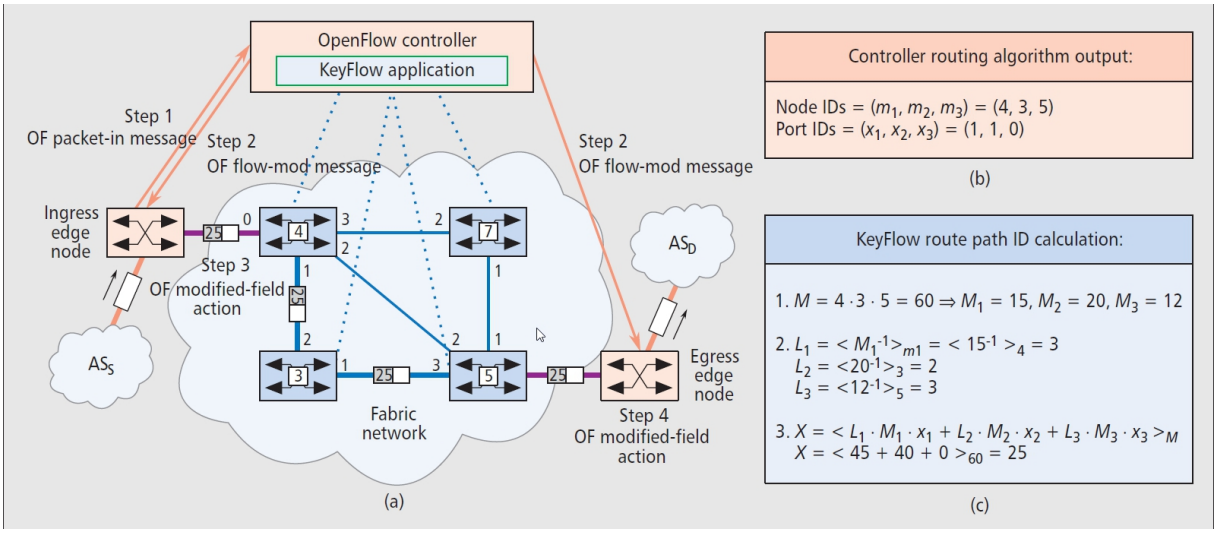

Figura 3.2: Passo a passo do modelo de encaminhamento KeyFlow. Fonte: (MARTINELLO et al., 2014)

Na figura, exemplificada como uma rede de borda OpenFlow, sendo o KeyFlow uma aplicação do plano de controle, no primeiro passo o elemento de borda solicita ao controlador uma ação para aquele fluxo, através de uma mensagem de packet-in. O controlador, ao identificar que aquele caminho necessita passar pelo núcleo KeyFlow, envia duas ações, mostradas nos passos 2 e 3. No passo 2, os nós de ambas as bordas são instruídos a adicionar uma nova entrada na tabela de fluxos(Mensagem flow-mod) contendo a ação de encaminhar os pacotes para a rede KeyFlow. O passo 3 é uma instrução de modificação(mensagem modified-field) de algum campo do pacote, inserindo a chave global. Os *switches* KeyFlow encaminham o pacote até a borda de destino, que recupera (passo 4) o campo modificado na borda anterior.

3.1.2 FlexForward

O FlexForward(VENCIONECK et al., 2014) é um *switch* virtual projetado para redes definidas por *software*. Trata-se de uma extensão do Open vSwitch (OVS), sendo base para criação facilitada de métodos de encaminhamento que utilizem o mínimo de memória, menor comunicação com o controlador e operações simples de CPU, ao invés de tabelas tradicionais baseadas em *matching* para encaminhar pacotes, configuradas diretamente por um plano de controle externo. Além disso, ele oferece a possibilidade de alternância entre os métodos de encaminhamento em tempo real. Neste trabalho, o FlexForward é o *switch* virtual da rede proposta, estando presente em todos os servidores para execução de encaminhamento virtual.

A latência de comunicação entre o plano de dados e o controlador, juntamente com a rela-

ção entre o tamanho da tabela de fluxos e o custo (tabelas pequenas aumentam a latência por ser necessário remover entradas quando cheias, enquanto tabelas maiores aumentam substancialmente o custo dos ativos) motivam esta abordagem. Por requerer uma rede centrada em servidores, ao eliminar equipamentos físicos, movendo a função de rede para elementos virtuais, há uma melhoria no custo-benefício envolvido. Entretanto, se forem utilizados os mesmos métodos de encaminhamento, além de competir em recursos com as aplicações, o encaminhamento torna-se muito mais lento, se comparado à abordagem com *hardware* dedicado. O diferencial do FlexForward é portanto permitir a eliminação da consulta em tabela através da possibilidade de criação e aplicação ágil de novos métodos que utilizem o mínimo possível de memória e aproveitem de operações de CPU, visto que os servidores possuem, como vantagem natural em relação à elementos de rede, processadores excessivamente mais rápidos.

Baseando-se então no argumento que pressupõe o *matching* em tabela armazenada na memória de servidores como mais lento que operações de CPU para encaminhar pacotes em redes centradas em servidores, o Flexforward modifica o OVS de acordo com o esquema ilustrado na figura 3.3. Os métodos de encaminhamento são armazenados no módulo do kernel, chamados pela função *flexforward_execute*. O controlador, embora seja OpenFlow, necessita também entender as mensagens adicionais, criadas como "extensões de fabricante", sem modificar as mensagens padrão. Toda informação de memória necessária pelo método de encaminhamento é armazenada no sistema de arquivos *sysfs*, do Linux. Esta informação é enviada pelo controlador, através da mensagem *SET_FWMODEL*. Assim que o método de encaminhamento é selecionado pelo controlador, os pacotes conseguintes são enviados diretamente para a função de saída de pacotes, passando como parâmetro a porta obtida anteriormente através do método de encaminhamento customizado, sem passar pela consulta em tabelas. O algoritmo fornece ainda a possibilidade de utilizar OpenFlow como método de encaminhamento, igualmente através de um comando do controlador, utilizando-se da tabela neste caso específico.

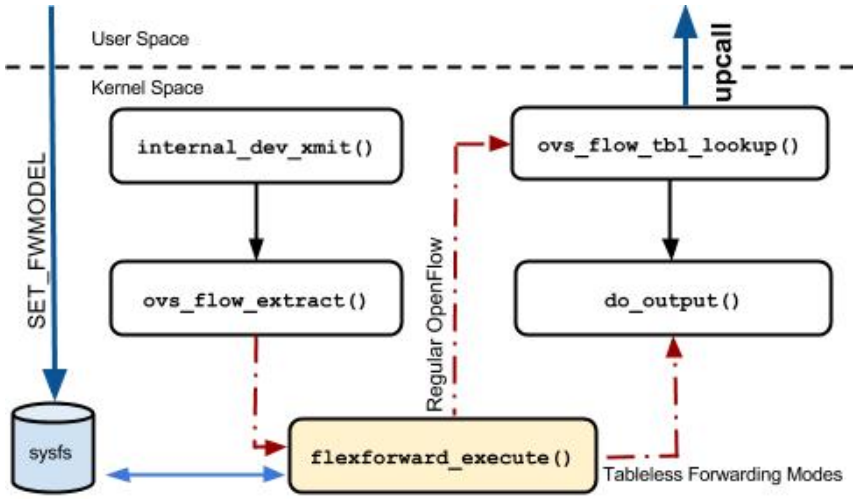

Figura 3.3: Visão de implementação do FlexForward em relação ao OVS fonte: (VENCIONECK et al., 2014)

Como prova de conceito, foram implementados dois métodos de encaminhamento existentes que possuem as características exigidas pela abordagem: KeyFlow, já apresentado anteriormente, e Hipercubo. O Hipercubo é um método que necessita uma topologia pré-definida. Sua configuração mínima é apresentada na figura 5.1, no capítulo 5. Quando completo, todos os nós possuem o mesmo número de vizinhos. Para cada nó é configurado um número identificador, sendo que nós vizinhos possuem apenas um bit de diferença entre seus identificadores. Quando um pacote precisa ser enviado, uma operação lógica XOR deve ser feita a fim de encontrar a porta correta de saída.

De forma genérica, o FlexForward segue a seguinte lógica. No princípio o controlador aprende a topologia da rede e determina o modelo de encaminhamento, enviando a mensagem de configuração de método para todos os elementos da rede. Então, quando um pacote passa por um *switch* configurado, não há necessidade de comunicação com o controlador para definir o que fazer com ele, bastando executar o método escolhido. Caso o controlador identifique que é necessário modificar o método, uma mensagem é enviada para cada elemento de rede no caminho. Os resultados dos experimentos realizados com esta abordagem encontram-se no capítulo 5.

3.2 Atrasos ao longo do caminho da transmissão

3.2.1 NOX-MT

Para Redes Definidas por *Software*, o controlador pode ser o gargalo. Tootoonchian et. al (TOOTOONCHIAN et al., 2012) citam estudos que mostram que, por exemplo, um *cluster* com 1500 servidores envia uma média de 100 mil fluxos por segundo para o controlador, 100 *switches* podem ter picos de 10 milhões de fluxos por segundo no pior caso, e, ainda, que 10 ms de atraso no tratamento de um fluxo pelo controlador pode adicionar 10 % de atraso na maioria dos fluxos para esta rede.

Para diminuir este atraso de comunicação, é proposto um controlador chamado NOX-MT (TOOTOONCHIAN et al., 2012) que utiliza técnicas como processamento em lote de entrada/-saída e uso de alocações de memória quem levam em consideração a existência de processadores com vários núcleos.

Este trabalho, ao contrário, propõe uma rede onde os elementos de encaminhamento se comuniquem o mínimo possível com o controlador, eliminando a necessidade de tabelas de fluxos nestes elementos. Do mesmo modo, nas comunicações inevitáveis com o plano de controle, técnicas como as utilizadas no NOX-MT podem ser utilizadas.

3.2.2 CLOTS

A fim de propor um *Datacenter* de baixo custo e com baixa latência, o CLOT (WANG et al., 2015) utiliza uma topologia do tipo torus e um algoritmo de roteamento denominado POW. A topologia torus permite várias dimensões. A 3D, por exemplo, normalmente utilizada em sistemas de computação paralela, conecta cada servidor com até 6 outros adjacentes, onde o roteamento acontece sem necessidade de *switches*. A comunicação pode fluir por 6 diferentes direções: X+,X-,Y+,Y-,Z+ e Z-. Para que seja diminuído o diâmetro do *Datacenter* e consequentemente sua latência, a topologia CLOTS inclui ainda alguns *switches* de baixo custo.

Para melhor comunicação, a pesquisa propõe que a camada IP seja substituída por um sistema de endereçamento por coordenadas, facilitando o roteamento. Para continuar compatível com o protocolo IP, existe um sistema de tradução de endereços. Os 32 bits do endereço IPv4 são utilizados para mapear até 6 dimensões, onde cada dimensão é representada por um grupo de 5 bits. Os 2 bits restantes são utilizados para identificar o número de dimensões que estão sendo usadas.

Por fim, o algoritmo de roteamento POW (*Probabilistic Oblivious Weighted*) oferece controle de fluxo e balanceamento de carga para a topologia. Para selecionar o próximo nó na necessidade de encaminhar pacotes, cada direção possível é associada com uma probabilidade baseada na distância entre o nó adjacente e o destino final, sendo que, no fim, o *next-hop* com maior probabilidade é escolhido. Para controlar o fluxo, o algoritmo utiliza um sistema de créditos, onde o pré-requisito para cada nó enviando dados é que a porta de destino tenha créditos suficientes. O valor padrão de créditos é o número de pacotes que o nó destino pode aceitar. Assim que um pacote é enviado, um crédito é removido. Se o nó destino liberar espaço no *buffer* abrindo caminho para mais um pacote, ele envia um crédito para o remetente. Por se tratar de um trabalho recente, ainda não existem resultados além dos teóricos para o CLOTS.

3.3 Atrasos internos dos hospedeiros finais

3.3.1 NIQ

Tendo em vista que as interfaces de rede Ethernet são projetadas com foco em atingir alta vazão com baixa utilização de CPU e, com isso, muitas vezes sacrificam a latência, o NIQ (*Network Interface Quibbles*)(FLAJSLIK; ROSENBLUM, 2013) é uma abordagem "losa branca", ou seja, substitui completamente a tecnologia antiga por uma nova. A proposta é criar uma nova placa de rede com foco em latência sem que a banda seja sacrificada.

Para isto, o projeto é focado em duas fontes de atraso: 1) a comunicação/transferência de dados entre a CPU e a placa de rede, juntamente com 2) a gerência de energia, implementado através da combinação de técnicas existentes como o encapsulamento de pequenos pacotes em descritores, com novas, como políticas de *caching*. O protótipo da placa de rede foi desenvolvido em FPGA, com uma aplicação simples e rápida de armazenamento, que utiliza *buffers* em anel. O driver da placa é executado em modo de usuário, com cópia-zero de pacotes e acesso direto através do *bypass* da pilha de protocolos do *kernel*.

Mesmo utilizando técnicas parecidas, o presente trabalho procura manter a infraestrutura atual, gerando menor impacto e obtendo os mesmos ganhos, em comparação com a NIQ. Além disso, o escopo é muito maior, abrangendo também a latência causada pela virtualização, pela consulta em tabela e pela comunicação com um controlador remoto.

3.3.2 SDNFV

A fim de integrar SDN com NFV, a proposta do SDNFV (WOOD et al., 2015) é utilizar um *switch* virtual chamado NetVM, que possui capacidade de fazer *chaining* de serviços mantendo o encaminhamento de pacotes próximo à *line-rate* de 10G com baixa latência, utilizando DPDK. A comunicação entre VMs e de VM para hospedeiro é através de memória compartilhada, onde o todos os elementos acessam a mesma área de memória, com acesso a todos os pacotes, como mostra a figura 3.4.

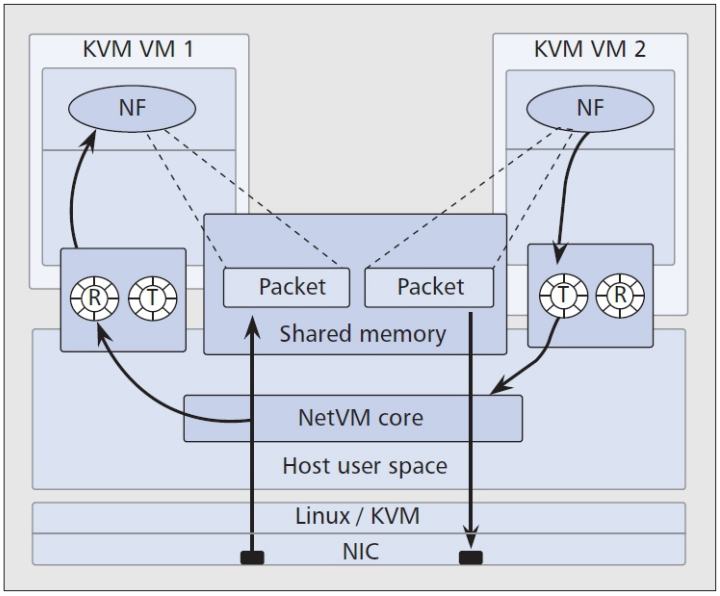

Figura 3.4: O NetVM compartilha memória entre todas as VMs. Fonte: (WOOD et al., 2015)

Em comparação com técnicas existentes que utilizam recursos de virtualização tradicionais como SR-IOV, a NetVM apresenta melhoria de mais de 250% em vazão e latência. O uso de memória compartilhada entre múltiplas VMs não é utilizado no FlexForward, havendo isolamento de fluxos, fator que permite *multitenancy* e segurança, apesar de haver um pequeno aumento na latência para comunicação entre VMs de um mesmo host.

