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

GILMAR LUIZ VASSOLER

TRIIIAD: UMA ARQUITETURA PARA ORQUESTRAÇÃO AUTONÔMICA DE REDES DE DATA CENTER CENTRADO EM SERVIDOR

VITÓRIA
2015

GILMAR LUIZ VASSOLER

TRIIIAD: UMA ARQUITETURA PARA ORQUESTRAÇÃO AUTONÔMICA DE REDES DE DATA CENTER CENTRADO EM SERVIDOR

Tese apresentada ao Programa de Pós-Graduação em Engenharia Elétrica do Centro Tecnológico da Universidade Federal do Espírito Santo, como requisito parcial para obtenção do Grau de Doutor em Engenharia Elétrica.
Orientador: Prof. Dr. Moisés Renato Nunes Ribeiro.

VITÓRIA
2015

Dados Internacionais de Catalogação-na-publicação (CIP)
(Biblioteca Setorial Tecnológica,
Universidade Federal do Espírito Santo, ES, Brasil)

Vassoler, Gilmar Luiz, 1975-

V339t TRIIIAD : uma arquitetura para orquestração autonômica de redes de data center centrado em servidor / Gilmar Luiz Vassoler. – 2015.

166 f. : il.

Orientador: Moisés Renato Nunes Ribeiro.

Tese (Doutorado em Engenharia Elétrica) – Universidade Federal do Espírito Santo, Centro Tecnológico.

1. Centros de processamento de dados. 2. Redes de informação. 3. Sistemas inteligentes de veículos rodoviários. 4. Hipercubo. 5. VM/CMS (Sistema operacional de computador). computador). 6. Computação em nuvem. 7. Redes definidas por software. 8. Comutação óptica. I. Ribeiro, Moisés Renato Nunes. II. Universidade Federal do Espírito Santo. Centro Tecnológico. III. Título.

CDU: 621.3

GILMAR LUIZ VASSOLER

TRIIAD: UMA ARQUITETURA PARA ORQUESTRAÇÃO AUTONÔMICA DE REDES DE DATA CENTER CENTRADO EM SERVIDOR

Tese submetida ao programa de Pós-Graduação em Engenharia Elétrica do Centro Tecnológico da Universidade Federal do Espírito Santo, como requisito parcial para a obtenção do Grau de Doutor em Engenharia Elétrica.

Aprova em 22 de maio de 2015

COMISSÃO EXAMINADORA

Prof. Dr. Moisés Renato Nunes Ribeiro
Universidade Federal do Espírito Santo – UFES
Orientador

Prof. Dr. Magnos Martinello
Universidade Federal do Espírito Santo – UFES

Prof. Dr. Antônio Marinho Pilla Barcellos
Universidade Federal do Rio Grande do Sul – UFRGS

Prof. Dr. Srinivasa Aditya Akella
University of Wisconsin-Madison – USA

Prof. Dr. Róbert Szabó
Ericsson, Budapest University of Technology and Economics

Dedico este trabalho a minha amada esposa Fernanda Scarpatti Pimentel, que me apoiou em todos os momentos da minha trajetória.

Agradecimentos

Como é difícil mensurar o quão importante foi a participação de cada pessoa nessa trajetória, pois cada interação foi única e trouxe grandes contribuições, diretas e indiretas, para este trabalho.

Olhando pelo lado técnico, preciso agradecer, imensamente, ao meu orientador Moisés. Não com um obrigado formal, mais sim com um obrigado sincero e honesto. Muito obrigado de verdade! Muito obrigado por ter me aceitado como seu orientando. Muito obrigado pelas horas e horas dedicadas a revisão de textos e aconselhamentos. Muito obrigado pela compreensão dos meus erros e desesperos. Muito obrigado pela paciência em esperar a finalização das tarefas no meu tempo. E muitíssimo obrigado por ter emprestado seu brilhantismo para que eu pudesse desenvolver algo, que acredito ser uma contribuição interessante para a área de data center e, principalmente, para os laboratórios LabTel e LabNerds.

Também pelo lado técnico, tenho que dizer Márcia Paiva, muitíssimo obrigado! Obrigado pela sua ajuda em tantas tarefas, em especial pela ajuda durante o desenvolvimento do artigo que aprovamos em revista. Você foi o máximo!

Ainda na linha de contribuições diretas, tenho que agradecer a você:

- Alextian, muito obrigado pela sua bravura em montar os equipamentos do LabNerds, pelas ideias que me emprestou e por toda ajuda oferecida;
- Cristina, muito obrigado por ajudar com tradução da minha tese;
- Diego Mafioletti, muito obrigado também, pela sua colaboração na estrutura do LabNerds, pelo apoio nos códigos do *OvS* e pela ajuda em tantas dúvidas pontuais;
- Dione, muito obrigado pela sua ajuda, encorajamento e incentivo durante o tempo que eu estava desbravando o OpenStack;
- Magnos, muito obrigado pelo incentivo, pelas ideias e pela disponibilização da infraestrutura do LabNerds; e
- Rafael Vencioneck, muito obrigado por sua extrema coragem em desbravar o código do *Open vSwitch*;

Por outro lado, as amizades de diversas pessoas favoreceram o desenvolvimento deste trabalho, ao tornar os dias muito mais alegres.

- A turma do LabTel: Breno, Camilo, Carlos, Diogo, Esequiel, Flávio Rabelo, Márcia Paiva, Reginaldo, Ricardinho, Pedro e Sabrina. Obrigado pela amizade de vocês, pelos ótimos momentos na cantina e tantas risadas que demos e vamos dar juntos.
- A turma do LabNerds: Alextian, Ana Carolina, Camilo, Diego(s), Dione, Eros, Magnos, Rafael, Rodolfo, foi muito bom trabalhar com vocês! E tenho certeza que será muito bom continuar trabalhando.
- Os professores do LabTel: Anselmo Frizera, Jair Silva, Marcelo Segatto e Maria José. Obrigado pela amizade e ensinamentos que obtive de vocês.

Agradeço também a meus pais, pois eles foram os grandes incentivadores e responsáveis por essa minha carreira de estudos. Mãe e Pai muito obrigado!

Agradeço aos meus irmãos e demais membros da minha família: Bruno e Carla; Fernanda, Bruno e Sofia; Geraldo e Márcia; Penha, Gustavo e Guilherme.

Agradeço a minha segunda família: Cleber e Conceição; Fábio, Claudiana e Laiza; Flávio, Arthur e Bernardo; e a Vovó Lúcia.

Finalizo agradecendo a minha esposa Fernanda Scarpatti Pimentel por tudo que ela representa para mim: amor, carinho, companheirismo, honestidade, sinceridade e muito mais. *Anja* você é tudo de melhor que já aconteceu na minha vida! Te amo!

Deus, muito obrigado pela excelente vida que tenho e por todas as pessoas que o Senhor adicionou e adiciona a minha vida!

Resumo

Esta tese apresenta duas contribuições para as redes de *data center* centrado em servidores. A primeira, intitulada *Twin Datacenter Interconnection Topology,* foca nos aspectos topológicos e demostra como o uso de *Grafos Gêmeos* podem potencialmente reduzir o custo e garantir alta escalabilidade, tolerância a falhas, resiliência e desempenho. A segunda, intitulada *TRIIIAD* ***TRI****ple-Layered* ***I****ntelligent and* ***I****ntegrated* ***A****rchitecture for* ***D****atacenters*, foca no acoplamento entre a orquestração da nuvem e o controle da rede. A *TRIIIAD* é composta por três camadas horizontais e um plano vertical de controle, gerência e orquestração. A camada superior representa a nuvem do *data center*. A camada intermediária fornece um mecanismo leve e eficiente para roteamento e encaminhamento dos dados. A camada inferior funciona como um comutador óptico distribuído. Finalmente, o plano vertical alinha o funcionamento das três camadas e as mantem agnósticas entre si. Este plano foi viabilizado por um *controlador SDN* aumentado, que se integrou à dinâmica da orquestração, de forma a manter a consistência entre as informações da rede e as decisões tomadas na camada de virtualização.

Palavras-chave: comutação óptica, *data center*, Hipercubo, máquina virtual, NVF, orquestração, SDN, Virtualização, redes centradas em servidores, computação em nuvem.

Abstract

This thesis presents two contributions for server-centric network data center. The first one is a *Twin Datacenter Interconnection Topology* and focus on topological aspects with high potential for decreasing cost while ensuring high scalability, fault tolerance, resilience and overall network performance. The second one is called *TRIIIAD* ***TRI****ple-Layered* ***I****ntelligent and* ***I****ntegrated* ***A****rchitecture for* ***D****atacenters* and deals with the coupling between cloud orchestration and network control. TRIIIAD consists of three horizontal layers and a vertical control, management and orchestration plane. The top layer offers the IaaS (*Infrastructure as a Service*). The middle layer provides a lightweight routing/forwarding mechanism. The bottom layer works as a distributed optical switching. Finally, the vertical plane is responsible for coordinating the interoperation of those three layers and keeping them agnostic to each other. This plane brings a new concept for server-centric designs: an *Augmented Software-Defined Networking,* in which a SDN controller can integrate network control with orchestration, so that consistency between decisions taken at network and virtualization layers can be ensured.

Keywords: *cloud computer, data center, hypercube, NFV, optical switch, orchestration, SDN, server-centric network, virtual machine, virtualization*

Sumário

Lista de Figuras

Lista de Quadros

Lista de Tabelas

Nomenclatura

Siglas

Símbolo	Descrição
AMQP	*Advanced Message Queuing Protocol (Protocolo de Enfileiramento de Mensagens Avançado)*
API	*Application Programming Interface (Interface de Programação de Aplicação)*
ARP	*Address Resolution Protocol (Protocolo de Resolução de Endereço)*
A-SDN	*Augmented Software-Defined Networking (Rede Definida por Software Aumentada)*
CDF	*Cumulative Distribution Function (Função de Distribuição Acumulada)*
CDN	*Content Delivery Network (Redes de Distribuição de Conteúdo)*
CPU	*Central Processor Unit (Processador)*
DPI	*Deep Packet Inspection (Inspeção Profunda de Pacotes)*
ECMP	*Equal Cost Multipath Protocolo*
HDD	*Hard Disc Drive (Disco Rígido)*
IaaS	*Infrastructure as a Service (Infraestrutura como Serviço)*
IETF	*Internet Engineering Task Force (Força Tarefa de Engenharia para Internet)*
IP	*Internet Protocol (Protocolo de Internet)*
LDP	*Label Distribution Protocol (Protocolo de Distribuição de Rótulos)*
LLDP	*Link Layer Discovery Protocol (Protocolo de Descoberta da Camada de Enlace)*
LSP	*Label Switched Paths (Caminhos Comutados por Rótulos)*
LTS	*Long Term Support (Suporte de Longo Prazo)*
MEMS	*MicroElectroMechanical System (Sistema Micro-Eletro-Mecânico)*
NFV	*Network Function Virtualization (Virtualização de Função de Rede)*
OSPF	*Open Shortest Path First*
OTN	*Optical Transport Network (Rede Óptica de Transporte)*
OvS	*Open vSwitch*
PaaS	*Platform as a Service (Plataforma como Serviço)*
RAM	*Random Accesss Memory (Memória de Acesso Aleatório)*
REST	*Representational State Transfer (Transferência de Estado Representacional)*
SaaS	*Software as a Service (Software como Serviço)*
SAN	*Storage Area Network (Rede para Área de Armazenamento de Dados)*
SGBD	*Database Management Systems (Sistema Gerenciador de Banco de Dados)*
SDN	*Software-Defined Neworking (Redes Definidas por Software)*
STP	*Spanning Tree*
TCAM	*Ternary Content Addressable Memory*
TCP	*Transmission Control Protocol (Protocolo de Controle de Transmissão)*
ToR	*Top-of-Rack (Topo de rack)*
VLAN	*Virtual LAN*
VM	*Virtual Machine (Máquina Virtual)*
WDM	*Wavelength Division Multiplexing (Multiplexação por Divisão de Comprimento de Onda)*

Capítulo 1 – Introdução

A necessidade das empresas e pessoas em acessarem seus dados, a qualquer hora e em qualquer lugar, acarretou uma mudança de paradigma, na qual o uso de serviços locais já não atende ao anseio de seus usuários. Desta forma, versões online de serviços comuns, tais como: armazenamento de arquivos e serviços de impressão, tornaram-se disponíveis a todos os tipos de usuários [1][2]. Associado a esta mudança de paradigma, a utilização massiva de dispositivos móveis, com pouca capacidade de armazenamento e processamento, tais como: *smartphones* e *tablets*, faz com que os serviços online, sejam cada vez mais demandados pelos usuários. Ademais, as interfaces inteligentes e intuitivas, que ocultam do usuário a complexidade de utilização, promovem o uso, cada vez maior, de serviços *online*. Como exemplo, pode-se citar o compartilhamento, em uma rede social, da cena presenciada por uma pessoa. Há poucos anos atrás, esta pessoa precisava estar portando uma máquina fotográfica para captar a cena. Em outro momento, a imagem seria transferida para um computador, então seria postada em sua rede social favorita. Hoje, essa mesma pessoa, simplesmente sacaria seu dispositivo móvel do bolso, captaria a imagem, e por meio da função compartilhar, a imagem estaria disponível para os membros de sua rede social, instantaneamente. No entanto, o poder de computação e armazenamento real por trás das redes sociais, serviços de busca, serviços de discos virtuais e lojas virtuais, não está fisicamente localizado na máquina do usuário, nem no provedor de serviços. De fato, tal computação poderosa e flexível é realizada de forma ubíqua [3], tanto para o cliente quanto para o provedor de serviços, graças à *computação em nuvem*.

A computação em nuvem tem ao menos três modelos de serviços: *Software* como Serviço (SaaS – *Software as a Service*), Plataforma como Serviço (PaaS – *Platform as a Service*) e Infraestrutura como Serviço (IaaS – *Infrastructure as a Service*) [4]. Com base em suas necessidades, os clientes podem escolher um desses serviços para criar entidades lógicas, tais como: servidores *web*, unidades de armazenamento, serviços bancários, ou um

computador propriamente dito. Estas entidades podem crescer ou encolher em tempo real, a fim de garantir os níveis desejados de latência, desempenho, escalabilidade, confiabilidade e segurança, para atender as exigências de uma aplicação específica. Dessa forma, a computação em nuvem pode reduzir significativamente o capital inicial e tornar as despesas operacionais proporcionais às demandas atuais, tanto no quesito armazenamento, quanto no quesito processamento. Por exemplo, um computador em nuvem de uma loja virtual pode demandar dezenas, centenas ou milhares de servidores dependendo da época do ano. No entanto, computadores em nuvem, usualmente criados por conjuntos de máquinas virtuais (VMs), exigem uma infraestrutura de rede complexa, essencialmente implementada a baixo custo. Esta rede deve ter capacidade de reconfiguração flexível desde a camada física até a camada virtual, por exemplo, para habilitar a migração de VMs (ou mesmo de toda rede virtual) em todo o conjunto de servidores físicos. Ademais, deve existir um mecanismo de encaminhamento de tráfego eficiente e capaz de suportar um balanceamento de carga adequado. Hoje, é cada vez mais comum, que os equipamentos físicos que fornecem os recursos para a computação em nuvem estejam agrupados, em grandes quantidades, em instalações conhecidas como *Data Centers* [5].

Em poucas palavras, os *data centers* são instalações para armazenamento e processamento de dados, capazes de hospedar centenas de milhares de servidores físicos, interconectados, entre si e ao mundo exterior, por meio de sistemas de comunicação. Ademais, os *data centers* são dotados de sistemas de apoio, tais como: sistemas de suprimento de energia, sistemas de segurança física, sistema de controle de temperatura e umidade, entre outros. Em virtude da complexidade inerente de um ambiente de *data center*, seu projeto precisa atender a diversos requisitos, no entanto, escalabilidade, desempenho, resiliência, tolerância a falhas, eficiência energética e baixo custo, são características primordiais destes projetos [6][7]. Neste ponto, é importante notar, que a escalabilidade, o desempenho, a resiliência e a tolerância a falhas são características fortemente dependentes da topologia física escolhida para estruturar a rede. Portanto, na opinião do autor, a rede de computadores representa o principal desafio para o sucesso de um *data center.*

1.1 Propostas e tecnologias para redes de data center

Na tentativa de atender os requisitos apresentados acima, as redes de *data center* poderiam utilizar *hardware* e *software* especializados, como *InfiniBand* [8] e *Myrinet* [9], que normalmente são empregados em supercomputadores, conforme lista do Top500.org [10]. No

entanto, o uso destas tecnologias é inviabilizado em diversos projetos, devido ao alto custo (cada servidor necessitaria de interfaces proprietária de rede), e ao fato de não serem nativamente compatíveis com os protocolos TCP/IP. Por estas razões, diversas propostas, tais como: *Fat-Tree* [11], *Portland* [12], *VL2* [13], *DCell* [14], *BCube* [15], *Jellyfish* [16], entre outras., têm construído suas redes de *data center* utilizando equipamentos de baixo custo (COTS, ou *Commodity Off-the-Shelf*), tanto para encaminhamento de pacotes (*switches*, roteadores, etc.), quanto para processamento dos dados (servidores).

Estas topologias de redes de *data center* baseadas em COTS podem ser classificadas quanto a sua necessidade ou não de utilizar equipamentos de redes. De um lado, estão as arquiteturas centradas em rede (*network-centric*), tais como: *Fat-Tree* [11] e *Portland* [12], que utilizam equipamentos tradicionais para encaminhar o tráfego da rede (*switches*, roteadores e balanceadores de carga). De outro lado, estão as arquiteturas centrada em servidores (*server-centric*), por exemplo, o *Hipercubo* [17], que dispensam os equipamentos de redes. Nesta arquitetura mais escalável, os servidores necessitam de mais interfaces de redes e desempenham, além do processamento dos dados, as funções de roteamento, encaminhamento e balanceamento de carga do trafego da rede. Um meio termo entre estas arquiteturas são as topologias mistas, nas quais os servidores são responsáveis pelo roteamento, porém utilizam *switches* de baixo custo em suas implementações, tais como: *DCell* [14] e *BCube* [15].

Corroborando com o princípio das arquiteturas centradas em servidores, onde as funções da rede são controladas via *software*, um novo conceito de projetar arquiteturas de rede, denominado *Redes Definidas por Software* (SDN – *Software-Defined Networking*) [18], tem sido proposto. Esta nova abordagem busca desacoplar e separar o controle da rede dos mecanismos de encaminhamento, permitindo a programabilidade das funções de rede. Para isso, em SDN existe um controlador logicamente centralizado que possui uma visão global da rede e controla múltiplos dispositivos de encaminhamento de pacotes, que podem ser configurados via uma interface de programação padronizada. Hoje, a interface de programação de aplicações (API – *Application Programming Interface*) mais utiliza em SDN é o *OpenFlow* [19], devido a grande adesão por parte de fabricantes tradicionais de equipamentos de rede, que passaram a fornecer esta API nativamente eu seus dispositivos.

A abordagem SDN permite aos programadores adicionar novas funcionalidades à rede para atender requisitos específicos rapidamente. Isso elimina a dependência das atualizações

por parte dos fabricantes que ocorrem de forma muito lenta, e por vezes, não atende as novas necessidades do cliente. Um exemplo do emprego do SDN/*OpenFlow* que ganhou muita atenção no cenário mundial foi a implementação desenvolvida pela Google em seus *data centers* [20]. Por outro lado, o uso de SDN permite utilizar equipamentos mais simples, sem *software* proprietários e consequentemente mais baratos. Uma iniciativa interessante neste sentido é o projeto aberto do *switch* modular em *hardware* denominado *6-pack* desenvolvido pelo Facebook [21].

Uma proposta interessante para fortalecer a integração entre o SDN e as redes de *data center* centrado em servidores é a utilização de *Virtualização de Funções de Rede* (NFV – *Network Function Virtualization*) [22], que consiste na virtualização de funções específicas de rede, tais como: encaminhamento, roteamento, balanceamento de carga, *firewall*, etc. Enquanto o SDN está focado na programabilidade do plano de controle, o NFV está focado na programabilidade do plano de dado. Assim, o NFV pode estender a programabilidade do SDN, uma vez que as soluções NFV podem ser controladas por um *controlador SDN* [18]. Por sua vez, as redes de *data center* centrado em servidores requerem a utilização de NFV, para rotear e encaminhar pacotes entre os servidores. Em contrapartida, oferecem recursos de CPU e memória abundantes, criando um cenário favorável a programabilidade da rede como um todo.

Outra proposta promissora para o desenvolvimento de redes de *data center* é comutação óptica, explorada nas propostas *c-Through* [23] e *Helios* [24]. Esta tecnologia, emprestada das redes de transporte de telecomunicação, adiciona a rede a capacidade de provisionamento dinâmico de atalhos entre elementos distantes, por exemplo, a criação de caminho direto entre dois servidores para transferência de grandes fluxos de dados, como *backups*. De modo mais geral, a comutação óptica confere a rede, a flexibilidade de reconfigurar seus enlaces físicos. Assim, se associada a um *controlador SDN*, este poderia utilizar a comutação óptica para garantir melhor distribuição do tráfego por toda a rede.

1.2 Redes de Data Center: problemas práticos de implementação

Apesar do grande esforço que vem sendo feito, as redes de *data center* ainda apresentam diversos problemas que dificultam atender plenamente as necessidades da computação em nuvem, tais como: (*i*) equipamentos *Ethernet* de baixa confiabilidade e pouca

programabilidade; (*ii*) arranjos topológicos; (*iii*) desacoplamento dos planos de dados, de controle e de orquestração da nuvem.

1.2.1 Equipamentos Ethernet de baixa confiabilidade e pouca programabilidade

Propostas baseadas em COTS para encaminhamento dos pacotes apresentam três problemas: O primeiro está relacionado à confiabilidade e ocorre, principalmente, nas topologias que utilizam *mini-switches*, tais como *DCell* e *BCube*, pois estes equipamentos não foram projetados para ambiente de alta disponibilidade. Isso pode acarretar interrupções nos serviços com consequências significativas para os provedores. O segundo problema está relacionado ao fato dos *switches Ethernet* (COTS ou não) utilizarem o protocolo ARP, que é baseado em mensagens de *broadcast*, inundando na toda rede para localizar as máquinas de interesse. Isso gera uma sobrecarga de mensagens de controle, que cresce proporcionalmente com o tamanho da rede. Com isso, a rede precisa ser segmentada para escalar, fragmentando o conjunto de servidores e impondo limitações de comunicação entre os seguimentos [13]. Ainda, o desempenho da rede é prejudicado, pois o protocolo *Spanning Tree* (STP), ao construir uma topologia lógica para evitar laços, limita a capacidade fim-a-fim da rede (*bisection bandwidth*), gerando um aumento da latência. Por fim, o uso do protocolo IP como um localizador global, em redes fragmentadas, além de gerar uma enorme sobrecarga de configuração, contribui para a perda de agilidade da rede. O terceiro problema relacionado aos COTS é a baixa flexibilidade que eles apresentam. Normalmente estes equipamentos são *hardware black boxes*, que não fornecem programabilidade suficiente para experimentar novos mecanismos, pois eles são controlados por um sistema operacional proprietário, que fornece um conjunto padronizado de funções e possibilidades de configurações da rede. Assim, tais equipamentos são verdadeiros obstáculos para evolução das arquiteturas de rede, pois eles dificultam ou impossibilitam a experimentação de inovações para atender, por exemplo, as necessidades da computação em nuvem. Devido aos problemas apresentados acima, as implementações do *Fat-Tree* e do *Portland*, de fato, foram realizadas utilizando cartões NetFPGA de quatro portas e não *switches* COTS. No caso do *Fat-Tree*, houve a necessidade de modificar a procedimento de consulta da tabela de roteamento. No caso do *Portland*, foi instalado o *OpenFlow* nas NetFPGA para controlar a tabela de encaminhamento. Por outro lado, o *DCell* e o *BCube* implementaram uma camada de encaminhamento *2,5* com cerca de *13k* e *16k* linhas de código, respectivamente. Por fim, a implementação do *VL2* exige que

switches com suporte para: encaminhamento *L3* na taxa de linha, *OSPF*, *ECMP*, e encapsulamento *IPinIP*, que podem limitar sua implantação [25].

1.2.2 Arranjos topológicos

No que ser refere às questões topológicas, algumas arquiteturas, apresentam problemas de resiliência, utilizando caminhos mais longos na presença de falhas. Por exemplo, se um *mini-switch* das topologias *DCell* e *BCube* falha, o caminho alternativo aumenta em dois ou mais saltos, aumentando o atraso médio da rede [26]. Isso ocorre pelo fato destas topologias não apresentarem duas geodésicas entre os pares de nós. Uma segunda questão relativa à topologia é a possibilidade de "*opticalização*" dos enlaces da rede. No caso das arquiteturas centradas em rede o uso de *switches* ópticos baseados em Sistema Micro-Eletro-Mecânico (MEMS – *MicroElectroMechanical System*), além de apresentarem alto custo, possuem problemas relativos ao tempo de comutação, que podem sobrepujar a maior parte dos benefícios trazidos pela comutação óptica [23][24]. Por outro lado, as arquiteturas centradas em servidores devem possuir algoritmos de roteamento capazes de suportar alterações nos enlaces físicos, em função da comutação dos circuitos ópticos. Ainda, possibilitar que a "*opticalização*" ocorra de forma gradual. Os modelos topológicos do *DCell* e *BCube*, não fornecem suporte para tal "*opticalização*". Por fim, enquanto arquiteturas centradas em redes, consomem cerca de *15%* do custo total do *data center* [27], em equipamentos específicos para encaminhar tráfego na rede. Nas arquiteturas centradas em servidores, esta tarefa vai sequestrar parte da CPU dos servidores para encaminhar os dados. Dependendo do cenário de tráfego, o uso de CPU pode atingir valores próximos a 90% [28].

1.2.3 Desacoplamento dos planos de dados, controle e de orquestração da nuvem

Diversas propostas exploram o uso do SDN para implementar o plano de controle separado do plano de dados da rede. Embora resolva vários problemas, a arquitetura estanque do SDN pode causar outros tipos de problemas. Em particular, em ambientes virtualizados, o modelo tradicional de endereçamento/localização (IP/MAC) [29], com VMs interligadas por tabelas de fluxos baseadas neste par, pode severamente prejudicar tanto a agilidade quanto a eficiência de políticas migratórias das VMs. Pois, a sobrecarga de comunicação entre o plano de dados e o *controlador SDN*, consequentemente aumenta o atraso de instalação de regras. Por exemplo, em [30] a migração das VMs envolve a cópia das regras de fluxos do *switch* de

origem para o *switch* de destino. A VM é migrada. Então, um túnel entre os *switches* de origem e destino é criado para manter a comunicação entre a VM migrada e as demais VMs conectadas ao *switch* de origem. Processos de migração de VM similares envolvendo instalações de regras de fluxos e/ou tunelamento são apresentados em [31] e [32].

Em paralelo, há ainda a utilização de um mecanismo de orquestração, para alocação das VMs sobre os recursos físicos de processamento. Esta abordagem impede que as entidades tenham uma visão holística sobre o ambiente a ser gerenciado. De modo que, por exemplo, uma decisão tomada pelo mecanismo de orquestração, sem conhecimento do estado da rede, possa conflitar com as políticas do *controlador SDN*. Em alguns casos, gerando inconsistência entre as redes virtuais e a rede física. Em redes de *data center* centrado em servidores, este desacoplamento pode gerar problemas ainda maiores, pois os servidores são responsáveis, tanto pelas funções de rede, quanto pela virtualização das máquinas.

1.3 Motivação: o problema fundamental da arquitetura centrada em servidores

O problema fundamental das topologias de *data center* centrado em servidores é o acoplamento da função fim (processamento) com a função meio (encaminhamento). Isso implica numa orquestração mais complexa do que no sistema convencional, onde as funções estão desacopladas. A Figura 1.1ilustra os modelos de processamento e encaminhamento das duas abordagens.

Observe na Figura 1.1(a) que as arquiteturas centradas em rede utilizam equipamentos específicos (*switches*) para realizar as funções de rede, desta forma, os equipamentos de processamento (servidores) recebem apenas os pacotes destinados a eles; e ocupa toda sua CPU nas atividades do usuário. Por outro lado, a Figura 1.1(b) mostra que as arquiteturas centradas em servidores utilizam o mesmo equipamento físico (servidores) para realizar as duas funções, de forma que todos os pacotes que chegam a suas interfaces devem ser tratados pela CPU. Isso pode acarretar em processamento deficiente de tarefas dos clientes (CPU tomada por funções de rede) ou limitar as taxas dos fluxos da rede (CPU tomada por tarefas de usuários em elementos intermediários).

A solução natural para abordar este problema aponta para uma forma eficiente e leve de encaminhamento em *kernel*. Todavia, neste caso a função meio (encaminhamento) ganha precedência sobre a função fim (processo em *kernel* versos processos de usuários).

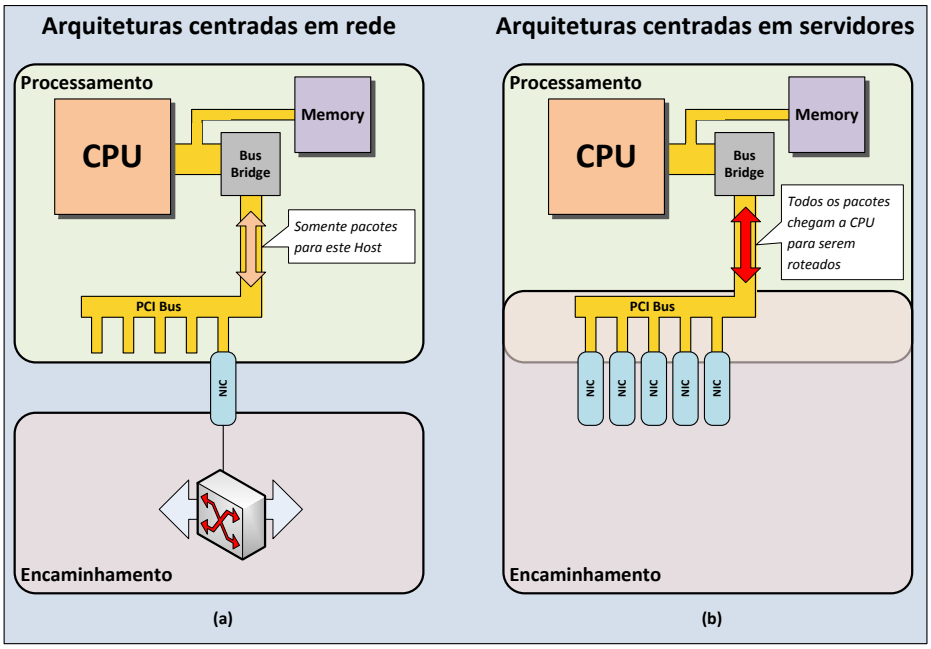


Figura 1.1: Modelo de processamento e encaminhamento: na arquitetura centrada em rede e na arquitetura centrada em servidores.

Outra solução é trabalhar a questão topológica da rede de interconexão, para minimizar o tráfego trânsito via reconfiguração física dos enlaces. Porém, isso tem consequências para o mecanismo de roteamento/encaminhamento, conforme comentado. Também são importantes aspectos da topologia: a multiplicidade de caminhos para favorecer o espalhamento do tráfego e possibilitar um balanceamento de carga de encaminhamento sobre os servidores; e a tolerância a falhas com preservação de resiliência.

Mecanismos autonômicos poderiam também auxiliar na distribuição de carga de encaminhamento via servidores pouco carregados. Todavia, tais iniciativas tornam-se muito mais complexas pelo acoplamento dos recursos comuns na arquitetura centrada em servidor, podendo facilmente levar o sistema a instabilidades. Por exemplo, a migração de uma VM sobrecarrega a rede, que por sua vez força outra migração por falta de recursos de CPU, implicando em sobrecarga da rede em outro local, e assim sucessivamente.

1.4 Hipóteses de trabalho e referências iniciais

A lista abaixo sistematiza as principais hipóteses que serão testadas durante o desenvolvimento deste trabalho. A fundamentação para formulação de cada hipótese baseia-se em resultados e modelos das respectivas referências:

1. Deve haver outras formas eficientes, em teoria de grafos, para interconectar redes de *data center* centrado em servidores, ainda não exploradas e com sucesso na interconexão de outras redes [33];
2. É possível implementar, em nível de *kernel,* um esquema leve de roteamento de fonte para reduzir o uso de CPU em nós intermediários, que permita ainda, a criação de caminhos contínuos e coerentes configurados pelo *controlador SDN* [34], porém sem a necessidade de sinalização em nível de fluxo;
3. Há espaço para o uso seletivo do SDN para viabilizar o desacoplamento do endereço IP com a localização física da VM, viabilizando a escalabilidade da solução para *data center* centrado em servidores e a migração transparente de VMs [12];
4. Para topologias com algum nível de regularidade, a "*opticalização*" do processo de comutação de forma distribuída e gradual é viável para reduzir a carga de encaminhamento sobre a CPU [35]. A integração da comutação óptica com o processo de roteamento/encaminhamento pode ainda ser integrada as funções SDN;
5. É necessária a integração das decisões do *controlador SDN* da rede [36] conjugadas com o orquestrador de uma arquitetura em nuvem [37], para viabilizar soluções autonômicas de operação dessa rede multicamada, envolvendo: reconfiguração de camada física, configuração dos elementos encaminhadores, criação de redes virtuais e migração de VM com base na utilização dos recursos.