3.3.3 SoftNIC

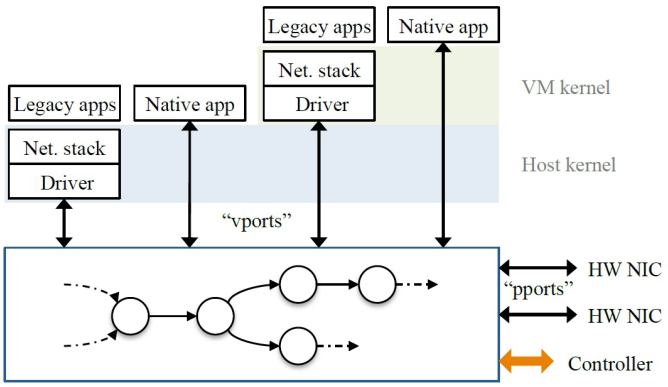

Figura 3.5: Arquitetura de aplicação com SoftNIC. Fonte: (HAN et al., 2015)

A SoftNIC (HAN et al., 2015) é uma arquitetura híbrida de *software* e *hardware* para controle e implementação de funções de placa de rede em servidores. A parte física é uma placa de rede sem modificações. Entretanto, o que é oferecido é uma espécie de *driver* executado no espaço de usuário da máquina física, que diminui a latência sem interferir na vazão, além de permitir que novas funções sejam adicionadas à placa de rede, como políticas de isolamento de performance ou limite de uso da banda disponível.

Para isto, o *driver*, que possui 14 mil linhas de código, é disposto em módulos, executados em *pipeline*, sendo possível adicionar ou removê-los facilmente, havendo possibilidade de incluir um controlador SDN. Apesar de haverem conceitos semelhantes aos do FlexForward, como portas virtuais e físicas, *bypass* do *kernel* através do DPDK, dedicação de *cores* para processamento de pacotes e aplicações legadas, a ideia de *switching* é tida como um módulo, enquanto o FlexForward é o *switch* em si. O principal objetivo da SoftNIC é permitir a adição de funcionalidades à placa de rede em *software*, enquanto que o do FlexForward é permitir adição facilitada de novos métodos de encaminhamento sem consulta em tabela.

No quesito latência, a figura 3.5 mostra que aplicações legadas na SoftNIC ainda executam o *driver* do *kernel* do Sistema Operacional, havendo um certo ganho apenas em VMs, onde os pacotes não passam pela pilha de protocolos nem *driver* do hospedeiro. Em contrapartida, este trabalho elimina mais camadas de atraso para aplicativos legados, visto que tanto no sistema hospedeiro quanto no convidado, apenas a pilha de protocolos precisa ser executada, através de um intermediário executado na VM e da utilização de uma interface de rede do tipo TAP.

4 *Uma Rede de Datacenter Centrada em Servidores, Definida por Software e de Baixa Latência*

4.1 Introdução

As redes definidas por *software* provêm principalmente da necessidade de habilitar inovação num ambiente restrito, com forte dependência de fabricantes de equipamentos. Sendo as exigências de banda, latência e provisionamento cada vez maiores no *Datacenter*, elas buscam alcançar um modelo mais aberto que permita que novos tipos de aplicações, serviços ou paradigmas que agreguem valor à rede sejam implantados de forma mais rápida e autônoma. Tendo em vista que o provisionamento manual de infraestrutura e serviços tornou-se obsoleto, automatização e flexibilidade passam a ser conceitos fundamentais das redes do futuro.

Entretanto, uma infraestrutura centrada em *hardware* sofre com várias insuficiências, como gerenciabilidade, flexibilidade e extensibilidade. Dispositivos de rede normalmente suportam uma gama de comandos e configurações baseados num sistema operacional ou *firmware* específico (FARHADY; LEE; NAKAO, 2015). Tendo em vista esta grande complexidade em conceber um modelo suficientemente flexível, aberto e que permita inovação, várias tendências parecem convergir ao ponto de tornar o *Datacenter* definido por *software* como um todo (WIPFEL, Robert, 2015).

Embora a virtualização completa dos serviços possa trazer automatização e implantação rápida dos mesmos, sua aplicação resulta no decaimento da performance e aumento de latência, chegando a níveis não tolerados para alguns casos. É desejável para a grande maioria dos serviços do *Datacenter* que haja baixa latência, ou seja, rápido encaminhamento de pacotes com alta vazão mesmo que exista virtualização. Existe ainda um modelo intermediário, onde, utilizando a mesma infraestrutura legada, são aplicados elementos virtuais de rede na borda, resultando assim numa SDN *overlay*.

Este capítulo propõe um modelo de rede para *Datacenter* executado em *software*, onde exista a possibilidade de automatização, flexibilidade, inovação, simplicidade de encaminhamento e principalmente baixa latência. Visto que muitas dessas vantagens são alcançadas com o modelo atual de SDN, a grande contribuição deste trabalho está na redução drástica de latência, alcançada pela identificação e eliminação de pontos gargalos pouco explorados em trabalhos anteriores.

4.2 Pontos de obstrução na rede de Datacenter

Tendo em vista os desafios de rede a serem enfrentados no *Datacenter*, as soluções existentes atacam problemas específicos que precisam ser integradas como um todo para amadurecimento das mesmas.

Permitir que os equipamentos físicos de rede sejam programados de maneira mais livre garante certa flexibilidade, porém é altamente dependente da tecnologia SDN escolhida. Com *hardware* de rede, uma nova versão do protocolo OpenFlow, por exemplo, exigiria substituição do equipamento, no caso de implementação em ASIC. Caso contrário, acarretaria em um aumento de latência, pois normalmente as CPUs e memórias que rodam *features* atualizáveis de equipamentos de rede são muito mais lentas que a combinação ASIC-TCAM.

Switches e roteadores virtuais, por outro lado, que possuem código aberto ou que sejam facilmente atualizáveis, são uma forma eficaz de se acompanhar a evolução constante do plano de dados. Isto garante programabilidade e flexibilidade, além de dar espaço para a implantação do NFV. Porém, transferir ASICs e TCAM para x86 e RAM resulta no decaimento de *performance* e aumento de latência. Passa a ser necessário que elementos virtuais de rede tenham processamento mais rápido de pacotes.

Esse requisito motivam os aceleradores de encaminhamento, como o DPDK. Eles identificam basicamente três gargalos na rede em servidores: as muitas cópias de memória para que o pacote atravesse a placa de rede, as interrupções exigidas pelo sistema operacional na pilha de protocolos e *drivers* de rede, com mudanças de contexto e chamadas de sistema até que se chegue na função efetiva de encaminhamento, e por último a competição que os processadores devem lidar entre a execução de funções de rede e o processamento necessário para o sistema operacional como um todo.

Para ultrapassar a pilha de protocolos, chamadas de sistema e *drivers*, o acelerador de encaminhamento provê APIs e bibliotecas com funções que permitem acesso direto à placa de rede, retirando portas físicas da gerência do kernel. É possível ainda obter uma reserva de processa-

dores lógicos, responsáveis apenas pelo processamento de pacotes, a fim de que outras funções existentes no servidor não sejam interferidas.

Estas propostas apresentadas permanecem separadas e validas apenas para problemas específicos, faltando uma junção completa entre elas. Este trabalho propõe uma arquitetura de *Datacenter* que integra esses elementos, tornando-os viáveis de serem implementados. Além disso, outro gargalo é identificado na comunicação envolvendo elementos virtuais de rede, que é a o operação de consulta em tabela, eliminada com a criação do *switch* chamado FlexForward, que possui a habilidade de executar métodos de encaminhamento que exigem mais processamento pela CPU do que consulta em memória, entre eles o KeyFlow.

4.3 A abordagem teórica

Apesar do FlexForward permitir outros métodos de encaminhamento, verifica-se que o KeyFlow possui uma motivação bastante similar à do FlexForward: rapidez no encaminhamento através da eliminação da consulta em tabela. Mesmo que a proposta do KeyFlow defenda uma aplicação apenas no núcleo da rede, sua inserção no *Datacenter* pode trazer como consequência secundária, além da diminuição da latência, sem necessidade de adição de nenhum outro recurso ou protocolo, simplificação, isolamento, políticas de acesso, encadeamento de serviços, criação facilitada de *overlays*, entre outros, revelando-se um eficaz modelo para a rede do futuro.

Ao tornar possível a integração de aceleradores de tratamento de pacotes com métodos de encaminhamento que não dependem de tabela como o KeyFlow, o FlexForward revela-se inadequado para implantação em *Datacenters* sem que haja nenhuma alteração, devido aos seus pré-requisitos implícitos, seja pelo método de encaminhamento, seja pelo próprio *vswitch*. Para que a baixa latência seja alcançada, o método de encaminhamento não pode ser executado de maneira sobreposta, exigindo uma infraestrutura centrada em servidores. Outra exigência é o roteamento em borda, visto que normalmente os métodos de encaminhamento não armazenam o estado da rede, estando ele no controlador ou nas bordas. Cada método de encaminhamento exige mudanças particulares no *Datacenter*, como por exemplo o modo de endereçamento ou a topologia.

A principal contribuição deste trabalho é apresentar um modelo arquitetural de rede de *Datacenter* que utilize o KeyFlow através do FlexForward. Esta rede é chamada LodeNet, possuindo este nome 2 significados. O primeiro é um acrônimo de **Lo**w **De**lay **Net**work, apontando para o fato de ser uma rede com baixa latência e menor atraso. O segundo significado provém

das palavras em inglês *Lode Network*, com *lode* sendo traduzido como filão ou veio, um termo de mineração que se refere a uma camada útil de algum tipo de minério que esteja envolto por camadas de outros tipos de matéria. A rede proposta neste trabalho é uma analogia desta camada útil, onde as camadas adjacentes desnecessárias são eliminadas.

O *Datacenter* centrado em servidores abre as portas para flexibilização do elemento de rede, uma vez que ele deixa de existir fisicamente. O grande problema dessa abordagem é o compartilhamento de recursos entre os aplicativos/serviços e o encaminhamento de pacotes, podendo causar sobrecarga no servidor. Para eliminar este problema, na LodeNet alguns núcleos de processador são alocados exclusivamente para tratar pacotes, aproveitando-se do grande número de núcleos que cada processador possui, além da grande quantidade de pacotes que cada núcleo atual é capaz de processar. Processadores recentes possuem capacidade de tratar, pela junção de seus núcleos, pacotes em *line-rate* de 50G, utilizando aceleradores de encaminhamento. (INTEL, 2015a)

Igualmente, centralizar o *Datacenter* no servidor significa executar um *switch* virtual. O modelo de consulta em tabela para encontrar uma ação àquele determinado fluxo ou pacote não funciona bem neste caso, dado que as memórias dos servidores em geral não são tão rápidas como TCAMs nem podem usufruir de ASICs especializados. O FlexForward juntamente com o KeyFlow eliminam esta barreira.

A LodeNet tem como elemento central a instância que possui processamento, armazenamento e rede, conceito que, embora se aplique à servidores, pode ser estendido para, por exemplo, racks que possuam estes elementos agregados. Pode-se, portanto, dizer que esta proposta possui uma arquitetura centrada em servidor. Isto significa que os equipamentos de rede são completamente eliminados e todo elemento responsável pelo encaminhamento é virtual. Não há nenhum tipo de atrelamento em relação à topologia. Há ainda uma redução no custo final, relacionada à economia realizada ao remover elementos de rede físicos, além da garantia de flexibilidade e abertura para inovação.

Cada servidor possui recursos que podem ser "fatiados" entre cada máquina virtual ou aplicação. Visto que servidores tornam-se também encaminhadores, naturalmente o controlador, além de ser responsável pelos elementos de rede, proporciona orquestração e automação ao gerenciar os recursos, criar *slices* e máquinas virtuais. Tendo como inspiração o plano de controle TRIIIAD (VASSOLER, 2015), o controlador torna-se o elemento central tanto de configuração de rede, como de configuração do *Datacenter* como um todo. Apesar do FlexForward ser a princípio virtual, não há impedimento para sua execução em, por exemplo, um chip, na própria placa ou bandeja de rede.

A tecnologia de NFV é fundamental na LodeNet, tendo em vista que todos os serviços, aplicações e elementos de rede do *Datacenter* são executados em *software*. A fim de eliminar então o gargalo causado pelas comunicações constantes entre o hospedeiro (servidor) e o convidado (aplicação, *container*, serviço ou máquina virtual), a abordagem prevê a existência de um acelerador de processamento de pacotes, que ultrapasse as camadas de virtualização, entregando diretamente o pacote da aplicação para o *switch* virtual.

Os servidores são fisicamente ligados uns aos outros utilizando elementos da primeira camada do modelo OSI. Cada servidor possui uma instância do FlexForward, sendo que todo pacote deve passar por ele, mesmo que a comunicação esteja sendo feita entre VMs de um mesmo *host*. A vantagem do KeyFlow como método de encaminhamento é sua rapidez, necessitando de apenas alguns bits de memória em cada elemento de rede para o cálculo da porta pela qual o pacote deve ser enviado. Devido a este fato, memórias mais rápidas e próximas do processador, como registradores, podem ser utilizados para guardar a chave local em cada *vswitch*, aumentando a velocidade de processamento do pacote. Se forem necessários, por exemplo, 96 bits, podem ser utilizados 3 registradores de 32 bits ou 2 de 64 bits.

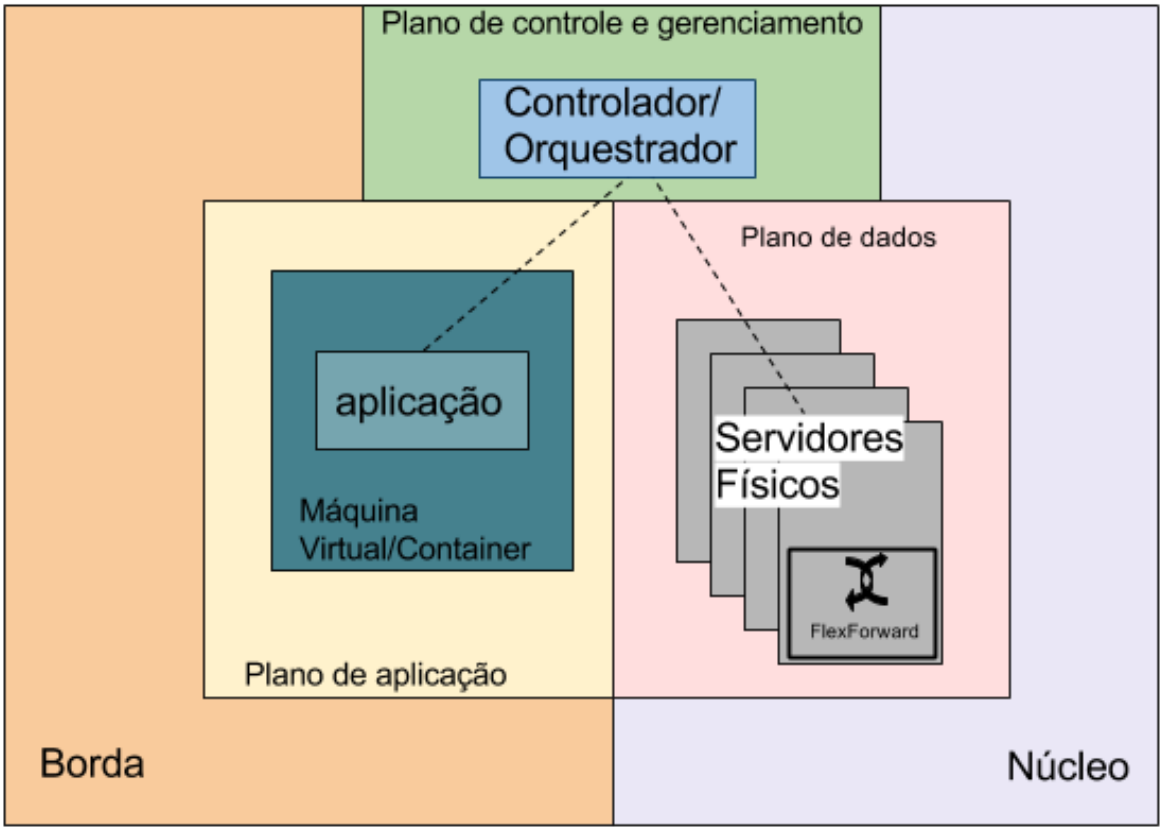

Figura 4.1: Arquitetura da LodeNet

A arquitetura com os elementos da LodeNet e suas classificações encontram-se na figura 4.1. As subseções conseguintes descrevem e posicionam cada elemento.

4.3.1 Núcleo e Borda

Devido à aplicação do método de encaminhamento KeyFlow, o conceito de borda e núcleo, assim como entendido na Internet, precisou ser aplicado no *Datacenter*. O núcleo do LodeNet tem como única função encaminhar pacotes. Ele deve ser o mais simples possível, a fim de que o mínimo de latência seja adicionado, quando um fluxo qualquer atravessá-lo. Faz parte do núcleo o *switch* FlexForward. Com o KeyFlow, a borda não necessita possuir estado da rede, nem grande memória para encaminhamento. Já a borda é responsável por obter informações mais complexas como rotas, políticas de encaminhamento e estado da rede, todas traduzidas na chave global do KeyFlow. Representada pela aplicação, ela recebe essas informações do controlador, elemento este que, além de possuir toda a inteligência da rede, está tanto na borda quanto no núcleo. Tendo como política, por exemplo, que uma aplicação X não deve se comunicar com a aplicação Y, a chave KeyFlow não é informada para X.

4.3.2 Modelo de camadas

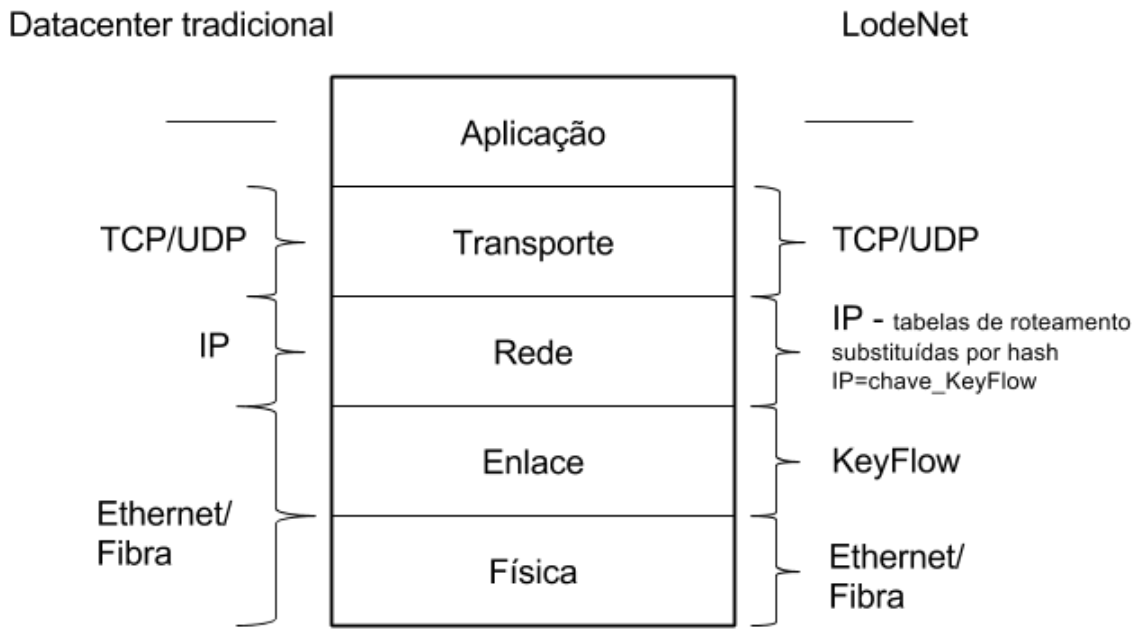

Figura 4.2: Modelo de camadas da LodeNet em comparação com o modelo TCP/IP

Há uma mudança nas tecnologias utilizadas pela LodeNet nas camadas de rede, quando comparada ao modelo TCP/IP. Como pode ser visto na figura 4.2, a camada de rede é modificada em menor grau, enquanto a camada de enlace é completamente substituída pelo KeyFlow. Para a camada de rede, o IP não é modificado. Porém, a aplicação que roda na borda substitui a sua tabela de rotas por uma hash que, para cada destino IP, retorna uma chave global KeyFlow para ser colocado no pacote. Existe ainda a possibilidade de execução de aplicativos legados. Neste caso, são adicionadas apenas duas entradas na tabela de rotas: a rede local e o *default gateway*. A seção 4.3.6 explica como funciona a tradução desta tabela para a hash, que permanece como intermediária. Já a camada de enlace mantém o cabeçalho ethernet, sendo portanto compatível com este padrão. Porém, os campos deste cabeçalho são rearranjados de forma a se adequar ao padrão KeyFlow definido.

4.3.3 O plano de controle e gerenciamento

O plano de controle nesta abordagem é uma extensão do modelo SDN, abarcando também outras funções como levantar máquinas virtuais e gerenciar recursos disponíveis, por isso chamado de plano de controle e gerenciamento. O controlador possui todas as possíveis rotas no *Datacenter* e as envia para as aplicações quando solicitado, estejam elas em *containers*, VMs ou quaisquer outras tecnologias que possam executar os serviços. Além disso, os elementos do núcleo são informados pelo plano de controle de suas chaves locais assim que a rede se inicia.

As políticas de rede e informações de *tenants* são de responsabilidade do controlador, que traduz esses dados em chaves KeyFlow locais e globais para os *switches* virtuais e aplicações, inserindo regras, podendo bloquear ou liberar tráfego se necessário.

4.3.4 O plano de dados

Cada servidor é associado à um *switch* virtual. Os *switches* virtuais recebem sua chave local KeyFlow do controlador, e não solicitam nenhuma informação à ele. Cada pacote que chega é então processado a partir de um campo de seu cabeçalho, que contém a chave global. O resultado da operação MOD entre chave global e a local informam a porta pela qual o pacote deve ser enviado e esta é sua única função. O FlexForward é o *switch* virtual do plano de dados.

4.3.5 A aplicação

A aplicação é o serviço que necessita ser executado, ou seja, o foco real do *Datacenter*. O modelo proposto de comunicação de baixa latência tem como objetivo final que as aplicações executem no ambiente mais propício possível, para que seus usuários tenham uma boa experiência de uso. Estando ela isolada do *vswitch*, é necessário um módulo de comunicação com o mesmo, para que a latência ganha não seja mascarada, por exemplo, pela camada de virtualização. Isto é possível através de um módulo do acelerador de pacotes, que roda dentro da VM. Este módulo substitui o *driver* original da placa de rede, sendo ele o responsável pela comunicação com o FlexForward. Mesmo aplicações legadas podem aproveitar das vantagens da aceleração de pacotes através da criação de uma interface de rede intermediária que se comunica com a interface do acelerador.

4.3.6 *Workflow* de funcionamento

No início o controlador precisa descobrir toda a rede, incluindo todas as ligações físicas, as aplicações sendo executadas em cada servidor e o IP de cada instância. Máquinas virtuais somente devem ser criadas e deletadas a partir da interface do controlador. Assim que a rede é identificada e as chaves calculadas, o controlador envia uma mensagem para todos os *switches*, configurando assim a chave local em cada um deles. Neste momento, todas as rotas possíveis estão calculadas.

O FlexForward possui portas virtuais e físicas, mas lógicamente são tratadas de maneira igualitária. As portas físicas ligam os servidores entre si, enquanto as portas virtuais ligam a VM ao *switch*. Portanto, as rotas na LodeNet são principalmente de VM para VM, ou de borda para borda, reforçando a ideia de núcleo apenas para transporte de dados.

A figura 4.3 ilustra o processo de um pacote para se comunicar com outro servidor. Apenas o processo acelerador se comunica com o controlador. O aplicativo pode ter ou não consciência do acelerador. Se tiver, ele chama a API do acelerador para que solicite a chave KeyFlow global ao controlador. Caso contrário, ou seja, se a aplicação for legada, ela estará sendo executada em modo de compatibilidade, onde uma interface de rede intermediária do tipo TAP é criada pelo processo acelerador. Neste caso, a aplicação gera um pacote do tipo ARP *request*, interceptada pelo acelerador, que então se comunica com o controlador. Em ambos os casos, a mensagem com o identificador do elemento de destino (por exemplo, o IP) é enviada ao controlador.

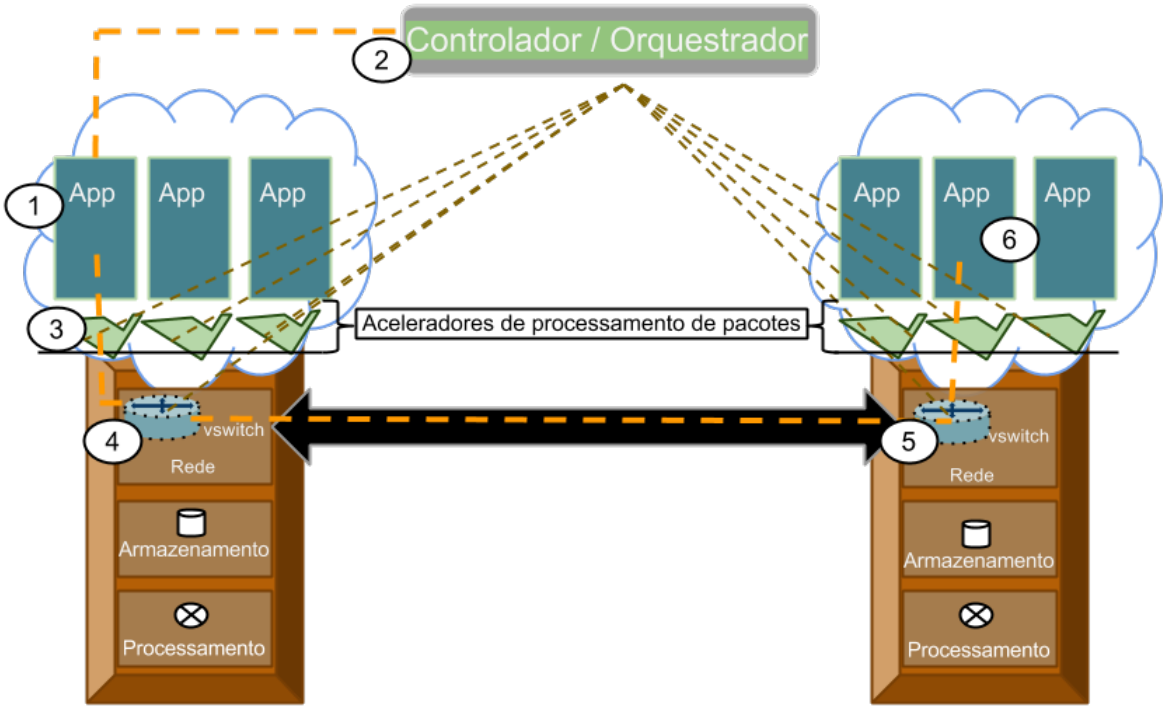

Figura 4.3: LodeNet: Representação gráfica da comunicação de pacotes

Quando o pacote chega no controlador, representado na figura pelo número 2, ele responde com uma mensagem que contém a KeyFlow do melhor caminho. É possível ainda configurar várias chaves KeyFlow que utilizem caminhos físicos diferentes. No caso do aplicativo solicitar ARP, o processo do acelerador recebe a chave e responde para a aplicação legada com um MAC aleatório, pois o pacote de camada 2 gerado por ela é substituído pelo cabeçalho KeyFlow, não importando os MACs de origem e destino para encaminhamento.

Uma vez que o controlador responde, o processo acelerador salva o identificador como chave de uma tabela hash, para acessar a chave KeyFlow global correspondente. Essa entrada na hash fica armazenada por tempo indefinido, não sendo necessária nenhuma outra comunicação com o controlador para os próximos pacotes deste destino.

O pacote é então enviado e chega no *vswitch*, representado em 4 na figura 4.3. Para este pacote, o algoritmo de encaminhamento do KeyFlow é executado, resultando na porta correta para chegar até o outro servidor. Em 5, o pacote chega ao outro servidor, mais especificamente ao FlexForward, que executa a operação MOD, resultando no número de porta virtual onde está localizada a VM destino. O pacote chega corretamente, como representado em 6, e a aplicação pode fazer o tratamento necessário.

Para redes externas ao *Datacenter*, é possível que seja solicitada ou enviada pró-ativamente

pelo controlador um ou vários *gateways*(no formato de chave global), de acordo com o prefixo de rede. Esta mensagem só é valida, porém, caso as redes internas do *Datacenter* sejam enviadas anteriormente pelo controlador. Neste caso, quando for necessário enviar um pacote, o processo acelerador checa se a rede é interna. Se positivo, ele solicita ao controlador a chave global para aquele IP em específico. Em caso negativo, o pacote é enviado ao *gateway* reservado para aquela rede. Isto ajuda a diminuir o tamanho da hash e evita comunicações desnecessárias com o controlador.