As contribuições do trabalho serão, em grande parte, originadas da confirmação e ou refutação parcial das hipóteses individuais, bem como as soluções de compromissos entre os objetivos conflitantes entre elas.

1.5 Compromissos

Durante o desenvolvimento deste trabalho os seguintes compromissos foram assumidos:

- Uso de plataforma e programas de código fonte aberto para: (*i*) desenvolvimento de aplicações SDN; (*ii*) virtualização de máquinas e redes; (*iii*) encaminhamento de pacotes em nível de *kernel*; (*iv*) sistemas embarcados;
- Congelamento das versões das plataformas e programas utilizados;
- Uso de recursos heterogêneos;
- Em especial, uso de servidores com poder de processamento inferiores às demandas de encaminhamento de rede, visando um cenário futuro de crescimento acentuado da capacidade de transmissão (*40 Gbps, 100 Gbps* ou mais), onde mesmo os processadores mais poderosos terão dificuldades em lidar com o volume de tráfego imposto a eles.

1.6 Tese a ser defendida

Defendemos que em arquiteturas centradas em servidores as reconfigurações em camada física; os planos de controle e gerenciamento da rede; e a orquestração das VMs são elementos que não podem ser tratados de forma isolada. Para realizar esta tarefa de forma integrada, faz-se necessário um novo conceito de SDN: um SDN aumentado (A-SDN – *Augmented Software-Defined Networking*), que além dos aspectos de rede, integra-se à dinâmica da orquestração, de forma a manter a consistência entre as informações da rede e as decisões tomadas; e não na direção contrária de uma ampliação da orquestração para observar as atividades da rede; nem com relação horizontal (*east/west bound interfaces*) entre orquestrador e *controlador SDN*, com feito no MANO (*Management and Orchestration*) em NFV [38].

1.7 Descrição do trabalho desenvolvido

A busca por uma topologia eficiente para redes de *data center* centrado em servidores deu origem a *Twin Datacenter Interconnection Topology* [26], que explorou o uso de uma classe de grafos 2-geodesicamente-conexo denominada *Grafos Gêmeos* [39]. A implementação a avaliação da topologia *Twin* foi confrontada com *Fat-Tree*, *DCell*, *BCube* e

Hipercubo em relação aos aspectos: custo, escalabilidade, tolerância da falhas, resiliência e desempenho. Com isso, verificou-se que as topologias *Twin*: (*i)* apresentam ótima relação custo-benefício, pois utilizam o menor número possível de enlaces, consumindo menos energia e reduzindo o CAPEX e o OPEX ; (*ii*) são escaláveis e com granularidade a partir de uma unidade; (*iii*) são tolerantes a falhas e resilientes, pois possuem duas geodésicas para cada par de nós não adjacentes. No entanto, algumas implicações práticas, ainda não resolvidas, inviabilizam o uso dos *Twin* em redes de grande escala, principalmente: (*i*) a dificuldade de encontrar uma topologia que atenda aos requisitos do projeto, devido à complexidade computacional de gerar todas as possiblidades para então escolher a que melhor se adequa; (*ii*) do melhor do nosso conhecimento, não existe um algoritmo de encaminhamento/roteamento específico para esta topologia. (*iii*) a dificuldade de implantar a comutação óptica distribuída para reconfigurar os enlaces, e ao mesmo tempo, manter a topologia *Twin*. Portanto, a limitação (*ii*) conflitou com a hipótese 2 e a limitação (*iii*) conflitou com a hipótese 4, de forma que a solução de compromisso limitou o desenvolvimento do restante do trabalho à utilização do *Hipercubo*.

Para testar as demais hipóteses, este trabalho apresenta o projeto, a implementação e a avaliação da *TRIIIAD (**TRI**ple-Layered **I**ntelligent and **I**ntegrated **A**rchitecture for **D**atacenters)*; uma arquitetura autonômica composta por três camadas horizontais e um plano vertical de controle, gerência e orquestração; baseada num *controlador SDN* aumentado, capaz de reconfigurar os enlaces da camada física e orquestrar a migração das máquinas virtuais com base nas cargas dos servidores físicos. Para isso o *controlador SDN* integrou-se à dinâmica da orquestração, de forma a manter a consistência entre as informações da rede e as decisões tomadas na camada de virtualização.

A *TRIIIAD* apoia-se sobre uma rede de *data center* centrado em servidores, interconectada por uma topologia de auto-roteamento/roteamento de fonte (*Hipercubo*), que se mantém estável mesmo com a reconfiguração de alguns de seus enlaces físicos. Esta abordagem conferiu a flexibilidade e a programabilidade necessárias para implementação de um mecanismo de encaminhamento de pacotes, sobre uma arquitetura x86, como uma *função virtual de rede* (NFV), cujos parâmetros são definidos pelo *controlador A-SDN*.

Para adicionar a capacidade de reconfiguração dos enlaces e testar a hipótese 4, foram distribuídas chaves ópticas 2x2 na camada física, de modo a criar planos de chaveamentos

capazes de reduzir o número médio de saltos no encaminhamento de pacotes, colaborando para redução do tráfego de trânsito.

Por fim, uma camada de virtualização permitiu o compartilhamento dos recursos físicos e compatibilizou a TRIIIAD com as tecnologias de redes amplamente difundidas, tais como: IPv4 e *Ethernet*.

Para comprovar a viabilidade técnica de programar essa arquitetura utilizando *hardware* de prateleira, bem como, para validar os conceitos e caracterizar o funcionamento da TRIIIAD, foi realizada uma implementação para demonstração do princípio, com oito servidores físicos e um par de chaves ópticas 2x2. Esta implementação foi ancorada na plataforma de computação em nuvem *OpenStack*, no sistema operacional *Linux Ubuntu* e no *switch* em *software Open vSwtich*, na plataforma *Ryu SDN* e no protocolo *OpenFlow.*

1.8 Contribuições da tese

As contribuições desse trabalho são:

- *Twin Datacenter Interconnection Topology.* Projeto, implementação e avaliação da classe de grafos 2-geodesicamente-conexo denominada *Grafos Gêmeos* aplicado a uma rede de *data center* centrado em servidores, avaliando e comparando os aspectos topológicos dessa propostas com outras encontras na literatura.
- *Arquitetura TRIIIAD*: Projeto, implementação e avaliação de uma arquitetura em três camadas, automatizada por um *controlador SDN* aumentado, capaz de reconfigurar os enlaces da camada física e orquestrar a migração das máquinas virtuais com base nas cargas dos servidores físicos. A arquitetura conta ainda com um mecanismo de encaminhamento eficiente implementado como uma função virtual de rede sobre uma topologia de rede de *data center* centrada em servidores.

1.9 Organização da tese

Este documento foi organizado da seguinte forma: o capítulo 2 discute com mais detalhes o contexto no qual o presente trabalho está inserido; o capítulo 3 apresenta a *Twin Datacenter Interconnection Topology*: a uma topologia para redes de *data center* centrado em servidores, inspirada na teoria dos grafo, que utiliza o menor número possível de enlaces para

interconectar os nós; o capítulo 4 apresenta em detalhes o projeto da arquitetura *TRIIIAD:* uma arquitetura autonômica composta por três camadas horizontais e um plano de: controle, gerência e orquestração; o capítulo 5 discute a integração entre os elementos da arquitetura *TRIIIAD* e apresenta o conceito de A-SDN (*Augmented Software-Defined Networking,* ou *Rede Definida por Software Aumentada);* o capítulo 6 apresenta em detalhes o ambiente experimental desenvolvido sobre *hardware* de prateleira, para validar os conceitos da *TRIIIAD*; o capítulo 7 apresenta e discute os resultados da caracterização do funcionamento da *TRIIIAD* em função dos recursos (CPU e memória) dos *nós de computação*, sobre o ambiente experimental construído; o capítulo 8 demostra e discute como o *controlador A-SDN* utiliza os parâmetros de carga dos *nós de computação* (CPU, memória e tráfego de trânsito) para realizar a orquestração autônoma da *TRIIIAD*; o capítulo 9 apresenta a conclusão do trabalho e indica os trabalhos futuros que serão investigados. Por fim, três apêndices são fornecidos: o Apêndice A apresenta os detalhes da implementação do mecanismo de roteamento sobre o *OvS*; O Apêndice B apresenta uma ilustração ampliada do diagrama de classes do *software* do *controlador A-SDN*; e o Apêndice C apresenta o ambiente OpenStack; seus componentes; suas vantagens e limitações.

Capítulo 2 – Contexto

O objetivo deste capítulo é fornecer é discutir com mais detalhes o contexto no qual o presente trabalho está inserido.

2.1 As redes de data center

Os *Data centers* são instalações que podem ser encaradas como verdadeiras fábricas de armazenagem e processamento de dados, capazes de hospedar centenas de milhares de servidores físicos, interconectados, entre si e ao mundo exterior, por meio de sistemas de comunicação, muitas vezes redundante. Outros sistemas de apoio como: sistemas de suprimento de energia, sistemas de segurança física, sistema de controle de temperatura e umidade, são essenciais para permitir o funcionamento destas fábricas. De forma que, a sinergia entre todos estes sistemas é fundamental para criar *data centers* cada vez mais produtivos, ou seja, capazes de manipular o maior volume de informações, no menor tempo possível.

Para interligar seus servidores, os *data centers* modernos impõe requisitos cada vez mais severos para as redes de computadores que os suportam. Entre os principais requisitos estão a escalabilidade, tolerância a falhas (robustez), resiliência, capacidade de agregação de banda, desempenho (latência) e baixo consumo de energia. Sendo que várias destas características são fortemente dependentes da topologia física, a rede tornou-se o elemento chave para o sucesso de um *data center*.

Ao longo dos últimos anos, diversas propostas topológicas foram desenvolvidas para criar redes eficientes e escaláveis, tais como: *Fat-Tree* [11], *VL2* [13], *DCell* [14] e *BCube* [15]. As topologias baseadas em COTS são classificadas quanto à necessidade ou não de equipamentos de redes. De um lado, estão as arquiteturas centradas em rede (*network-centric*) que utilizam equipamentos tradicionais para encaminhar o tráfego da rede. De outro lado,

estão as arquiteturas centrada em servidores (*server-centric*), que utilizam os servidores para as funções de: roteamento, encaminhamento e balanceamento de carga do trafego da rede, além do processamento dos dados. A Figura 2.1 ilustra as duas abordagens de construção de redes de *data center.*

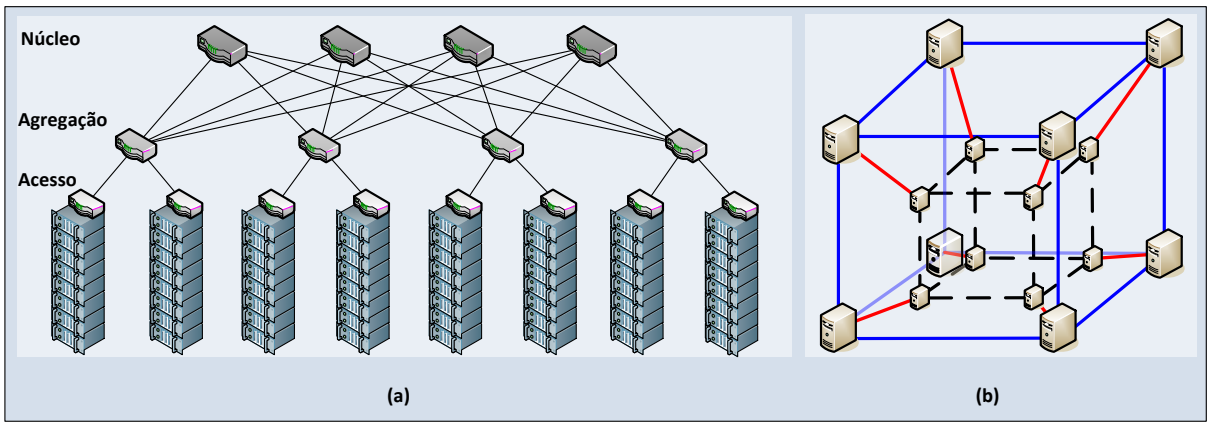


Figura 2.1: Classificação das redes de *data center*: (a) arquiteturas tradicionais centradas em redes (*network-centric*), (b) arquiteturas centradas em servidores (*server-centric*).

2.1.1 Redes de data center centrado em redes (network-centric)

As redes de data center centrado em redes, tais como: *Fat-Tree* [11], *Portland* [12], *VL2* [13], são as topologias tradicionais, constituídas por servidores físicos agrupados em armários (*racks*) e interligados através de um *switch Ethernet* de acesso no topo desses armários, comumente chamados de *ToR (Top-of-Rack*). Tipicamente estas redes utilizam duas ou três camadas de *switches* de alta capacidade. Entretanto, dispositivos como roteadores e balanceadores de carga também são necessários para rotear os pacotes pelos vários segmentos da rede e distribuir a carga entre os elementos de encaminhamento, respectivamente [11].

No exemplo ilustrado na Figura 2.1(a), é possível observar uma rede de três níveis, onde os *switches* de acesso (*Access*) conectam os servidores aos *switches* de agregação (*Agregation*), que agregam armários de servidores. Por fim, estes se ligam a um terceiro nível de *switches* que formam o núcleo da rede (*Core*). É por meio desta abordagem que as topologias centrado em redes conseguem escalar em número de máquinas. Entretanto, esta topologia tradicional, associada aos protocolos legados, apresentam diversas dificuldades para atender os requisitos das redes de *data center* atuais.

2.1.2 Redes de data center centrado em servidores (server-centric)

Por outro lado, as redes de data centrado em servidores, tais como: *DCell* [14], *BCube* [15], *Hipercubo* [17], são topologias que dispensam equipamentos tradicionais de rede, para roteamento e encaminhamento dos pacotes, bem como para balanceamento da carga na rede. No entanto, alguns casos podem utilizar *mini-switches* como a função de expandir o número de portas dos servidores, como é o caso do *DCell*, *BCube*. Estas topologias mais escaláveis, evidentemente, implicam que os servidores físicos devem possuir mais de uma interface de rede, a fim de realizarem as funções de: roteamento, encaminhamento e balanceamento de carga do trafego da rede, além do processamento dos dados os usuários.

No exemplo ilustrado na Figura 2.1(b), é possível observar uma rede de *data center* centrado em servidores estruturada como um *Hipercubo* de grau 4. Observe que não existe nenhum elemento tradicional de rede e que cada elemento possui 4 interfaces de rede para estabelecer as conexões com seus vizinhos.

Devido a sua habilidade de multiplexar os núcleos da CPU entre as aplicações e os processos de encaminhamento de pacotes [40], a abordagem de redes de *data center* centrado em servidores abre novas possibilidades para testar novos projetos e serviços. Ainda, tal como mostrado em [27] esta redução pode alcançar até *15%* do custo total de um *data center*. O consumo de energia pode ser reduzido e proporcional à utilização do *data center*. Estas características tornam as arquiteturas centradas em servidores uma forma de interconexão muito atrativa financeiramente. No entanto, um problema muito conhecido das arquiteturas centradas em servidores não foi ainda propriamente resolvido: a sobrecarga que o encaminhamento por múltiplos saltos gera sobre a CPU do servidores. O *DCell*, o *BCube* utilizam em média 40% e 36% de da CPU para encaminhamento do tráfego de trânsito. Dependendo do cenário de tráfego, o uso de CPU pode atingir valores próximos a 90% [28]. Como resultado a latência pode se tornar um fator proibitivo para aplicações sensíveis ao atraso, tais como: bolsas de valores (100 μs a 1 ms) e computação de alto desempenho (1 μs a 10 μs) [41].

2.2 Redes Definidas por software

As *Redes Definidas por Software* (SDN – *Software-Defined Networking*) buscam desacoplar e separar o controle da rede dos mecanismos de encaminhamento, criando uma forma de, verdadeiramente, programar as funções de rede [18]. Em SDN, um controlador

logicamente centralizado trabalha sobre uma visão global da rede que lhe confere informações para: calcular as melhores rotas com base em diversos parâmetros; distribuir os fluxos de dados pela rede; isolar elementos ou enlaces que apresentem falhas; particionar a rede física em fatias, etc. Isso tudo é realizado programando os múltiplos dispositivos de encaminhamento de pacotes, via uma interface de programação padronizada, por exemplo, *OpenFlow* [19].

Quanto aplicado a topologias centrada em rede, o SDN pode: criar mecanismos eficiente de balanceamento de carga, facilitar o encaminhando dos pacotes e criar topologias virtuais, etc. [12] [34]. Entretanto, os *switches* e demais elementos de rede devem ser adquiridos de fabricantes que fornecem o *OpenFlow* em seus dispositivos. Adicionalmente, seus *hardware/firmware* normalmente são compatíveis com uma distribuição específica do *OpenFlow*. Isto pode representar um obstáculo para atualização e programação de novos serviços na rede.

Por remover os dispositivos de rede, e com eles as limitações das implementações dos fabricantes, as arquiteturas de rede centrada em servidores podem abrir novos horizontes para o uso de SDN em ambientes de *data center*. Especialmente depois que o *Open vSwitch* foi integrado ao *kernel 3.3* do sistema operacional Linux, em substituição da *bridge* tradicional. Isso trouxe dois grandes melhoramentos para o Linux: (*i*) O *OvS* é especialmente projetado para ser utilizado com um *switch* virtual em ambientes de virtualização como é o caso da nuvem; e (*ii*) o *OvS* habilita o Linux em um "dispositivo" *OpenFlow*. Porém, problemas relacionados com a escalabilidade do controlador logicamente centralizado e o uso de *hardware* de propósito geral para encaminhamento, podem representar grandes desafios que precisam ser superados. Primeiro, os *switches* em *software* não páreo para suas contrapartes em *hardware*. Por exemplo, a operação de busca da *tupla* do *OpenFlow (*na versão 1.0 composta por dez campos), na tabela de fluxos, pode ser realizada em paralelo por uma memória em *hardware* do tipo TCAM (*Ternary Content Addressable Memory*). Segundo, a alocação de novos fluxos pode sobrecarregar tanto o canal de controle como o controlador SDN, pois um conjunto muito grande (dezenas de milhares de servidores) irão enviar requisições para o controlador centralizado, ao invés de um conjunto limitado de switches. Por outro lado, a inserção de novos fluxos pode ser mais rápida nas topologias centradas em servidores. Resultados experimentais em [42] revelaram valores de latência para o tempo de inserção de fluxo em torno de 0,1 ms por fluxo no *OvS*. Apesar do tempo de inserção ser dez

vezes mais rápido que os *switches* em *hardware*, que possuem CPU menos potentes que os servidores, a convergência de encaminhamento das redes centrada em servidores podem ser significativamente maiores, devido a seus caminhos de múltiplos saltos.

2.3 Virtualização de Funções de Rede

Nos últimos anos, as empresas de Telecomunicações e Tecnologia da Informação têm sofrido uma enorme pressão para reduzir os custos de capital (CAPEX – *Capital expenditures*) e custos operacionais (OPEX – *Operational Expenditure*) como forma de otimizar os investimentos e obter maior retorno de capital. A intensa evolução das tecnologias de virtualização, que permitem compartilhar, simultaneamente, recursos de processamento, armazenamento e comunicação de um mesmo *hardware* físico entre diversas máquinas virtuais, tem sido uma ferramenta importante para atingir esses objetivos [43].

Neste contexto, a existência de diversos *appliances* com *hardware* e *software* dedicados e especializados em realizar uma função específica da rede, como, por exemplo, *firewalls*, redes de distribuição de conteúdo (*CDN – Content Delivery Network*), equipamentos para inspeção profunda de pacotes (*DPI – Deep Packet Inspection*) e roteadores, têm representado um desafio para os provedores de serviços de rede e telecomunicações. Os principais problemas estão relacionados ao fato desses equipamentos terem um ciclo de vida relativamente reduzido e demandarem um alto investimento de CAPEX e OPEX, limitando, assim, a escalabilidade e flexibilidade da rede [44]. Outro fator importante, é o tempo que um novo serviço demora em chegar ao mercado, dificultando iniciativas de inovação.

A tecnologia conhecida como Virtualização das Funções de Rede (*NFV – Network Functions Virtualization*) tem como objetivo resolver esses problemas e tornar as redes mais simples e flexíveis, se beneficiando da evolução das tecnologias de virtualização e minimizando a dependência de restrições de *hardware* [45]. NFV busca transferir as funções de rede de *appliances* comerciais, com *hardware* e *software* dedicado, para equipamentos de prateleira, executando funcionalidades baseadas em *software* sobre *hardware* padrão.

Em NFV, devido ao uso das tecnologias de virtualização, é possível que um mesmo equipamento físico execute diversas funções de rede para diferentes aplicações e usuários. Dessa forma, pode-se citar como benefícios esperados com a adoção da tecnologia: redução de custos CAPEX e OPEX, diminuição do tempo para um novo produto chegar ao mercado,

melhor escalabilidade e flexibilidade para instanciação de novos serviços, incentivo à inovação e fortalecimento de plataformas abertas. Entretanto, a introdução de NFV traz também vários desafios que ainda precisam ser endereçados pela academia e pela indústria para sua adoção bem sucedida [45], tais como: desempenho aceitável em relação ao *hardware* de propósito específico; gerenciamento, alocação e orquestração de *appliances* virtuais; segurança e resiliência.

2.4 Comutação óptica aplicada as redes de data center

Originalmente empregada nas redes de transporte de telecomunicação para implementar engenharia de tráfego e/ou realizar balanceamento de carga, a *comutação de circuitos ópticos* é uma tecnologia promissora para flexibilizar a alocação de banda nas redes de *data center*, pois adiciona à rede a capacidade de provisionamento dinâmico de atalhos entre elementos distantes. Quando empregada em conjunto com a multiplexação por divisão de comprimento de onda (*WDM – Wavelength Division Multiplexing)*, sua capacidade de transmissão é aumenta em dezenas de vezes, permitindo, por exemplo, a transmissão de vários fluxos de 10 *Gbps* sobre um único cabo de fibra óptica, desde que todo o tráfego seja para um mesmo destino. Dessa forma, o seu uso é mais adequado à demanda de tráfego agregado dos *switches top-of-rack* das arquiteturas de *data center* centradas em rede.

Porém, a comunicação óptica também apresenta algumas desvantagens. A primeira é relacionada ao alto custo dos *switches* ópticos. Assim, quando para uma grande quantidade de rajadas de demanda de tráfego agregado entre diferentes pares de *switches top-of-rack,* o número de circuitos ópticos para suportar simultaneamente essas comunicações em rajadas no *data center* seria proibitivo, em função do alto custo dos equipamentos ópticos. O segundo problema está relacionado à proposta usual de uso da comutação óptica como uma facilidade de rede centralizada, que se adequa melhor as arquiteturas centradas em rede, a que as arquiteturas centradas em servidores. Ademais, esta proposta apresenta um único ponto de falha, de forma que uma falha eventual no comutador óptico tornaria toda a rede óptica indisponível. O terceiro problema é relacionado com *tempo de comutação* do Sistema Micro-Eletro-Mecânico (MEMS – *MicroElectroMechanical System*), utilizado como tecnologia usual para redes ópticas reconfiguráveis. Nesta tecnologia a comutação acontece através do redirecionamento de espelhos, controlados por microcontroladores. Esses espelhos estão acoplados a pequenos motores que os giram para alcançarem ângulos desejáveis e assim

refletirem da melhor maneira os sinais injetados, criando conexões entre as portas de entrada e saída especificadas.

Algumas propostas como o *c-Through* [23] e *Helios* [24] utilizam *switches* baseados em MEMS para criar arquiteturas híbridas, também conhecidas como *HyPaC Network* (*Hybrid Packet and Circuit Network*), que tenta encontrar uma solução que combine os benefícios da comutação óptica e elétrica. No entanto, essas propostas mostram que o tempo gasto para reconfigurar os micros-espelhos é em torno de *12 ms* e que essa ausência de sinal é tempo suficiente para a maioria dos transceptores ópticos detectarem a perda do enlace. Na prática, isso gera atrasos ainda maiores na rede, dado que o tempo para os transceptores se recuperarem gira em torno de *15 ms*. Portanto, o tempo total de atraso (em torno de *27 ms*) pode sobrepujar a maior parte dos benefícios trazidos pela comutação óptica de circuitos.

2.4.1 Comutação óptica distribuída

Outra abordagem para se construir arquiteturas do tipo *HyPaC Network* é distribuir comutadores ópticos de baixo custo por toda a topologia da rede, conforme proposto em [35]. Entre as vantagens do espalhamento desses dispositivos pela rede em relação a grandes *switches* ópticos pode-se destacar: (*i*) o sinal é regenerado a cada salto; (*ii*) não existem cascateamentos ou chaveamentos multi-estágio para criar uma grande matriz de chaveamento; (*iii*) permitem o uso de dispositivos de baixo custo; e (*iv*) não apresentam problemas de *crosstalking*. Experimentos conduzidos pelo autor deste trabalho, publicados em [46], mostram que, com chaves magneto-ópticas 2x2 de prateleira, é possível reconfigurar a camada física de forma que esta operação não afete os transceptores ópticos, em outras palavras, os receptores não detectam a perda do sinal. Ainda, foi demostrado que este princípio se mantém, inclusive, em situações onde os servidores (ou serviços sendo executados nestes servidores) demandem alta disponibilidade de suas conexões. Portanto, esta abordagem se mostra ideal para ser aplicada as redes de *data center* centrado em servidores, no intuito de criar atalhos para reduzir o tráfego de trânsito inerente a essa arquitetura. Ademais, esta abordagem permite a implantação parcial de dispositivos ópticos na rede, tornando o projeto economicamente viável. Por fim, uma implantação em 50% dos planos de comutação, representaria em torno de ¾ da redução total do tráfego de trânsito obtida com a implantação em 100% dos planos, conforme demostrado em [35].

Capítulo 3 – Aspectos topológicos das redes de data center

A interconexão topológica dos equipamentos tem sido tratada como o fator chave por diversas propostas de *data centers*, tais como: *Fat-Tree* [11], *Portland* [12], *VL2* [13], *DCell* [14], *BCube* [15] e Jellyfish [16], baseadas em COTS. Isso porque, a topologia influência diretamente nos seguintes requisitos de projeto: custo, escalabilidade, tolerância da falha, resiliência e desempenho.

De forma geral, todas as propostas possibilitam a criação de topologias com milhares de servidores, mas com uma granularidade grossa devido à regularidade exigida por suas estruturas de interconexão. Além disso, a característica de tolerância a falhas dessas propostas é alcançada com um custo muito alto. Este fato pode ser constatado, dependendo da topologia, tanto em termos do grande conjunto de enlaces "sobressalentes", quanto em termos do aumento excessivo no número de saltos na presença de falhas, como será demonstrado mais adiante.

Para abordar esses aspectos, foi proposto neste trabalho um modelo inovador de topologia intitulado *Twin Datacenter Interconnection Topology* [26], para redes de *data center* centrado em servidores, baseado em na classe de grafos 2-geodesicamente-conexo denominada *Grafos Gêmeos* [39]. As topologias *Twin* beneficiam-se das propriedades destes grafos, para aprimorar a escalabilidade e resiliência das redes, com uma ótima relação custo-benefício. Os resultados dos experimentos realizados mostram que as topologias *Twin*: (*i)* apresentam um custo de enlaces menor que as topologias confrontadas, pois utilizam o menor número possível de enlaces, consumindo menos energia e reduzindo o CAPEX e o OPEX ; (*ii*) são escaláveis e apresentam granularidade de crescimento a partir de uma unidade; (*iii*) são tolerantes a falhas e resilientes, possuindo propriedades atrativas para distribuição de fluxos em condições de operação normal e na presença de falhas.

3.1 Topologia *Twin*

Seja $G=G(V,E)$ um grafo que consiste de um conjunto $V(G)$ de vértices ou nós, e um conjunto $E(G)$ de arestas ou enlaces, cada enlace uv conecta um par de nós $(u,v) \in G$. A ordem de G é $n=|V(G)|$, e seu tamanho $m=|E(G)|$. Todos os grafos deste capítulo são conexos, não direcionados e sem pesos. Um grafo é conexo se existe um caminho conectando cada par de nós $(u,v) \in G$.

Dois nós $u,v \in V(G)$ são ditos adjacentes se o enlace uv pertence a $E(G)$. A vizinhança $\Gamma(v)$ de um nó $v \in V(G)$ é o conjunto de nós adjacentes a v em G. A cardinalidade, isto é, o número de elementos de $\Gamma(v)$ define o grau do nó v. Dois nós $u,v \in V(G)$ formam um par gêmeo (u,v) em G, se e somente se eles possuem a mesma vizinhança [39].

O menor caminho, em número de saltos, conectando dois nós $u,v \in V$ em G é chamado de distância geodésica u-v. O diâmetro $d=d(G)$ é o número de enlaces da maior geodésica em G.

Um grafo é 2-conexo, se e somente se existem no mínimo dois caminhos disjuntos entre cada par de nós $u,v \in V(G)$. Como os caminhos disjuntos também são disjuntos por enlaces, uma topologia física 2-conexo sobrevive a uma falha individual de nó ou de enlace, pois neste caso cada par de nós ainda está conectado por um caminho disjunto. Entretanto, não há restrição quanto ao número de saltos do novo caminho, com relação ao número de saltos da geodésica no grafo original. Neste sentido, a classe de grafos 2-geodesicamente-conexos foi definida como um subconjunto particular da classe de grafos 2-conexos [47]. Um grafo G e 2-geodesicamente-conexo (*2-GC*), se e somente se existem no mínimo 2 geodésicas disjuntas por nós entre cada par de nós não adjacentes $u,v \in V(G)$. Assim, no caso de uma única falha em qualquer nó ou enlace, cada par de nós não adjacentes ainda é interligado por uma geodésica alternativa, contornando o elemento que falhou.

Com base nas definições acima, todo *Grafo Gêmeo*, daqui em diante chamado de *Twin,* é um grafo *2-GC* de tamanho mínimo em relação ao número de enlaces, de forma que cada *Twin* tem ordem $n \geq 4$, e tamanho $m=2n-4$ [39]. Os *Twin* se beneficiam do processo de construção baseado no conceitos de pares gêmeos de nós. Por exemplo, por meio de um par gêmeo, um nó pode ser sempre adicionado a um *Twin*, formando outro. De forma geral, este processo pode ser realizado de diferentes maneiras, uma vez que um *Twin* possui vários pares gêmeos. Portanto, para cada n, existe uma família de grafos *Twin* [39]. Além disso, dois *Twin*

podem ser sempre unidos para forma um terceiro e este processo de união requer apenas 4 enlaces adicionais. Ambos os processos de construção serão apresentados em mais detalhes nas subseções 3.1.1 e 3.1.2.

A Figura 3.1(a) ilustra um *Twin* de ordem *8*, onde se pode verificar que ele possui *12* enlaces, *4* pares gêmeos [*(u1,v1)*, *(u2,v2)*, *(u3,v3)*, e *(u4,v4)*], e no mínimo *2* geodésicas disjuntas por nós entre cada par de nós não adjacentes. O *Hipercubo* mostrado na Figura 3.1(b) também possui *8* nós, *12* enlaces e no mínimo *2* geodésicas disjuntas por nós entre cada par de nós não adjacentes. Porém, ele não possui nenhum par gêmeo. Portanto, ele não pode se beneficiar dos processos de construção dos *Twin*.

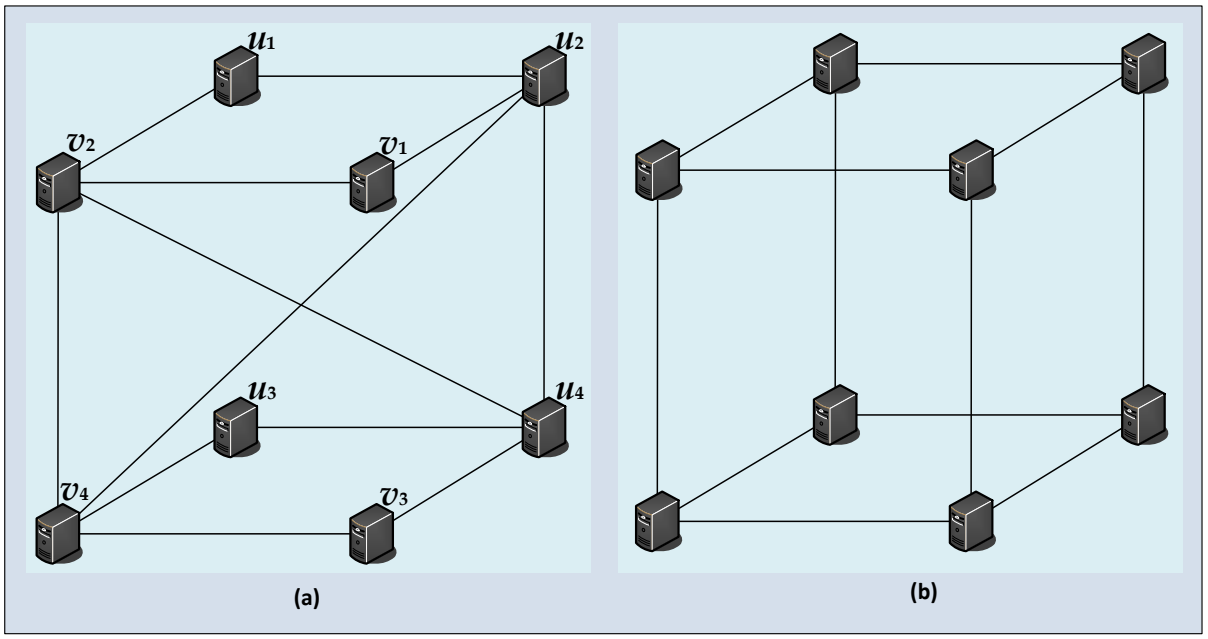


Figura 3.1: Ilustração de uma topologia centrado em servidores: (a) *Twin*, e (b) *Hipercubo*, ambos com *8* nós e *12* enlaces.

3.1.1 Processo de crescimento dos Twin

Os *Twin* podem ser definidos recursivamente da seguinte forma [39]: o ciclo de ordem *4* é um *Twin*. Se *G* é um *Twin* de ordem n, e *(u,v)* é um par gêmeo em *G*, então a adição de um novo nó *v′* usando os enlaces *uv′* e *vv′* geram um *Twin G′* de ordem *n+1*. Ademais, *(u,v)* permanece um par gêmeo em *G′*. Portanto, o processo de crescimento de um *Twin* consiste em:

1. Identificar o par gêmeo *(u,v)* em *G*;
2. Construir *G′*, onde $V(G')=V(G)\cup\{v'\}$ e $E(G')=E(G)\cup\{uv', vv'\}$.

As linhas pontilhadas da Figura 3.2 representam duas possíveis maneiras (não-isomorfas) para aumentar a topologia *Twin* de ordem *8* mostrada na Figura 3.1(a), a fim de se obter um *Twin* de ordem *9*, pela inserção de um novo nó v' representado pelo servidor branco.

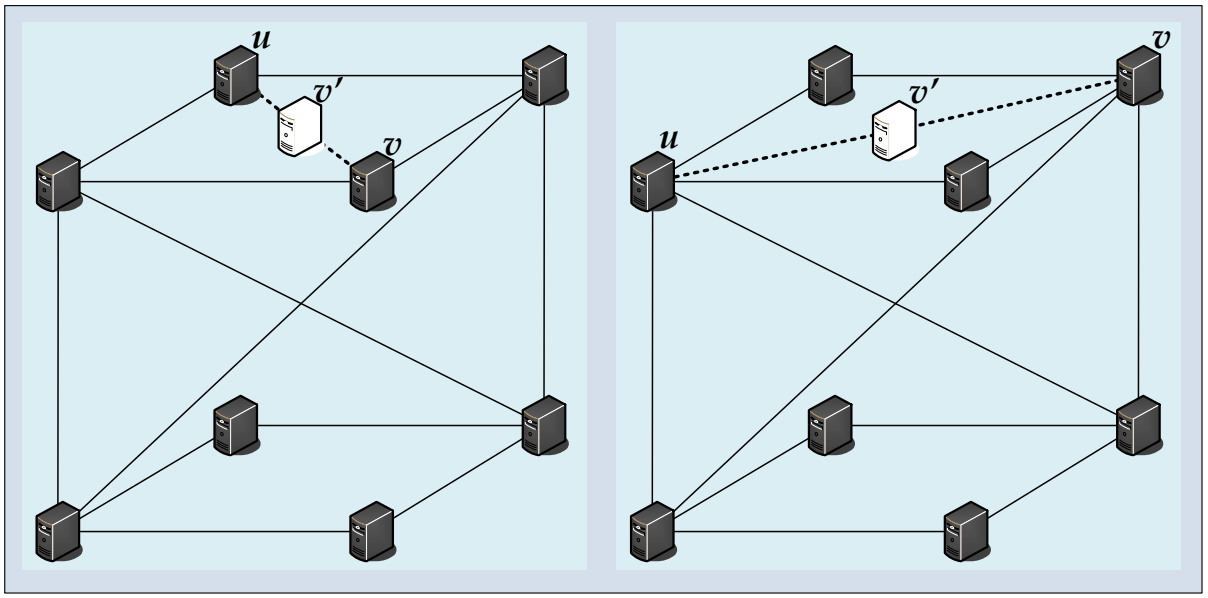


Figura 3.2: Processo de crescimento dos *Twin*. As linhas pontilhadas representam dois possíveis modos (não isomorfas) para o crescimento da topologia *Twin* de ordem 8 para construir um *Twin* de ordem 9.

3.1.2 Processo de união dos Twin

O processo de união entre dois *Twin* G_1 e G_2 consiste em:

1. Identificar o par gêmeo (u_1,v_1) em G_1 e o par gêmeo (u_2,v_2) em G_2;
2. Construir $G=G(V,E)$, onde $V=V(G_1) \cup V(G_2)$ e $E=E(G_1) \cup E(G_2) \cup \{u_1u_2, u_1v_2, u_2v_1, v_1v_2\}$.

Duas maneiras de unir dois *Twin* da Figura 3.1(a), são ilustradas na Figura 3.3(a) e na Figura 3.3(b). Os servidores brancos representam os pares gêmeos selecionados e as linhas pontilhadas representam os 4 novos enlaces. Se o diâmetro é a principal preocupação do projeto, pode-se unir os *Twin* como mostrado na Figura 3.3(a). Por outro lado, se a principal preocupação for com grau máximo dos nós, pode-se escolher a forma apresentada na Figura 3.3(b). Naturalmente, os novos *Twin* gerados possuem 16 nós e 28 enlaces. Por sua vez, a Figura 3.3(c) representa o processo padrão de união de dois *Hipercubos* da Figura 3.1(a), que necessita de *8* novos enlaces. Observe que não é possível utilizar o processo de união do *Twin*, porque os *Hipercubos* não possuem pares gêmeos. Note ainda, que o resultado do

processo também resultou em um *Hipercubo* com *16* nós, porém utilizando *32* enlaces. Isso mostra a vantagem dos grafos *Twin* sobre os *Hipercubos* com relação ao custo, que será investigado mais adiante na seção 3.3.1. Por fim é importante enfatizar que os *Twin* permitem diversas formas de união, tornando mais flexível o projeto da topologia.

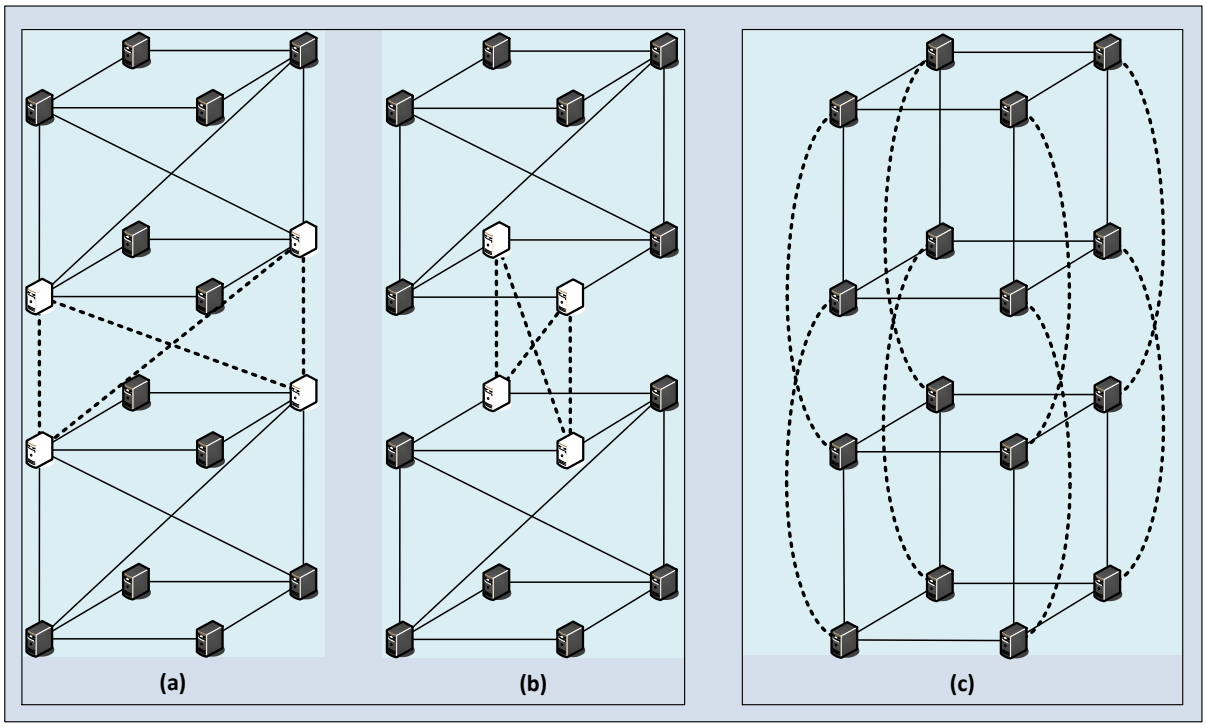


Figura 3.3: Processo de união dos *Twin* e do *Hipercubo*: (a) processo de união de *Twin* com restrição de diâmetro; (b) processo de união com restrição do grau máximo dos nós; (c) união de dos *Hipercubos* pelo processo padrão.

3.2 Projeto de topologias de redes de data center

Esta seção, apresenta uma definição particular dos seguintes aspectos relativos às topologias de rede de *data center*: custo, escalabilidade, tolerância da falha, resiliência e desempenho.

3.2.1 Custo

O custo de uma rede de *data center* varia de acordo com os equipamentos de *hardware* utilizados para encaminhar ou processar o tráfego da rede. Assim, o custo das interfaces de rede e dos cabos estão presentes em todas as arquiteturas: centradas em servidores, centradas em rede e híbridas. Entretanto, conforme apresentado na seção 2.1.2, a arquitetura centrada em servidor elimina os equipamentos de rede (*switches*, roteadores e balanceadores de carga) , multiplexando os núcleos da CPU entre as aplicações e as funções de rede [40]. Com isso,

reduz do custo total de um *data center*, tornando assim, uma forma de interconexão muito atrativa financeiramente [27]. Por fim, neste capítulo foi investigado o uso de uma topologia baseada em grafos gêmeos para redes de *data center* centrado em servidores, entretanto, a topologia baseada em *Grafos Gêmeos* poderia ser aplicada, também, para arquiteturas centradas em rede ou arquiteturas híbridas.

Para comparar o custo para diferentes arquiteturas, foi considerado que todas as topologias utilizam servidores de prateleira do mesmo modelo.

3.2.2 Escalabilidade

Ao longo do tempo, podem ser necessárias algumas expansões incrementais na rede do *data center* para acomodar, por exemplo, demandas de novos usuários ou novas aplicações que demandem muita largura de banda. Em geral, a expansão incremental em *data center* que utilizam equipamento de prateleira é realizada pelo modelo *scale out*, ou seja, pela adição de novos servidores na rede. Contudo, as topologias *Fat-Tree*, *DCell*, *BCube* e *Hipercubo* não permitem a inclusão de um número arbitrário de servidores. De fato, todas estas topologias podem alcançar um grande número de servidores, porém sua expansão incremental requer a adição de tantos servidores quanto necessários para preencher a topologia, mesmo quando apenas alguns novos servidores seriam suficientes.

Topologias de redes baseadas em *Grafos Gêmeos* eliminam o problema de expansão incremental, pois seus processos de crescimento ou união permitem adicionar quaisquer números de novos servidores. Assim, quando um *data center* precisa ser expandido lentamente, pode-se usar o processo de crescimento para adicionar o número exato de servidores necessários. No caso de uma expansão de grande volume, pode-se utilizar o método de *data center* modulares, isto é, os servidores podem ser arranjados em *containers*, e o processo de união pode ser usado para conectá-los.

3.2.3 Tolerância a falha, resiliência e desempenho

A fim de prover um serviço ininterrupto com a melhor desempenho, a rede de *data center* deve ser tolerante a falha e resiliente, ou seja, ela deve permanecer conexa e manter o seu desempenho após a falha. A tolerância da falha é uma característica relativa ao número de caminhos disjuntos entre os nós que a topologia oferece para cada par de nós, enquanto a resiliência diz respeito sobre a qualidade desses caminhos. As topologias *DCell*, *BCube*,

Hipercubo e *Twin* provêm, no mínimo, dois caminhos disjuntos para cada par de nós. Por outro lado, a topologia *Fat-Tree* apresenta baixo nível de tolerância a falha, uma vez que cada servidor é conectado com a rede por um único enlace com o *switch* de acesso. No entanto, o desempenho para estas topologias é afetado de forma diferente após uma falha simples, como será apresentado na próxima seção.

Neste trabalho, o desempenho da rede é medido pelo *tempo de completude do fluxo* [48] para finalizar todas as conexões. A resiliência é medida pelo aumento do tempo de completude do fluxo sobre condições de falhas, pela *penalidade na contagem de saltos*, e aumento do diâmetro da rede. A penalidade na contagem de saltos é definida como a diferença entre o número de saltos do caminho de trabalho (menor caminho) e o número de saltos do caminho alternativo.

3.3 Comparativo entre as topologias

Nesta seção, a topologia *Twin* é comparada com as topologias *Fat-Tree*, *DCell*, *BCube* e *Hipercubo*, em relação aos aspectos discutidos na seção anterior. A topologia *Jellyfish* não foi utilizada no comparativo, pois devido a sua natureza aleatória de interconexão, seria necessário a criação de uma população de topologias para uma analise representativa.

3.3.1 Custo

Para comparar o custo entre as topologias *Fat-Tree*, *DCell*, *BCube*, *Hipercubo* e *Twin* com n servidores, foi utilizado o número de enlaces necessários para interconectar todos os servidores. Considerando que a topologia *Fat-Tree* utiliza *switches* de prateleira, elas podem ser, tipicamente, construídas para $n = 1024$, 3456, 8192, 27648, ou 65536 servidores. Assim estes valores foram utilizados como parâmetros para todas as topologias.

O número de enlaces para a topologia *Twin* é calculado pela equação $2n - 4$, e para as outras topologias estudadas, seus números de enlaces são calculados com base em seus parâmetros. Assim para a *Fat-Tree* foi utilizado o número de portas do *switch,* para *DCell* e *BCube* foram utilizados o nível da recursão e o número de portas dos *mini-switches* e para o caso do *Hipercubo* foi utilizado o grau dos nós.

A Figura 3.4 ilustra o número de enlaces em relação ao número de servidores para as topologias comparadas. Observe que o *Twin* utiliza o número mínimo de enlaces, seguido

pelo *DCell*. As topologias *Fat-Tree* e *BCube* apresentam suas curvas sobrepostas, pois elas utilizam praticamente o mesmo número de enlaces, enquanto o *Hipercubo* apresente o maior número de enlaces. Analisando os dados para *65356* servidores, observa-se que o número do enlaces da topologia *Twin* é 20% menor, quando comparado ao *DCell*, e 75% menor quando comparado ao *Hipercubo*.

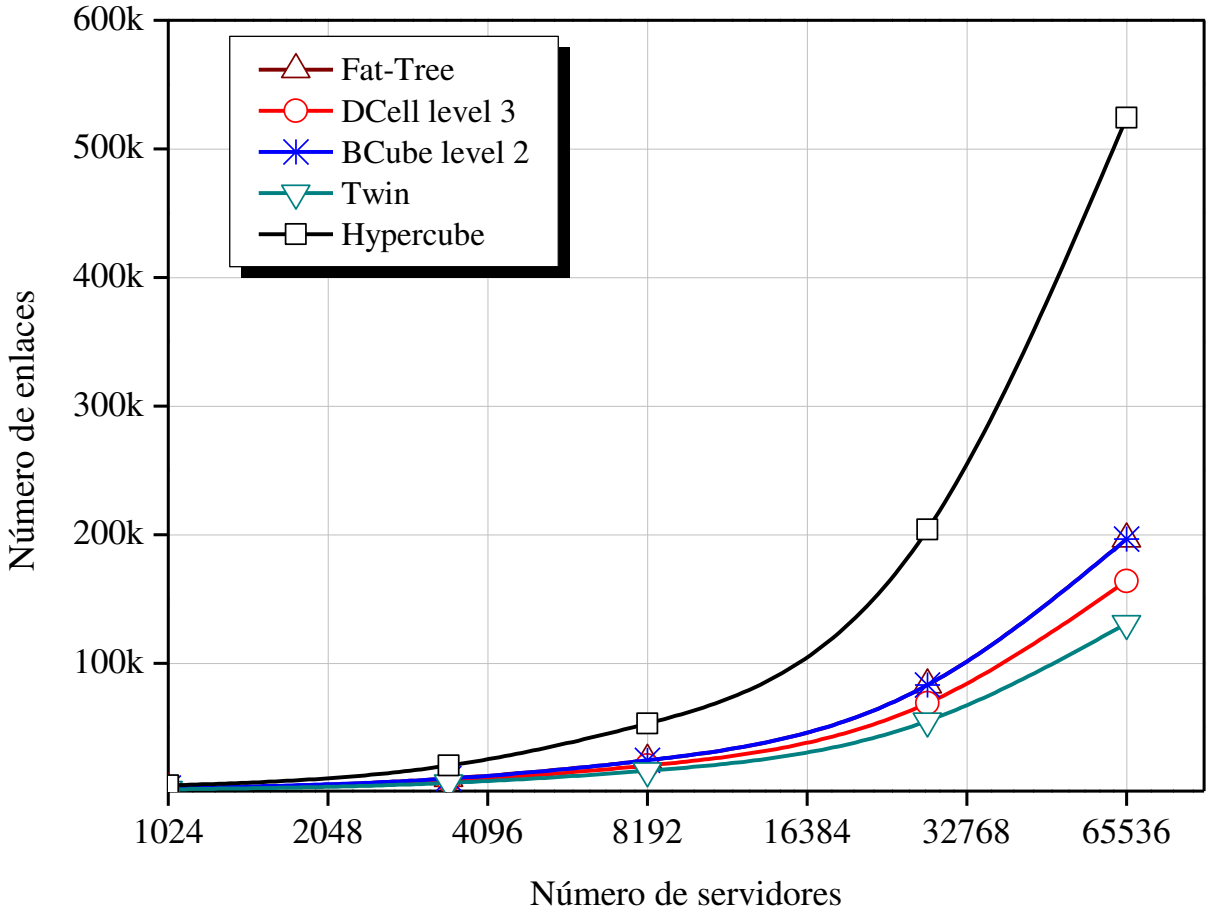


Figura 3.4: Número de enlaces em relação ao número de servidores para as topologias *Fat-Tree*, *DCell*, *BCube*, *Hipercubo* e *Twin*.

3.3.2 Escalabilidade

A escalabilidade de um *data center* baseado em equipamentos de prateleira pode ser analisada pelos seus números típicos de servidores, representados *n* na Tabela 3.1. Para topologia *Fat-Tree n* depende somente do número de portas dos *switches*; para *DCell* e *BCube n* depende do nível da recursão e do número de portas dos *mini-switches*; no caso do *Hipercubo n* depende do grau do nó representado por δ. Assim, um *data center* com *10K* servidores: (*i*) usaria um *Fat-Tree* com *switches* de *48* portas, que proveria capacidade de interconexão para *24648* servidores; (*ii*) para as topologias *DCell* e *BCube* poderia se

encontrar uma combinação entre o nível e o número de portas dos *switches* que fosse próximo de *10K*; (*iii*) para o *Hipercubo* o grau do nó deveria ser *14*, o que implicaria em uma topologia com *16384* servidores.

Outro problema do *Fat-Tree*, *DCell* e *BCube* surge quando um novo servidor precisa ser adicionado em uma topologia de rede totalmente ocupada. Por exemplo, se *1000* servidores extras precisassem ser acomodados na topologia *Fat-Tree* com *switches* de *16* portas, todos os *switches* deveriam ser substituídos por novos *switches* de *24* portas, por exemplo, e então *2432* novas portas ficariam em espera. O *DCell* e o *BCube* apresentam problema similares. Por sua vez, no *Hipercubo* o número de servidores interconectados deve ser duplicado. Apesar da possibilidade de construirmos topologias incompletas, não há garantias que uma topologia parcial irá apresentar a mesma resiliência e desempenho em relação a uma implementação completa da topologia.

Conforme já foi destacado, utilizando a topologia *Twin* um *data center* pode-se acomodar qualquer quantidade de servidores, sem sobre-provisionamento de recursos.

Tabela 3.1: Número típico de servidores para *Fat-Tree*, *DCell*, *BCube*, *Hipercubo* e *Twin*, utilizando equipamentos de prateleira.

Fat-Tree	***n***	***DCell***	***n***	***BCube***	***n***	***Hipercubo***	***n***
sw 16 p.	1024	Nível 2, *sw* 6 p.	1806	Nível 2, *sw* 16 p.	4096	$\delta = 12$	4096
sw 24 p.	3456	Nível 2, *sw* 8 p.	5256	Nível 2, *sw* 24 p.	13824	$\delta = 13$	8192
sw 32 p.	8192	Nível 2, *sw* 10 p.	12210	Nível 2, *sw* 32 p.	32768	$\delta = 14$	16384
sw 48 p.	24648	Nível 2, *sw* 12 p.	24492	Nível 3, *sw* 16 p.	65536	$\delta = 15$	32768
sw 64 p.	65536	Nível 3, *sw* 4 p.	176820	Nível 3, *sw* 24 p.	331776	$\delta = 16$	65536

3.3.3 Tolerância a falha, resiliência e desempenho

Para comparar o desempenho entre as topologias *Fat-Tree*, *DCell*, *BCube*, *Hipercubo* e *Twin*, um modelo para cada topologia foi construído usando o ambiente de emulação de rede *Mininet* [49]. Adicionalmente, um *controlador SDN* foi implementado, permitindo a avaliação do desempenho dessas topologias com base em dois métodos de roteamento: (*i*) o *OSPF* (*Open Shortest Path First*) que é um protocolo aberto definido pela IEFT (*Internet Engineering Task Force*), documentado na RFC 2328 [50] e que utiliza o menor caminho entre dois nós; (*ii*) e o *ECMP* (*Equal Cost Multipath Protocolo*) que é um protocolo, também, definido pela IEFT, documentado nas RFC 2991 [51] e RFC 2992 [52], e permite que fluxos diferentes entre dois nós sejam encaminhados por caminhos alternativos de mesmo custo. Os mesmos mecanismos de roteamento foram utilizados em todas as topologias, para garantir um

comparativo mais justo, dado que o principal objetivo é avaliar os aspectos da topologia física. Finalmente, foi adotado um cenário de tráfego uniforme e imparcial para todas as topologias, ou seja, uma comunicação de todos para todos.

Os modelos foram testados utilizando uma topologia de *16* servidores, e o tráfego para cada par de nós foi equivalente a um fluxo de *500 MB*, envidados na taxa máxima das interfaces, ou seja, *1 Gbps*. Para propósitos de comparação, um modelo *Full Mesh* também foi testado com os mesmo parâmetros. No caso do *DCell*, a topologia foi construída com *20* servidores, assim foi utilizando um fluxo de *316 MB* para alcançar o mesmo volume de tráfego trocado nas outras topologias, de *120 GB*. Os fluxos individuais foram gerados pela ferramenta *Iferf* [53]. Neste senário, as topologias *Fat-Tree*, *DCell*, *BCube*, *Hipercubo* e *Twin* continham *36*, *30*, *32*, *33* e *28* enlaces, respectivamente.

Para *16* servidores, existe uma família de *Twin* com *182* membros [33], os quais forma todos testados, e dois deles, chamados *Twin-1* e *Twin-2*, foram selecionados para apresentar os resultados. O *Twin-1* foi selecionado, pois é o membro da família que minimiza do diâmetro da rede, em contrapartida maximiza do grau de alguns nós, enquanto o *Twin-2* foi selecionado por ter o mesmo diâmetro do *BCube* e do *Hipercubo*.

Primeiramente será discutido o desempenho de cada topologia em um cenário sem falhas. A primeira e quarta barra, de cada topologia, na Figura 3.5 mostram o tempo de completude do fluxo utilizando *OSPF* e *ECMP*, respectivamente. A topologia *Full Mesh* gastou *70s* para finalizar os *240* fluxos, como mostrado pela linha horizontal pontilhada na Figura 3.5. O melhor desempenho utilizando o OSFP foi de *73s,* enquanto para o *ECMP* foi de *72s*. Paras as topologias *Twin*, o tempo gasto variou de acordo como membro da família escolhido. O *Twin-1* gastou *73s* em ambos os métodos. Observe que este é um tempo muito próximo do tempo gasto pela topologia *Full Mesh* e que não pode ser melhorado pelo *ECMP*, devido ao encaminhamento de fluxos que ocorre entre os servidores não adjacentes. Por outro lado o *ECMP* obteve um desempenho melhor para as topologias *Twin-2*, *Fat-Tree* e *Hipercubo*, dado que estas topologias possuem, no mínimo, duas geodésicas para a maioria dos pares de nós. No entanto, o *DCell* e o *BCube* possuem somente uma geodésica para cada par de nós, com isso pouco ou nenhuma melhora pode ser observada na Figura 3.5 para estas topologias utilizando o *ECMP*. É importante notar que, mesmo para este cenário reduzido, a topologias *Twin* apresenta uma redução de até *28%* no número de enlaces, com um desempenho sobre o *ECMP* muito próximo das outras topologias.

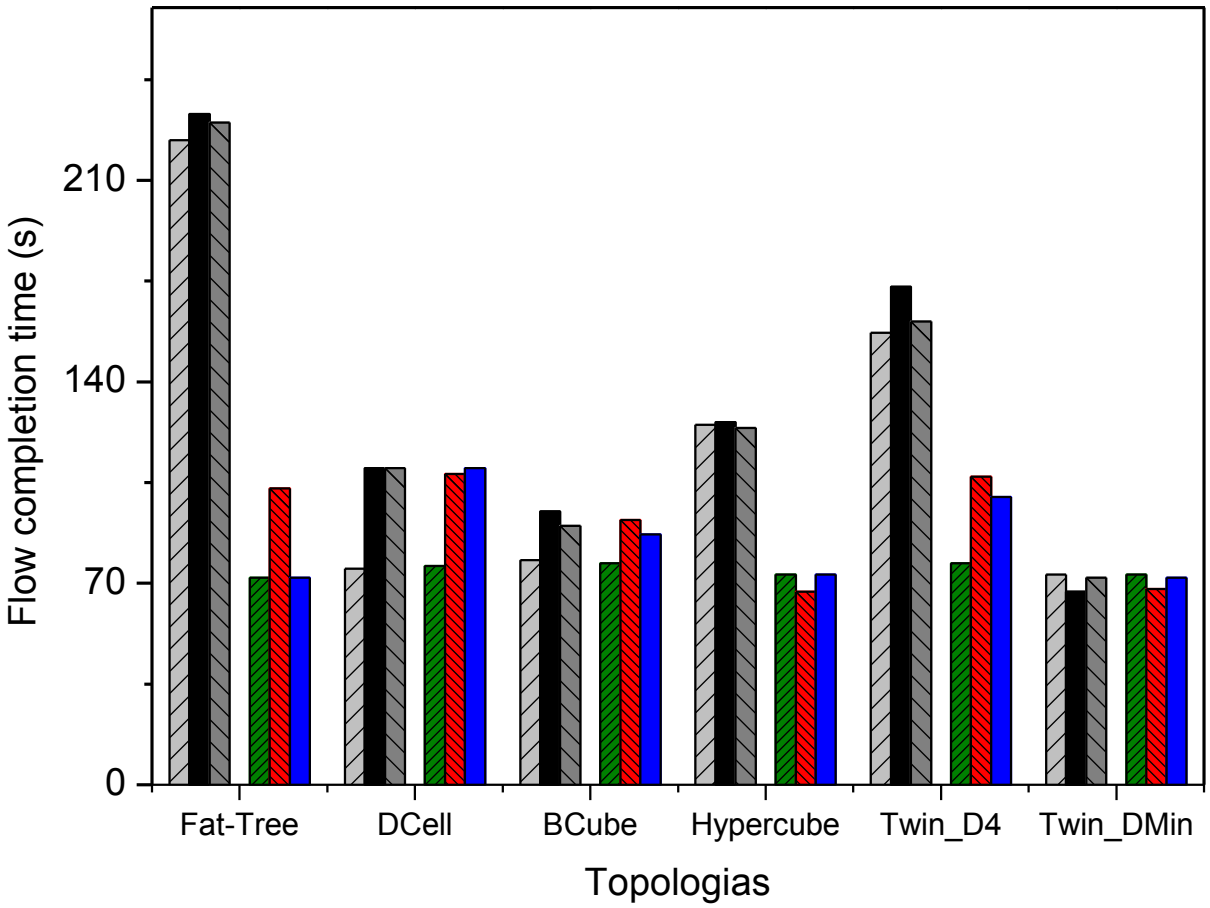


Figura 3.5: Tempo de completude do fluxo para finalizar todas as conexões em cada topologia, sobre condições de operação: normal, com falha de nó e falha de enlace, usando *OSPF* e *ECMP*.

Para analisar o mesmo cenário, porém na presença de falhas, foi introduzido tanto falhas de nós de grau máximo, como falhas de enlaces aleatoriamente escolhidos. Conforme afirmado anteriormente, todas as topologias 2-conexos tem o mesmo nível de tolerância a falha, isto é, suportam, no mínimo, uma falha de nó ou uma falha de enlace mantendo a topologia conectada. Entretanto, elas apresentam diferentes níveis de resiliência. Nos experimentos desta seção, a resiliência está sendo medida pelo aumento do tempo de completude do fluxo sobre condições de falhas, pela penalidade na contagem de saltos, e aumento do diâmetro da rede.

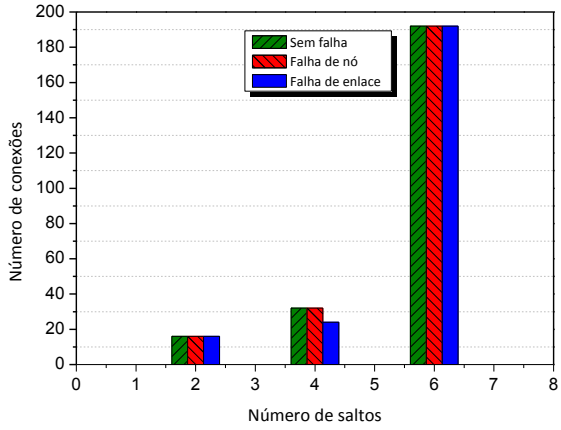
Na Figura 3.5, a segunda e a quinta barra, de cada topologia, mostram o tempo de completude do fluxo após uma falha de nó, usando o *OSPF* e *ECMP*, respectivamente; e os resultados correspondentes a uma falha de enlace são mostrados na terceira e na sexta barra, de cada topologia. Quando comparados os tempos de completude do fluxo dessas topologias, pode-se observar que *Twin-1* apresenta o melhor desempenho entre todos, tanto para falha de

enlace quanto para falha de nó, usando o *OSPF*. Para o caso do *ECMP*, as topologias *Twin-1*, *Fat-Tree*, e *Hipercubo* apresentam praticamente o mesmo desempenho na presença de falha de enlace. No entanto, para falha de nó o *Twin-1* e o *Hipercubo* apresentam um melhor desempenho que a *Fat-Tree*. Finalmente, é imponte observar que, nas topologias sem *switches*, tais como: *Twin* e *Hipercubo*, a falha de nó reduz o trafego total, que por sua vez reduz o tempo de completude do fluxo.

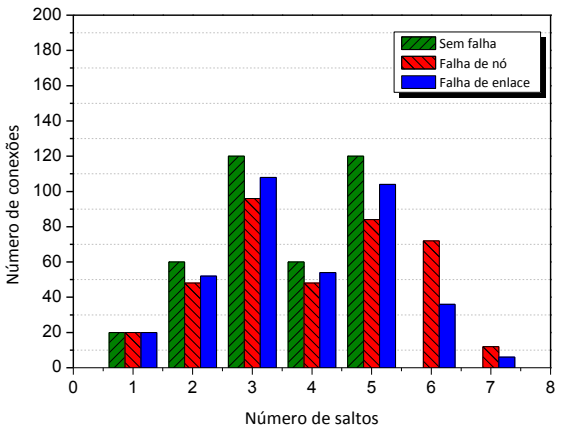
A Figura 3.6 ilustra a distribuição de fluxo em relação ao número de saltos, de onde foram analisados a penalidade na contagem de saltos e o diâmetro da rede. As triplas representam (d, d^n, d^l), onde d é o diâmetro da rede, d^n é o diâmetro após uma falha no nó de grau máximo, e d^l é o diâmetro após falha em um enlace aleatório. Observe que, para as topologias que possuem, no mínimo, duas geodésicas para a maior parte dos pares de nós, o diâmetro da rede não altera substancialmente após uma falha. Entretanto, para *BCube* e *DCell* o diâmetro sofre um aumento de duas unidades.

A distribuição de fluxos em relação ao número de saltos é praticamente a mesma para as topologias *Twin*, mesmo nas situações de falhas, ao passo que, para *DCell* e *BCube*, o número de fluxos usando caminhos mais longos é aumentado, consequentemente o atraso médio da rede (*delay*) é maior. As conexões usando *6* ou *7* saltos na Figura 3.6(*b*), e as conexões usando *6* saltos na Figura 3.6(*c*) ilustram o efeito da penalidade na contagem de saltos na resiliência. A topologia *Fat-Tree* também apresenta uma degradação de desempenho graciosa, porém suas distribuições de fluxos mostram a maioria das conexões usando caminhos longos. Finalmente, em termos de distribuição de fluxos, diâmetro, e penalidade na contagem de saltos, os resultados para as topologias *Hipercubo* e *Twin-2* são similares.