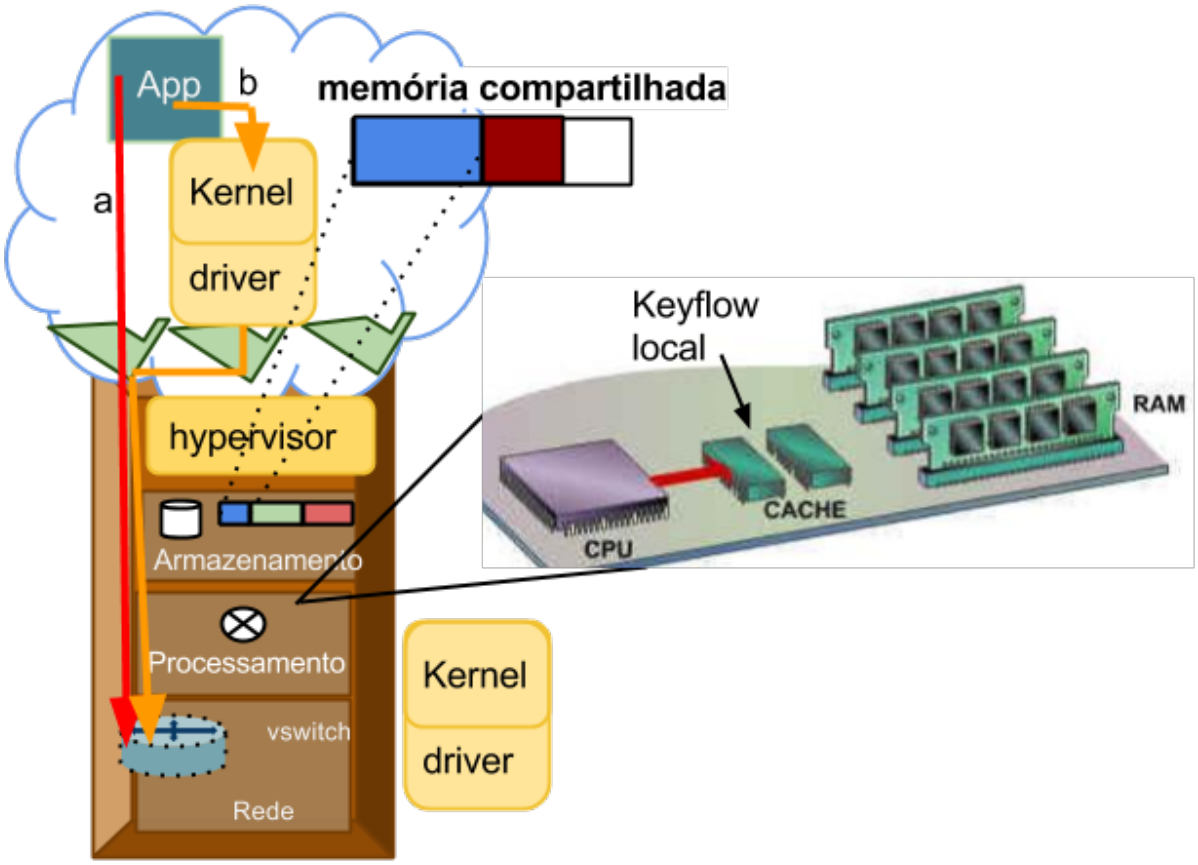

Figura 4.4: LodeNet: Mecanismos de eliminação de camadas e diminuição da latência

Quando uma VM ou aplicativo fica inoperante, o controlador atualiza os elementos de borda com uma mensagem cujo conteúdo é o identificador único (IP, por exemplo) da aplicação que ficou inoperante. Quando o acelerador recebe essa mensagem, o identificador é apagado de sua tabela hash, impossibilitando o envio posterior de pacotes para este destino. Para evitar *loops*, um tempo de vida é previsto no cabeçalho KeyFlow.

Para eliminar as camadas e prover um encaminhamento de baixa latência, algumas camadas são eliminadas através do esquema representado na figura 4.4. O caminho representado pela letra b é o tradicional do *Datacenter*, onde o pacote, para sair do servidor, precisa passar pelo

Kernel e driver da VM, pelo hipervisor, pelo Kernel e driver da máquina física, para então, finalmente, sair pela placa de rede. Na LodeNet, o caminho é representado pela letra a, onde todas essas camadas são ultrapassadas e o pacote pode ir direto para o *switch* virtual. Isto é feito através de um integração entre o hipervisor e o acelerador de pacotes. Para pular o hipervisor, o kernel e o driver da VM, existe uma memória compartilhada entre a VM e o servidor. Quando um pacote é copiado para essa região da memória, o pacote está automaticamente no servidor. O acelerador de pacotes permite então ultrapassar o Kernel e driver do servidor. Para acelerar ainda o encaminhamento KeyFlow, a chave local é armazenada em memória rápida próxima ao processador, como por exemplo o cache ou registrador.

4.3.7 Desafios e possíveis soluções

Migração de máquinas virtuais em tempo real e integração com Hipervisor

Migrar VMs em tempo real é fundamental para o ambiente de *datacenter*. Quando ocorre uma migração em tempo real, ou *live migration*, o hipervisor copia toda a máquina virtual de um servidor a outro, ou seja, toda memória e disco. Devido ao fato da máquina continuar escrevendo constantemente nesses elementos, ela precisa ser pausada. Porém, isso é feito de maneira extremamente rápida, de modo que as conexões não se percam, os processos não parem e a VM continue executando como se nada tivesse acontecido. Durante o momento de pausa, o hipervisor envia pacotes ARP (*Address Resolution Protocol*) , chamado ARP gratuito, para atualizar todos os *switches*.

Numa migração em tempo real, na LodeNet, a chave global deve mudar em todos os pacotes cujo destino for a máquina virtual migrada. Para isto, o controlador tem certa integração com o hipervisor, e, enquanto a VM é copiada, a nova chave é calculada. No momento da pausa, o controlador envia uma atualização para todos os elementos de borda, terminando assim o processo.

A reconfiguração de chaves do KeyFlow

O controlador que executa KeyFlow necessita, antes de distribuir as chaves, obter a lista completa de *switches*, o número de portas de cada um e a ligação entre eles. Uma vez obtida essa relação, é possível calcular as chaves e enviar para cada equipamento de rede. O problema, porém, acontece quando algum evento de rede, como inserção ou deleção de um novo switch, adição ou subtração de uma nova porta, que neste caso é equivalente a criação e remoção de uma máquina virtual e, por fim, a inoperância de algum enlace ou elemento. O padrão do

KeyFlow, em alguns destes casos, é realizar uma reconfiguração completa das chaves da rede, e enviar para os *switches*. Num *Datacenter* dinâmico como os de computação em nuvem, refazer esse processo repetidamente pode causar aumento da latência na comunicação, por isso, são necessárias técnicas para evitá-la.

É possível utilizar a seguinte estratégia (não analisada na pratica, necessitando de estudo futuro dos possíveis impactos): servidores "fantasmas"com chaves locais já computadas são criados previamente e ficam aguardando uma alocação. No caso de uma remoção de servidor, sua chave local é considerada fantasma e pode ser alocada para uma adição futura. É importante salientar que os servidores fantasmas possuem ligações igualmente fantasmas com servidores reais. Estas ligações devem ser atendidas no caso de uma inserção, para que não haja necessidade de novos cálculo de rota.

Seguindo o mesmo raciocínio, no caso da criação de uma nova porta virtual haveria a necessidade de reconfiguração, porém, como se tratam de *switches* virtuais, pode ser criado um número alto de portas fantasmas com antecedência, sendo muitas delas não utilizadas, para que, quando uma aplicação ocupe aquela porta, a chave já esteja computada.

O controlador, no começo da rede, descobre todos os caminhos para gerar a chave, mas normalmente escolhe apenas um principal, baseado em requisitos configuráveis, embora isto não seja obrigatório. No caso de um *link* físico ficar inoperante, o controlador precisa apenas escolher outro dos caminhos disponíveis e enviar a nova informação para as máquinas virtuais, sem necessidade de reconfiguração. Também é possível utilizar um rótulo sobressalente que definirá em tempo real o caminho alternativo para o fluxo, ou seja, disponibilizar a informação de caminho alternativo no próprio pacote, para o caso onde o *switch* detecta um evento de queda de enlace em uma de suas portas. Apenas seria necessária uma reconfiguração se acabassem os caminhos alternativos.

A escalabilidade das chaves KeyFlow

A quantidade de bits necessários para as chaves do KeyFlow cresce muito rapidamente, de acordo com certas variáveis. O problema pode ser visualizado através da figura 4.5. A quantidade de portas, juntamente com o número de nós também influenciam no tamanho da chave, porém o número de saltos é a variável que mais tem impacto neste fator.

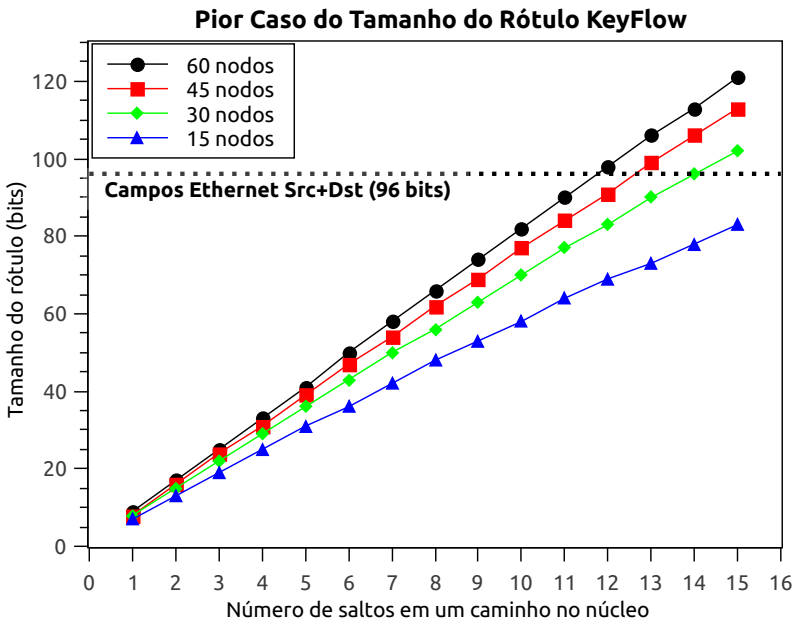

Figura 4.5: Quantidade de bits necessários para o KeyFlow de acordo com a quantidade de saltos. Fonte:(MARTINELLO et al., 2014)

A figura 4.5 omite que foi considerado que cada nó possuía 24 portas. Considerando para fins de exemplo que um *datacenter* não tenha mais que 7 saltos entre todas as VMs, que cada servidor seja um salto e que as máquinas virtuais sejam tratadas como borda, se este *datacenter* tiver 913 servidores serão necessárias no máximo 9 portas físicas. Tal resultado foi obtido com o algoritmo disponível no anexo A.4 que ligava os servidores fictícios linearmente e caso a ligação entre 2 deles estivesse num caminho com mais de 7 saltos, uma nova ligação era criada entre estes elementos.

Utilizando o mesmo algoritmo para gerar o gráfico da figura 4.5, disponível em (MARTINELLO et al., 2014), para calcular o maior número de bits necessário para o pior caso de distribuição das chaves, dadas as variáveis supracitadas, considerando 15000 portas para cada servidor (dado que cada VM conta como uma porta no vswitch), verifica-se que para este caso são necessários apenas 119 bits, sendo um valor aceitável para tamanho de cabeçalho e para armazenamento no *switch*. Se for tomado 128 bits como ideal de quantidade tanto para manipulação em código, quanto para cabeçalho em pacote, cada servidor pode ter até 45000 portas, num ambiente de 1500 servidores e 8 saltos. A escalabilidade do KeyFlow, portanto, é viável para o modelo proposto.

5 *Prova de Conceito e Análise de Resultados*

5.1 Introdução

O modelo de *Datacenter* proposto por este trabalho deve ser possível e viável de ser implantado. Além disso, a comunicação entre os elementos deve ter como característica a baixa latência e a rápida capacidade de processamento de pacotes. Por isso, como prova de conceito, foi implementado um protótipo de *switch* virtual a partir do DPDK-OVS, juntamente com o *software* para ser executado nas bordas. O controlador não foi implementado, apenas simulado através da geração de pacotes para os outros elementos. Esse protótipo foi executado em ambiente físico e virtual, a fim de comparar o modelo proposto com outros modelos em diversos ambientes possíveis.

A principal novidade deste trabalho é propor uma arquitetura de *Datacenter* que torne possível a redução da latência através da aplicação do KeyFlow, e consequente eliminação da operação de *matching* em tabela para encaminhamento, ao mesmo tempo que integra elementos que a diminuem ainda mais, como a aceleração de pacotes e o armazenamento de variáveis em memória rápida. Alguns outros fatores de complexidade, porém, devem ser eliminados para que a eliminação da consulta em tabela seja eficiente.

5.1.1 Avaliação em Ambiente completamente virtualizado

O FlexForward (VENCIONECK et al., 2014) foi inicialmente criado e testado em ambiente puramente virtual, com placas de rede emuladas, sendo um primeiro passo em busca da redução da latência no *Datacenter* definido por *software*, através da eliminação da consulta em tabela, a fim de implantar novas tecnologias como NFV.

Sendo uma extensão do Open vSwitch[1], o FlexForward atua principalmente no módulo do

[1] http://openvswitch.org

kernel, permitindo a escolha de qualquer método customizado de encaminhamento, bastando um comando do controlador para modificar entre os métodos disponíveis. Uma vez configurado o método, não há mais necessidade de comunicação entre o controlador e o plano de dados, eliminando dois gargalos da rede SDN: a comunicação constante com o controlador e a consulta em tabela.

Tal modificação, por eliminar esses gargalos, reduz a latência, como ilustrado nos dois principais testes, das figuras 5.2 e 5.3. Os experimentos foram executados utilizando 8 VMs ligadas por redes virtuais ponto a ponto. A máquina física utilizada foi um servidor IBM x3200M3, com um processador Intel Xeon X3430 de 2.40Ghz e 32GB de RAM, executando o hipervisor VMWare ESXi 5.1.0. Em cada máquina virtual estava instalado o Sistema Operacional Debian versão 7, com o kernel versão 3.2 e 4GB de RAM. Os pacotes foram enviados como mostra a figura 5.1, do ponto A ao ponto B.

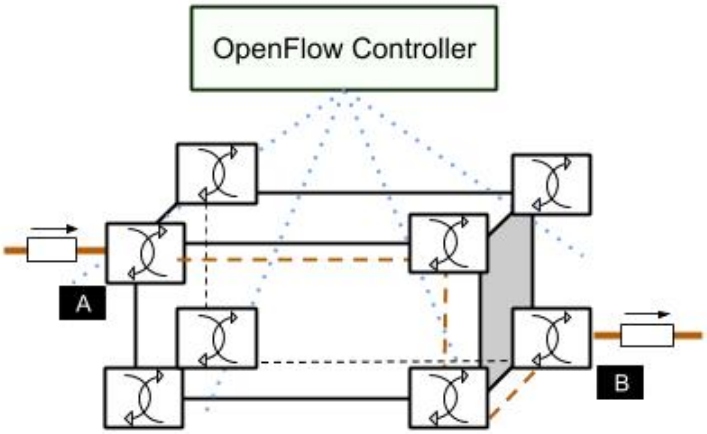

Figura 5.1: Ambiente utilizado para primeira execução do Flexforward

As máquinas virtuais foram organizadas de acordo com uma topologia de hipercubo mínima, como mostra a figura 5.1. Foram utilizados 3 métodos de encaminhamento para comparação. Em primeiro lugar, o Openflow, que é a execução padrão até que uma ordem do controlador mude para outro disponível. O KeyFlow, já apresentado no capítulo 3, é a segunda forma de encaminhamento, seguida pelo hipercubo.

O hipercubo é uma topologia de rede com um método de encaminhamento próprio. Nele, cada elemento possui seu próprio endereço e cada pacote contém o endereço de destino no hipercubo. Na organização mínima, representada na figura 5.1, cada nó tem 3 vizinhos e seu endereço deve ser diferente em apenas 1 bit, quando comparado com cada vizinho. A porta pela qual o pacote deve ser enviada é obtida através de uma operação lógica XOR entre o endereço

de destino e os endereços disponíveis em cada nó. Pelo fato dos experimentos serem executados em uma topologia de hipercubo, seu algoritmo obtém um resultado levemente superior ao do KeyFlow.