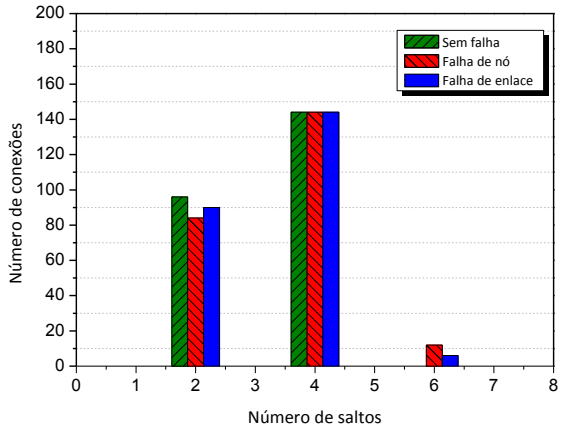
Adicionalmente, observe que a falha de nó, por definição, nunca leva ao aumento de diâmetro nas topologias *Twin*. Mesmo no pior caso, onde o servidor de maior número de conexões falha, todos os demais servidores podem se comunicar usando os menores caminhos. Estes resultados asseguram tolerância da falha, resiliência e desempenho das topologias de rede de *data center Twin*.



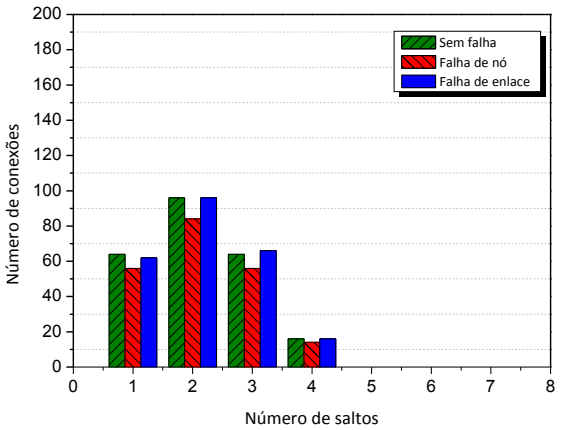
(a) *Fat-Tree*; tripla de diâmetro: (6,6,6)



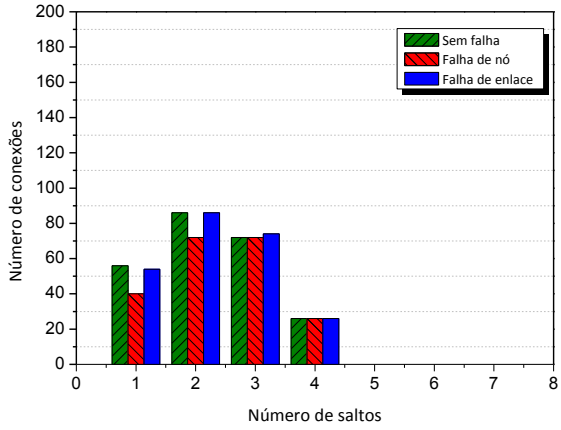
(b) *DCell*; tripla de diâmetro: (5,7,7)



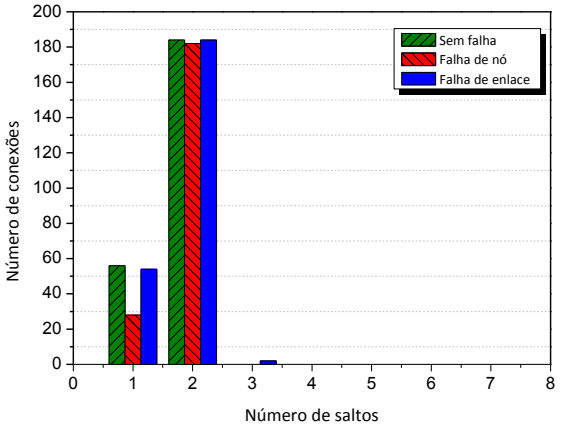
(c) *BCube*; tripla de diâmetro: (4,6,6)



(d) Hipercube; tripla de diâmetro: (4,4,4)



(e) *Twin-2*; tripla de diâmetro: (4,4,4)



(f) *Twin-1*; tripla de diâmetro: (2,2,3)

Figura 3.6: Distribuição de fluxos, em relação ao número de saltos. A tripla representada por (d, d^n, d^l), onde d é o diâmetro da rede, d^n é o diâmetro após uma falha no nó de grau máximo, e d^l é o diâmetro após falha em um enlace aleatório.

3.4 Implicações prática do uso da topologia *Twin*

Esta seção destaca algumas implicações práticas do uso de topologias *Twin* em redes de *data center*, em especial, em redes de grande escala.

3.4.1 Como escolher uma topologia *Twin*

Para o projeto de uma *data center Twin* com n servidores, existe uma família de topologias *Twin* de ordem n da qual se pode escolher uma topologia de acordo com os requisitos particulares do projeto. Para facilitar a escolha da topologia, esta família pode ser reduzida usando uma sequência de filtros, tais como: grau máximo dos nós, diâmetro, e número médio de saltos. Ainda em uma família *Twin* de ordem n, é possível encontrar topologias com bons compromissos entre o diâmetro e o grau máximo do nó, como ilustrado na Figura 3.3(a) para uma rede pequena. Para cada n, existe uma topologia *Twin* de diâmetro *2* [39]; e existe no mínimo uma topologia *Twin* com diâmetro na faixa de *2* a $\lfloor n/2 \rfloor$, para cada $n \leq 17$ [33].

3.4.2 Como identificar uma topologia *Twin*

As topologias *Twin* são escaláveis, devido ao seu processo de crescimento e união, porém ambos os processos deve iniciar sobre uma topologia *Twin*. Felizmente, é computacionalmente fácil identifica se uma topologia é *Twin*, usando o algoritmo de reconhecimento criado por Chang & Ho [54], o problema de decidir se uma topologia de ordem n e tamanho $m = 2n - 4$ é um *Twin* pode ser resolvido com complexidade dada por $O(mn) = O(n^2)$. Assim, uma aplicação proposta para este algoritmo seria reconectar uma dada topologia com n nós em um grafo *Twin*, simplesmente rearranjando suas interconexões, e/ou aumentando ou reduzindo seu número de enlaces.

3.4.3 Como construir uma topologias *Twin* de grande escala

Como afirmado anteriormente, é simples construir uma topologia *Twin* de ordem n, nó por nó, a partir do clico de ordem 4; ou pela união de duas topologias *Twin* de ordem $n1$ e $n2$, onde $n = n1 + n2$. Assim, dado n, poder-se-ia construir todas as topologias *Twin* de ordem n, e então escolher a topologia que melhor se adequada aos requisitos do projeto. No entanto, um processo de construção recursiva nó a nó não é recomendado para grandes

valores de n, por exemplo, para centenas ou milhares de nós. Isso porque o número de topologias aumenta rapidamente com n, por exemplo, para *17* nós existem *2,45E+26* grafos 2-conexos dentre os quais apenas *310* são *Twin* [55]. Ademais, usando o processo de merge, não há garantias de se obter a melhor topologia *Twin*. Portanto, a aplicação prática do *Twin* em grandes redes de *data center* depende do desenvolvimento de um algoritmo eficiente para construir grandes topologias com base nos requisitos do projeto.

3.4.4 Qual a complexidade em cabear os servidores em uma topologia *Twin*

A topologia *Twin* utiliza o número mínimo de enlaces de uma topologia 2-conexos. Isso significa uma redução substancial da complexidade de interconectar os nós da topologia. Processos minimamente invasivos são necessários para crescimento ou união de topologias *Twin*, conforme ilustrado na Figura 3.2 e na Figura 3.3, respectivamente. Ademais, todos os grafos *Twin* são planares [56], logo existe um arranjo espacial do nós, tal que, nenhum enlace cruze com outros. Todas estas características favorecem a acomodação do cabeamento em um modo mais compacto e até mesmo um arranjo em camadas poderia ser investigado, tirando proveito da planaridade dos *Twin*.

3.4.5 Como lidar com nós de alto grau em uma topologia *Twin*

Em um *data center*, os servidores usualmente possuem um número limitado de *slots* para alocar cartões de interfaces de rede. Isso representaria uma dificuldade para acomodar nós com alto grau de conexões em uma arquitetura centrada em servidores. Por outro lado, modelos de interconexões híbridos como *DCell* e *BCube* fazem uso de *switches* para expandir o grau dos servidores como apresentado na Tabela 3.1. Portanto, esta mesma estratégia poderia ser empregada para viabilizar nós com alto grau de conectividade em uma arquitetura híbrida baseada na topologia *Twin*. Em contaste com *DCell* e *BCube* em que todos os nós tem o mesmo grau, na topologia *Twin* somente alguns nós possuem grau máximo, como ilustrado na Figura 3.3(a).

3.5 Análise comparativa para aplicação em data centers

Apesar dos *Twin* apresentarem melhores características topológicas quando comparados ao *Fat-Tree*, *DCell*, *BCube* e *Hipercubo*, algumas implicações práticas inviabilizam o uso dessa topologia em redes de grande escala, principalmente: (*i*) a

dificuldade de encontrar uma topologia que atenda aos requisitos do projeto, devido a complexidade computacional de gerar todas as possiblidades para então escolher a que melhor se adequa; (*ii*) do melhor do nosso conhecimento, o fato de não existir um algoritmo de encaminhamento/roteamento específico para esta topologia, que possa atender as hipótese 4 deste trabalho, ou seja, a integração da comutação óptica com o processo de roteamento/encaminhamento; (*iii*) a dificuldade de espalhar dispositivos de chaveamento para reconfigurar os enlaces, e ao mesmo tempo, manter a topologia *Twin*.

Por sua vez, o *Hipercubo* apresentou resultados muito interessantes, empatando com o *Twin* no critério de penalidade na contagem de saltos, consequentemente na manutenção do diâmetro após a falha. As várias geodésicas entre seus pares de nós favorecem o espalhamento dos fluxos, resultando em excelente desempenho ao utilizar o *ECMP*. Diferentemente do *DCell* e *BCube*, que se apoiam sobre *mini-switches* (COTS) que não foram projetados para ambiente de alta disponibilidade, no *Hipercubo* o encaminhamento do tráfego da rede é realizado exclusivamente por servidores, que mesmo sendo COTS apresentam alta disponibilidade. A regularidade da topologia do *Hipercubo* favorece a "*opticalização*" da rede de forma distribuída e gradual. Para suportar as alterações topológicas promovidas pela comutação óptica, o *Hipercubo* conta com um algoritmo de roteamento nativo, que utilizada a operação XOR para escolher o menor caminho, apenas com o conhecimento local do nó, isto é, sua localização no espaço de endereçamento. Outra característica que habilita a "*opticalização*", é a disponibilidade de múltiplos caminhos entre os pares de nós da topologia, fornecendo rotas alternativas.

Por fim, para o restante desta tese é prioritário que a topologia para rede de *data center* centrado em servidores possua uma camada de encaminhamento eficiente, tanto em nível elétrico, quanto em nível óptico. Ainda, esta topologia deve: (*i*) permitir uma distribuição parcial de dispositivos ópticos pela na rede; (*ii*) não alterar o grau dos nós físicos da rede; e (*iii*) possuir um algoritmo de roteamento tolerante a comutação óptica. Portanto, este trabalho optou pela utilização do *Hipercubo* como a topologia base da arquitetura que será apresentada no próximo capítulo.

Capítulo 4 – Arquitetura TRIIIAD

A arquitetura TRIIIAD vai muito além da concepção de uma nova rede de *data center*. De fato ela é uma proposta para automatizar ambientes de *data centers,* desde a reconfiguração dos enlaces físicos da rede, até a orquestração de máquinas e redes virtuais. Para lidar com aspectos tão amplos, a arquitetura foi dividida em três camadas sobrepostas, alinhas por um plano vertical de controle gerência e orquestração. Ademais, um dos fatores chaves para alcançar os objetivos almejados, foi a opção por uma rede de *data center* centrado em servidores como base da arquitetura. Isso possibilitou a implementação de uma *função virtual de rede (NFV) controlada por software (SDN)*, que além de otimizar o encaminhamento dos dados entre as VMs, conferiu flexibilidade e programabilidade ao projeto. Ao longo deste capítulo, são discutidos em detalhes os conceitos e tecnologias que foram empregados nesta arquitetura, iniciando por suas camadas.

4.1 As camadas da arquitetura TRIIIAD

O nome TRIIIAD é um acrônimo para ***TRI****ple-Layered* ***I****ntelligent and* ***I****ntegrated* ***A****rchitecture for* ***D****atacenters*, *arquitetura para data center em três camadas, inteligente e integrada*. Em parte este nome foi escolhido em função de a arquitetura ser composta por três camadas sobrepostas, integradas por um plano vertical, conforme ilustrado na Figura 4.1. De forma sucinta, a camada superior, ou camada de virtualização, representa as redes virtuais e máquinas virtuais do *data center*. A camada intermediária, ou camada de encaminhamento, representa a rede de transporte de dados. A camada híbrida reconfigurável, representa os dispositivos capazes de reconfigurar os enlaces entre os elementos físicos de rede. Além das camadas horizontais, existe um plano vertical de controle, gerência e orquestração para alinhar a funcionamento destas camadas. As subseções a seguir apresentam os detalhes

conceituais e requisitos específicos de cada camada da arquitetura e do plano vertical de controle, gerência e orquestração.

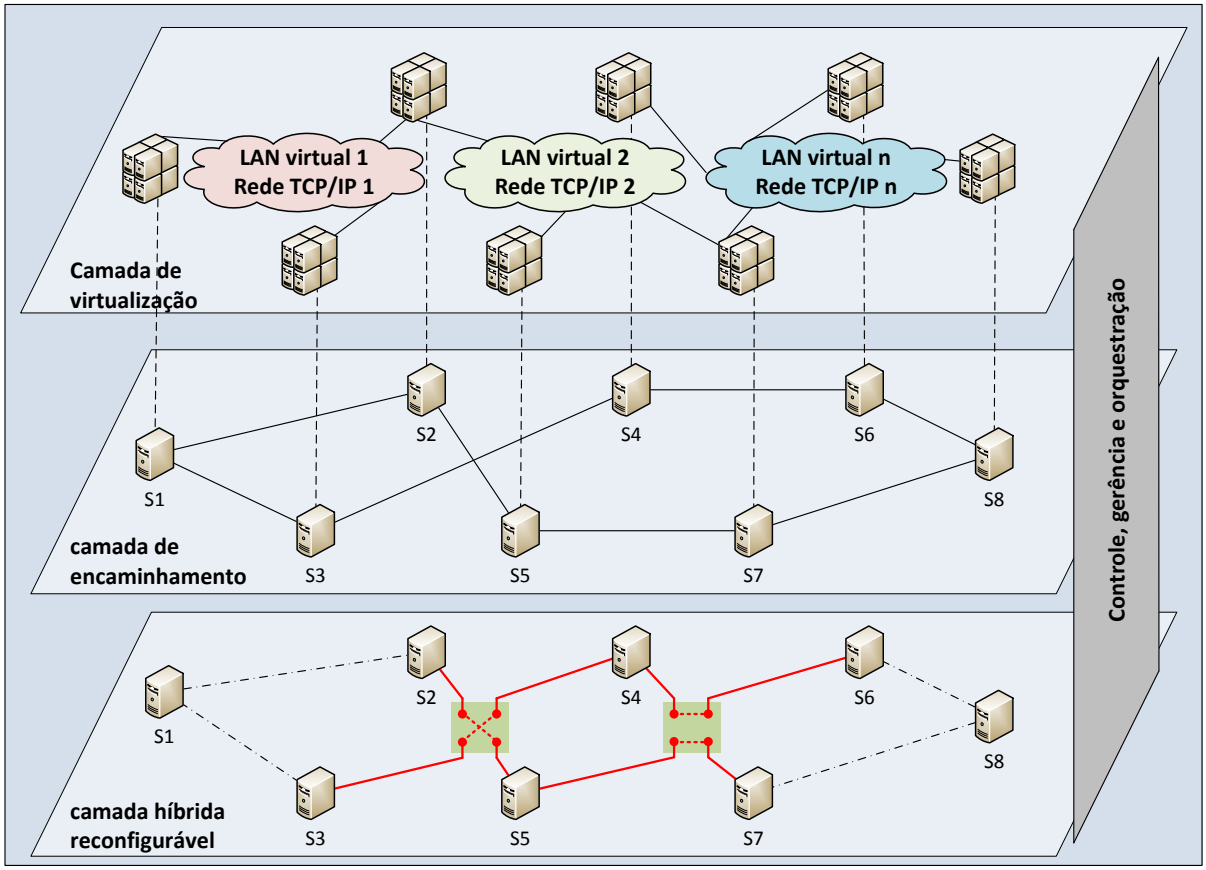


Figura 4.1: Visão geral da arquitetura TRIIIAD

4.1.1 Camada de virtualização

A camada de virtualização desempenha dois importantes papéis. O primeiro é prover o compartilhamento dos recursos físicos, o que corrobora para a redução dos custos, atendendo assim um requisito importantíssimo nos *data centers* atuais. O segundo é compatibilizar a arquitetura com as tecnologias de redes amplamente difundidas, tais como: IPv4 e *Ethernet*, entre outras. A Figura 4.1 ilustra um cenário no qual a camada de virtualização apresenta oito VMs por servidor físico e três redes virtuais, tanto em nível 2 (camada de enlace) quanto em nível 3 (camada de redes TCP/IP), nas quais as VMs podem ser alocadas. Observe ainda, que é possível que uma VM esteja alocada em uma ou mais redes virtuais.

Para configurar o cenário descrito acima, a camada de virtualização deve permitir as operações básicas de criação e destruição de redes virtuais e VMs. Porém, para tornar o ambiente virtual realmente flexível, esta camada deve prover a operação de migração de VMs, atendendo o requisito de não alterar seus endereços IP após a migração. Isso possibilita, que

em uma migração a quente, isto é, com a VM ligada, as conexões TCP pré-existentes e os estados das aplicações sejam mantidos. Assim, as migrações serão transparentes do ponto de vista dos clientes conectados a VM. Deste modo, a camada de virtualização da arquitetura TRIIIAD permite entregar produtos/serviços mais flexíveis e customizados para os clientes de um *data center*.

4.1.2 Camada de encaminhamento

A camada intermediária representa a rede de transporte de dados da nossa arquitetura. Assim, sua principal funcionalidade é fornecer um mecanismo eficiente para encaminhamento dos pacotes gerados pelas VMs. A Figura 4.1, ilustra um exemplo desta camada composta por oito servidores físicos, os quais estão conectados a seus vizinhos pelos enlaces físicos da camada de inferior. Cada servidor físico suporta um grupo de VMs que fazem parte da camada de virtualização. Neste ponto, é importante enfatizar que em *data centers* modernos, cada servidor físico pode hospedar centenas de VMs, de forma que a rede de dados deve ser capaz de lidar com centenas de milhares de VMs. Com isso, o mecanismo de encaminhamento deve evitar problemas como: buscas complexas em tabelas de roteamento, convergência de rotas, protocolos de descoberta pesados e controladores centralizados para fornecer as rotas.

Ainda, por esta camada ser o elo entre a camada de virtualização e a camada híbrida de reconfiguração, três importantes requisitos foram impostos:

1. Ser transparente para a camada superior, ou seja, as VMs podem utilizar os protocolos de rede padrão (*Ethernet* e TCP/IP) sem nenhuma adaptação.
2. Ser agnóstica sobre a criação, destruição e migração de máquinas virtuais, ou seja, o gerenciamento das VMs não deve impactar no funcionamento da camada de encaminhamento.
3. Ser agnóstica a reconfiguração da camada híbrida que a suporta, ou seja, a modificação dos enlaces físicos da rede não deve impactar no funcionamento da camada de encaminhamento.

4.1.3 Camada híbrida reconfigurável

A camada híbrida reconfigurável é composta por enlaces elétricos, enlaces ópticos e comutadores 2x2. Seu principal objetivo é minimizar o alto tráfego de trânsito, isto é, o

tráfego encaminhado pelos servidores, inerente às redes de *data centers* centrados em servidores. Para isso, os enlaces físicos são reconfigurados, de forma a criar atalhos na rede entre os nós que participam de grandes fluxos de dados. Estes atalhos são criados seguindo a filosofia de chaveamento de circuito, onde todos os fluxos são encaminhados pelo mesmo caminho físico. Consequentemente, o uso destes atalhos pode prover uma redução significativa da latência da rede como um todo.

Um exemplo da aplicabilidade deste conceito pode ser observado na camada híbrida reconfigurável da Figura 4.1, que representa os enlaces elétricos por linhas traço-ponto (pretas), os enlaces ópticos por linhas sólidas (vermelhas). Um comutador, em estado de *cruz*, conecta *S2* a *S5* e *S3* a *S4*. Enquanto um segundo comutador, em estado *barra*, conecta *S4* a *S6* e *S5* a *S7*. Porém, caso seja mais vantajoso, este cenário poderia ser reconfigurado de forma a conectar *S2* a *S4*, e *S3* a *S5*, por exemplo. Ademais, estas reconfigurações permitiriam contornar enlaces e servidores que apresentassem falhas, bem como fazer um balanceamento de carga sobre toda a rede. Para que esta camada seja viável, ela deve atender os seguintes requisitos:

1. Utilizar dispositivos de baixo custo para reconfiguração dos enlaces;
2. Possibilitar a distribuição destes dispositivos na rede;
3. Possibilitar a implantação de forma parcial e incremental;
4. Possibilitar uma reconfiguração rápida o suficiente para prevenir eventos de perca da conexão;
5. Não alterar o grau dos nós físicos da rede e;
6. Não criar laços na topologia.

4.1.4 Plano de controle, gerência e orquestração

O plano de controle, gerência e orquestração atua de forma vertical na arquitetura, alinhando o funcionamento das três camadas e mantendo-as agnósticas entre si. Assim, suas responsabilidades foram estendidas e abrangem desde tarefas de alto nível, por exemplo, ordenar a migração de VMs, até tarefas de baixo nível, por exemplo, reconfigurar os enlaces físicos da rede. Isso porque, em uma rede centrada em servidores, a orquestração das máquinas virtuais tem impacto direto no encaminhamento dos pacotes, uma vez que as tarefas de computação e as tarefas de rede competem por recursos do servidor, principalmente CPU.

4.2 Projeto da arquitetura

Esta seção discute as soluções e compromissos assumidos para o projeto das camadas, bem como apresenta os elementos necessários para realização da TRIIIAD. Para que a leitura seja linear, será discutido primeiramente o projeto da camada de encaminhamento, que apresenta conceitos necessários para explicar as demais camadas.

4.2.1 Camada de encaminhamento

Conforme pode ser observado na Figura 4.1, a camada intermediária não possui nenhum elemento tradicional de rede, tais como: *switches* ou roteadores. Assim, os servidores físicos são responsáveis tanto por hospedar as VMs, quanto por encaminhar dos pacotes na rede. Esta abordagem, conhecida como rede de *data center* centrado em servidores, conferiu flexibilidade e programabilidade necessárias para implementação de um mecanismo eficiente de encaminhamento de pacotes, sobre uma arquitetura *x86*, como uma *função virtual de rede (NFV)*, cujos parâmetros são definidos por um *controlador SDN*. Esta seção, apresenta o mecanismo de encaminhamento escolhido e discute algumas limitações das rede de *data center* centrado em servidor.

O mecanismo de encaminhamento para a *função virtual de rede* deveria ser simples, eficiente e livre dos problemas apresentados na seção 4.1.2. Assim, candidatos potenciais, tais como: MPLS e soluções puramente *OpenFlow*, se mostraram inadequados. No caso do MPLS, apesar de ser uma tecnologia consolidada e com características interessantes, o fato de necessitar do protocolo LDP (*Label Distribution Protocol*), para criar e manter a base de dados com os LSP (*Label Switched Paths*), associado à necessidade de introduzir um novo cabeçalho para rotular os pacotes (aumentando *sobrecarga* na rede), tornou o seu uso inviável neste projeto. Por sua vez, as soluções puramente *OpenFlow* podem apresentar um desempenho baixo, quanto empregadas em topologias de redes centradas em servidores, devido problemas de escalabilidade, uso de *hardware* de propósito geral, sobrecarga tanto no canal de controle como no *controlador SDN*, que foram discutidas na seção 2.2.

A solução escolhida foi utilizar uma topologia com roteamento em fonte, que não necessitasse de um protocolo para descoberta de caminho, e tão pouco um elemento central para calcular as rotas. Especificamente, optou-se pela topologia em *Hipercubo* para construir nossa camada de encaminhamento, conforme anunciado na seção 3.5. Além das justificavas já apresentadas, o fato da topologia em *Hipercubo* ser base para outras arquiteturas utilizadas em

data centers, tais como: *BCube* [15], *MCube* [57], e por fim no fato desta topologia ter sido amplamente estudada e utilizada em arquiteturas de alto desempenho, foram levados em conta durante a escolha.

De forma resumida, o mecanismo de encaminhamento utilizado em topologias em *Hipercubo* é baseado em um algoritmo de roteamento na origem, Tal como um *fabric* não demanda nenhum sistema de controle. Para encaminhar os pacotes, um servidor utiliza uma simples operação XOR sobre seus vizinhos para encontrar o vizinho mais próximo do destino. Se o algoritmo de roteamento for capaz de detectar que o vizinho mais próximo está inoperante, ele pode escolher um novo vizinho, fazendo uso da capacidade intrínseca de tolerância à falha do *Hipercubo*, conferida pelos seus múltiplos caminhos. É importante enfatizar que os múltiplos caminhos, também, podem ser utilizados para realizar um balanceamento de carga sobre a rede. Embora a topologia apresente várias vantagens, a presença de inconsistências topológicas na rede, tais como: particionamento da rede ou laços fechados devido a erros de conexão dos enlaces físicos, podem gerar pacotes que circulem indefinidamente pela rede. Para evitar este tipo de problema, mecanismos de controle como TTL, podem ser adicionados ao algoritmo de encaminhamento.

Por sua vez, o uso de uma rede de *data center* centrado em servidor apresenta a desvantagem de utilizar parte da capacidade da CPU para encaminhamento do tráfego de trânsito, que está sendo minimizada pelo uso do mecanismo leve de encaminhamento e pela possibilidade de reconfigurar os enlaces físicos. Por outro lado, a utilização de uma arquitetura centrada em servidores possibilitou que o algoritmo de roteamento em *Hipercubo* fosse codificado no *switch* em *software,* utilizado pelo ambiente de virtualização. Isso trouxe uma série de benefícios para a arquitetura, dentre os quais se pode destacar:

1. Economia em relação ao uso de equipamentos de rede, tais como: *switches e* roteadores;
2. Integração entre o ambiente de virtualização e o mecanismo de encaminhamento utilizado na rede física;
3. Comunicação via protocolo *OpenFlow* promovida pela *switch* em *software*, que foi explorada para ajustar os parâmetros do mecanismo de encaminhamento;

4. Ainda, a codificação foi realizada com menos de 700 linhas de código, pois o *switch* virtual já implementa todas as funções de tratamento e encaminhamento de pacotes necessárias.

Como foi argumentado nesta subseção, o uso de uma rede de *data center* centrado em servidores permitiu atendermos os requisitos desta camada, e abriu espaço para automatização de alguns aspectos da arquitetura que serão discutidos em seções futuras.

4.2.2 Camada de virtualização

O projeto da camada de virtualização teve como base um sistema monitor de máquina virtual, comumente conhecido como *hypervisor,* associado a um *switch* virtual, ambos sendo suportados por um sistema operacional hospedeiro. Optou-se pelo uso de um sistema operacional hospedeiro, ao invés de utilizar o *hypervisor* diretamente sobre o *hardware*, devido à flexibilidade que a primeira opção proporciona. Com isso, pode-se escolher um *hypervisor* que atendesse as necessidades especificas da arquitetura, em especial a de realizar a migração das VMs tanto "a frio" como "a quente". De forma independente, também se pode escolher o *switch* virtual para a arquitetura. A Figura 4.2 ilustra a organização lógica de um servidor físico da arquitetura TRIIIAD. No topo, estão as VMs, as quais podem executar quaisquer sistemas operacionais (Linux, Windows, etc.) suportados pelo *hypervisor* escolhido. No centro, há o sistema operacional hospedeiro que integra o monitor de máquinas virtuais ao *switch* virtual. Um módulo de monitoramento de carga (módulo monitor) foi criado para registrar continuamente o uso de recursos de CPU memória e tráfego de rede. Na base, existe o *hardware* físico do servidor.

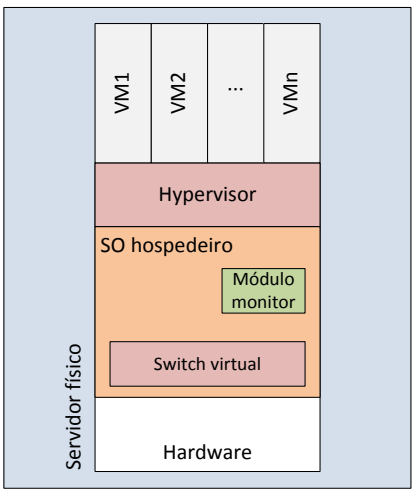


Figura 4.2: Organização lógica de um servidor físico de computação da arquitetura TRIIIAD.

Do ponto de vista da rede virtual, tomando o caso mais simples, cada VM está alocada em uma rede virtual nível 2, e possui um endereço IP único, que é utilizado como identificador global. O endereço MAC de cada VM é definido de tal forma que os 32 bits mais significativos representem o endereço binário do servidor físico no *Hipercubo*, e os 16 bits restantes representam o identificador da VM no servidor físico. Assim, é possível mapear mais de 1 bilhão de servidores físicos com mais de 65.000 VMs cada. Esta ideia é similar à utilizada em *Portland* [12], onde o identificador global é substituído por um identificador posicional. No entanto, a solução TRIIIAD difere da utilizada em *Portland*, principalmente pelo fato de não depender do protocolo *OpenFlow* e de regras instaladas em *switches* para encaminhar os pacotes de dados. Como foi explicado, o mecanismo de roteamento em *Hipercubo* de cada servidor é responsável pelo encaminhamento dos pacotes de dados para o vizinho mais próximo do destino.

De forma ilustrativa, a Figura 4.3 apresenta a VM2 hospeda no servidor físico 1 enviando dados para a VM1 hospedada no servidor físico 3. Inicialmente a VM2 envia uma mensagem de *broadcast* do tipo ARP *request* com o endereço IP da VM1. Esta mensagem é interceptada pelo *switch* virtual e encaminhada para o *controlador SDN*. Este resolve o endereço e retorna uma mensagem de ARP *replay* para *switch* virtual, que por sua vez encaminha para a VM2. Neste momento, a VM2 pode enviar dados normalmente para a VM1 pela rede virtual IP. O endereço binário de *Hipercubo* do servidor de físico de destino é extraído, de cada pacote de dados que chega ao *switch* virtual, dos 32 bits mais significativos do endereço MAC de destino usando uma operação AND. Após isso, o protocolo de roteamento *Hipercubo* envia o pacote para o vizinho mais próximo do destino. O endereço MAC da VM2 é preservado ao longo de todo o caminho, de forma que a VM1 pode enviar pacotes de volta para a VM2, sem a necessidade de uma operação de ARP *request*.

Esta solução atende ao requisito de prover a operação de migração mantendo o endereço IP da VM. No entanto, a operação de migração deve alterar o endereço MAC da máquina migrada para refletir sua nova posição no *Hipercubo*. Além disso, quando comparada com soluções *OpenFlow* puras, a proposta apresenta as seguintes vantagens:

1. O mecanismo de encaminhamento em *Hipercubo* precisa ler apenas o endereço de destino do quadro *Ethernet* (6 bytes), ao invés de ler os cabeçalhos *Ethernet*, IP e TCP;

2. Os servidores físicos operam de forma descentralizada, com isso, eles não sobrecarregam o *controlador SDN* nem o canal de controle seguro;

3. O *controlador SDN* pode ser utilizado para outras tarefas, visto que, do ponto de vista de encaminhamento dos pacotes, ele realiza a simples operação de mapear um endereço IP em um endereço MAC. Assim, a arquitetura utiliza o *controlador SDN* para tarefas de automatização, conforme será discutido na seção 5.6.

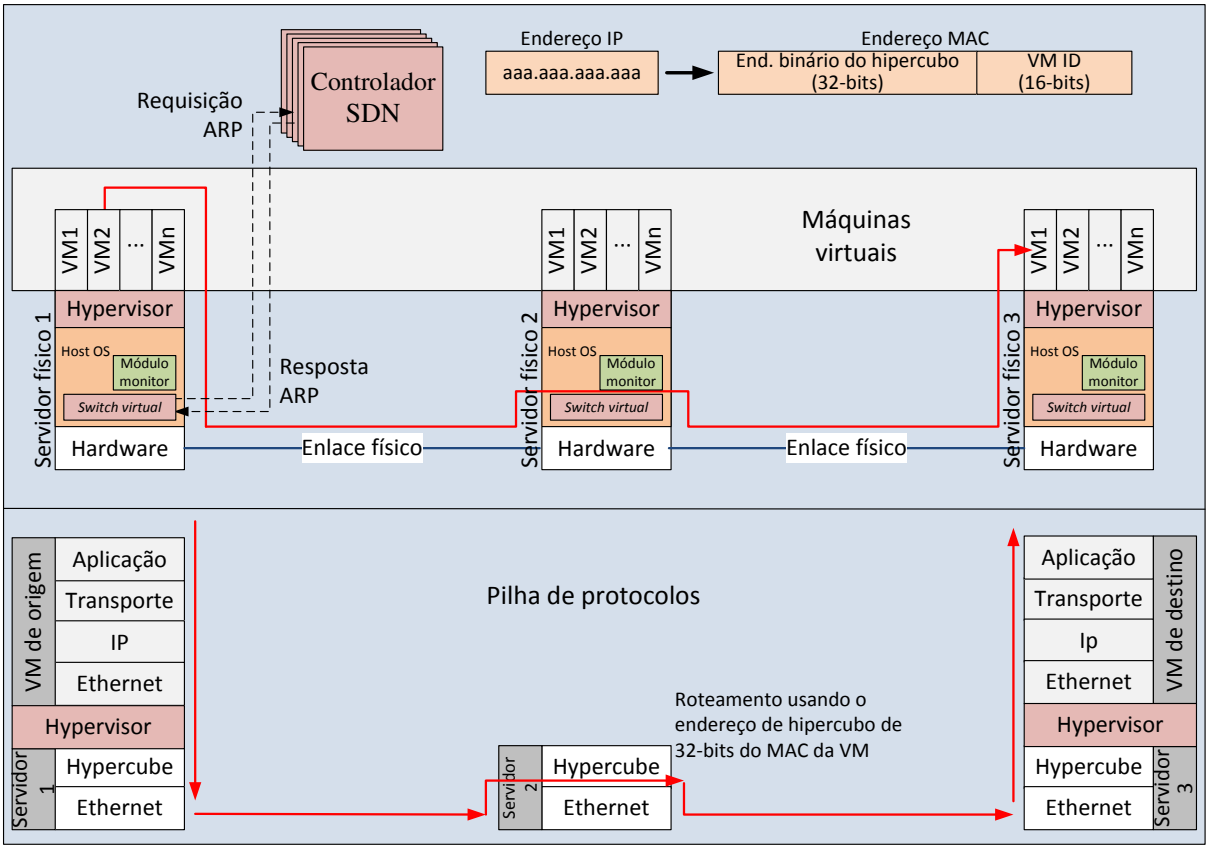


Figura 4.3: Encaminhamento de pacotes na arquitetura TRIIIAD

4.2.3 *Camada híbrida reconfigurável*

A camada híbrida reconfigurável foi projetada fazendo uso do *Cross-Braced Hypercube* [35], que investigou como simples chaves ópticas 2x2 combinadas com topologia *Hipercubo* tradicional poderiam criar atalhos para grandes fluxos da rede, reduzindo o tráfego de trânsito sobre os servidores. Em resumo, o *Cross-Braced Hypercube* reconfigura dinamicamente as conexões com o intuito de reduzir o número médio de saltos na rede, sem alterar o grau dos nós e sem impor modificações no algoritmo original de roteamento utilizado no *Hipercubo*. Isso assegura que os servidores da camada de encaminhamento sejam totalmente agnósticos em relação a topologia na camada física. Por sua vez, este isolamento

entre a camada de encaminhamento e a topologia na camada física possibilita que o chaveamento óptico seja realizado com sucesso.

Como caso ilustrativo, foram utilizadas seis chaves ópticas 2x2 (uma em cada face do cubo) para possibilitar a religação dinâmica dos enlaces em um *Hipercubo* de grau 3, conforme apresentado na Figura 4.4. Na configuração inicial todas as chaves ópticas estão no estado *barra* formando as conexões padrão de um *Hipercubo*, como pode ser visto na Figura 4.4a. Porém, se a maior parte do trafego que vai pelo enlace *000-001* for encaminhada para o enlace *001-101*, seria melhor mudar a chave óptica verde (face frontal) para o estado *cruz*, conectando assim os nós *000* e *101* diretamente como na Figura 4.4b. Após esta mudança, o nó *000* poderia continuar enviando os pacotes pela mesma interface de rede (enlace vermelho), por que ele não tem ciência da reconfiguração a nível físico que ocorreu. Com isso, estes pacotes não teriam que ser encaminhados pelo nó intermediário *001*, pois eles chegariam diretamente no destino. A outra pequena parte do tráfego, destinada ao nó *001*, seria encaminhada pelo enlace *101-001*, graças a capacidade de tolerância a falha nativa do roteamento em *Hipercubo*.

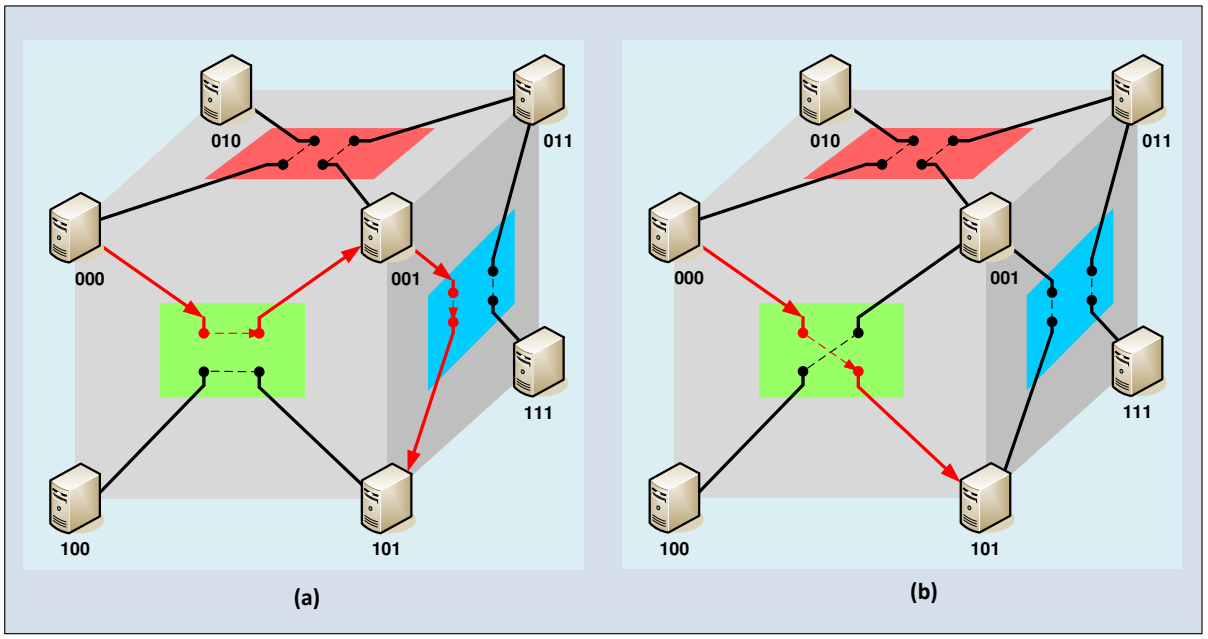


Figura 4.4: Hipercubo de 3 dimensões reforçado com chaves ópticas: a) chaves óptica no estado *barra*; b) chave óptica no estado *cruz*.

Para o caso geral de um *Hipercubo* de dimensão *n*, foi definido o *k-ésimo* bit do endereço (da esquerda para direita) como *dimensão k*, os enlaces que conectam nós com seus *k-ésimo* bits diferentes como *enlaces da dimensão k*, e os planos formados por dois enlaces da dimensão *k* e dois enlaces da dimensão *k+1* como *planos da dimensão k*. Onde *k* varia de *0* a

n-1, e *k+1* é igual a *0*, se *k* for igual a *n-1*. Assim, uma chave óptica pode ser inserida em um plano da dimensão *k,* conectando os dois enlaces da dimensão *k* as extremidades das chaves ópticas. Desta forma , pode-se calcular o total de chaves ópticas pela expressão $n \times 2^n/4$, para enlaces unidirecionais. Sendo naturalmente o dobro, para enlaces bidirecionais. Esta proposta permite implantar as chaves óptica de forma incremental no *Hipercubo*. De fato, os autores do *Cross-Braced Hypercube* argumenta que a redução de tráfego de trânsito com a implantação de chaves ópticas em *50%* dos planos, representa em torno de $3/4$ da redução total que seria obtida com a implantação em *100%* dos planos. Outras vantagens do espalhamento das chaves ópticas na rede em relação a grandes *switches* ópticos são: (*i*) o sinal é regenerado a cada salto; (*ii*) não existem cascateamentos ou chaveamentos multi-estágio para criar uma grande matriz de chaveamento; (*iii*) permitem o uso de dispositivos de baixo custo; e (*iv*) não são afetados por *crosstalking*.

De acordo com Bari *et al.* [25] as propostas de arquitetura de redes para *data centers* somente permitem o controle da camada 2 e 3. Por outro lado, a TRIIIAD apresenta o diferencial de permitir o controle da camada 1 (camada física), possibilitando a redução do trafico de trânsito inerente as redes de *data center* centrado em servidores. Por fim, a camada de encaminhamento não é impactada pelas reconfigurações dos enlaces físicos, desde que estas reconfigurações não criem laços na topologia e mantenha a topologia conexa.

4.2.4 Framework de controle, gerência e orquestração

Ambientes complexos, como os *data centers*, exigem o uso de elementos adicionais que viabilizem o seu gerenciamento. Desta forma, a arquitetura TRIIIAD utiliza, além dos *nós de computação*, uma base de dados compartilhada e quatro controladores logicamente centralizados, que irão supervisionar e atuar sobre os *nós de computação* e as chaves ópticas para orquestrar a execução das tarefas da arquitetura. Para um melhor entendimento do ambiente, a Figura 4.5 ilustra os elementos da arquitetura e suas interações.

Próximo ao centro da Figura 4.5 está o *servidor físico de computação* ou *nó de computação*, que é responsável pelo serviço de computação do *data center* e também pelo encaminhamento dos pacotes. A computação é realizada nas máquinas virtuais suportadas pelo *hypervisor*, enquanto o encaminhamento é realizado pelo protocolo de roteamento em *Hipercubo* implementado no *switch* virtual. A supervisão das atividades da arquitetura é

dividida entre quatro elementos: o *controlador da Nuvem*, o *controlador da Rede*, o *controlador SDN* e o *controlador das Chaves Ópticas.*

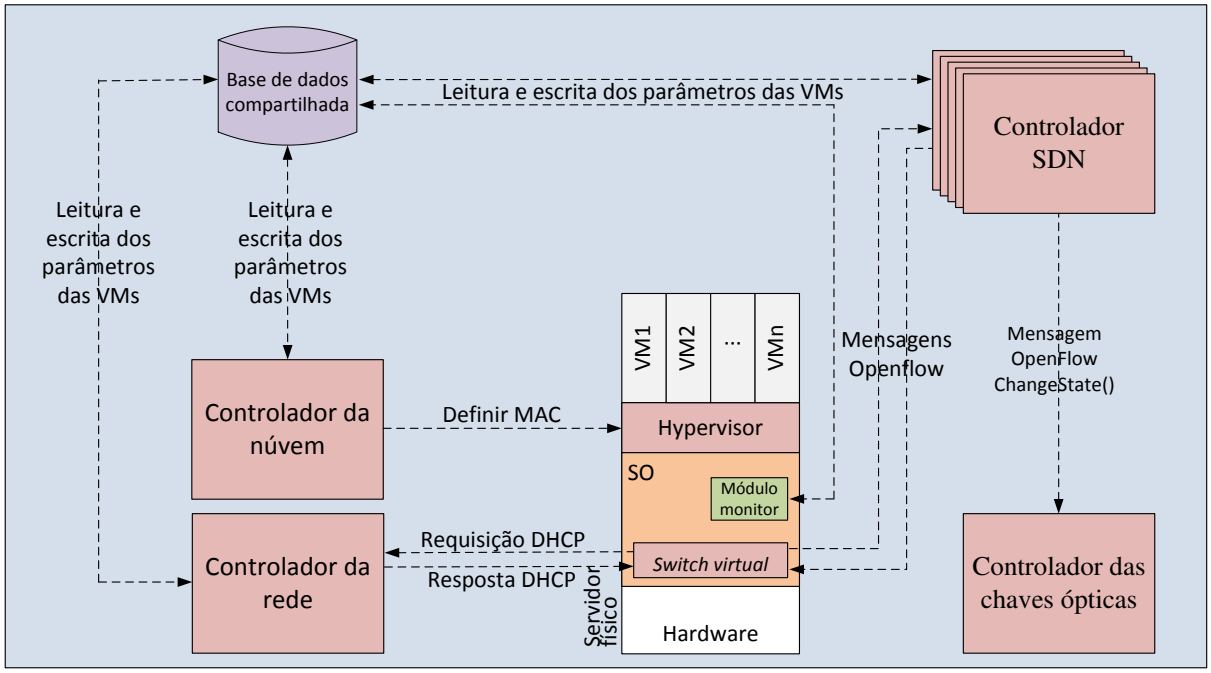


Figura 4.5: Elementos da arquitetura TRIIIAD

O primeiro, *controlador da Nuvem* da Figura 4.5, é responsável pelas operações que envolvem as máquinas virtuais, para isso, ele interage diretamente com o *hypervisor* de cada *servidor físico de computação*. No exemplo da ilustração, este controlador define o endereço MAC da máquina virtual durante a operação de criação ou migração. O segundo, *controlador da Rede*, atua de forma semelhante coordenando as atividades que envolvem as redes virtuais, para isso ele interage com o *switch* virtual de cada *nó de computação*. A título de exemplo, foi ilustrado a troca de mensagens DHCP, necessária para configurar automaticamente os parâmetros de rede de uma VM. O terceiro, *controlador SDN*, é responsável por diversas atividades, deste a inicialização da arquitetura, até a gerência de aspectos do *Hipercubo*. Sua interação, com o *nó de computação,* é feita por meio de mensagens *OpenFlow* trocadas com o *switch* virtual. Estes três controladores e o *nó de computação* possuem acesso de leitura e escrita a *base de dados compartilhada,* onde estão armazenados os dados relativos às máquinas virtuais, tais como: endereço MAC *Ethernet*, endereço IP, servidor físico no qual a VM está hospedada, entre outras. O quarto controlador, *controlador das Chaves Ópticas*, traduz a requisição enviada pelo *controlador SDN* em sinais elétricos que fisicamente irão alterar os estados das chaves ópticas. A Figura 4.5 ilustra o *controlador SDN* enviando uma mensagem *OpenFlow* denominada *ChangeState()* para o *controlador das Chaves Ópticas.* Por fim, para

efeito de simplicidade, a Figura 4.5 ilustra um único *servidor físico de computação*. No entanto, um grande *data center*, normalmente contém milhares destes elementos.

Capítulo 5 – Integração e consolidação dos elementos da arquitetura TRIIIAD

A supervisão da arquitetura TRIIIAD exige uma intensa troca de informações entre seus elementos, de forma que alocar banda na rede de dados para o gerenciamento da arquitetura, poderia comprometer o uso desta rede. Somado a isso, a rede de dados da arquitetura é disposta em uma topologia em *Hipercubo*, que é muito interessante para distribuir carga, porém, apresentaria um desbalanceamento se fossem incluídos alguns elementos centralizadores, como é o caso dos controladores. Assim, a solução adotada foi criar uma rede de gerenciamento fisicamente independente da rede de dados, conforme a Figura 5.1, que ilustra uma visão geral da interconexão dos elementos da arquitetura.

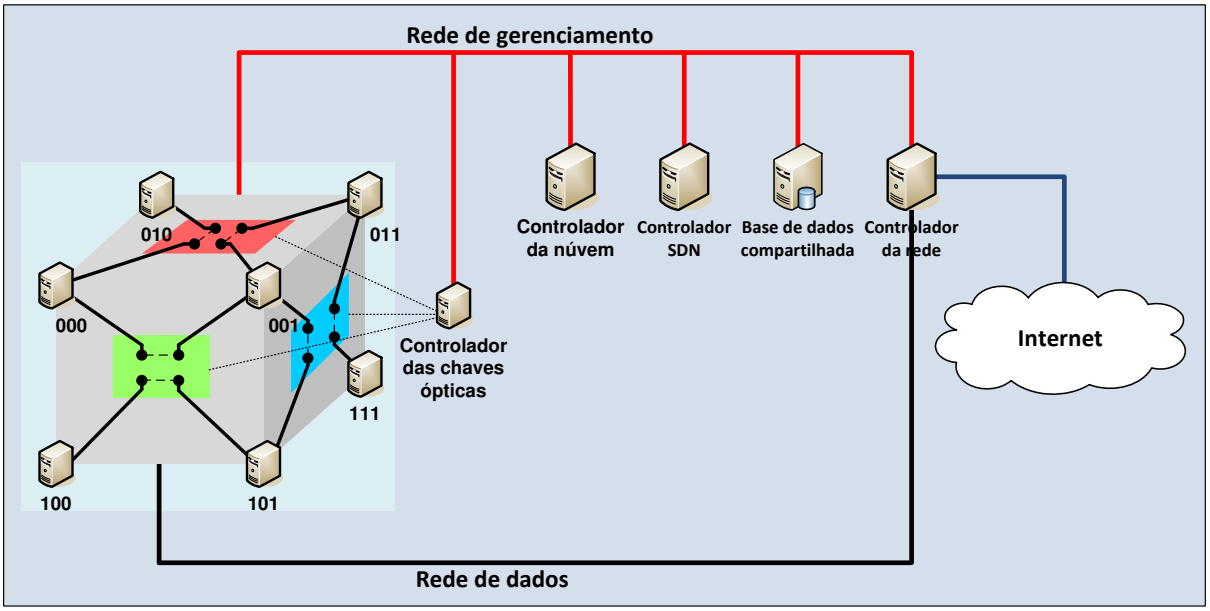


Figura 5.1: Visão da interconexão dos elementos da arquitetura TRIIIAD

A parte superior da Figura 5.1 ilustra a rede de gerenciamento (*management network*), onde estão conectados a base de dados compartilhada, os quatro controladores e os *nós de computação*, representados pelo *Hipercubo* de dimensão *3*. Por sua vez, a rede de dados (*data network*) conecta os *nós de computação* entre si, efetivamente formando a topologia em *Hipercubo*, e ao *controlador da Rede*, para que as VMs possam acessar os serviços fornecidos por ele, incluindo o acesso a Internet. O *controlador das Chaves Ópticas* precisa estar localizado junto aos planos de chaveamento, a fim de enviar os sinais elétricos necessários para acionar as chaves, por este motivo ele foi ilustrado próximo ao *Hipercubo*.

5.1 Servidores físicos de computação

Na Figura 5.1, está implícito que, cada servidor físico de computação necessita de uma interface física de rede para se conectar a rede de gerenciamento, uma segunda interface para se conectar ao *controlador da Rede*, e três de interfaces para se conectar aos seus vizinhos e formar o *Hipercubo*. De fato, o número de interfaces de um *nó de computação*, vai depender do grau do *Hipercubo* que ele está conectado. De forma genérica, para um *Hipercubo* de grau n, cada *nó de computação* deverá possuir $n + 2$ interfaces físicas de rede. Por exemplo, para um *Hipercubo* de grau $n=14$, que conecta $2^{14} = 16384$ nós, cada nó irá necessitar 16 interfaces de rede. O que pode ser resolvido de forma simples com o uso de placas com múltiplas interfaces, por exemplo, 4 placas de com 4 interfaces cada. Outro ponto importante que deve ser destacado está ilustrado na Figura 5.2. Observe que apenas as interfaces identificadas como *NIC 1, NIC 2, NIC 3* e *NIC 4*, que pertencem a rede de dados, são gerenciadas pelo *switch* virtual. Isso assegura que o algoritmo de roteamento do *Hipercubo*, implementado no *switch* virtual, irá tratar apenas os pacotes de dados das VMs. Por sua vez, os pacotes da rede de gerenciamento, que trafegam pela interface *NIC 0*, são tratados diretamente pelo sistema operacional hospedeiro.

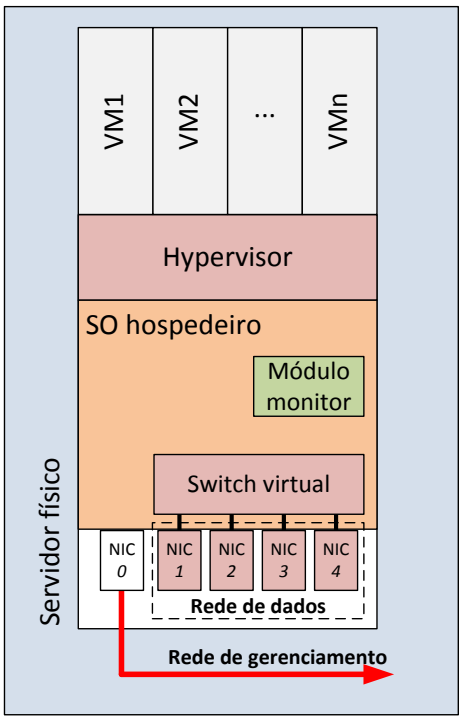


Figura 5.2: Detalhe das interfaces de um *nó de computação*

5.2 Controlador da Nuvem

Imagine a situação na qual um conjunto de *1000* VMs precisam ser instanciada para atender a demanda de um cliente de um *data center*. Imagine ainda, que este cliente exija que estas VMs sejam divididas igualmente entre duas redes virtuais, que cada VM tenha pré-instalado o servidor *web Apache* e que todas devam ter os seus horários sincronizados. Por fim, imagine que este ambiente deva estar operacional em questões de minutos. A situação descrita neste paragrafo evidencia a necessidade de um conjunto de ferramentas, para viabilizar a operacionalização da camada de virtualização de um *data center*.

Na TRIIIAD, este conjunto de ferramentas é fornecido pelo *controlador da Nuvem*, por meio dos serviços de: *gerenciamento de hypervisor, gerenciamento de imagens, autenticação e identidade, redes virtuais, armazenamento persistente, sincronização de tempo, e painel de operação.*

O principal serviço do *controlador da Nuvem* é sem dúvida o *serviço de gerenciamento* de *hypervisor*, que possibilita controle remoto dos *hypervisors*, instalados nos *nós de computação*. Assim, durante a criação de máquina virtual, o *controlador da Nuvem* pode fornecer os parâmetros do *hardware* virtual, bem como a imagem do disco que deve ser utilizado. Para uma operação de migração os parâmetros seriam: a VM que será migrada e o

nó de computação de destino. Já para a operação de exclusão bastaria fornecer a VM que deve ser excluída. Ainda, é importante que existam outras operações, tais como: desligar uma VM, reiniciar uma VM, alterar parâmetros do *hardware* virtual, etc.

O serviço de gerenciamento de imagens é encarregado de criar de um repositório para imagens de discos virtuais, as quais são utilizadas para criação de VMs. Enquanto o *serviço de autenticação e identidade* permite criar usuários para gerenciamento da nuvem (ou parte dela), bem como autenticar estes usuários. Por sua vez, o *serviço de redes virtuais* é responsável por definir os parâmetros das redes virtuais utilizadas pelas VMs. Embora, é importante destacar que as redes virtuais são criadas pelo *controlador da Rede*. O *serviço de armazenamento persistente* cria dispositivos de armazenamento em bloco (HDD virtuais), para as VMs que necessitem armazenar dados de forma persistente. Para atender a necessidade de sincronização entre os relógios das VMs, existe o *serviço de sincronização de tempo*. Por fim, é fundamental a existência de um *serviço de painel de operação,* que proporcione uma visão da camada de virtualização e, ao mesmo tempo, forneça uma interface para executar tarefas corriqueiras, tais como: iniciar VMs, associar endereços IP a uma VM, criar usuários, etc.

5.3 Base de dados compartilhada

A base de dados compartilhada armazena as informações geradas pelo: *controlador da Nuvem*, *controlador da Rede*, *controlador SDN* e pelos *nós de computação*. Entre os dados mais importantes estão:

- *endereço MAC* dos *nós de computação*. Este parâmetro é fundamental para que no momento da criação ou migração de uma máquina virtual, o *controlador da Nuvem* possa definir o endereço MAC da VM com base no endereço no endereço do nó, permitindo assim que o esquema de roteamento em *Hipercubo* possa funcionar. Conforme será na subseção *5.6.1.1*, o MAC dos *nós de computação* é armazenado na base de dados pelo *controlador SDN*.
- *endereço MAC* e *endereço IP* das VMs. Estes parâmetros são utilizados pelo *controlador SDN* para resolver as requisições de ARP. Conforme apresentado acima, o endereço MAC da VM é definido pelo *controlador da Nuvem*, enquanto o endereço IP é definido pelo *controlador da Rede*, conforme será apresentado na seção *5.4*.

- Carga média dos *nós de computação*. Estas cargas são registradas na base de dados pelo *Módulo monitor* de cada *nó de computação* e é utilizada pelo *controlador SDN* para orquestrar a arquitetura, conforme será apresentado na subseção *5.6.2*.

5.4 Controlador da Rede

Atuando de forma semelhante ao *controlador da Nuvem*, o *controlador da Rede* viabiliza a operacionalização das redes (*L2* e *L3*) da camada de virtualização. Para isso este controlador conta com os serviços de: *isolamento em camada 2, roteamento em camada 3, tradução de endereço de rede de camada 3 e configuração automática de parâmetros de rede.*

O *serviço isolamento em camada 2* permite ao *controlador da Rede* criar segmentos de camada 2, virtualmente isolados. Para isso, ele configura nos *switches* virtuais, dos *nós de computação*, mecanismos bem conhecidos, tais como: *virtual LAN* e tunelamentos. O *serviço de roteamento em camada 3* é realizado por roteadores em *software*, para possibilitar a comunicação entre VMs de redes distintas. Por sua vez, o *serviço de tradução de endereço de rede de camada 3* (NAT) é oferecido para permitir que as VMs possam acessar o mundo externo, por exemplo, a internet. E por fim, para cada rede de camada 3 existe um servidor de DHCP que implementa o *serviço de configuração automática de parâmetros de rede*.

A Figura 5.3 ilustra um controlador de rede, onde a interface *NIC 0* está conectada a rede de gerenciamento e é tratada pelo sistema operacional. As interfaces *NIC 1* e *NIC 2* são gerenciadas pelo *switch* virtual e conectadas a rede de dados e a internet, respectivamente. O *serviço de isolamento em camada 2 (L2 Segmentation)* é um processo que atua no *switch* virtual, do *controlador da Rede* e dos *nós de computação*, configurando o mecanismo de isolamento de tráfego em camada 2. Por sua vez, os *serviços de roteamento em camada 3 (Router), tradução de endereço de rede de camada 3* (NAT) e *configuração automática de parâmetros de rede* (DHCP) estão conectados ao *switch* virtual, para atender os seguintes casos:

- *Configuração da interface de rede de uma VM*. Quanto uma VM utiliza o protocolo DHCP para configurar sua rede, ela envia um pacote de *broadcast* para descobrir os servidores de DHCP disponíveis. Este pacote é analisado pelo mecanismo de roteamento em *Hipercubo*, que identifica o pacote como *broadcast* e encaminha para o *controlador da Rede*. Ao chegar ao controlador, o *serviço de configuração automática*

de parâmetros de rede (DHCP) irá tratar este pacote e retornar os parâmetros de configuração desta VM, onde o *gateway default* é o roteador virtual do *controlador da Rede*. Então, o par (MAC, IP) é salvo na base de dados compartilhada.

- *Acesso a internet por uma VM*. De forma natural, a pilha de protocolo TCP/IP da VM, utilizado *gateway default*, para encaminhar qualquer pacote para fora da sua rede. Isso faz com que o mecanismo de roteamento em *Hipercubo*, encaminhe estes pacotes para o *controlador da Rede*, que por sua vez, utiliza os *serviços de roteamento em camada 3* (*Router*) e *tradução de endereço de rede de camada 3* (NAT), possibilitando assim que o pacote trafegue pela internet e as respostas sejam retornadas para a VM correta.
- *Comunicação entre VMs em redes virtuais distintas*. De forma semelhante ao acesso a internet, o acesso entre VMs localizadas em rede distintas é realizado pelos *serviços de roteamento em camada 3* (*Router*) e *tradução de endereço de rede de camada 3* (NAT).

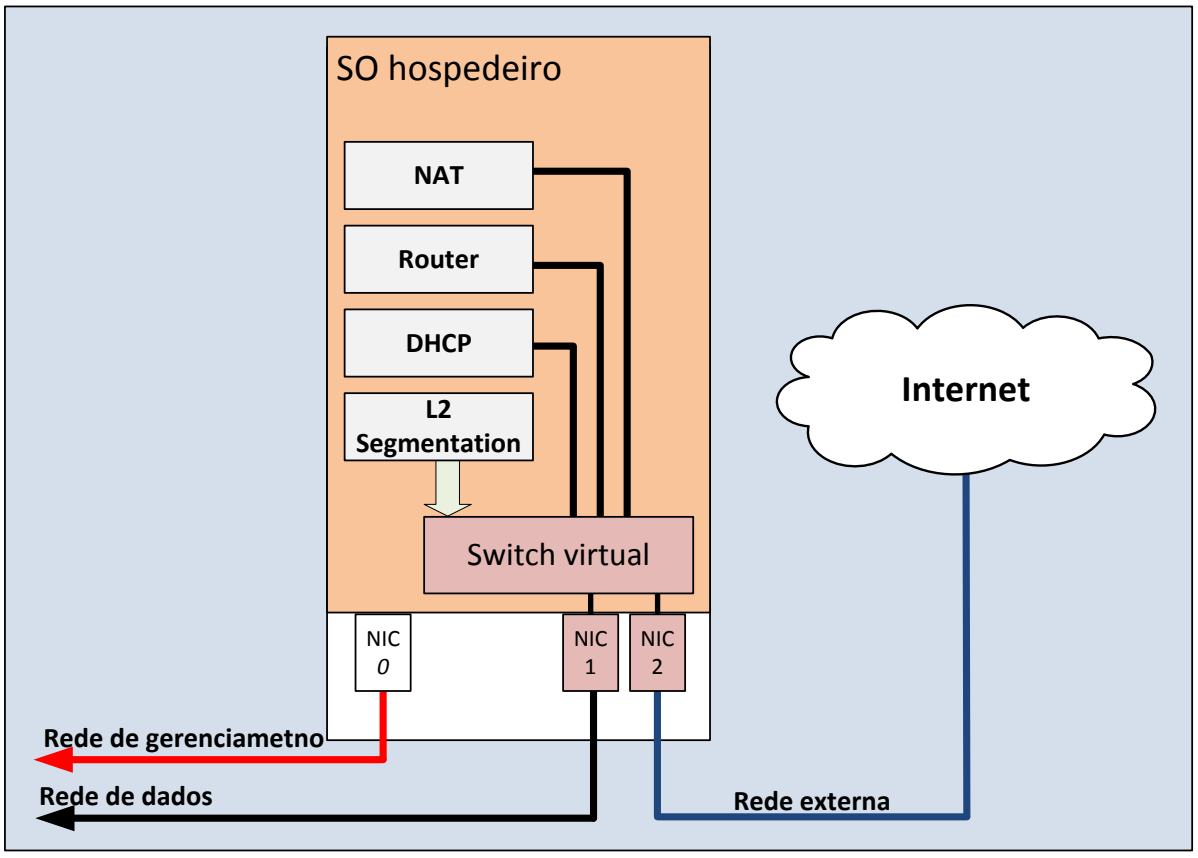


Figura 5.3: Organização lógica do *controlador da Rede* na TRIIIAD.

5.5 Controlador das Chaves Ópticas

O *controlador das Chaves Ópticas* é o elemento da arquitetura responsável por efetivar a comutação das chaves, que irá reconfigurar os enlaces da camada híbrida de reconfiguração, entre os *nós de computação* que compõem o *Hipercubo*. No entanto, não é este controlador que decide quais chaves serão comutadas. Esta tarefa compete ao *controlador SDN*, conforme será apresentado a posteriori. Ademais, um único *controlador das Chaves Ópticas* não é capaz de atender a todas as chaves da arquitetura. Esta limitação ocorre devido ao grande número de chaves que podem existir, bem como ao espalhamento destas chaves na topologia. Por outro lado, isso favorece o uso de computadores de baixíssimo custo, por exemplo, os computadores integrados em uma única placa do tamanho de um cartão de crédito. Em nível de *hardware*, este controlador deve possuir uma interface de rede (com ou sem fio); e pinos de entrada e saída para comandar o circuito que irá acionar as chaves ópticas. Em nível de *software*, o controlador deve permitir a instalação de um sistema operacional e do *software OpenFlow* que irá se comunicar com o *controlador SDN*. Esta última exigência foi imposta para homogeneizar o protocolo utilizado pelo *controlador SDN* na arquitetura.

5.6 Augmented Software-Defined Networking (A-SDN)

As redes tradicionais de *data center* estão fundamentadas na separação entre os elementos de rede e os elementos de computação, tanto em nível físico como em nível lógico. Desde modo, a orquestração das VMs é realizada independentemente do controle e gerenciamento da rede. Por sua vez, a rede de *data center* centrado em servidores e a NFV promovem uma reviravolta neste cenário, pois o acoplamento dos elementos de computação aos elementos de rede, gera uma competição pelos recursos do servidor, inviabilizando o modelo de controle/gerenciamento e orquestração, outrora utilizado. Nesse novo cenário, os planos de controle e gerenciamento da rede e a orquestração das VMs não podem ser tratados de forma isolada. De fato, o *controlador da Rede* deve ampliar sua visão, integrando-se à dinâmica da orquestração, de forma a manter a consistência entre as informações da rede e as decisões tomadas.