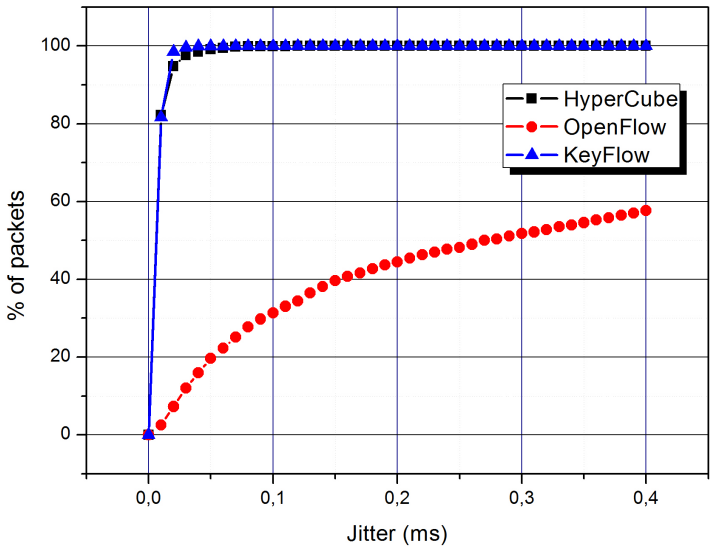

Figura 5.2: Comparação de jitter entre métodos de encaminhamento no Flexforward executados em ambiente virtual

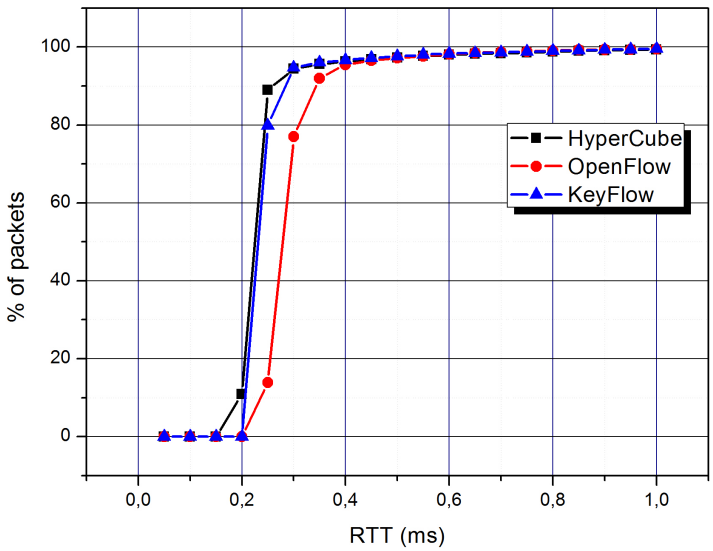

Figura 5.3: Comparação de latência entre métodos de encaminhamento no Flexforward executados em ambiente virtual

O teste de variação de latência ou jitter, parâmetro importante para serviços em tempo real como Voz sobre IP, encontrado na figura 5.2 mostrou que 99% dos valores obtidos ficaram abaixo dos 0.05 ms para os métodos sem consulta na tabela, ou seja, Hipercubo e KeyFlow. No

caso do Openflow, 50% dos pacotes tiveram jitter superior a 0.3ms.

Por volta de 90% dos pacotes testados com o método hipercubo obtiveram RTT abaixo de 0,25 ms enquanto o KeyFlow apresentou 80%. OpenFlow apresentou apenas 15% desta latência. A média de latência do Openflow foi 35% maior, quando comparado com os outros métodos. Estes resultados podem ser verificados na figura 5.3.

Foi comprovado assim que é possível diminuir a latência através do FlexForward. Uma operação na CPU provou-se mais rápida do que uma busca na tabela de fluxos, para o caso de um *Datacenter* centrado no servidor. Os testes realizados com OpenFlow possuíam poucas entradas na tabela (cerca de 5) e mesmo assim foi possível obter uma diferença, visto que em ambientes de produção o número de fluxos seria bem maior. A diferença de latência torna-se pequena pois a pilha de protocolos dos sistemas operacionais das VMs mascaram o ganho, mesmo em ambiente com apenas placas de rede virtuais. Essa diferença torna-se ainda menor em ambientes físicos, pela adição de mais um gargalo, como pode ser verificado pelos próximos experimentos.

5.1.2 Avaliação em Ambiente Físico

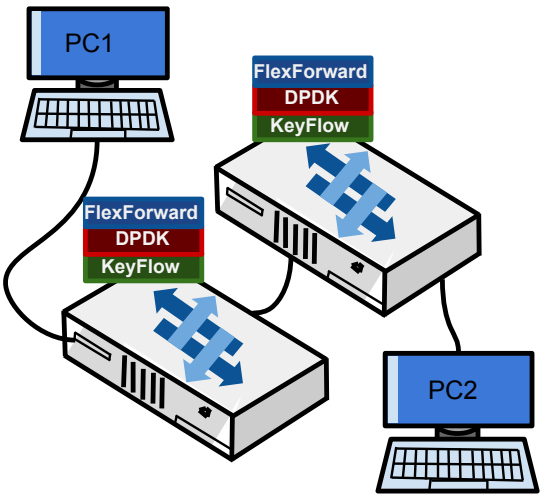

Figura 5.4: Ambiente de avaliação do DPDK-OVS e FlexForward

A tabela de fluxos revelou-se como um gargalo da rede. Entretanto, quando o mesmo experimento é executado em ambiente físico, este ganho mostrou-se quase imperceptível, como mostram as figuras 5.5 e 5.6, onde o Flexforward foi executado em duas máquinas físicas diretamente conectadas, a primeira com um processador Intel Xeon E5520 de 2.27GHz e 35G de

memória RAM. A segunda um processador Intel Xeon E5335 de 2GHz e 25G de RAM. Pôde-se então concluir que outro gargalo existia, que mascarava o ganho obtido pela eliminação da consulta em tabela.

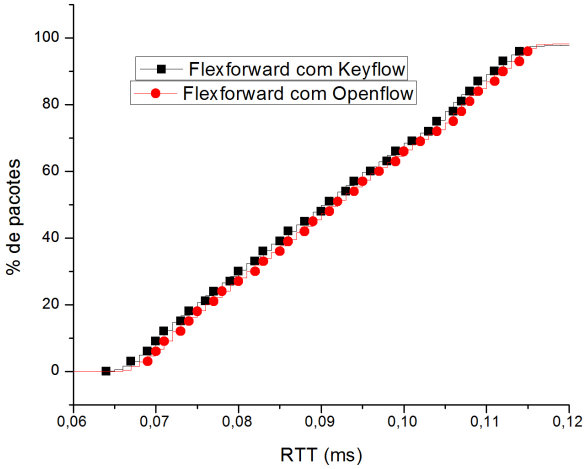

Figura 5.5: Comparação de latência entre métodos de encaminhamento no Flexforward executados em ambiente físico

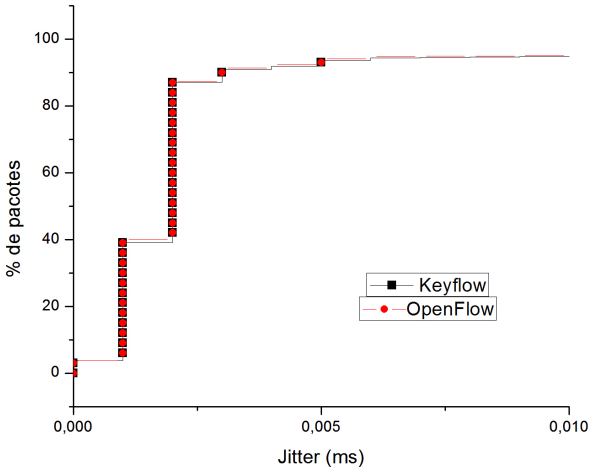

Figura 5.6: Comparação de jitter entre métodos de encaminhamento no Flexforward executados em ambiente físico

A pilha de protocolos do *Kernel* do Sistema Operacional e o driver da placa de rede estavam obscurecendo o ganho do FlexForward. Para validar esta hipótese, a implementação do *vswitch* foi melhorada a fim de eliminar essas camadas de *overhead*. Os aceleradores de processamento

de pacotes, como o DPDK, utilizam artifícios para burlar a pilha de protocolos, através de funções que acessam diretamente a placa de rede, além de rotinas que auxiliam na velocidade do encaminhamento do pacote. Foi então adicionado o DPDK ao Flexforward, utilizando como base o *switch* virtual DPDK-OVS[2]. Como o DPDK-OVS é apenas capaz de encaminhar pacotes, sem tratar pacotes destinados a ele mesmo, houve a necessidade de adicionar dois computadores simples entre os servidores, sem modificação alguma no método de encaminhamento deles, como mostra a figura 5.4. Foi adicionado um *notebook* HP Pavilion LZ015LA com CPU AMD E-350 Processor e 2GB de RAM e um computador com processador Intel Xeon E5335 de 2GHz e 25G de RAM entre os servidores encaminhadores.

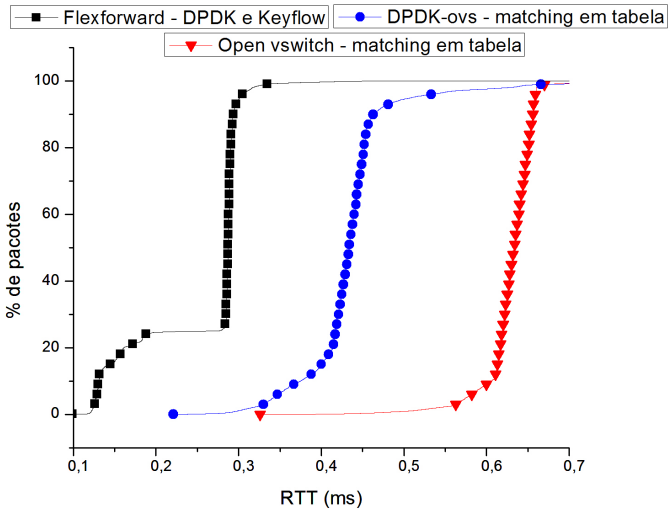

Figura 5.7: Comparação entre DPDK-OVS utilizando *matching* em tabela, Open vSwitch e FlexForward utilizando KeyFlow

Eliminando a pilha de protocolos do *kernel* através do DPDK, o gargalo volta a ser a consulta em tabela. A figura 5.7 mostra a comparação entre o *switch* DPDK-OVS, o Open vSwitch (que utilizam *matching* em tabela para encaminhamento) e o FlexForward com DPDK, utilizando KeyFlow como método de encaminhamento. Sem o matching de tabela e com acelerador de pacotes, a latência caiu pela metade, quando comparado com o ambiente sem acelerador de encaminhamento. A média de latência do DPDK-OVS foi de 0,33 ms enquanto a do FlexForward com DPDK e Keyflow foi 0,12 ms, havendo portanto uma latência média 2,6 vezes maior para o método de consulta em tabela.

O auxílio do DPDK neste caso foi a reserva de processador lógico e o acesso direto à placa de rede, que resultou num tipo de *bypass* do *driver* da placa de rede, das chamadas de sistema e

[2]https://github.com/01org/dpdk-ovs

da pilha de protocolos. Tal modificação resultou num alto ganho de performance pelo DPDK-OVS, em comparacão com o OVS. A eliminação da consulta em tabela porém, trás um ganho ainda maior, mesmo comparado ao DPDK.

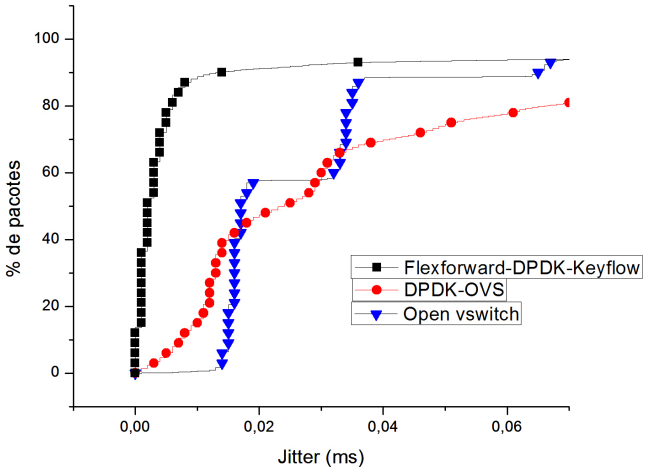

Figura 5.8: Comparação de jitter entre DPDK-OVS, Open vSwitch e FlexForward

5.1.3 Avaliação em ambiente físico com Máquinas Virtuais

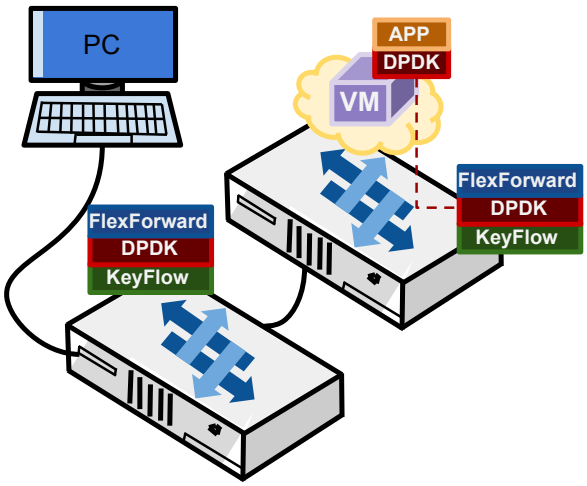

Figura 5.9: Ambiente para testes com FlexForward-DPDK

A partir dos resultados anteriores pode-se concluir que tanto em ambiente físico quanto virtual é possível reduzir drasticamente a latência através da eliminação da consulta em tabela e da pilha de protocolos do kernel. Ainda assim há a necessidade de um experimento que esteja mais próximo ao de um ambiente NFV, ou seja, com funções de rede virtualizadas que inclui

comunicação entre sistemas virtualizados e físicos. É preciso que neste ambiente também exista uma melhoria na latência, visto que a virtualização adiciona mais uma camada de *overhead*, resultando em maior lentidão no encaminhamento. Para isto, o ambiente ilustrado na figura 5.9 foi configurado, onde a VM representa um serviço virtualizado e o PC um cliente. Como o acelerador elimina a pilha de protocolos, é preciso que exista uma interface entre as aplicações legadas que a utilizem, além de permitir também que aplicações sejam criadas diretamente para utilização com o acelerador. Para este fim, foi criada uma interface de rede virtual (TAP) na VM. A figura 5.10 mostra os testes realizados com e sem esta interface de rede virtual.

As mesmas máquinas físicas dos experimentos anteriores foram utilizadas, com a adição da máquina virtual com 3GB de RAM e 2 processadores lógicos alocados para ela. O hipervisor utilizado na configuração foi o qemu[3], modificado para que possa reconhecer as portas virtuais do *vswitch* na máquina virtual. Essa modificação é disponibilizada juntamente com o DPDK-OVS.

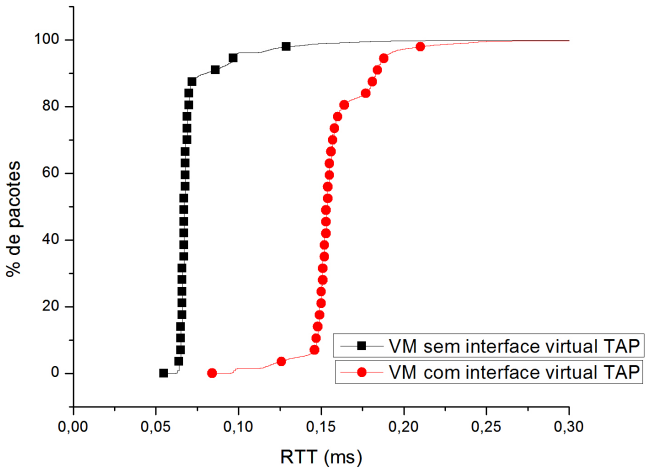

Figura 5.10: Experimento de latência realizado com FlexForward-DPDK abrangendo um ambiente de virtualização e físico

Para efeito de comparação, criou-se a mesma configuração física, porém num ambiente Linux tradicional de virtualização. Foi utilizado o qemu sem modificações e todas as interfaces controladas pela pilha de protocolos do kernel. Verifica-se pelo gráfico 5.11 que todo o *overhead* gerado pela virtualização em ambiente tradicional é eliminado pelo acelerador de encaminhamento, tendo a latência diminuída ainda mais com o FlexForward.

A comunicação entre a VM e o servidor físico, no caso do FlexForward, utiliza do recurso

[3]http://qemu.org

de cópia zero disponibilizado pelo DPDK, no qual existe uma memória compartilhada entre os elementos. Assim que o pacote está na região compartilhada, ele já está simultaneamente nas duas máquinas. Tal recurso aumenta ainda mais o ganho na velocidade de encaminhamento. Existe portanto um processo DPDK também na máquina virtual, que se comunica com o *vswitch*. Tal momória, entretanto é compartilhada apenas entre Vm e host, sendo que outras VMs alocam outras regiões de memória, havendo, portanto, um isolamento de redes virtuais.

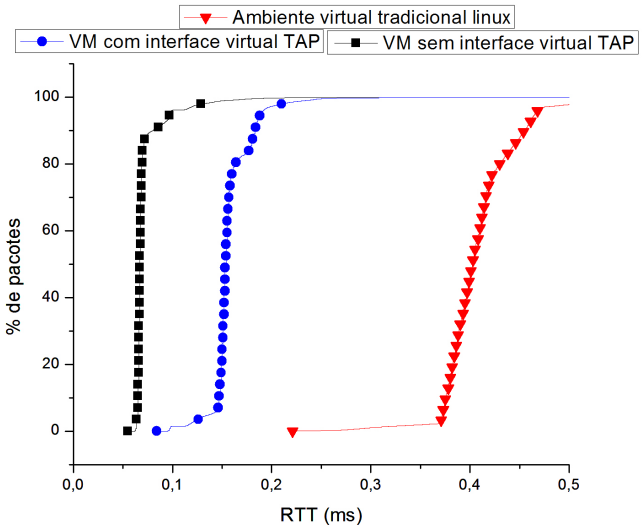

Figura 5.11: Experimento de latência realizado com FlexForward-DPDK abrangendo um ambiente de virtualização e físico, em comparação com ambiente tradicional

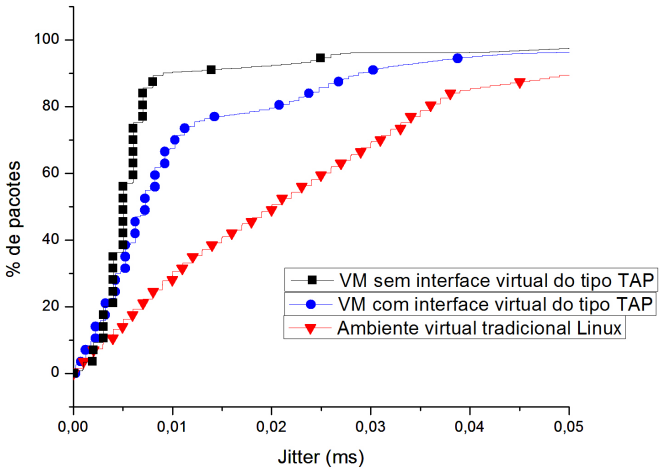

Figura 5.12: Experimento de jitter realizado com FlexForward-DPDK abrangendo um ambiente de virtualização e físico

A interface do tipo TAP aumenta a latência pois inclui o processamento de pacote do kernel do sistema operacional. Os aplicativos legados utilizam a TAP, que se comunica com o *vswitch* através do processo DPDK, que por sua vez se comunica com o *vswitch*. Mesmo assim, porém, foi possível verificar um ganho relativamente alto em comparação com o ambiente Linux tradicional.

A média com a interface virtual TAP ainda é 2,5 vezes menor que a obtida em ambiente tradicional. Na configuração ideal, isto é, sem a interface virtual, a latência média é 5,6 vezes menor quando comparada com a configuração tradicional e 2,1 vezes menor quando comparada com o experimento utilizando uma interface virtual TAP.

5.2 Conclusão

A prova de conceito do modelo de rede de *Datacenter* proposto neste trabalho contribuiu para reduzir algumas barreiras que impediam a adoção do NFV em redes SDN para aplicações com requisitos de baixa latência, através da utilização do FlexForward, juntamente com o DPDK e o KeyFlow. A latência foi reduzida drasticamente através da eliminação da consulta em tabela, das sucessivas cópias de memória realizadas pela virtualização e da pilha de protocolos implementada no kernel de cada SO. Embora as soluções já existissem separadamente, foi necessário uni-las a fim de criar uma solução completa para o *Datacenter*, que mostrou-se viável, veloz e flexível, além de permitir inovações futuras.

6 *Considerações Finais e Trabalhos Futuros*

Este trabalho identificou os gargalos da rede do *Datacenter*, a fim de diminuir a latência de comunicação, requisito para tornar viável a programabilidade das redes definidas por *software* e a flexibilidade do NFV. Foi identificado que a pilha de protocolos dos *drivers* de rede do sistema operacional, a cópia de memória para encaminhamento do pacote, a virtualização e a consulta em tabela são problemas que aumentam a latência da rede. Os primeiros problemas são resolvidos com um acelerador de pacote, enquanto o último necessita de um método de encaminhamento diferente do tradicional, o qual foi escolhido o KeyFlow. A proposta tratou de um modelo de *Datacenter* que integra todos estes elementos, obtendo suas vantagens de maneira conjunta.

Desta maneira, torna-se desnecessária a aquisição de equipamentos de rede nesta arquitetura e ainda assim obtém-se uma rede veloz, flexível, gerenciável e programável. Como prova de conceito, o *switch* FlexForward foi implementado a partir do DPDK-OVS, a primeira modificação do Open vSwitch realizada para DPDK. A implementação provou que o método de encaminhamento proposto é mais vantajoso em comparação com métodos de consulta em tabela, reduzindo a latência pela metade, ou em 6 vezes se comparado com o ambiente Linux tradicional.

O ambiente de testes disponibilizado para este trabalho foi básico, sendo importante mais servidores para testes mais expressivos. Acredita-se que se estes requisitos forem cumpridos, os resultados sejam ainda mais favoráveis. Para trabalhos futuros recomenda-se processadores com mais funcionalidades compatíveis com DPDK como por exemplo a possibilidade de *huge-pages* com 1G de memória. A placa de rede disponibilizada é capaz de encaminhar apenas 1G de tráfego, medida que o kernel do Sistema Operacional consegue manipular sem problemas. Por isso, é necessário num trabalho futuro testes com interfaces de no mínimo 10G, onde um teste de *throughput* seria facilmente visualizado numa comparação entre abordagens.

Um ambiente com escala no rack reduz drasticamente o número de *switches* virtuais, per-

mitindo maior escalabilidade do KeyFlow, melhor gerência dos recursos disponíveis e menor carga no controlador. Um teste futuro utilizando racks gerenciáveis com prateleiras de recursos é importante para evolução da proposta.

Embora o *hardware* especializado de rede com ASICs e TCAMs seja mais caro em comparação com processadores, é necessária uma comparação entre essas duas abordagens, onde deve ser verificado se a LodeNet trás a mesma ou melhor velocidade e latência, com um custo menor.

Um outro trabalho futuro seria a criação de hardware especializado em executar switches virtuais do tipo FlexForward, com memórias pequenas e processadores dedicados, de maneira que a memória onde fique armazenada a chave seja equivalente ao atual registrador encontrado nos processadores. A abordagem de um *pool* de rede, que é o caso encontrado em um *Datacenter* centrado no rack é considerado ideal devido ao fato de necessitar de um número menor de elementos, permitindo escalabilidade para as chaves do KeyFlow.

O método de encaminhamento apresentado neste trabalho não depende de uma topologia específica. Entretanto, um estudo aprofundado no tema poderia revelar uma topologia que diminuísse ainda mais a latência no encaminhamento, através da diminuição geral no número de saltos.

Por fim, é necessária uma implementação completa do controlador e suas devidas integrações e agregações, seja com o orquestrador, seja com o hipervisor, pois sua função é fundamental na rede proposta, sendo ele o responsável por computar e distribuir as chaves KeyFlow de cada elemento de rede, além de gerenciar todo o *Datacenter*.

Referências Bibliográficas

ANDERSON, T. et al. Overcoming the internet impasse through virtualization. *Computer*, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 38, n. 4, p. 34–41, abr. 2005. ISSN 0018-9162. Disponível em: <http://dx.doi.org/10.1109/MC.2005.136>.

BRISCOE, B. et al. Reducing internet latency: A survey of techniques and their merits. *Communications Surveys Tutorials, IEEE*, PP, n. 99, 2014. ISSN 1553-877X.

CASADO, M. et al. Ethane: taking control of the enterprise. In: *Proceedings of the 2007 conference on Applications, technologies, architectures, and protocols for computer communications*. New York, NY, USA: ACM, 2007. (SIGCOMM '07), p. 1–12. ISBN 978-1-59593-713-1. Disponível em: <http://doi.acm.org/10.1145/1282380.1282382>.

CHAO, H.; LIU, B. *High Performance Switches and Routers*. [S.l.]: Wiley, 2007. ISBN 9780470113943.

CHIOSI, Margaret et al. *Network Functions Virtualisation An Introduction, Benefits, Enablers, Challenges and Call for Action*. 2013. Acesso em: 6 jun. 2015. Disponível em: <https://portal.etsi.org/nfv/nfv_white_paper.pdf>.

CHOWDHURY, N. M. K.; BOUTABA, R. A survey of network virtualization. *Comput. Netw.*, Elsevier North-Holland, Inc., New York, NY, USA, v. 54, n. 5, p. 862–876, abr. 2010. ISSN 1389-1286. Disponível em: <http://dx.doi.org/10.1016/j.comnet.2009.10.017>.

CISCO SYSTEMS. *Application Centric Infrastructure - IT Insights*. 2014. Acesso em: 1 jul. 2015. Disponível em: <http://www.cisco.com/c/dam/en/us/solutions/collateral/data-center-virtualization/application-centric-infrastructure/i-dc-05212014-application-centric-infrastructure.pdf>.

CUMULUS NETWORKS. *The first, true Linux OS for data center networking*. 2015. Acesso em: 1 jul. 2015. Disponível em: <http://cumulusnetworks.com/cumulus-linux/overview/>.

DAVIE, B.; GROSS, J. *A Stateless Transport Tunneling Protocol for Network Virtualization (STT)*. [S.l.], April 2014. Disponível em: <http://www.ietf.org/internet-drafts/draft-davie-stt-06.txt>.

FARHADY, H.; LEE, H.; NAKAO, A. Software-defined networking: A survey. *Computer Networks*, v. 81, p. 79 – 95, 2015. ISSN 1389-1286. Disponível em: <http://www.sciencedirect.com/science/article/pii/S1389128615000614>.

FLAJSLIK, M.; ROSENBLUM, M. Network interface design for low latency request-response protocols. In: *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 2013. (USENIX ATC'13), p. 333–346. Disponível em: <http://dl.acm.org/citation.cfm?id=2535461.2535502>.

GARG, P.; WANG, Y.-S. *NVGRE: Network Virtualization using Generic Routing Encapsulation*. [S.l.], April 2015. Disponível em: <http://www.ietf.org/internet-drafts/draft-sridharan-virtualization-nvgre-08.txt>.

GENG, H. *Data Center Handbook*. [S.l.]: Wiley, 2014. ISBN 9781118436639.

GORANSSON, P.; BLACK, C. *Software Defined Networks: A Comprehensive Approach*. [S.l.]: Elsevier Science, 2014. ISBN 9780124166844.

GREENBERG, A. et al. A clean slate 4d approach to network control and management. *SIGCOMM Comput. Commun. Rev.*, ACM, New York, NY, USA, v. 35, n. 5, p. 41–54, out. 2005. ISSN 0146-4833. Disponível em: <http://doi.acm.org/10.1145/1096536.1096541>.

GUO, C. et al. Dcell: A scalable and fault-tolerant network structure for data centers. In: *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication*. New York, NY, USA: ACM, 2008. (SIGCOMM '08), p. 75–86. ISBN 978-1-60558-175-0. Disponível em: <http://doi.acm.org/10.1145/1402958.1402968>.

GUO, C. et al. Bcube: A high performance, server-centric network architecture for modular data centers. In: . New York, NY, USA: ACM, 2009. (SIGCOMM '09), p. 63–74. ISBN 978-1-60558-594-9. Disponível em: <http://doi.acm.org/10.1145/1592568.1592577>.

HAMILTON, J. *Networking: The last bastion of mainframe computing*. 2011. Último acesso: 06/06/2013.

HAN, S. et al. *SoftNIC: A Software NIC to Augment Hardware*. [S.l.], May 2015. Disponível em: <http://www.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-155.html>.

HERBERT, John. *IPv6 and SDN Adoption Buddies*. 2014. Acesso em: 2 jul. 2015. Disponível em: <http://movingpackets.net/2014/06/02/ipv6-sdn-adoption-buddies/>.

INTEL. *Intel, Facebook Collaborate on Future Data Center Rack Technologies*. 2013. Acesso em: 21 jul. 2015. Disponível em: <http://www.intel.com/content/www/us/en/research/intel-labs-silicon-photonics-research.html>.

INTEL. *Intel Silicon Photonics Research*. 2013. Acesso em: 21 jul. 2015. Disponível em: <http://www.intel.com/content/www/us/en/research/intel-labs-silicon-photonics-research.html>.

INTEL. *Impressive Packet Processing Performance Enables Greater Workload Consolidation*. 2015. Acesso em: 8 jun. 2015. Disponível em: <http://www.intel.com/content/dam/www/public/us/en/documents/solution-briefs/communications-packet-processing-brief.pdf>.

INTEL. *Intel Data Plane Development Kit*. 2015. Acesso em: 8 jun. 2015. Disponível em: <http://dpdk.org>.

KUSNETZKY, D. *Virtualization: A Manager's Guide*. [S.l.]: O'Reilly Media, 2011. ISBN 9781449313180.

LEBLANC, K. *Performance - Still Fueling the NFV Discussion*. 2015. Acesso em: 28 jul. 2015. Disponível em: <https://www.sdxcentral.com/articles/contributed/vnf-performance-fueling-nfv-discussion-kelly-leblanc/2015/05/>.

LERNERM Andrew. *The State of SDN Adoption*. 2014. Acesso em: 7 jun. 2015. Disponível em: <http://blogs.gartner.com/andrew-lerner/2014/06/16/the-state-of-sdn-adoption/>.