Na TRIIIAD, a função de orquestração das VMs e o acionamento dos dispositivos de comutação da camada híbrida reconfigurável fazem parte das atividades do *controlador SDN*. No entanto, estas atividades são exemplos de tarefas que estão fora do escopo da SDN tradicional, pois sua realização exige que controlador tenha uma visão mais ampla da

arquitetura, associando os parâmetros tradicionais fornecidos pela rede aos parâmetros fornecidos pelas VMs e pelos *nós de computação*. Devido a este aumento da visão, atuação e capacidade da SDN nesta arquitetura em relação à SDN tradicional, daqui em diante será utilizado a sigla A-SDN (*Augmented Software-Defined Networking,* ou *Rede Definida por Software Aumentada)* para representar as funções da rede definida por *software* das subseções a seguir.

5.6.1 Inicialização da TRIIIAD

Quando o *controlador A-SDN* é inicializado, três subtarefas são disparadas. A primeira realiza a criação e a inicialização do *Hipercubo* que representa a rede de encaminhamento, a segunda realiza a criação e a inicialização do comutador óptico formado pelos elementos de comutação da camada híbrida de reconfiguração, e a terceira realiza a criação e a inicialização da estrutura que gerencia a carga dos *nós de computação*.

5.6.1.1 Inicialização Hipercubo da rede de encaminhamento

O controlador cria um objeto lógico chamado *Hipercubo* com seguintes parâmetros: grau, lista *switches* e tipo de *switches*. O primeiro parâmetro representa o grau do *Hipercubo* que será criado, o segundo parâmetro representa a lista de *switches* conectados a este controlador, inicialmente esta lista pode estar vazia. Por fim, o último parâmetro representa o tipo de *switch* que será considerado como nó do *Hipercubo*. Este último parâmetro é necessário para que seja possível diferenciar os *switches* que representam os *nós de computação* dos *switches* que representam os *controladores das Chaves Ópticas*.

À medida que cada *nó de computação* é inicializado, seu *switch* virtual conecta-se ao *controlador A-SND* da arquitetura e fica aguardando que este envie os parâmetros de configuração relativos a sua posição no *Hipercubo*. Desta forma a lista de *switches* vai sendo preenchida. Por sua vez, o objeto *Hipercubo* seleciona os *switches* que representam os *nós de computação* e constrói logicamente a topologia em *Hipercubo*, com base nas conexões entre os elementos. Uma vez que todos os *nós de computação* foram adicionados à topologia, o objeto *Hipercubo* envia um conjunto de mensagens para cada *switch*, onde a primeira mensagem define a posição do *nó de computação* no espaço de endereçamento do *Hipercubo* e as demais mensagens informam quais seus vizinhos utilizando tupla (vizinho, interface). Por

fim, o *controlador A-SDN*, atualiza a base de dados compartilhada, definindo o endereço MAC *Ethernet* de cada *nó de computação* com base em sua posição do *Hipercubo*.

5.6.1.2 Inicialização do Comutador Óptico

De forma semelhante, um objeto lógico denominado *ComutadorÓptico* é inicializado com seguintes parâmetros: lista *switches* e tipo de *switches*. O primeiro parâmetro representa a lista de *switches* conectados a este controlador, o segundo parâmetro representa o tipo de *switch* que será considerado como nó do comutador óptico, representados pelos *controladores das Chaves Ópticas*. À medida que cada controlador é detectado, ele é adicionado a lista de *switches*, e em seguida, a lista de nós do objeto *ComutadorÓptico*. Porém, diferentemente do caso dos *nós de computação*, nenhuma mensagem especial é enviada para os *controladores das Chaves Ópticas*, pois não há nada a ser configurado neste momento. Assim, apenas a mensagem padrão do protocolo *OpenFlow* estabelecendo a conexão e enviada.

Cada nó do *ComutadorÓptico* é identificado de forma única pelo seu nome, que deve ser criado de acordo com o padrão "*key-xxxx*", onde *x* é um dígito entre 0 e 9. Por meio deste nome, é possível buscar quais os *planos de chaveamentos* (planos da dimensão *k* conforme descrito no item 4.2.3) são controlados por este nó, e para cada plano, quais os *nós de computação* que estão associados a ele. Desta forma, o *controlador A-SDN*, é capaz de identificar e atuar no plano de chaveamento correspondente aos *nós de computação* com alto tráfego de trânsito, por exemplo.

5.6.1.3 *Inicialização da estrutura que gerencia a carga dos nós de computação*

Um terceiro objeto lógico denominado *NósDeComputação* é criado para gerenciar as cargas de trabalho dos *nós de computação*. Ao ser inicializado, este objeto recebe seis parâmetros, que representam os limites superiores e inferiores de CPU, memória e tráfego de trânsito, que foram convencionados. Os nós do objeto *NósDeComputação* são adicionados a partir de uma busca na base de dados compartilhada. Entretanto, isso ocorre somente após o término do procedimento de inicialização do *Hipercubo*. Esta restrição foi imposta para garantir que os endereços (MAC) dos *nós de computação* tenham sido atualizados na base de dados compartilhada. Ainda, o objeto *NósDeComputação* pode consultar esta base de dados, a qualquer instante, para obter informações sobre as VMs que estão hospedadas em cada *nó de computação*. Após este momento, toda a arquitetura está inicializada e pronta para operar.

5.6.2 Orquestração de políticas na TRIIIAD

A *orquestração da* TRIIIAD compreende as tarefas para automatizar o gerenciamento da migração de máquinas virtuais e a religação dos enlaces da camada híbrida reconfigurável. Este conjunto de tarefas é desempenhado pelo *controlador A-SND* em parceria com os *nós de computação*, e segue o fluxograma ilustrado na Figura 5.4.

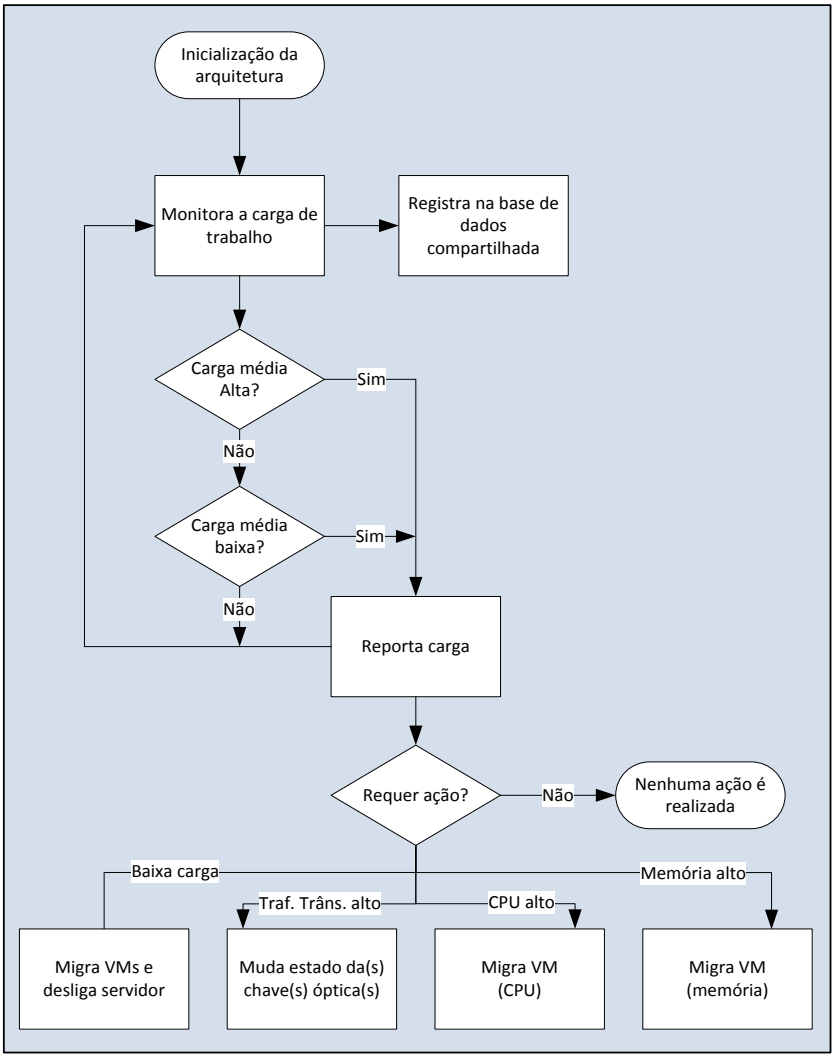


Figura 5.4: Fluxograma da orquestração da TRIIIAD

Observe no fluxograma da Figura 5.4, que o inicio da orquestração se dá com a *inicialização da arquitetura*, mais especificamente durante o processo criação do objeto lógico para gerenciar as cargas de trabalho dos *nós de computação*. Uma vez que a arquitetura foi inicializada, os *nós de computação* passam a monitorar suas cargas de trabalho, em termos de uso de CPU, memória e tráfego de trânsito. Em intervalos de tempo pré-determinados, cada *nó de computação* armazena sua carga de trabalho na base de dados compartilha, ao mesmo tempo, testa se sua carga média está entre os limites mínimos e máximos pré-estabelecidos.

Caso a carga média esteja fora dos limites, uma mensagem é encaminhada para o *controlador A-SDN* reportando o evento. Por sua vez, o controlador analisa cada evento reportado e verifica se ele requer uma ação. Para isso, o controlador utiliza o objeto *NósDeComputação* para checar as cargas de trabalho dos demais elementos e tomar a decisão mais apropriada para o momento, inclusive decidir que nenhuma ação deva ser realizada.

As ações tomadas pelo *controlador A-SDN* dependem do evento que foi reportado. Assim, caso um *nó de computação* reporte que está com alto uso de CPU, o controlador executa a sua rotina *Migra VM (CPU)*, que seleciona uma VM do nó sobrecarregado e um nó de destino com baixo uso de CPU, por fim, envia uma mensagem com estes parâmetros, ao nó sobrecarregado, ordenando a migração. Uma decisão semelhante é tomada para o caso alto uso de memória. Caso um nó reporte que está com alto tráfego de trânsito, o controlador executa a rotina *Muda estado da(s) chave(s) óptica(s)*, que verifica se o nó pertence a um plano de chaveamento. Caso afirmativo, o controlador avalia o impacto do chaveamento sobre a rede, se este chaveamento não reduzir o tráfego de trânsito global, nada é realizado. Por outro lado, se houver redução do tráfego de trânsito global, o *controlador A-SND*, envia uma mensagem para o *controlador das Chave Ópticas* responsável por este plano de chaveamento, informando o novo estado das chaves.

Um caso especial ocorre quando um *nó de computação* reporta que está com carga abaixo dos limites pré-estabelecidos. Este evento faz com que o *controlador A-SDN* execute sua rotina *Migra VMs e desliga servidor*, que irá remover todas as VMs deste *nó de computação* e desligá-lo. Esta situação foi adicionada a arquitetura para possibilitar o gerenciamento de energia, por meio do desligamento de máquinas físicas em momento de subutilização do *data center*. No entanto, sua utilização fica condicionada ao aprimoramento do mecanismo de roteamento *Hipercubo*, para trabalhar sobre uma topologia incompleta.

5.6.3 Resolução das requisições de ARP

A opção de sobrecarregar os endereços MAC das VMs com a posição que ela ocupa no *Hipercubo*, criou uma rede plana com possiblidades de milhões de VMs. Para contornar o problema de inundação (*flooding*), a resolução das requisições ARP foi atribuída ao *controlador A-SDN* e operacionalizada da seguinte forma: Os pacotes de ARP *request* gerados pelas VMs são interceptados pelo *switch* virtual dos *nós de computação*, e encaminhados para o *controlador A-SDN*. Ao receber este tipo de pacote, o controlador

consulta a base de dados compartilha e busca o endereço MAC (definido na criação da VM pelo *controlador da Nuvem*) associado ao endereço IP da consulta (durante a inicialização do VM pelo *controlador da Rede*). O próximo passo executado pelo *controlador A-SDN* é montar um pacote de ARP *replay* e encaminhá-lo para VM de origem via o *switch* virtual do servidor físico que hospeda esta VM. Observe que todo o processo é transparente para a VM, de forma que nenhuma modificação da pilha TCP/IP foi necessária.

5.6.4 Potenciais problemas e limitações tratados pela A-SDN

O uso de uma topologia de *data center* centrado em servidor associado a NFV, apesar de fornecer um cenário flexível e programável, apresenta problemas e limitações que precisam ser contornados. O mais evidente destes problemas é a competição pelos recursos dos servidores físicos, entre as tarefas de computação e de encaminhamento de pacotes. Para minimizar esse problema foi utilizada: uma camada híbrida reconfigurável, que distribuiu enlaces ópticos à topologia; e uma *função virtual de rede* para roteamento de pacotes em *Hipercubo*. Portanto, por um lado, buscou-se minimizar o tráfego de trânsito global na arquitetura, possibilitando o uso de atalhos entre servidores, e consequentemente minimizando o número de saltos. Por outro lado, utilizou-se um algoritmo de encaminhamento leve para aliviar a CPU.

Beneficiando-se do acoplamento entre a camada de virtualização e a camada de encaminhamento, foi criado um esquema de endereçamento para as VMs, no qual o endereço de *hipercubo* do *nó físico de destino* é parte do endereço MAC da VM. Assim, a operação de migração de VMs foi modificada, de forma a alterar o endereço MAC da VM migrada para corresponder ao endereço de *Hipercubo* do servidor físico de destino. Todos estes fatores corroboraram para que os planos de controle/gerenciamento da rede e a orquestração das VMs não pudessem ser tratados de forma independente. Portanto, a dinâmica da orquestração foi integrada as atividades de controle e gerenciamento da rede, expandindo a visão do *controlador A-SDN* sobre a arquitetura, de forma a manter consistente as informações da rede e as decisões tomadas.

Outros problemas e limitação que são de responsabilidade do *A-SND* estão descritos nas subseções abaixo.

5.6.4.1 Aspectos topológicos

A complexidade envolvendo o tratamento de eventos topológicos, por exemplo, a falha de um *nó de computação* (ou de um *switch* virtual) implica numa falha de rede. Num sistema com camada física reconfigurável o desafio é ainda maior. Assim, a A-SDN tem que consolidar informações da orquestração com as informações topológicas para tomar a decisão acertada neste contexto de redes de *data center* centrado em servidores. Por outro lado, a descoberta automática de recursos de redes tornou-se mais intricada, devido ao compartilhamento dos recursos dos *nós de computação* entre o processamento dos usuários e as funções de rede. Ainda, a localização e identificação dos recursos tornaram-se atividades chaves, perpassando funções de: orquestração, controle e gerência da rede. Assim, conforme foi apresentado, os mecanismos tradicionais de inicialização (DHCP) e de correspondência entre diferentes identificadores (ARP para conectar endereços IP e MAC) exigiram um tratamento de forma integrada na arquitetura.

5.6.4.2 Aspectos de roteamento e de encaminhamento

O aproveitamento de rotas alternativas nas redes de *data center* centrado em servidores deve levar em conta à ocupação dos servidores com tarefas de usuários (VMs). Por outro lado, o mecanismo de encaminhamento *kernel* tem prioridade sobre as tarefas de usuários. Para abordar este problema, o *controlador A-SDN* deveria enviar informações para o mecanismo de encaminhamento, para que este possa escolher o próximo salto em função da carga dos seus vizinhos, gerando o mínimo de interferência, tanto nas tarefas dos usuários quanto no encaminhamento do tráfego.

A simplificação do mecanismo de encaminhamento (roteamento em fonte) pode gerar problemas se a topologia apresentar laços. Por exemplo, alguns pacotes seriam encaminhados infinitamente, tomando a capacidade dos *nós de computação* envolvidos. Este tipo de problema, naturalmente, são mais críticos se a camada física tem capacidade de reconfiguração. Apesar de um mecanismo de proteção como TTL, com número de saltos máximo igual ao diâmetro da rede, minimizar este problema. A solução mais efetiva seria dotar o *controlador A-SDN* com um mecanismo de tratamento de laços.

5.6.4.3 Aspectos de multiplicidade, segurança e de mobilidade de VMs

Os aspectos topológicos e de roteamento devem permitir a multiplicidade e a mobilidade das VMs. A multiplicidade diz respeito à capacidade da rede de *data center* centrado em servidores ser escalável em termos de número de VMs. Ao mesmo tempo, este

ambiente deve fornecer mecanismos de segurança e isolamento para todo o datacenter. Isso representa um grande desafio que deverá ser tratado pelo *controlador A-SDN*.

Em relação à mobilidade, num ambiente centrado em servidores, ao se migrar VMs pode-se disparar eventos de sobrecarga nos *nós de computação* intermediários. Por sua vez, esta sobrecarga pode iniciar mais processos migratórios, de forma que a rede entre num regime instável, devido ao acoplamento entre as funções da rede e de computação. Neste caso, o *controlador A-SDN* deve ser dotado de mecanismos capazes de estabilizar os eventos oriundos deste acoplamento.

Capítulo 6 – Implementação da TRIIIAD

Dois objetivos motivaram a implementação da arquitetura TRIIIAD. O primeiro foi comprovar a viabilidade técnica de programar essa arquitetura utilizando *hardware* de prateleira; o segundo foi validar os conceitos para permitir a caracterização do funcionamento da arquitetura. Assim, foi criado um ambiente experimental, limitado em escala, para demonstração dos princípios nos Capítulos 4 e 5.

6.1 Componentes de hardware do ambiente experimental

O ambiente experimental consiste em dois conjuntos de servidores. O primeiro formado por oito máquinas físicas, representando os *nós de computação*, e o segundo composto pelo *controlador A-SDN, controlador da Nuvem, controlador da Rede* e *controlador das Chaves Ópticas*. Além dos servidores, o ambiente experimental contou com um *par de chaves magneto-ópticas 2x2* [58], quatro transceptores de *1 Gbps* e um *switch* físico. A Figura 6.1 ilustra a visão geral do ambiente físico experimental. À esquerda está o *Hipercubo* de três dimensões formados pelos *nós de computação*. Observe que na face frontal do cubo foram removidos os enlaces diretos *000–001* e *100–101*, e introduzido um *par de chaves magneto-ótica* para conectar os nós desta face. Os demais enlaces diretos entre os nós do *Hipercubo* foram mantidos. Próximo ao centro da figura, dois conjuntos de cabos conectam os *nós de computação* ao *switch* físico. O primeiro, na parte superior do *Hipercubo*, conecta cada *nó de computação* a rede de gerenciamento. O segundo estende a rede de dados permitindo aos *nós de computação* acessar os serviços do *controlador da Rede*. Ainda no centro da figura, é ilustrado o *controlador das Chaves Ópticas*, conectado: a rede de gerenciamento pelo *switch* físico e ao *par de chaves magneto-ópticas* do *Hipercubo*. À direta, está o ambiente virtualizado pelo *VMWare ESXi*, que suporta o *controlador da Nuvem, o controlador A-SDN e o controlador da Rede.* As conexões desses controladores com as redes de gerenciamento e de

dados são realizadas, respectivamente, pelas interfaces *NIC 0* e *NIC 1* do servidor físico *Dell PowerEdge R820*. Por fim, o *switch* físico possui uma conexão com a Internet que é compartilhada por todos os elementos conectados a rede de gerenciamento.

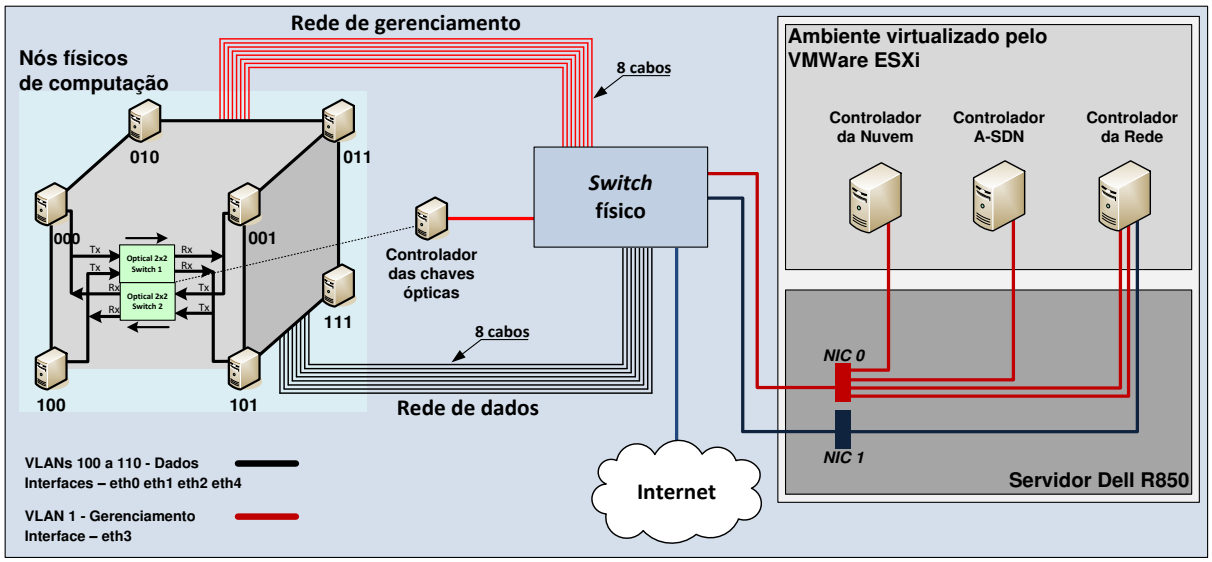


Figura 6.1: Visão geral do ambiente físico experimental.

6.1.1 Switch virtual

Um equipamento *Cisco Catalystic 2960* de 26 portas, sendo as portas de 1 a 24 *Fast Ethernet* e as portas 25 e 26 *Gigabit Ethernet*, foi utilizado para materializar o *switch* físico da Figura 6.1. A rede de dados foi isolada da rede de gerenciamento utilizando o mecanismo de VLAN, suportado por este equipamento. Para isso, as portas foram distribuídas conforme o esquema ilustrado na Figura 6.2, onde a rede de gerenciamento foi alocada nas portas 1 a 8, 21, 23 e 25 (na cor vermelha), enquanto a rede de dados foi alocada nas portas 9 a 16 e 24 (na cor azul). As demais portas (na cor branca) não foram utilizadas. As portas da rede de dados foram configuradas em modo *trunk* para encaminhar somente os pacotes com *tag* de VLAN de 100 a 110, enquanto as portas da rede de gerenciamento foram mantidas na configuração padrão do *switch*.

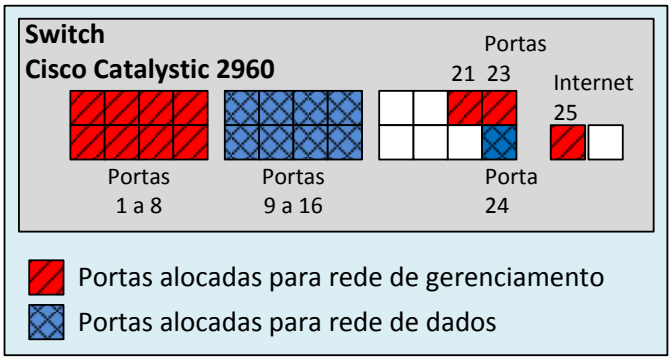


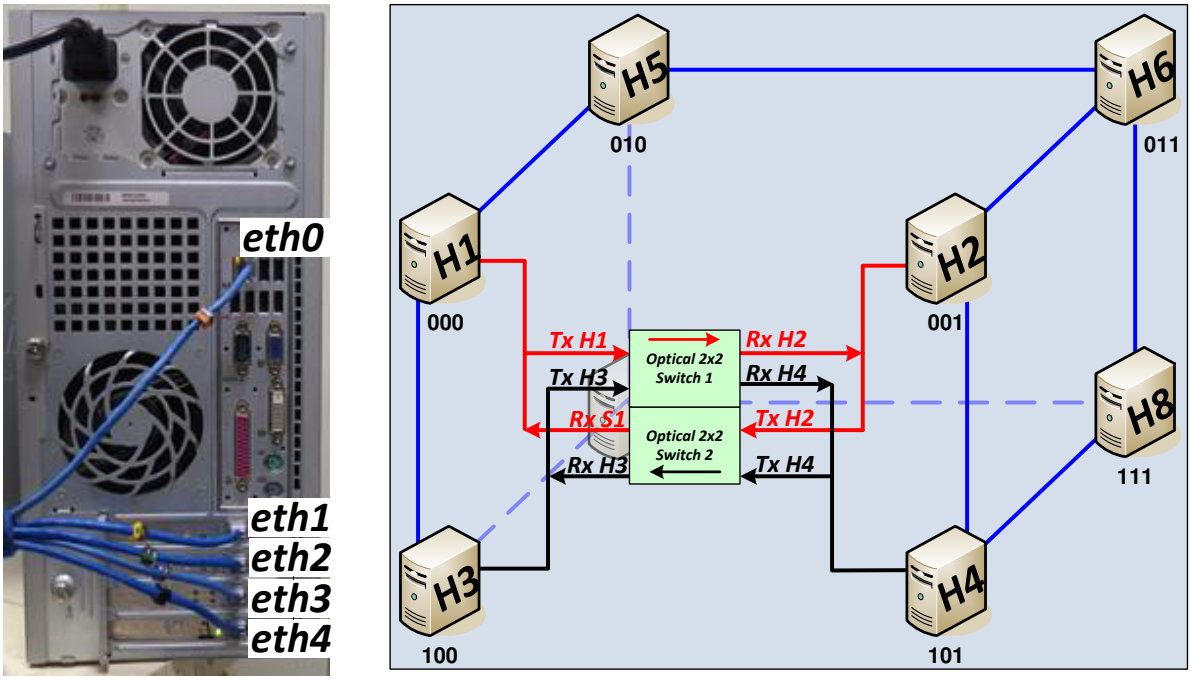
Figura 6.2: Distribuição das portas do *switch* físico

Uma imagem do *switch* físico do ambiente experimental é apresentada na Figura 6.3

Figura 6.3: Imagem do *switch* físico *Cisco Catalystic 2960*

6.1.2 Nós de computação

Para os *nós de computação* foram utilizados oitos *desktops HP Compac dc5750 Microtower com processador AMD Athlon X2 64, 2 núcleos @ 1 GHz, 4 GB de RAM e 80 GB de HDD*. Cada equipamento contou com cinco interfaces de rede *Gigabit Ethernet*, sendo uma *onboard* e quatro *offboard*, conforme a imagem da Figura 6.4(a).



(a) *(b)*

Figura 6.4: Interfaces dos nós físicos de computação

Em cada máquina, a interface *eth3* (*onboard*) foi conectada a *rede de gerenciamento*, enquanto a interface *eth4* foi conectada a *rede de dado*. As interfaces *eth0*, *eth1* e *eth2* foram utilizadas para formar o *Hipercubo* ilustrado na Figura 6.4(b). Na face frontal do *Hipercubo*, foi introduzido um *par de chaves magneto-ópticas 2x2*, para permitir a comutação dos enlaces físicos entre os nós: *H1* e *H2*; *H3* e *H4*, da seguinte forma. Quando as chaves estão no estado *barra*, *H1* se conecta a *H2*, e *H3* se conecta a *H4*. No entanto, quando as chaves são comutadas para o estado *cruz*. *H1* se conecta a *H4*, enquanto *H2* se conecta a *H3*. Isso permite, por exemplo, que um grande fluxo entre *H1* e *H4* não tenha atravessar um servidor intermediário, que não participa da comunicação.

6.1.3 Ambiente de virtualização e controladores A-SDN, da Nuvem e da Rede

O ambiente virtualizado, apresentado na Figura 6.1, foi implementado pelo *software VMware vSphere Hypervisor ESXi* [59], versão 5.5, sobre um servidor físico *Dell PowerEdge R820, dois processadores Intel® Xeon® CPU E5-4603, 8 núcleos @ 2 GHz, 384 GB de RAM, 4 HDDs de 1 TB* cada, configurados em *Raid 10*, duas interfaces de rede *Gigabit Ethernet*.

Para cada controlador (*A-SDN, da Nuvem* e *da Rede)* foi criado um *hardware* virtual básico composto por: *um processador com quatro núcleos, 4 GB de RAM, 64 GB de HDD e uma interface de rede*. De modo a atender a particularidade do *controlador da Rede*, duas novas interfaces de rede virtuais foram adicionadas a ele conforme a Figura 6.5, que ilustra a interconexão das interfaces virtuais do *controlador de Rede*.

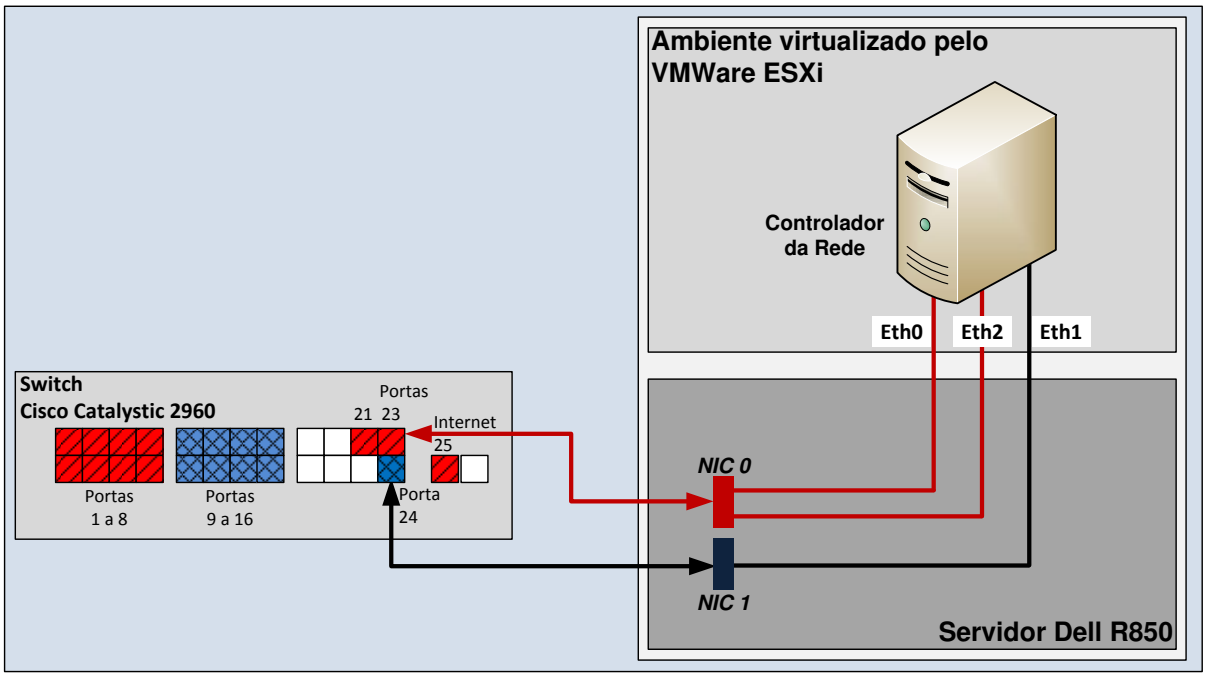


Figura 6.5: Interconexão das interfaces do controlador de redes

Observe na Figura 6.5 que a interface *Eth0, controlador de Rede,* foi mapeada para a interface *NIC 0*, do servidor físico, e conectada a rede de gerenciamento. A interface *Eth1* foi mapeada para a interface *NIC 1* e conectada a rede de dados. A interface *Eth2* foi mapeada para a interface *NIC 0* e utilizada como interface externa para permitir acesso das máquinas virtuais a Internet. Por sua vez as interfaces físicas *NIC 0* e *NIC 1*, do servidor físico, foram conectadas, respectivamente, as portas 23 e 24 do *switch* físico. O *controlador SDN* e o *controlador da Nuvem* tiveram sua única interface mapeada para a interface *NIC 0*, do servidor físico, e conectada a rede de gerenciamento (ver Figura 6.1).

6.1.4 Controlador das chaves ópticas

O *controlador das Chaves Ópticas* foi implementado utilizando um *Beagle Bone Black* [60], que consiste de uma placa de *8,63cm* por *5,33cm*, na qual estão integrados: *um processador de 1GHz, 512 MB de RAM, 2 GB de armazenamento persistente, 1 porta Fast Ethernet*, e diversos conectores de expansão, incluindo uma GPIO de 65 pinos, que foi

utilizada para acionar o circuito das chaves ópticas. A escolha deste equipamento ocorreu pelas seguintes razões: o laboratório já possuía o equipamento; é um dispositivo de baixo custo (US$ 35,00 em 2013); possibilita o empilhamento de placas (*shields*); e principalmente, é compatível com diversas distribuições do sistema operacional Linux, o que possibilitou uma homogeneização da plataforma de *software* da arquitetura.

As *chaves magneto-óptica 2x2* utilizadas foram do modelo YS-1200-155-US [58], que trabalham com os comprimentos de onda entre 1530 a 1565 nm. Além do baixo custo, essas chaves apresentam baixo consumo energético, pois são semi-passivas e requerem uma corrente de 200 mA durante 2 ms apenas para mudança de estado. Seu funcionamento baseia-se em dois terminais elétricos (*t1 e t2*), de modo que, se a corrente é aplicada no sentido de *t1* para *t2*, a chave vai para o estado *barra*, mapeando a *entrada 1* para *saída 1* e a *entrada 2* para *saída 2*. Caso a corrente seja aplicada no sentido inverso, a chave vai para o estado *cruz*, mapeando a *entrada 1* para *saída 2* e a *entrada 2* para *saída 1*. No entanto as portas de E/S do *Beagle Bone Back* não são capazes de fornecer corrente suficiente para alimentar estas chaves. Assim, um circuito de acionamento das chaves foi desenvolvido e acoplado ao *controlador das Chaves Ópticas*. A Figura 6.6 ilustra o esquemático do circuito de acionamento, incluindo o *Beagle Bone Black* e o *par de chaves magneto-ópticas*. Esse circuito consiste em dois CIs (circuitos integrados), sendo um L293D que implementa duas pontes H e um HD74LS04P que implementa seis inversores. Este último foi utilizado por dois motivos. Primeiro, para garantir que as entradas da ponte H nunca estejam em estados proibidos (valores iguais), segundo, minimizar o número de conexões com a placa do controlador.