MACVITTIE, L. *Stacks and Flows: Decoupling Hype From Reality*. 2014. Exhibit at Data Center World [Acesso em: 10 jun. 2015]. Disponível em: <http://www.datacenterworld.com/fall/education/data-center-trends-track/>.

MARTINELLO, M. et al. Keyflow: a prototype for evolving sdn toward core network fabrics. *Network, IEEE*, v. 28, n. 2, p. 12–19, March 2014. ISSN 0890-8044.

MCKEOWN, N. et al. Openflow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, ACM, New York, NY, USA, v. 38, n. 2, p. 69–74, mar. 2008. ISSN 0146-4833. Disponível em: <http://doi.acm.org/10.1145/1355734.1355746>.

MELL, P.; GRANCE, T. *The NIST Definition of Cloud Computing*. 2011. Acesso em: 7 jun. 2015. Disponível em: <http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf>.

NADEAU, T.; GRAY, K. *SDN: Software Defined Networks*. [S.l.]: O'Reilly Media, 2013. ISBN 9781449342449.

ONF. *SDN Architecture Overview*. 2013. Acesso em: 7 jun. 2015. Disponível em: <https://www.opennetworking.org/images/stories/downloads/sdn-resources/technical-reports/SDN-architecture-overview-1.0.pdf>.

ONF. *OpenFlow switch specification version 1.4*. 2014. Acesso em: 20 jul. 2015. Disponível em: <https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.4.0.pdf>.

ONF - Open Network Foundation. *ONF SDN Definition*. 2015. Acesso em: 2 jul. 2015. Disponível em: <https://www.opennetworking.org/sdn-resources/sdn-definition>.

SAAD, Y.; SCHULTZ, M. H. Data communication in hypercubes. *Journal of Parallel and Distributed Computing*, v. 6, n. 1, p. 115 – 135, 1989. ISSN 0743-7315. Disponível em: <http://www.sciencedirect.com/science/article/pii/0743731589900452>.

SALISBURY, Brent. *The Control Plane, Data Plane and Forwarding Plane in Networks*. 2015. Acesso em: 2 jul. 2015. Disponível em: <http://networkstatic.net/the-control-plane-data-plane-and-forwarding-plane-in-networks/>.

SAMANI, Raj and Reavis, Jim and Honan, Brian. *CSA Guide to Cloud Computing v3*. 2011. Acesso em: 7 jun. 2015. Disponível em: <https://cloudsecurityalliance.org/guidance/csaguide.v3.0.pdf>.

TENNENHOUSE, D. L.; WETHERALL, D. J. Towards an active network architecture. *SIGCOMM Comput. Commun. Rev.*, ACM, New York, NY, USA, v. 26, n. 2, p. 5–17, abr. 1996. ISSN 0146-4833. Disponível em: <http://doi.acm.org/10.1145/231699.231701>.

TOOTOONCHIAN, A. et al. On controller performance in software-defined networks. In: *Proceedings of the 2Nd USENIX Conference on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services*. Berkeley, CA, USA: USENIX Association, 2012. (Hot-ICE'12), p. 10–10. Disponível em: <http://dl.acm.org/citation.cfm?id=2228283.2228297>.

VASSOLER, G. *TRIIIAD: Uma Arquitetura para Orquestracao autonomica de Redes de Data Center Centrado em Servidor*. Tese (Doutorado em engenharia eletrica) — Universidade Federal do Espirito Santo (UFES), 2015.

VENCIONECK, R. D. et al. Flexforward: Enabling an sdn manageable forwarding engine in open vswitch. In: IEEE. *Network and Service Management (CNSM), 2014 10th International Conference on*. [S.l.], 2014. p. 296–299.

VERAS, M. *Virtualizacao: Componente Central do Datacenter*. [S.l.]: Brasport, 2011. ISBN 9781449313180.

VMWARE. *NSX - Visao Geral*. 2015. Acesso em: 1 jul. 2015. Disponível em: <www.vmware.com/br/products/nsx>.

WANG, T. et al. Clot: A cost-effective low-latency overlaid torus-based network architecture for data centers. In: *Communications (ICC), 2015 IEEE International Conference on*. [S.l.: s.n.], 2015. p. 5479–5484.

WANGUHGU, K. VXLAN: Extending networking to fit the Cloud. v. 37, n. 5, out. 2012. ISSN 1044-6397. Disponível em: <https://www.usenix.org/publications/login/october-2012-volume-37-number-5/vxlan-extending-networking-fit-cloud>.

WIPFEL, Robert. *Software already defines your Datacenter*. 2015. Acesso em: 7 nov. 2015. Disponível em: <http://www.nextplatform.com/2015/08/07/software-already-defines-your-datacenter>.

WOOD, T. et al. Toward a software-based network: integrating software defined networking and network function virtualization. *Network, IEEE*, v. 29, n. 3, p. 36–41, May 2015. ISSN 0890-8044.

Anexos

A.1 Modificações no DPDK 1.7.1

DPDK/lib/librte_eal/linuxapp/eal/eal_thread.c

```
@@ -56,6 +56,19 @@
 #include "eal_thread.h"
 RTE_DEFINE_PER_LCORE(unsigned, _lcore_id);
+RTE_DEFINE_PER_LCORE(uint8_t, _keyflow0) = 0;
+RTE_DEFINE_PER_LCORE(uint8_t, _keyflow1) = 0;
+RTE_DEFINE_PER_LCORE(uint8_t, _keyflow2) = 0;
+RTE_DEFINE_PER_LCORE(uint8_t, _keyflow3) = 0;
+RTE_DEFINE_PER_LCORE(uint8_t, _keyflow4) = 0;
+RTE_DEFINE_PER_LCORE(uint8_t, _keyflow5) = 0;
+RTE_DEFINE_PER_LCORE(uint8_t, _keyflow6) = 0;
+RTE_DEFINE_PER_LCORE(uint8_t, _keyflow7) = 0;
+RTE_DEFINE_PER_LCORE(uint8_t, _keyflow8) = 0;
+RTE_DEFINE_PER_LCORE(uint8_t, _keyflow9) = 0;
+RTE_DEFINE_PER_LCORE(uint8_t, _keyflow10) = 0;
+RTE_DEFINE_PER_LCORE(uint8_t, _keyflow11) = 0;
+RTE_DEFINE_PER_LCORE(uint8_t, _method) = 0;
```

DPDK/lib/librte_pipeline/rte_pipeline.c

```
@@ -1204,6 +1204,34 @@
rte_pipeline_action_handler_drop(
                    struct rte_pipeline *p, uint64_t pkts_mask)}}}
+int rte_ffw_pipeline_run(struct rte_pipeline *p)
+{
+       struct rte_port_in *port_in;
+       uint64_t lookup_hit_mask = 0;
+       uint64_t pkts_mask = 0;
+        for (port_in = p->port_in_first; port_in != NULL;
+                port_in = port_in->next) {
```

```
+               uint32_t n_pkts, table_id;
+               struct rte_table *table;
+
+
+        /* Input port RX */
+        n_pkts = port_in->ops.f_rx(port_in->h_port, p->pkts,
+        port_in->burst_size);
+        if (n_pkts == 0)
+            continue;
+
+               port_in->f_action(p->pkts, n_pkts, &pkts_mask,
+                                 port_in->arg_ah);
+
+               /*get table just to fulfill requirements
+               table_id = port_in->table_id;
+               table = &p->tables[table_id];
+               table->f_action_hit(p->pkts, &lookup_hit_mask, p->entries,
+                                                   &n_pkts);
+        }
+       return 0;
+}
```

```
 int rte_pipeline_run(struct rte_pipeline *p)

@@ -1350,6 +1378,15 @@ rte_pipeline_flush(struct rte_pipeline *p)
int rte_pipeline_ffw_port_out_packet_insert(struct rte_pipeline *p,
+                uint32_t port_id, struct rte_mbuf *pkt)
+{
+        struct rte_port_out *port_out = &p->ports_out[port_id];
+       port_out->ops.f_tx(port_out->h_port, pkt);
+       return 0;
+}
+
int rte_pipeline_port_out_packet_insert(struct rte_pipeline *p,
               uint32_t port_id, struct rte_mbuf *pkt){
```

DPDK/lib/librte_pipeline/rte_pipeline.h

```
@@ -150,6 +150,7 @@ int rte_pipeline_check(struct rte_pipeline *p);
```

```
  *   0 on success, error code otherwise
 int rte_pipeline_run(struct rte_pipeline *p);
+int rte_ffw_pipeline_run(struct rte_pipeline *p);

@@ -657,6 +658,10 @@ int rte_pipeline_port_out_packet_insert(
                                        struct rte_pipeline *p,

        uint32_t port_id,
        struct rte_mbuf *pkt);
+int rte_pipeline_ffw_port_out_packet_insert(struct rte_pipeline *p,
+       uint32_t port_id,
+       struct rte_mbuf *pkt);
+
 #ifdef __cplusplus
```

A.2 Modificações no DPDK-OVS

openvswitch/datapath/dpdk/ovdk_flow.c

```
@@ -55,6 +55,29 @@
 static void flow_key_extract(uint32_t in_port, struct rte_mbuf *pkt);
 int ffw_prefetch(struct rte_mbuf **pkts, uint32_t num_pkts,
+                 uint64_t *pkts_mask __attribute__((unused)) ,void *arg) {
+
+       for (i = 0; i < PREFETCH_OFFSET && i < nb_rx; i++) {
+         rte_prefetch0(RTE_MBUF_METADATA_UINT32_PTR(pkts[i], 0));
+         rte_prefetch0(rte_pktmbuf_mtod(pkts[i], void *)); }
+
+       for (i = 0; i < nb_rx - PREFETCH_OFFSET; i++) {
+         rte_prefetch0(RTE_MBUF_METADATA_UINT32_PTR(pkts[i +
+                                PREFETCH_OFFSET], 0));
+         rte_prefetch0(rte_pktmbuf_mtod(pkts[i +
+                                     PREFETCH_OFFSET], void *));}
+       ovdk_vport_get_vportid(*(uint32_t *)arg, &vport_id);
+       ovdk_stats_vport_rx_increment(vport_id, num_pkts);
+       return 0; }
+
int flow_keys_extract(struct rte_mbuf **pkts, uint32_t num_pkts,
```

openvswitch/datapath/dpdk/ovdk_pipeline.c

```
@@ -86,6 +91,20 @@
 #define OVDK_PIPELINE_KEY_OFFSET        32
 #define OVDK_PIPELINE_PORT_OFFSET       64
+RTE_DECLARE_PER_LCORE(uint8_t, _keyflow0);
+RTE_DECLARE_PER_LCORE(uint8_t, _keyflow1);
+RTE_DECLARE_PER_LCORE(uint8_t, _keyflow2);
+RTE_DECLARE_PER_LCORE(uint8_t, _keyflow3);
+RTE_DECLARE_PER_LCORE(uint8_t, _keyflow4);
+RTE_DECLARE_PER_LCORE(uint8_t, _keyflow5);
+RTE_DECLARE_PER_LCORE(uint8_t, _keyflow6);
+RTE_DECLARE_PER_LCORE(uint8_t, _keyflow7);
+RTE_DECLARE_PER_LCORE(uint8_t, _keyflow8);
+RTE_DECLARE_PER_LCORE(uint8_t, _keyflow9);
+RTE_DECLARE_PER_LCORE(uint8_t, _keyflow10);
```

```
+RTE_DECLARE_PER_LCORE(uint8_t, _keyflow11);

 static struct rte_ring *create_ring(const char *name);

@@ -117,19 +136,38 @@
 /* Handle actions */
 static int ovdk_pipeline_exception_actions_execute(...)
-static int ovdk_pipeline_actions_execute(struct rte_mbuf **pkts,
+//static int ovdk_pipeline_actions_execute(struct rte_mbuf **pkts,
+  //      uint64_t *pkts_mask, struct rte_pipeline_table_entry **entries,
+    //    void *arg);
+ static int ffw_execute(struct rte_mbuf **pkts,
+         uint64_t *pkts_mask,
+         struct rte_pipeline_table_entry **entries,
+         void *arg);
+void ffw_keyflow_execute(struct rte_mbuf *m);

@@ -256,7 +292,7 @@ ovdk_pipeline_init(void)
                /* each time an entry is hit f_action_hit() is called */
-               .f_action_hit = ovdk_pipeline_actions_execute,
+               .f_action_hit = ffw_execute,

@@ -425,6 +463,125 @@ ovdk_pipeline_init(void)
        return;
 }

+void ffw_keyflow_execute(struct rte_mbuf *m)
+{
+       int ret;
+       struct ether_hdr *eth_hdr;
+       uint8_t keyflow_global[12], i, keyflow_local[12];
+       __uint128_t glob=0, loc=0, port=0;
+       uint32_t port_id=0, port_dpdk=0;
+       unsigned lcore_id = 0;
+
+       keyflow_local[0] = RTE_PER_LCORE(_keyflow0);
+       keyflow_local[1] = RTE_PER_LCORE(_keyflow1);
+       keyflow_local[2] = RTE_PER_LCORE(_keyflow2);
```

```
+       keyflow_local[3] = RTE_PER_LCORE(_keyflow3);
+       keyflow_local[4] = RTE_PER_LCORE(_keyflow4);
+       keyflow_local[5] = RTE_PER_LCORE(_keyflow5);
+       keyflow_local[6] = RTE_PER_LCORE(_keyflow6);
+       keyflow_local[7] = RTE_PER_LCORE(_keyflow7);
+       keyflow_local[8] = RTE_PER_LCORE(_keyflow8);
+       keyflow_local[9] = RTE_PER_LCORE(_keyflow9);
+       keyflow_local[10] = RTE_PER_LCORE(_keyflow10);
+       keyflow_local[11] = RTE_PER_LCORE(_keyflow11);
+
+       eth_hdr = rte_pktmbuf_mtod(m, struct ether_hdr *);
+       for(i=0;i<6;i++){
+               keyflow_global[i] = eth_hdr->s_addr.addr_bytes[i];
+               keyflow_global[i+6] = eth_hdr->d_addr.addr_bytes[i];
+       }
+       loc = ((__uint128_t) keyflow_local[0] << 88) |
+         ((__uint128_t) keyflow_local[1] << 80) |
+                 ((__uint128_t) keyflow_local[2] << 72) |
+         ((__uint128_t) keyflow_local[3] << 64) |
+         ((__uint128_t) keyflow_local[4] << 56) |
+         ((__uint128_t) keyflow_local[5] << 48) |
+         ((__uint128_t) keyflow_local[6] << 40) |
+         ((__uint128_t) keyflow_local[7] << 32) |
+                 ((__uint128_t) keyflow_local[8] << 24) |
+         ((__uint128_t) keyflow_local[9] << 16) |
+         ((__uint128_t) keyflow_local[10] << 8) |
+         ((__uint128_t) keyflow_local[11]);
+
+       glob = ((__uint128_t) keyflow_global[0] << 88) |
+          ((__uint128_t) keyflow_global[1] << 80) |
+                  ((__uint128_t) keyflow_global[2] << 72) |
+          ((__uint128_t) keyflow_global[3] << 64) |
+              ((__uint128_t) keyflow_global[4] << 56) |
+          ((__uint128_t) keyflow_global[5] << 48) |
+                  ((__uint128_t) keyflow_global[6] << 40) |
+          ((__uint128_t) keyflow_global[7] << 32) |
+                  ((__uint128_t) keyflow_global[8] << 24) |
+          ((__uint128_t) keyflow_global[9] << 16) |
```

```
+                  ((__uint128_t) keyflow_global[10] << 8) |
+          ((__uint128_t) keyflow_global[11]);
+
+       port = glob % loc;
+       port_id = (uint32_t) port;
+       lcore_id = rte_lcore_id();
+       if ((ret = ovdk_vport_get_out_portid(port_id,
+
&port_dpdk))) {
+               return;
+       }
+       rte_pipeline_ffw_port_out_packet_insert(
+                           ovdk_pipeline[lcore_id].pf_pipeline,
+                           port_dpdk, m);
+       ovdk_stats_vport_tx_increment(port_id, 1);
+}
+
+static int ffw_execute(struct rte_mbuf **m,
+   uint64_t *pkts_mask __attribute__((unused)),
+       struct rte_pipeline_table_entry **entries __attribute__((unused)),
+   void *arg)
+{
+RTE_LOG(INFO, APP, "FFW execute\n");
+       int i;
+       struct rte_mbuf *mbuffer;
+       unsigned lcore_id = 0;
+       uint32_t n_pkts;
+       n_pkts = *(uint32_t*) arg;
+       for(i=0;i<n_pkts;i++){
+               struct ether_hdr *eth_hdr;
+               uint16_t ethertype, op;
+           mbuffer = m[i];
+               eth_hdr = rte_pktmbuf_mtod(mbuffer, struct ether_hdr *);
+               ethertype = ((eth_hdr->ether_type & 0x00FF) << 8) |
+                   ((eth_hdr->ether_type & 0xFF00) >> 8);
+               if (ethertype == 0x0806 ){
+                 struct arp_packet *arp;
+                 arp = (struct arp_packet *)(rte_pktmbuf_mtod(mbuffer,
```

```
+                  unsigned char *) + sizeof(struct ether_hdr));
+                  op = ((arp->op & 0x00FF) << 8) | ((arp->op & 0xFF00) >> 8);
+                  if (likely(op == 1)){
+                                 lcore_id = rte_lcore_id();
+               rte_pipeline_ffw_port_out_packet_insert(
+                     ovdk_pipeline[lcore_id].pf_pipeline,128,
+                     mbuffer);
+                                 }
+                                 else{
+                                         ffw_keyflow_execute(mbuffer);
+                                 }
+                        }
+                        else{
+                                 ffw_keyflow_execute(mbuffer);
+                        }
+               }
+               return 0;
+
+}

@@ -442,8 +596,10 @@ ovdk_pipeline_run(void)

-       rte_pipeline_run(pf_pipeline);
+       rte_ffw_pipeline_run(pf_pipeline);
```