De forma resumida, o circuito segue a seguinte lógica: o pino *P8.12* do controlador, passa por um diodo de proteção e se conecta ao pino *6A* do *CI* HD74LS04P, que produz uma saída com valor invertido no pino *6Y*. Uma derivação do pino *P8.12* é enviada para o pino *In1* do *CI* L293D (entrada 1 da primeira ponte H), enquanto o pino *In2* do *CI* L293D (entrada 2 da primeira ponte H) recebe o complemento de *In1* ao ser conectado ao pino *6Y* do *CI* HD74LS04P. O pino *Out1* do *CI* L293D (saída 1 da primeira ponte H) se conecta ao terminal *t1* da chave óptica 1, enquanto o pino *Out2* do *CI* L293D (saída 2 da primeira ponte H) se conecta ao terminal *t2* da chave óptica 1. Uma lógica análoga é utilizada para acionar a chave óptica 2.

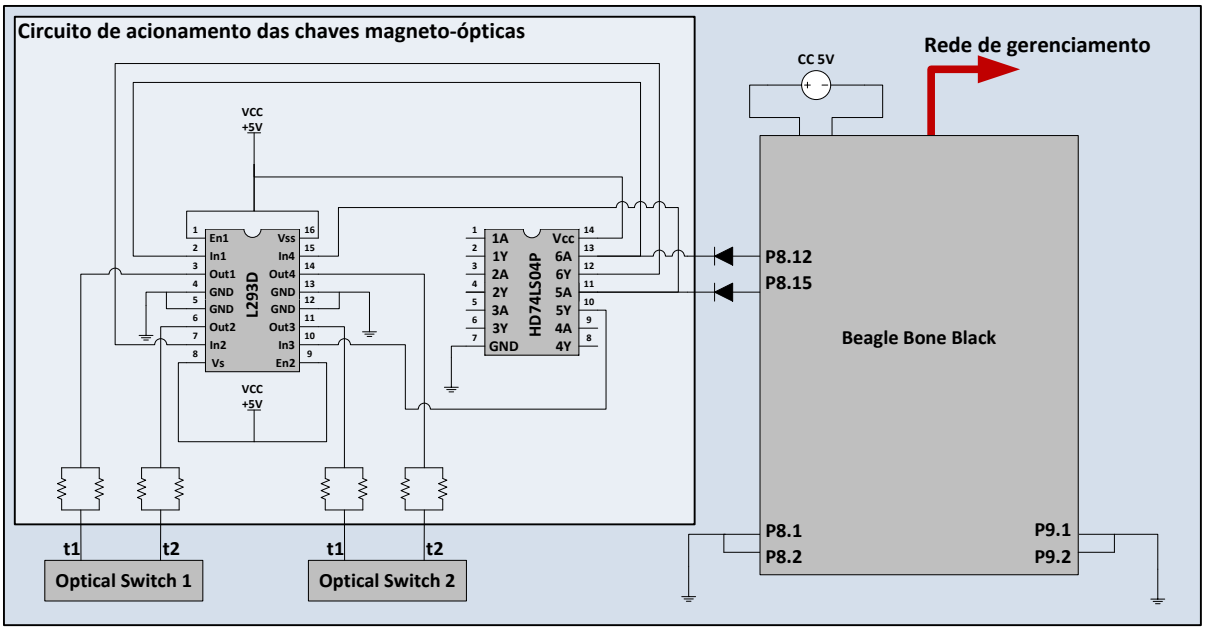


Figura 6.6: Esquemático do circuito de acionamento das chaves ópticas

Com base no esquemático da Figura 6.6, os valores da Tabela 6.1, que representa a tabela verdade do circuito de acionamento das chaves magneto-ópticas, foram preenchidos.

Tabela 6.1: Tabela verdade do circuito de acionamento das chaves.

Beagle Bone Black	*CI* HD74LS04P		*CI* L293D				Terminais da chave	
Chave óptica 1								
P8.12	*6A*	*6Y*	*In1*	*In2*	*Out1*	*Out2*	*t1*	*t2*
1	1	0	1	0	1	0	1	0
0	1	0	0	1	0	1	0	1
Chave óptica 2								
P8.15	*5A*	*5Y*	*In3*	*In4*	*Out3*	*Out4*	*t1*	*t2*
1	1	0	1	0	1	0	1	0
0	1	0	0	1	0	1	0	1

Conforme pode ser observado na Tabela 6.1, quando pino *P8.12* do controlador está em estado 1, a corrente circula do sentido de *t1* para *t2*, fazendo com que a *chave óptica 1* mude para o estado *barra.* Invertendo o valor do pino *P8.12*, a corrente sobre os terminais *t1* e *t2* muda de direção e a *chave óptica 1* é comutada para o estado *cruz.* O mesmo raciocínio vale para a outra chave.

Uma imagem da montagem do *controlador das Chaves Ópticas* do ambiente experimental é apresentada na Figura 6.7.

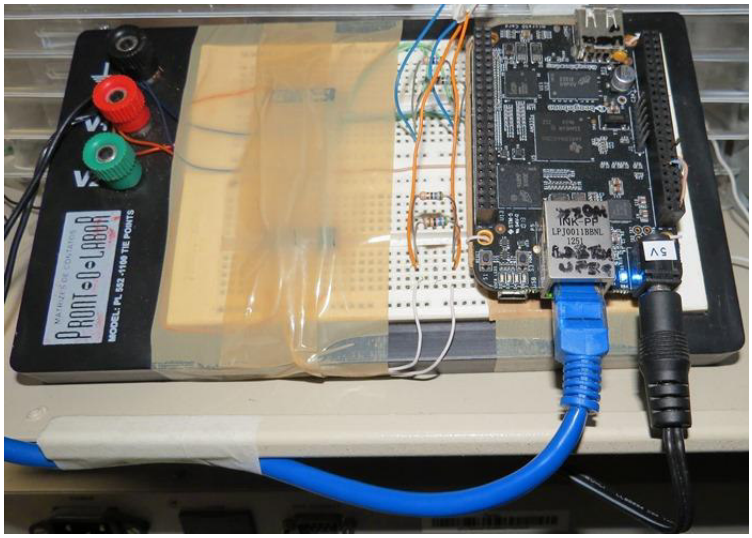


Figura 6.7: Imagem do circuito de acionamento das chaves ópticas

6.2 Componentes de software do ambiente experimental

A implementação da TRIIIAD está ancorada sobre a plataforma de computação em nuvem OpenStack [37], que foi escolhida com base nos seguintes critérios:

1. Obrigatório – ser uma plataforma de código aberto para permitir as modificações necessárias, tais como: a definição do endereço MAC da VM com base no endereço de *Hipercubo* do nó físico;
2. Obrigatório – ser compatível com SDN para permitir a integração com o *controlador A-SDN* projetado;
3. Obrigatório – ser suportada por componentes de *hardware* padrão, possibilitando a implementação nos equipamentos disponíveis no laboratório;
4. Obrigatório – possuir uma documentação completa, possibilitando um amplo entendimento da arquitetura.
5. Obrigatório – ser escalável, possibilitando a implementação em um pequeno grupo de máquinas;
6. Desejável – ser um projeto ativo, pois desta forma ele recebe constantes atualizações e melhoramentos;
7. Desejável – ter o apoio de grandes empresas. Isso demostra o interesse pela plataforma e favorece sua adoção por mais e mais empresas.

8. Desejável – ser implementada por uma linguagem de alto nível, facilitando o entendimento do código para efetuar as alterações necessárias.

A plataforma OpenStack foi projetada como uma ambiente de Infraestrutura como Serviço (IaaS), integrando um conjunto de *software* de forma modular. Seu desenvolvimento segue um cronograma rigoroso onde uma nova versão é liberada a cada 6 meses. Assim, este projeto utilizou a versão *IceHouse*, lançada em 17 de Abril de 2014[1], que na época da montagem do ambiente experimental era a versão estável mais recente. Outros detalhes sobre o OpenStack estão disponíveis no Apêndice C.

O segundo componente de *software* foi o sistema operacional *Ubuntu Server 12.04 LTS* [61]. Ele foi instalado em todos os controladores e nos *nós de computação*. Entre os motivos para a escolha dessa distribuição, podemos destacar:

1. O Ubuntu foi a primeira distribuição Linux a adotar o OpenStack como parte da sua arquitetura, em 2011[2]. Assim, todas as versão do Ubuntu a partir da versão 11.04 possuem os *software* do OpenStack em seus repositórios;
2. O Ubuntu 12.04 LTS é uma versão de suporte de longo prazo, que utiliza apenas pacotes exaustivamente testados que garantem maior estabilidade ao sistema. Ainda, seu suporte se estende até Abril de 2017[3];
3. O Ubuntu 12.04 LTS está disponível para diversas plataformas, incluindo para o *Beagle Bone Black*. Com isso foi possível criar um ambiente homogêneo em relação ao sistema operacional.
4. Por fim, o autor possui experiência no uso desta distribuição.

O intuito nesta seção não é descrever em detalhes os componentes de *software* utilizados, tão pouco apresentar as configurações realizadas para operacionalizar o ambiente. Assim, o restante dessa seção é apresenta um mapeamento entre os serviços apresentados no Capítulo 5 e os componentes de *software* utilizados para efetiva implementação da TRIIIAD. Para facilitar o entendimento, o mapeamento será divido de acordo com a função de cada componente da arquitetura.

[1] https://wiki.openstack.org/wiki/Releases
[2] http://blog.canonical.com/2011/02/03/canonical-joins-the-openstack-community/
[3] https://wiki.ubuntu.com/Releases

6.2.1 Controlador da Nuvem

O *controlador da Nuvem* é elemento que concentra a maior parte dos serviços da plataforma OpenStack. De fato, todos os serviços de gerenciamento da nuvem OpenStack estão instalados nesse servidor. Restando para os *nós de computação* e o *controlador da Rede* as atividades de operacionalização do ambiente, como será apresentado mais adiante. A Tabela 6.2 apresenta o mapeamento dos serviços do *controlador da Nuvem* em serviços do OpenStack. Observe que para alguns serviços, vários componentes do OpenStack são necessários.

Tabela 6.2: Mapeamento dos serviços do *controlador da Nuvem* em componentes do OpenStack.

Serviço	Componentes do OpenStack
Gerenciamento* de *hypervisor	***Servidor Nova:*** fornece uma API de comunicação entre o controlador e os nós de computação, para prover: 1) controle de acesso a base de dados Nova; 2) escalonamento de VMs entre os nós de computação; 3) certificados digitais; 4) gerenciamento de autenticação por *token*; 5) console de acesso remoto via VNC; entre outros.
Gerenciamento de imagens	***Servidor Glance***: fornece uma API para armazenamento e recuperação de imagens de máquinas virtuais, incluindo os metadados a cada imagem.
Autenticação e identidade	***Servidor Keystone***: fornece uma API para gerenciamento e autenticação de usuários, que é utilizada por todos os demais serviços do OpenStack.
Redes virtuais	***Servidor Neutron:*** fornece uma API de *conectividade de rede como serviço* entre as interfaces virtuais gerenciadas pelo OpenStack. É neste serviço que são realizadas todas as configurações relativas às redes virtuais, tais como: segmentos *L2*, redes *L3*, roteadores, etc.
Armazenamento persistente	***Servidor Cinder:*** fornece uma API para de *armazenamento em bloco como serviço*, possibilitando a criação de HDD virtuais que podem ser conectados as VMs.
Painel de operação	***Servidor Horizon:*** fornece uma interface gráfica, via *web,* para todos os serviços OpenStack. Esse serviço fornece aos usuários uma visão de toda a nuvem, possibilitando que um operador possa realizar a maioria das tarefas graficamente.

Além dos componentes do OpenStack, outros serviços foram instalados no *controlador da Nuvem*. Esses serviços foram implementados por *software* do sistema operacional Linux, e estão apresentados na Tabela 6.3.

Tabela 6.3: Mapeamento dos serviços do *controlador da Nuvem* em serviços do Linux.

Serviço	Serviços do Linux
Sincronização de tempo	***Servidor NTP:*** implementa o protocolo de sincronização de tempo em rede (*Network Time Protocolo – NTP)* para sincronizar a hora das VMs da nuvem OpenStack.
Serviço de mensagens	***Servidor RabbitMQ:*** implementa o protocolo de enfileiramento de mensagens avançado (*Advanced Message Queuing Protocol* – AMQP [62]), que é utilizado pelos serviços do OpenStack.

Por fim, este controlador foi utilizado para hospedar a base de dados compartilhada, uma vez que a maioria dos serviços que acessam essa base está instalada nele. Para esta tarefa, foi utilizado o gerenciado de banco de dados MySQL Server, que é indicado na documentação do OpenStack.

6.2.2 Controlador da Rede

No *controlador da Rede* foram instalados os componentes do OpenStack que operacionalizam o funcionamento da rede virtual utilizada pela nuvem, conforme apresentado na Tabela 6.4.

Tabela 6.4: Mapeamento dos serviços do *controlador da Rede* em componentes do OpenStack.

Serviço	Componentes do OpenStack
Autenticação e identidade	***Cliente Keystone***: acessa o serviço de identidade instalado no *controlador da Nuvem*, permitindo que os demais serviços OpenStack possam se autenticar.
Redes virtuais	***Serviço Neutron-Common:*** fornece os serviços básicos do Neutron. Sua instalação é necessária para os *plugins* de operacionalização da rede.
Isolamento em camada 2	***Neutron plugin ML2:*** O ML2 (*Modular Layer 2*) fornece ao Neutron a capacidade de utilizar simultaneamente diversas tecnologias de rede de camada 2, tais como: VLAN, tunelamento GRE, tunelamento VxLAN, etc. Ainda, permite o uso de

Serviço	Componentes do OpenStack
	diversos mecanismos de encaminhamentos, tais como: *Open vSwitch*, Linux Bridge, Arista, etc.
Roteamento em camada 3	***Neutron plugin L3 Agent:*** fornece uma API que possibilita a criação de roteadores em *software* para conectar seguimentos de camada 2 (virtuais e físicos).
Tradução de endereço de rede de camada 3	***Neutron plugin L3 Agent:*** cada roteador virtual criada por esse *plugin* tem a capacidade de realizar NAT em suas portas configuradas como *gateway*.
Configuração automática de parâmetros de rede	***Neutron plugin DHCP Agent:*** fornece o serviço de configuração automática dos parâmetros de rede das VMs de acordo com a configuração das redes virtuais realizada no servidor da nuvem.

O *controlador da Rede* também fez uso de alguns serviços básico fornecidos pelo Linux, conforme mapeado na Tabela 6.5.

Tabela 6.5: Mapeamento dos serviços do *controlador da Rede* em serviços do Linux.

Serviço	Serviços do Linux
Sincronização de tempo	***Cliente NTP:*** sincroniza a hora desse servidor com o servidor da nuvem.
Acesso à base de dados	***Cliente MySql:*** utilizado pelos serviços do OpenStack para acessar a base de dados compartilha.

Conforme descrito no item 5.4, o *controlador da Rede* necessita de um *switch* virtual, para que os serviços do Neutron possam ser executados. Para isso, utilizamos o *software Open vSwitch – OvS* [63], que é o *switch* virtual padrão OpenStack e se integra a serviço de rede Neutron por meio do componente *Neutron Plugin Open vSwitch* e *Neutron Plugin Open vSwitch Agent*. O primeiro define o *Open vSwitch* como mecanismo de encaminhamento do Neutron, enquanto o segundo habilita o Neutron configurar os parâmetros do *Open vSwitch*. Outros critérios que orientaram a escolha do *OvS* foram:

1. Obrigatório – ser uma projeto de código aberto para possibilitar a implementação do algoritmo de roteamento em *Hipercubo*;

2. Obrigatório – ser compatível com SDN para permitir a integração com o *controlador A-SDN* projetado;
3. Obrigatório – ser integrado ao *kernel* do sistema operacional Linux permitindo um encaminhamento mais eficiente.

6.2.3 Nós de computação

Em cada *nó de computação* foram instalados os componentes do OpenStack que efetivamente criam, migram, modificam e destroem máquinas virtuais, bem como os componentes necessários para o funcionamento das redes virtuais, conforme apresentado na Tabela 6.6.

Tabela 6.6: Mapeamento dos serviços dos *nós de computação* em componentes do OpenStack.

Serviço	Componentes do OpenStack
Autenticação e identidade	***Cliente Keystone***: acessa o serviço de identidade instalado no *controlador da Nuvem*, permitindo que os demais serviços OpenStack possam se autenticar.
Virtualização de máquinas	***Servidor Nova-Compute:*** fornece uma API para gerenciar *hypervisors*, tais como: KVM/libvirt, QEMU/libvirt, XenServer/XCP, entre outros.
	Serviço Nova-Common: fornece os serviços básicos do Nova. Sua instalação é necessária para comunicação com o servidor da nuvem.
Redes virtuais	***Serviço Neutron-Common:*** fornece os serviços básicos do Neutron. Sua instalação é necessária para os *plugins* de operacionalização da rede.
Isolamento em camada 2	***Neutron plugin ML2:*** Sua instalação é necessária para os *plugins* relativos ao *Open vSwitch*.
Mecanismo de encaminhamento	***Neutron plugin Open vSwitch:*** define o *Open vSwitch* como mecanismo de encaminhamento do Neutron
	Neutron plugin Open vSwitch Agent: habilita o Neutron configurar os parâmetros do *Open vSwitch*.

Assim, como os demais componentes, os *nós de computação* também utilizaram serviços básicos fornecidos pelo Linux, conforme mapeado na Tabela 6.7.

Tabela 6.7: Mapeamento dos serviços dos *nós de computação* em serviços do Linux.

Serviço	Serviços do Linux
Sincronização de tempo	***Cliente NTP:*** sincroniza a hora desse servidor com o servidor da nuvem.
Acesso à base de dados	***Cliente MySql:*** utilizado pelos serviços do OpenStack para acessar a base de dados compartilha.
Virtualização de máquinas	***Hypervisor KVM/Libvirt:*** fornece o ambiente para criação de máquinas virtuais com aceleração em *hardware*.
	Hypervisor QEMU/Libvirt: fornece o ambiente para criação de máquinas virtuais, porém sem aceleração em *hardware*.

Na Tabela 6.7 é possível observar que foram instalados dois *hypervisors* nos *nós de computação*. Isso foi feito com o propósito de verificar os ganhos de desempenho que a aceleração em *hardware* confere a virtualização de máquinas.

6.2.3.1 Roteamento em Hipercubo entre os nós de computação

Para habilitar o encaminhamento em *Hipercubo* nos *nós de computação*, foi alterado o componente do *Open vSwitch*, conhecido como *datapath*, que faz parte do *kernel* do sistema operacional Linux. Desta forma, o encaminhamento em *Hipercubo* é tratado com prioridade máxima pelo sistema. Especificamente, a implementação foi realizada sobre o código do *Open vSwitch* 2.30 LTS, por se tratar de uma versão de suporte de longo prazo, e implementar os mais recentes protocolos. Os detalhes da implementação realizada podem ser observados no Apêndice A.

6.2.3.2 Monitoramento de carga de CPU, memória e tráfego de trânsito

Os *nós de computação* foram equipados com um módulo de monitoramento de carga, desenvolvido pelo autor, que coleta informações sobre o uso de recursos (CPU, memória e tráfego de trânsito) a cada *1s*. Estas informações são então registradas na base de dados compartilhas, em intervalos de *15s*. Adicionalmente, o módulo de monitoramento de carga compara sua carga média com os limites mínimos e máximos pré-estabelecidos, em intervalos de *30s*, e reporta ao *controlador A-SDN* os eventos que estejam fora desses limites. Com base nas informações reportas pelos *nós de computação* o controlador A-SND realiza uma

orquestração autonômica sobre a TRIIIAD, por meio da reconfiguração dos enlaces ópticos e da migração de máquinas virtuais, conforme explicado na seção 5.6.2.

6.2.4 Controlador das Chaves Ópticas

No *controlador das Chaves Ópticas*, foi instalado uma versão o sistema operacional Ubuntu 12.04 LTS, especialmente compilada para o *hardware* desse controlador e disponibilizada por ARMhf [64]. Ainda, para manter a homogeneidade da arquitetura, foi utilizado o *Open vSwitch* para: realizar a comunicação com o *controlador A-SND* e comandar os pinos que controlam o circuito de acionamento das chaves magneto-ópticas.

A comunicação com o *controlador A-SDN* foi realizada utilizando o protocolo *OpenFlow*, que é nativo na implementação do *Open vSwitch*. Por outro lado, uma nova função foi codificada, em nível de usuário no *Open vSwitch*, para permitir ativar ou desativar os pinos de E/S do *Beagle Bone Black*. Com isso, o *Open vSwitch* foi capaz de gerar sinais elétricos para a entrada do circuito que efetivamente altera o estado de uma ou mais chaves magneto-ópticas simultaneamente.

6.2.5 Controlador A-SDN

O desenvolvimento do *controlador A-SDN* foi realizado sobre a plataforma *Ryu SDN (versão 13)* [36]. Entre os motivos para a escolha dessa plataforma, podemos destacar:

1. O Ryu é uma plataforma de código aberto que permite a extensão do protocolo *OpenFlow*, por meio de novas mensagens;
2. O Ryu suporta as versões mais recentes do protocolo *OpenFlow*, bem como as extensões Nicira que foram exploradas neste trabalho;
3. O Ryu possui uma boa documentação, favorecendo o aprendizado da plataforma.
4. O Ryu utiliza a linguagem de programação Python [65], que suporta programação orienta a objeto.
5. O Ryu é um projeto ativo com frequentes atualizações e melhoramentos;
6. Por fim, é a plataforma adota pelos membros do laboratório onde este trabalho foi realizado.

Beneficiando-se das capacidades do Ryu, o *software* do *controlador A-SND* foi modelado seguindo os preceitos da orientação a objetos, conforme ilustrado no diagrama de classes da Figura 6.8. Uma imagem ampliada desta ilustração pode ser vista no Apêndice B.

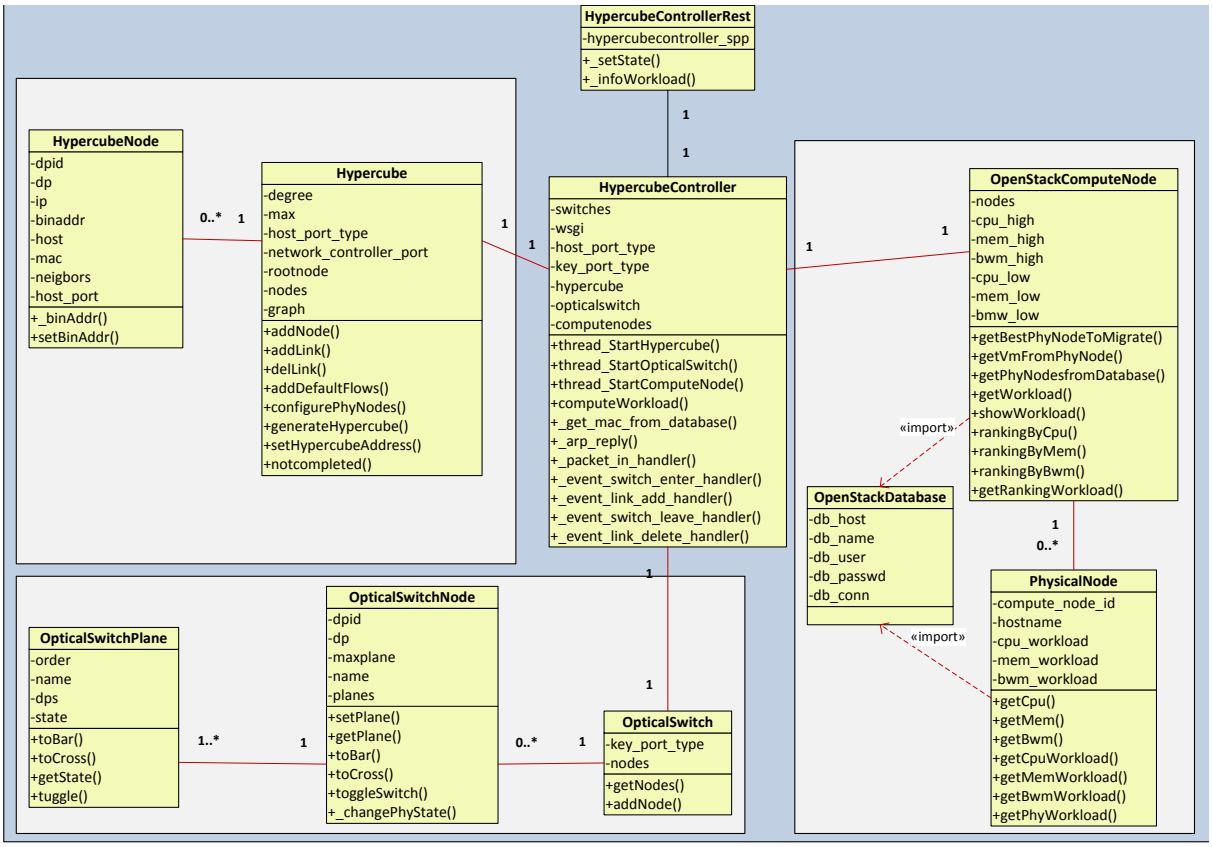


Figura 6.8: Diagrama de classes do *software* do *controlador A-SDN*

No centro do diagrama da Figura 6.8, está a classe principal da aplicação denominada *HypercubeController*. Essa classe é responsável pelas tarefas básicas de um *controlador SDN*, tais como: descoberta de topologia e comunicação com os *switches* via protocolo *OpenFlow* [19]. Para isso, ela herda os atributos e métodos da classe RyuApp, que é uma classe interna da plataforma Ryu (não está representada no diagrama). Além das tarefas básicas, a classe *HypercubeController* realiza as tarefas de inicialização da arquitetura, ARP *replay*, e orquestração da arquitetura:

- A inicialização da arquitetura é realizada pelos métodos: *thread_StartHypercube()* que inicializa um objeto da classe *Hypercube*; *thread_StartOpticalSwitch()* que inicializa um objeto da classe *OpticalSwitch*; e *thread_StartComputeNode()* que inicializa um objeto da classe *OpenStackComputeNode*.

- A tarefa de responder as requisições de ARP é realizada pelos métodos: *_packet_in_handler()* para receber os pacotes de ARP *request* das VMs; *_get_mac_from_database()* para resolver o endereço MAC; e *_arp_replay()* para montar e enviar o pacote de ARP *replay* para as VMs.
- Por fim a tarefa de orquestração é realizada pelo método *computeWorkload()* conforme apresentado no tópico *5.6.2 Orquestração de políticas* na TRIIIAD.

À esquerda da classe *HypercubeController* está classe *Hypercube*, que é responsável pelo gerenciamento do *Hipercubo* formado pelos *nós de computação*. Entre os seus atributos está a lista de objetos da classe auxiliar *HypercubeNode*, que possui os atributos e métodos necessários para representar um nó do *Hipercubo*.

À direita da classe *HypercubeController* está classe *OpenStackComputeNone*, que é responsável pela coleta e ordenação das cargas de trabalho dos *nós de computação* da arquitetura OpenStack. Para isso ela possui diversos métodos para ordenar os nós físicos, representados pelos objetos da classe *PhysicalNode*, por uso de CPU, memória ou tráfego de trânsito. Como esta classe faz uso constante da base de dados compartilhada, ela importa um objeto da classe *OpenStackDatabase* para gerenciar a conexão com o banco de dados MySql Server.

Abaixo da classe *HypercubeController* está a classe *OpticalSwtich*, que é responsável pelo gerenciamento das chaves ópticas da face do *Hipercubo*. Basicamente esta classe é composta de uma lista de chaves ópticas, representada pelos objetos da classe *OpticalSwitchNode*, que possui os atributos e métodos necessários para representar um controlador de chaves ópticas. Observe que cada controlador de chave óptica possui um ou mais planos de chaveamento, representados pelos objetos da classe *OpticalSwtichPlane*. Cada plano de chaveamento representa um plano da dimensão k¸ conforme descrito no tópico 4.2.3 Camada híbrida reconfigurável.

Por fim, o diagrama de classes apresenta a classe *HypercubeControllerRest* que confere a habilidade de receber dados das cargas dos *nós de computação* via API REST *(Representational State Transfer)* [66].

Capítulo 7 – Caracterização do Funcionamento da TRIIIAD

O objetivo desse capítulo é caracterizar o funcionamento da arquitetura em função dos recursos (CPU e memória) dos *nós de computação*, do ambiente experimental construído em nosso laboratório, de acordo a metodologia apresentada na seção abaixo.

7.1 Metodologia de caracterização do ambiente

A metodologia proposta visa avaliar as características da TRIIIAD, em especial, as implicações do roteamento em *Hipercubo* e do uso de uma camada óptica reconfigurável sobre o encaminhamento do tráfego da rede, realizado pelos servidores. Por outro lado, essa metodologia também avalia como o OpenStack se encaixa nessa arquitetura. Devido a diversidade de cenários avaliados, a caracterização da arquitetura foi dividida em três fases:

- *Fase 1 – Caracterização das operações na nuvem*. Avaliação do uso dos recursos dos *nós de computação* para atender demandas de criação e migração de VMs na nuvem OpenStack;
- *Fase 2 – Caracterização dos cenários de tráfego.* Avaliação do uso dos recursos dos *nós de computação* sobre cenários de tráfego sintetizados, sem a utilização da reconfiguração dos enlaces ópticos; e
- *Fase 3 – Caracterização dos impactos da comutação óptica*. Avaliação dos benefícios que podem ser alcançados com a comutação óptica, realizada de forma manual, em função dos recursos dos *nós de computação*.

Para garantir maior confiabilidade dos resultados, foi desenvolvido um conjunto de *scripts shell* para permitir a condução semiautomática dos experimentos. Com isso, a maioria dos amostrou *30* curvas, que foram estatisticamente processadas para produzir os resultados que serão apresentados ao longo deste capítulo. Especificamente, foi calculado a mediana ($\bar{x}$),

o desvio padrão amostral (s) e o erro padrão amostral ($\sigma/\sqrt{n}$), e estimamos o intervalo de confiança de 95%, baseado na tabela da distribuição *Normal* de probabilidade, utilizando a Equação (1). Entretanto, para não poluir os gráficos, os intervalos de confiança não foram plotados.

$$\left(\bar{x} - 1,96 \times \frac{\sigma}{\sqrt{n}}, \bar{x} + 1,96 \times \frac{\sigma}{\sqrt{n}}\right) \quad (1)$$

Os *scripts shell* se valeram da sincronização dos relógios dos *nós de computação* e das máquinas virtuais com o *servidor da Nuvem*, para iniciar e finalizar cada experimento. Isso permitiu simular cenários de tráfego e coletar o uso de recursos, simultaneamente, das diversas máquinas (físicas ou virtuais) participantes de cada experimento, com uma precisão de um segundo. A geração dos cenários de tráfegos simulados foi realizada com o auxílio da ferramenta *Iferf* [53]. Por outro lado, a coleta de dados utilizou duas ferramentas. O monitoramento de CPU e memória foi realizado pela ferramenta *sar*, que é parte do pacote *sysstat* [67] de monitoramento de desempenho do sistema Linux. Enquanto o monitoramento do tráfego de rede foi realizado pela ferramenta *bwm-ng* (*Bandwidth Monitor NG*) [68], por ser um programa utilizado em diversos trabalhos, tais como: *Performance issues in clouds* [69] e *Resource usage monitoring in clouds* [70], por apresentar um ótimo desempenho e fornecer os dados em um formato adequado para calcular o tráfego de trânsito nos *nós de computação*, conforme será descrito neste capítulo. Por fim, visando uma melhor organização, a descrição dos experimentos, seus resultados e discursão foram divididos em seções de acordo com as fases da metodologia aqui apresentada.

7.2 Fase 1 – Caracterização das operações na nuvem

Durante a funcionamento do *data center* é comum a ocorrência de operações para criar e/ou migrar VMs na nuvem. Assim, o objetivo dessa fase da metodologia foi entender como o OpenStack realiza essas operações e quais os seus impactos sobre os recursos dos *nós de computação* e sobre a rede. Para isso, foram realizados experimentos para avaliar: o tempo consumido, a alocação de memória, o consumo de CPU e tráfego gerado na rede durante a criação/migração de máquinas virtuais sobre o ambiente experimental, de acordo com os cenários das subseções a seguir.

7.2.1 Criação de máquinas virtuais na nuvem

Neste cenário, o *controlador da Nuvem* utilizou um *script shell* para instanciar uma VM a cada n segundos. Para avaliar com mais clareza o impacto da criação de uma única VM, o *script* do controlador foi parametrizado para lançar, a cada *60s*, uma VM com *512 MB de RAM, 1 CPU virtual (VCPU), 5 GB de disco, 1 interface de rede,* utilizando uma imagem de *1,5 GB* do sistema operacional *Linux Ubuntu 14.04*. Do outro lado, cada *nó de computação*, por meio de um *script shell de monitoramento,* registrou o uso seus recursos (CPU e memória) e do tráfego de rede em suas interfaces em sincronia com processo de criação das VMs. Da analise dos dados coletados, foram plotados os gráficos da Figura 7.1, que ilustram o comportamento dos *nós de computação*, ou simplesmente *Hosts,* identificados na legenda desta e das demais figuras como *H1 a H8*, em função do uso de CPU e memória, e o tráfego rede gerado para atender a demanda de criação de VMs.