openvswitch/datapath/dpdk/ovdk_vport.c

```
@@ -169,7 +167,7 @@ ovdk_vport_init(void)
-               vport_info[i].port_in_params.f_action = flow_keys_extract;
+               vport_info[i].port_in_params.f_action = ffw_prefetch;
```

openvswitch/include/openflow/nicira-ext.h

```
@@ -316,6 +314,7 @@ enum nx_action_subtype {
     NXAST_SET_MPLS_TC,          /* struct nx_action_ttl */
+       NXAST_SET_KEYFLOW_KEY,      /* [extension] struct
+                                nx_action_set_keyflow_key */
@@ -442,6 +441,19 @@ struct nx_action_pop_queue {
+/* [Extension] Action structure for NXAST_SET_KEYFLOW_KEY.
+ *
```

```
+ * Set the key when the method is keyflow */
+struct nx_action_set_keyflow_key {
+    ovs_be16 type;                  /* OFPAT_VENDOR. */
+    ovs_be16 len;                   /* Length is 22. */
+    ovs_be32 vendor;                /* NX_VENDOR_ID. */
+    ovs_be16 subtype;               /* NXAST_SET_KEYFLOW_KEY. */
+    uint8_t key[12];                /* The Key to be setted */
+       uint8_t pad[2];
+};
+OFP_ASSERT(sizeof(struct nx_action_set_keyflow_key) == 24);
```

openvswitch/lib/ofp-actions.c

```
@@ -232,6 +231,20 @@ dec_ttl_from_openflow(struct ofpbuf *out, enum
+static void
+keyflow_from_openflow(const struct nx_action_set_keyflow_key *naskk)
+{      uint8_t i, localkf[12];
+       for(i=0;i<12;i++){
+               localkf[i] = (uint8_t) ntohs(naskk->key[i]);
+               VLOG_INFO("%x:", localkf[i]);}
+       ffw_link_set_local_keyflow(localkf)}

@@ -377,6 +389,7 @@ ofpact_from_nxast(const union ofp_action *a, enum
     enum ofperr error = 0;
+       const struct nx_action_set_keyflow_key *naskk;
     switch (code) {

@@ -505,7 +519,12 @@ ofpact_from_nxast(const union ofp_action *a, enum
         error = sample_from_openflow(&a->sample, out);
         break;
+
+       case OFPUTIL_NXAST_SET_KEYFLOW_KEY:
+        naskk = (const struct nx_action_set_keyflow_key *) a;
+        keyflow_from_openflow(naskk);
+        break; }
```

openvswitch/lib/ofp-parse.c

```
@@ -874,6 +874,10 @@ parse_named_action(enum ofputil_action_code code,
+       case OFPUTIL_NXAST_SET_KEYFLOW_KEY:
```

```
+         OVS_NOT_REACHED();
+         break;
```

openvswitch/lib/ofp-util.def

```
@@ -77,6 +77,7 @@ NXAST_ACTION(NXAST_DEC_MPLS_TTL,    nx_action_header,
0, "dec_mpls_ttl")
 NXAST_ACTION(NXAST_PUSH_MPLS,       nx_action_push_mpls,    0, "push_mpls")
 NXAST_ACTION(NXAST_POP_MPLS,        nx_action_pop_mpls,     0, "pop_mpls")
 NXAST_ACTION(NXAST_SAMPLE,          nx_action_sample,       0, "sample")
+NXAST_ACTION(NXAST_SET_KEYFLOW_KEY,      nx_action_set_keyflow_key,
0, NULL)
```

A.3 Pseudo-código a ser executado na máquina virtual

```
static int main_loop(){
 struct rte_mbuf *pkts_burst[PKT_READ_SIZE];
 struct rte_mbuf *m;
 if ((1ULL << lcore_id) & input_cores_mask) {
  /* Loop forever reading from NIC and writing to tap */
  for (;;) {
   rx_count = rte_ring_count(rx_ring);
   free_count = rte_ring_free_count(free_q);
   pkts_count = RTE_MIN(rx_count, free_count);
   pkts_count = RTE_MIN(pkts_count, PKT_READ_SIZE);
   if (unlikely(pkts_count == 0))
    continue;
   ret = rte_ring_dequeue_bulk(rx_ring, (void**) pkts_burst,
                                                pkts_count);
   if (unlikely(ret < 0))
    continue;
   for (i=0; likely(i < pkts_count); i++){
    m = pkts_burst[i];
    eth_hdr = rte_pktmbuf_mtod(m, struct ether_hdr *);
    ethertype = ((eth_hdr->ether_type & 0x00FF) << 8) |
                        ((eth_hdr->ether_type & 0xFF00) >> 8);
    if (ethertype == 0x0806 ){
     arp = (struct arp_packet *)
           (rte_pktmbuf_mtod(m, unsigned char *) +
                          sizeof(struct ether_hdr));
     op = ((arp->op & 0x00FF) << 8) | ((arp->op & 0xFF00) >> 8);
    if (likely(op == 2)){
     ip_or = ((uint32_t)arp->sndr_ip_addr[3] << 24) |
             ((uint32_t)arp->sndr_ip_addr[2] << 16) |
             ((uint32_t)arp->sndr_ip_addr[1] << 8)  |
             ((uint32_t)arp->sndr_ip_addr[0]);
     for(i=0;i<6;i++){
      setkeyflow[i] = arp->sndr_hw_addr[i];
      setkeyflow[i+6] = arp->targ_hw_addr[i];
```

```
  arp->targ_hw_addr[i] = mac[i];
  eth_hdr->d_addr.addr_bytes[i] = mac[i];}
uint32_t key = ip_or;
ret = rte_hash_add_key(ffw_lookup_struct[master_socket],
                                              (void *) &key);
if (ret < 0) {
 rte_exit(EXIT_FAILURE, "Unable to add entry\n");}
ffw_keyflow_key[ret] = malloc(sizeof(struct ffw_hash_keyflow));
for(i=0;i<12;i++){
 ffw_keyflow_key[ret]->keyflow[i] = setkeyflow[i];}
ret = rte_hash_lookup(buf_hash[master_socket],
                        (const void *)&ip_or);
if(ret > 0){
 for (i=0;i<buff[ret]->buflen;i++){
  eth_hdr2 = rte_pktmbuf_mtod(buff[ret]->buf[i],
                            struct ether_hdr *);
  for(j=0;j<6;j++){
   eth_hdr2->s_addr.addr_bytes[j] = setkeyflow[j];
   eth_hdr2->d_addr.addr_bytes[j] = setkeyflow[j+6];}
   rte_ring_enqueue(tx_ring, (void*)buff[ret]->buf[i]);}
  buff[ret]->buflen = 0;
  ret = rte_hash_del_key(buf_hash[master_socket],
                          (const void *)&ip_or);
  free(buff[ret]);}}
else if(arp->op == 9){ //9 = IP down
 ip_or = ((uint32_t)arp->sndr_ip_addr[3] << 24) |
         ((uint32_t)arp->sndr_ip_addr[2] << 16) |
         ((uint32_t)arp->sndr_ip_addr[1] << 8)  |
         ((uint32_t)arp->sndr_ip_addr[0]);
 ret = rte_hash_del_key(buf_hash[master_socket],
                         (const void *)&ip_or);
 if (ret > 0){
  rte_ring_enqueue_bulk(free_q, (void*)buff[ret]->buf,
                                    buff[ret]->buflen);
  free(buff[ret]);}
```

```
    ret = rte_hash_del_key(ffw_lookup_struct[master_socket],
                                        (const void *)&ip_or);
    free(ffw_keyflow_key[ret]);}}
  for(i=0;i<6;i++){
   eth_hdr->d_addr.addr_bytes[i] = mac[i];}
  ret = write(tap_fd, rte_pktmbuf_mtod(m, void*),
                          rte_pktmbuf_data_len(m));
  ret = rte_ring_enqueue_bulk(free_q, (void**) pkts_burst, 1);}}}
else if ((1ULL << lcore_id) & output_cores_mask) {
 /* Loop forever reading from tap and writing to NIC */
 for (;;) {
  ret = rte_ring_dequeue_bulk(alloc_q, af_pkts, 1);
  if (unlikely(ret < 0))
   continue;
  m = af_pkts[0];
  ret = read(tap_fd, m->pkt.data, 1500);
  if (unlikely(ret < 0)) {
   FATAL_ERROR("Reading from %s interface failed", tap_name);}
  if (unlikely(ret == 0)){
   ret = rte_ring_enqueue_bulk(free_q, af_pkts, 1);
   continue;}
  m->pkt.nb_segs = 1;
  m->pkt.next = NULL;
  m->pkt.pkt_len = (uint16_t)ret;
  m->pkt.data_len = (uint16_t)ret;
  eth_hdr = rte_pktmbuf_mtod(m, struct ether_hdr *);
  ethertype = ((eth_hdr->ether_type & 0x00FF) << 8) |
              ((eth_hdr->ether_type & 0xFF00) >> 8);
  if (ethertype == 0x0806 ){
   arp = (struct arp_packet *)(rte_pktmbuf_mtod(m, unsigned char *)
                                      + sizeof(struct ether_hdr));
   uint32_t key;
   key = ((uint32_t) arp->targ_ip_addr[3] << 24) |
         ((uint32_t) arp->targ_ip_addr[2] << 16) |
         ((uint32_t)arp->targ_ip_addr[1] << 8)   |
```

```
        ((uint32_t)arp->targ_ip_addr[0]);
 if ((rte_hash_lookup(ffw_lookup_struct[master_socket],
                          (const void *) &key)) < 0 ){
  ret = rte_ring_enqueue(tx_ring, (void*)m);}
 else{
  uint32_t iptoarp;
  iptoarp = ((uint32_t) arp->targ_ip_addr[3] << 24) |
            ((uint32_t) arp->targ_ip_addr[2] << 16) |
            ((uint32_t)arp->targ_ip_addr[1] << 8)   |
            ((uint32_t)arp->targ_ip_addr[0]);
  send_arp_reply_to_tap(iptoarp);
else{
 ipv4_hdr = (struct ipv4_hdr *)
                (rte_pktmbuf_mtod(m, unsigned char *)
                + sizeof(struct ether_hdr));
 uint32_t key = ipv4_hdr->dst_addr;
 ret = rte_hash_lookup(ffw_lookup_struct[master_socket],
                                  (const void *) &key);
 if (ret >= 0){
  i=0;
  while(i<6){
   eth_hdr->d_addr.addr_bytes[i] =
                    ffw_keyflow_key[ret]->keyflow[i];
   eth_hdr->s_addr.addr_bytes[i] =
                    ffw_keyflow_key[ret]->keyflow[i+6];
   i++;}
 ret = rte_ring_enqueue(tx_ring, (void*)m);}
else{
 ret = rte_hash_lookup(buf_hash[master_socket],
                           (const void *) &key);
 if (ret < 0){
  ret = rte_hash_add_key(buf_hash[master_socket], (void *) &key);
  buff[ret] = malloc(sizeof(struct buftable));
  buff[ret]->buflen = 0;
  buff[ret]->buf[0] = malloc(sizeof(struct rte_mbuf));
```

```
        if(buff[ret]->buflen == 0){
         send_arp_request_to_controller(key);}
        buff[ret]->buf[buff[ret]->buflen] = m;
        buff[ret]->buflen++;
        buff[ret]->buf[buff[ret]->buflen] =
                    malloc(sizeof(struct rte_mbuf));}}}}
    else {
     return 0;}}

int main(int argc, char *argv[]){
 socketid = rte_socket_id();
 create_hash(socketid);
 create_bufhash(socketid);
 rte_eal_mp_remote_launch(main_loop, NULL, CALL_MASTER);
 RTE_LCORE_FOREACH_SLAVE(i) {
  if (rte_eal_wait_lcore(i) < 0)
   return -1;}
 return 0;}
```

A.4 Código de geração de topologia simples

```
from collections import defaultdict
dist = {}
def dijkstra(graph,src,dest,visited=[],distances={},predecessors={}):
[...]

if __name__ == '__main__':
    grafo = {0: {}}
    visitado = defaultdict(list)
    for i in range(0, 1500):
        w1 = {i + 1: {}}
        grafo.update(w1)
        w1 = {i + 1: 1}
        grafo[i].update(w1)
        w1 = {i: 1}
        grafo[i + 1].update(w1)
        for j in range(0, i + 2):
            for x in range(0, i + 2):
                if j != x and int(j) < int(x)
                                        and (x not in visitado[j]
                        or j not in visitado[x]):
                    dijkstra(grafo, j, x, visited=[],
                                distances={}, predecessors={})
                    dj = dist[x]
                    if dj > 8:
                        w1 = {x: 1}
                        grafo[j].update(w1)
                        w1 = {j: 1}
                        grafo[x].update(w1)
                    visitado[j].append(x)
                    visitado[x].append(j)
    print grafo
```