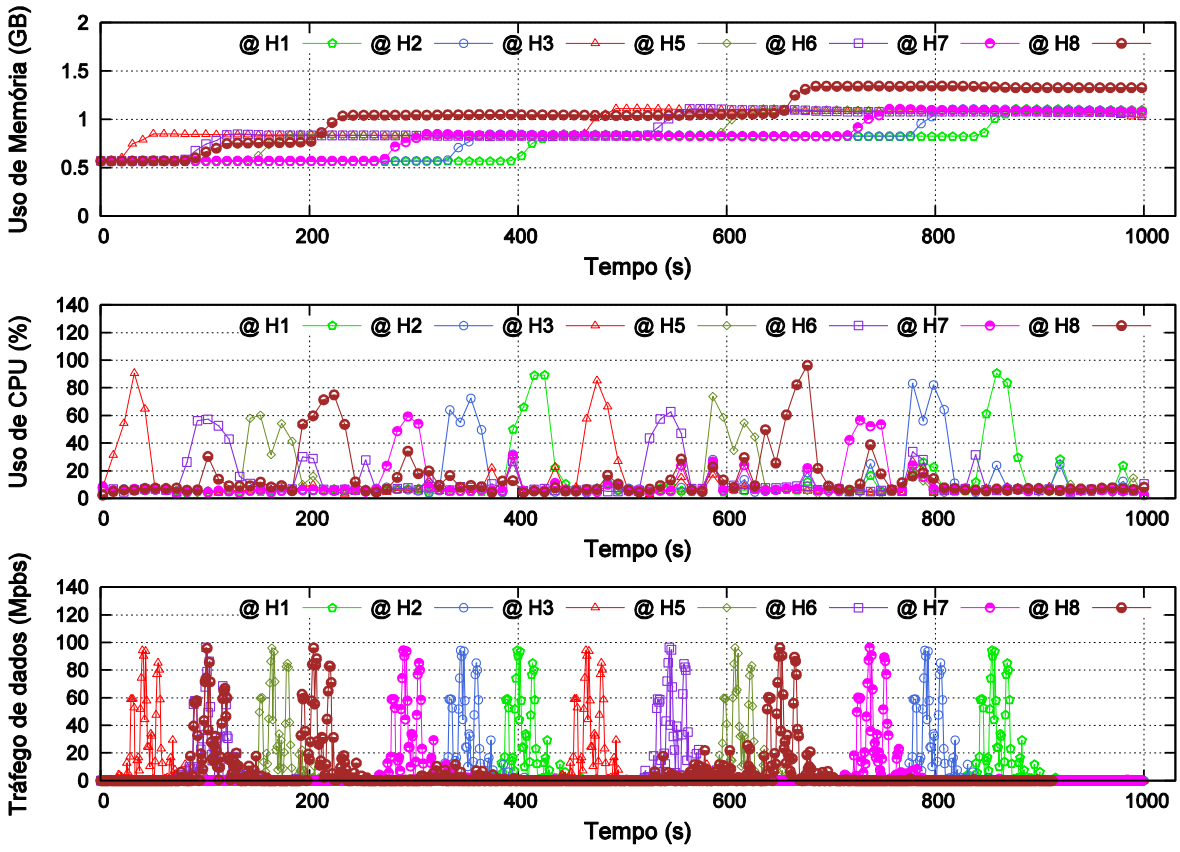


Figura 7.1: Criação de máquinas virtuais na nuvem em função dos recursos dos *Hosts* (*nós de computação*) e da rede.

O gráfico na parte superior da Figura 7.1 apresenta o aumento do uso de memória dos *nós de computação* à medida que as VMs são alocadas nos *Hosts*. A curva foi suavizada para favorecer a visualização dos degraus. Apesar das VMs terem sido configuradas para utilizar

512 MB de RAM, o *hypervisor* aloca a memória de acordo com a necessidade da cada VM, limitando ao valor que foi configurado. Por essa razão, cada degrau representa um incremento em torno de *250 MB de RAM*. Ainda é possível observar que, no início do experimento, que algumas VMs foram instanciadas praticamente juntas, apesar de terem sido lançadas em intervalos regulares de *60s*. Esse comportamento se deve ao escalonador do OpenStack, que enfileira as requisições, durante o processo de ordenação para escolha do nó físico mais adequado, e dispara simultaneamente as requisições enfileiradas, assim que a escolha dos nós foi finalizada. Ainda em relação ao escalonador do OpenStack, observou-se que a quantidade de memória física instalada no *nó de computação* é prioritária, em relação a CPU. Tanto que o *Host H4* não recebeu nenhuma VM, por estar com *1 GB de RAM* a menos que os demais. O gráfico na parte intermediária da Figura 7.1 mostra o comportamento das CPUs dos *nós de computação*, que atingem picos entre 60% e 100% durante a instanciação de uma VM. Como nenhum processo é inicializado junto da VM, após a sua criação, não há demanda significativa da CPU dos *nós de computação*. Por fim, o gráfico na parte inferior da Figura 7.1 apresente o comportamento da rede durante o processo de criação. Devido ao fato da imagem do sistema operacional estar fisicamente armazenada no servidor da Nuvem, a transferência dessa imagem foi realizada pela rede de gerência, que está conectada por meio de um *switch* físico com portas de *Fast Ethernet*. O quê explica o limite de *100 Mbps* do tráfego de dados. É importante notar que para uma rede com milhares de máquinas, o uso da rede de gerenciamento para trafegar os arquivos de imagem das VMs pode se tornar um gargalo durante o processo de criação de um grande conjunto de VMs.

Para demostrar como os tempos de criação de VMs foram distribuídos neste cenário, foi elaborado o gráfico da Figura 7.2, que ilustra a função de distribuição acumulada (CDF – *Cumulative Distribution Function*) dos tempos de criação das máquinas virtuais.

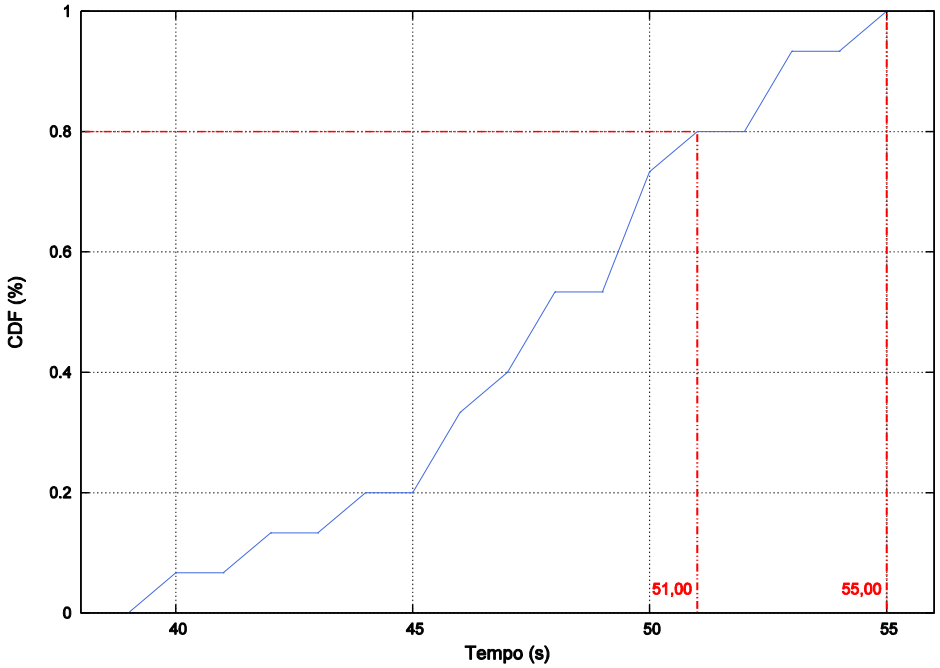


Figura 7.2: CDF dos tempos de instanciação de VMs. O detalhe mostra que 80% das VMs são instanciadas até o tempo de 51s.

Os tempos de instanciação das VMs apresentados na Figura 7.2 foram coletados pelo *controlador da Nuvem*. Assim eles representam a duração entre a requisição e a inicialização do sistema operacional das VM. É possível observar que 100% das VMs foram instanciadas em até 55s, sendo 80% delas foram instanciadas em até 51s. Boa parte deste tempo foi devido à transferência da imagem do servidor da Nuvem para o *nó de computação*. Nos cálculos realizados, aproximadamente 28s, o que representa uma taxa de transferência efetiva média de 13,21 *Mbps* sobre a rede vazia.

Para melhor avaliar o comportamento do escalonador OpenStack em uma arquitetura heterogênea, removeu-se todas as VMs alocadas anteriormente. Então o *Host H4* foi substituído por um servidor físico com mais processamento e memória (*Dell PowerEdge T310, 01 processador Intel® Xeon® CPU X3440, 4 núcleos @ 2.53 GHz, 24 GB de RAM, 1 HDD de 2 TB*). Os experimentos foram repedidos e os dados plotados nos gráficos da Figura 7.3. Devido ao interesse no comportamento do escalonador OpenStack, apenas os gráficos de consumo de memória e CPU foram apresentados na Figura 7.3.

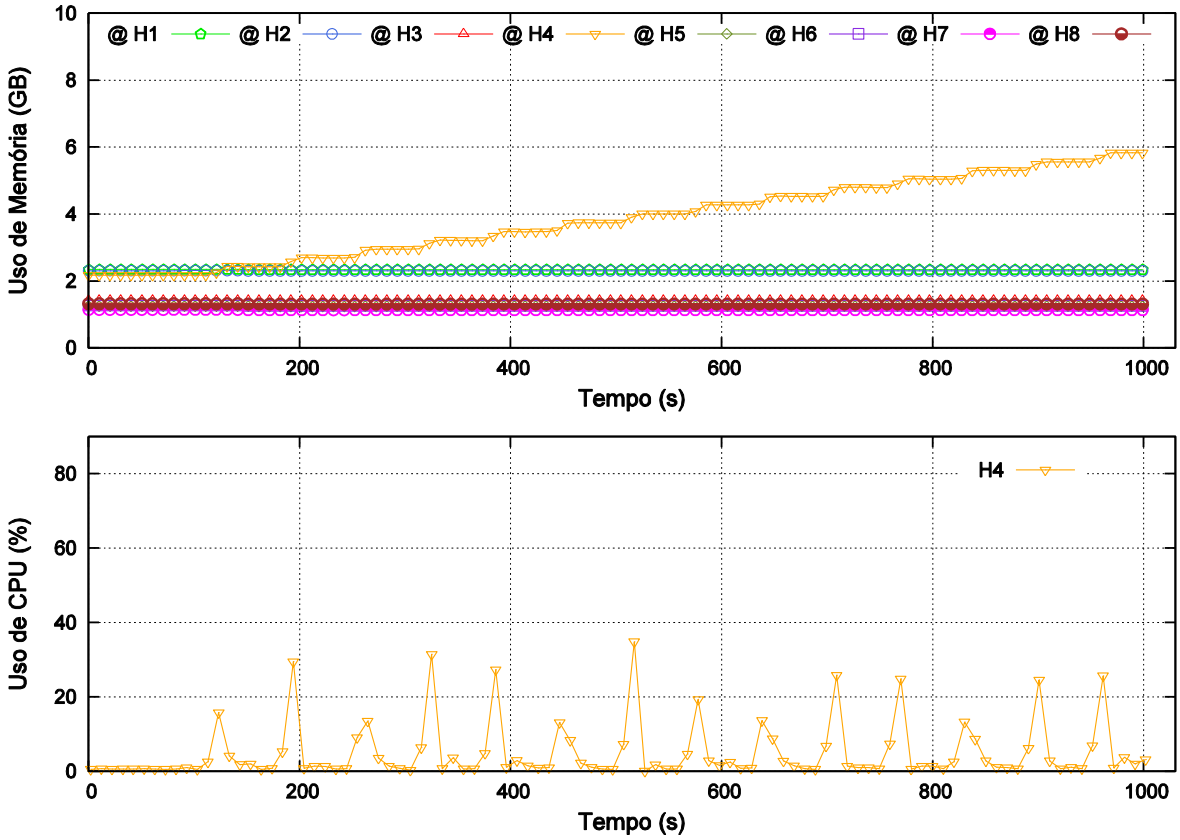


Figura 7.3: Criação de máquinas virtuais na nuvem, com a substituição do *Host H4* por outro servidor com mais poder de processamento e memória.

Tal como no experimento anterior, o gráfico de alocação de memória da Figura 7.3 mostra que o escalonador do OpenStack priorizou a alocação de VMs no *Host H4* que possui mais memória física em relação a seus pares, alocando todas as 15 VMs em *H4*. A fim de definir o ponto no qual o escalonador escolheria outro *Host* para alocar VM. O número de requisições foi elevado e observou-se que apenas quando a memória física do servidor *Host H4* foi totalmente esgotada, uma nova VM foi direcionada para outro servidor. Isso mostra que, em um ambiente heterogêneo, o escalonador OpenStack distribui as VMs entre os *Hosts* proporcionalmente a sua memória física, não se importando com o carga de CPU dos *nós de computação*. Por outro lado, o gráfico de uso de CPU da Figura 7.3 mostra um uso de CPU percentualmente menor quando comparado ao primeiro experimento. Isso era esperado devido a maior capacidade de processamento do novo servidor. No entanto, é importante notar que em mais da metade das instanciações, o uso de CPU foi acima de 20%. Isso mostra que a CPU é um requisito muito demandado durante o processo de criação da VMs, podendo interferir no processamento de dados das VMs em execução, mesmo que por um curto período de tempo. Portanto, ao direcionar muitas VMs novas para um nó com grande capacidade de memória, o escalonador do OpenStack penaliza as VMs que demandam muita

CPU. Assim, um critério mais justo de distribuição das VMs deveria levar em conta as cargas médias dos *nós de computação* em relação a CPU, memória e, para o caso das redes de *data center* centrados em servidores, o tráfego de trânsito.

7.2.2 Migração de máquinas virtuais na nuvem

Neste cenário, o novo *Host H4* foi carregado com 30 VMs, cada uma com de *512 MB de RAM, 1 CPU virtual (VCPU), 5 GB de disco, 1 interface de rede,* utilizando uma imagem do sistema operacional *Linux OpenWrt* [71], de apenas de *13Mb* . Esta imagem foi escolhida, não por acaso, mais por ser uma distribuição específica para roteadores de pequeno porte, muito otimizada em relação ao uso de CPU e memória (a especificação mínima é um *hardware* com *16 MB de RAM*). Ainda, apresenta uma vasta gama de *software* para análise e monitoramento de rede, ideal para o propósito desse trabalho.

A condução desse experimento ocorreu da seguinte forma. No *controlador da Nuvem*, um *script shell* solicitava a migração de uma VM e aguardava o seu término, para então migrar outra VM, permitindo avaliar o impacto da migração de uma única VM. Da mesma forma que no cenário anterior, foi utilizado o *script shell de monitoramento* em cada *nó de computação* durante o processo de migração das VMs. O comportamento deste cenário foi divido entre duas figuras. A Figura 7.4 apresenta o gráfico de uso de memória em detalhes, enquanto a Figura 7.5 apresenta os gráficos de uso de CPU e tráfego da rede.

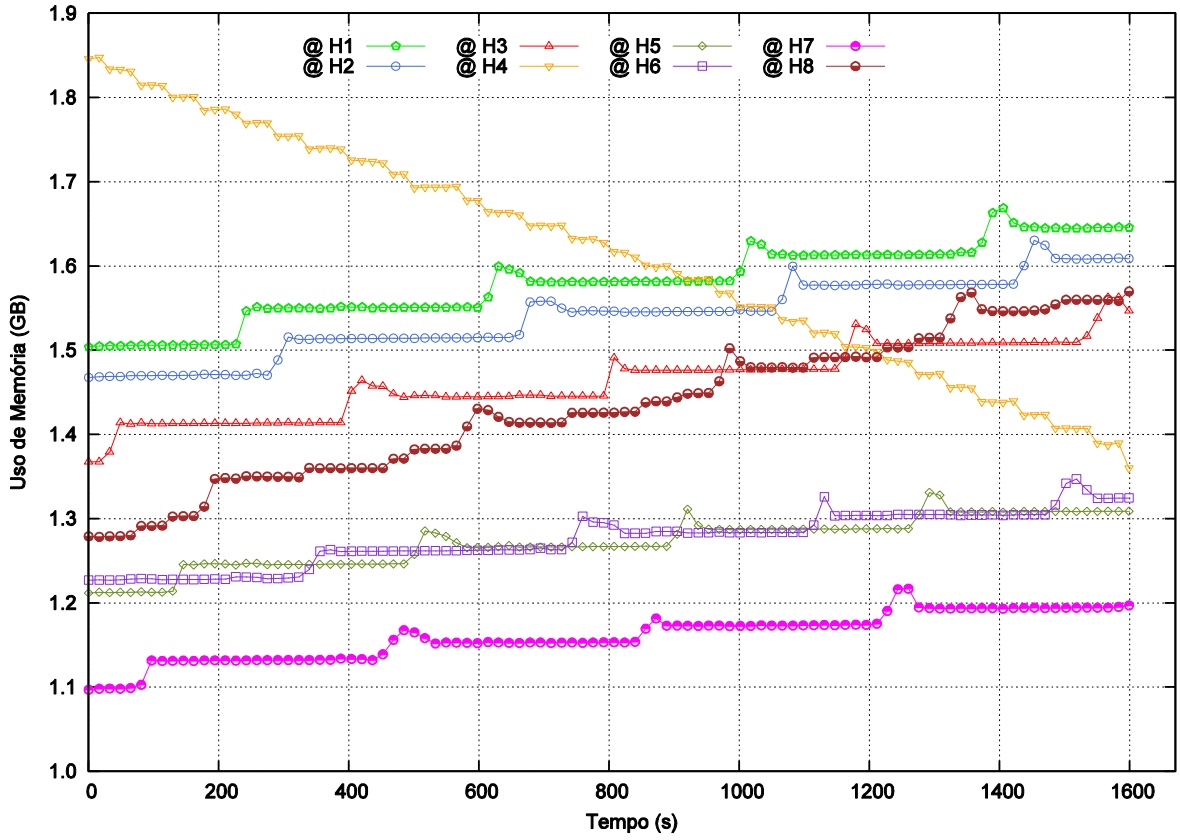


Figura 7.4: Uso de memória durante o processo de migração das VMs do *Host H4* para os demais *Hosts*.

O gráfico da Figura 7.4 apresenta o uso de memória dos *Hosts* à medida que as VMs são migradas de *H4* para os demais *Hosts*. Novamente, a curva foi suavizada para favorecer a visualização dos degraus. Para estas VMs, cada degrau representa um incremento médio de *28 MB,* alocados pelo *hypervisor,* bem abaixo do limite de *512 MB* configurado para o *hardware* virtual. Note que enquanto o uso de memória de *H4* diminui, os demais *Hosts* alocam mais memória para suportar as VMs que estão chegando. Ainda, por se tratar de uma VM muito compacta, a migração ocorre em menos de 35s (ver Figura 7.8 para detalhes). Neste experimento, o escalonador OpenStack distribuiu melhor a carga entre os *nós de computação*, pelo fato de todos os nós disponíveis para migração possuírem a mesma configuração. No entanto, notou-se que mesmo os HDs virtuais estando fisicamente nos *nós de computação*, o processo de migração do OpenStack utilizou a rede de gerenciamento para mover as VMs entre os *Hosts*.

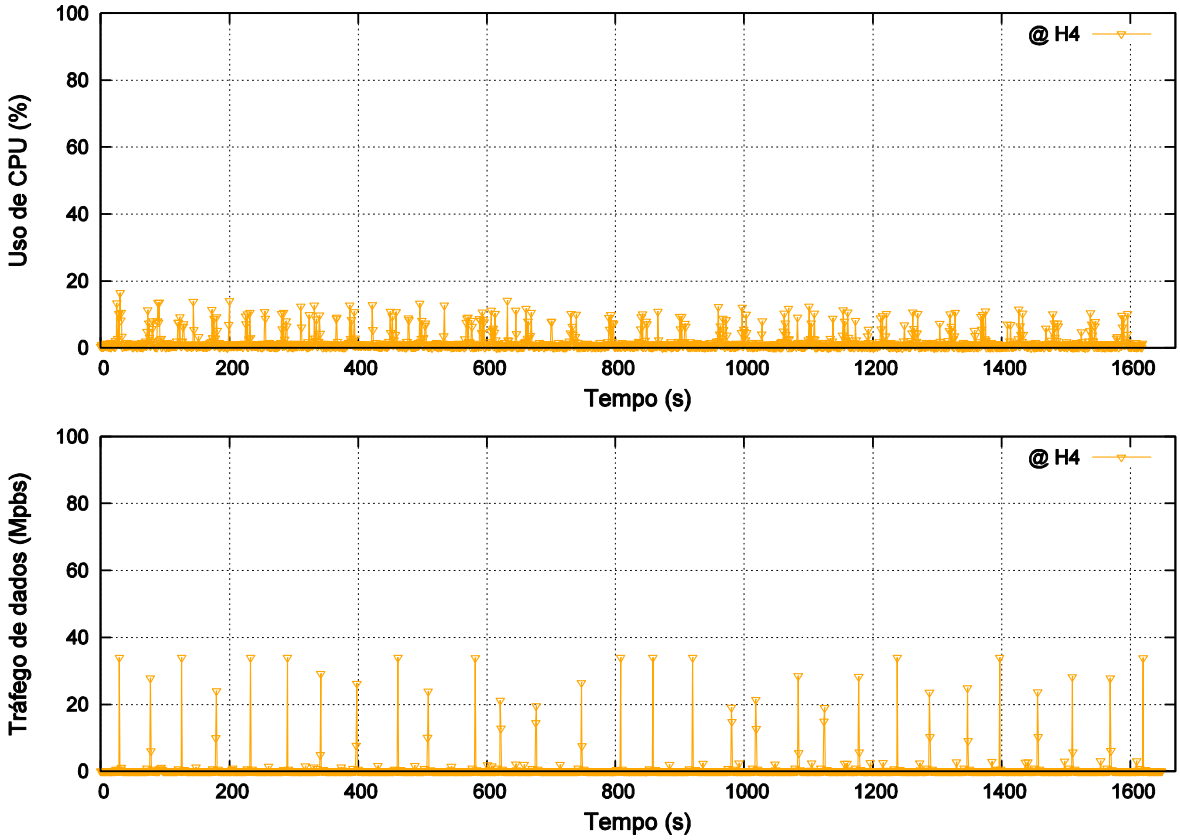
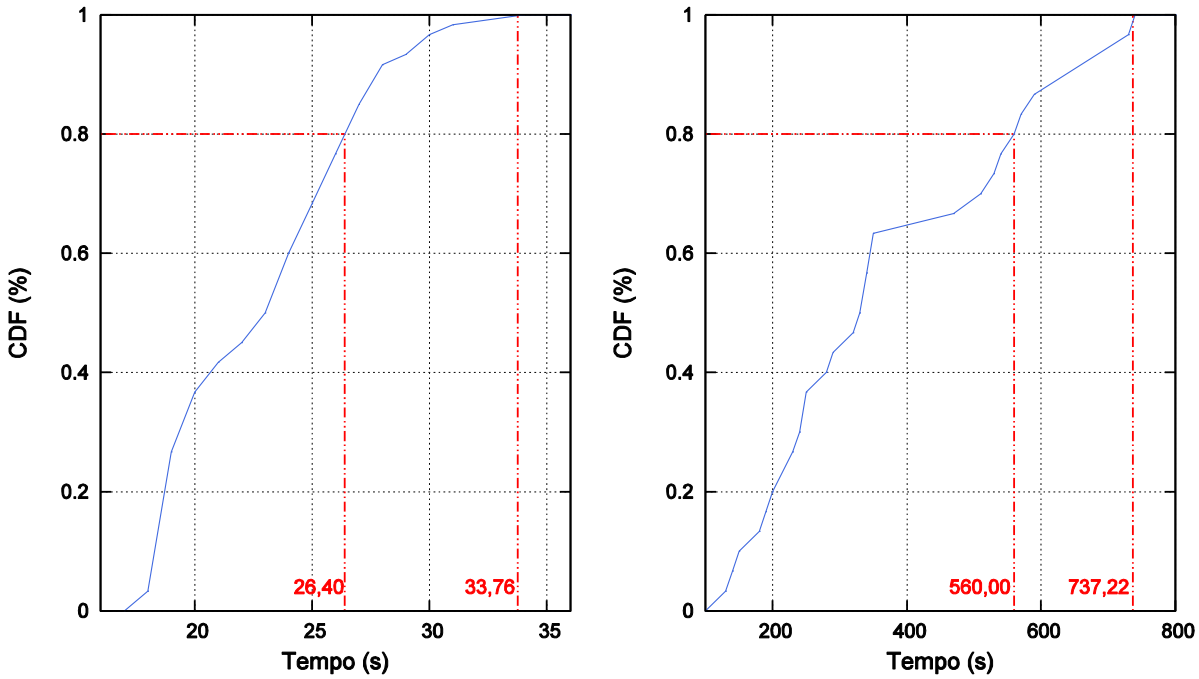


Figura 7.5: Uso de CPU e tráfego de rede durante o processo de migração das VMs do *H4* para os demais *Hosts*.

Os gráficos da Figura 7.5 apresentam o uso de CPU do *Host H4* e o tráfego de rede observado em sua interface de acesso a rede de gerenciamento. O uso de CPU manteve-se abaixo de 20% durante todo o processo, devido aos modestos requisitos do sistema operacional utilizado nas VMs. Por sua vez, o tráfego de dados foi realizado em pequenas rajadas, que não ultrapassaram a taxa de *40 Mbps*. Por sua vez, o tráfego total observado durante o período, atingiu o montante de *1,16 GB*, em média *38 MB* por VM. Esse tráfego é condizente com o tamanho de cada VM em disco, que gira em torno de *30 MB*, uma vez que foi observado o tráfego bruto, em nível de camada 2.

Uma forma de contornar a limitação de banda do processo de migração do OpenStack é modificando seu algoritmo para que encaminhe os HDs virtuais pela rede de dados, e utilize a rede de controle apenas para os metadados das operações. Por outro lado, o mecanismo de reconfiguração da camada física proposta neste trabalho, poderia ser utilizado, não para aliviar o tráfego da rede de controle, mais para aliviar os *nós de computação* que estão participando de uma grande migração, minimizando o tráfego de trânsito da rede de dados sobre eles.

Para finalizar a *Fase 1*, foi comparado os tempo de migração de VMs, em função do seu tamanho em disco. Para isso, o cenário desse experimento foi repetido utilizando a VM Ubuntu do primeiro experimento. Com os dados coletados, foram calculadas as funções de distribuição acumulada para cada tipo de VM e plotados os gráficos da Figura 7.8.



(a) OpenWRT: 512MB RAM, 1 VCPU, 3GB HD, ocupando 30MB em disco

(b) Ubuntu: 512MB RAM, 1 VCPU, 5GB HD, ocupando 1.9GB em disco

Figura 7.6: CDF dos tempos de migração de VMs. (a) CDF para migrar uma VM OpenWrt que ocupa 13 MB em disco. (b) . CDF para migrar uma VM Ubuntu que ocupa 1,9 GB em disco.

Assim como os tempos de criação, os tempos de migração das VMs apresentados na Figura 7.8 foram coletados pelo *controlador da Nuvem*. A Figura 7.8(a) mostra que todas as VMs OpenWrt, que ocupam apenas 30 MB em disco, foram migradas em até 33,76s, sendo 80% delas iniciadas em até 26,40s. Por sua vez, as VMs Ubuntu, que ocupam 1,9 *GB* em disco, foram todas migradas em até 737,22s, sendo 80% delas iniciadas em até 560,00s, conforme apresentado na Figura 7.8(b). Isso demonstra que existe uma relação direta entre o tamanho em disco da VM (ou da imagem no caso da criação) com o tempo de migração (ou criação). Calculando as razões entre os tempos para 80% e 100% das distribuições, verifica-se que os tempos de migração das VMs OpenWrt foram, respectivamente, *21,21* e *21,83* vezes menores que os das VMs Ubuntu. Por outro lado, calculando a razão entre os tamanhos em disco das VMs, verifica-se que as VMs OpenWrt são cerca de *63* vezes menores que as VMs

Ubuntu. Essa diferença entre as razões, de tempo e tamanho, pode ser explicada pelo fato de que parte do tempo de migração é para transportar o HD virtual pela rede, enquanto a outra parte é para efetivamente criar o *hardware* virtual. De forma que o tamanho somente influencia no tempo de transporte e não no tempo de criação do *hardware* virtual no *Host* de destino.

7.3 Fase 2 – Caracterização dos cenários de tráfego

O objetivo da *Fase 2* foi entender como os recursos dos *nós de computação* foram afetados pelo trabalho extra dos servidores de rotear e encaminhar o tráfego das VMs pela rede de dados. Ainda, objetivou-se entender o funcionamento da rede virtual criada pelo OpenStack, seu desempenho e seus impactos sobre os *nós de computação*. Para isso, foram criados experimentos para avaliar: o consumo de CPU e a vazão da rede sobre os cenários de tráfego sintetizados, sem a utilização da reconfiguração dos enlaces ópticos.

Todos os cenários de tráfego desta seção utilizaram a VM OpenWrt, apresentada na seção anterior, visando minimizar os recursos necessários para sustentar uma VM e maximizar a disponibilidade de recursos para as tarefas de rede.

7.3.1 Encaminhamento de tráfego entre VMs no mesmo Host físico e na mesma rede virtual

No cenário apresentado na Figura 7.7(a), os *nós de computação* destacados receberam duas VMs (*VM1* e *VM2)* cada*,* conectadas pela mesma rede virtual. Em seguida foi carregado em cada *VM1* o *script shell gerador* e em cada *VM2* o *script shell monitor*. Assim, as *VM1* foram utilizadas para gerar tráfego, utilizando a ferramenta *Iperf*, enquanto as *VM2* foram utilizadas para monitorar o uso de CPU e o tráfego de rede em suas interfaces, conforme Figura 7.7(b). Em seguida, todos os *scripts* foram parametrizados com: horário de início, *30* repetições, *60*s de duração por repetição, além dos endereços IP de cada VM.

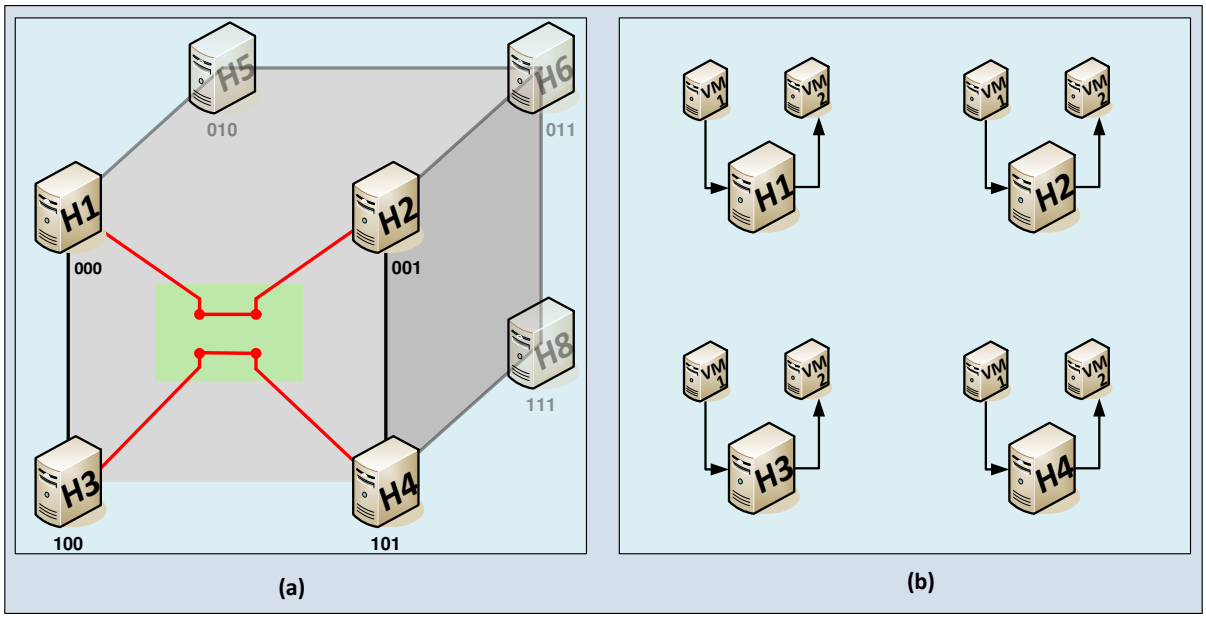


Figura 7.7: Cenário de teste de encaminhamento de tráfego entre VMs no mesmo *Host* e na mesma rede virtual: (a) Destaque dos *Hosts* participantes na face do cubo. (b) *VM1* enviando tráfego para a *VM2* por intermédio do *Host* que as hospeda.

O *script shell de monitoramento* foi utilizado em cada *nó de computação*, conforme descrito nos cenários anteriores. Ao executar os *scripts*, foram gerados 30 amostras por VM e por nó físico, perfazendo um total de 120 amostras para cada. Estas amostras foram tratadas estatisticamente, resultando nos gráficos ilustrados na Figura 7.8.

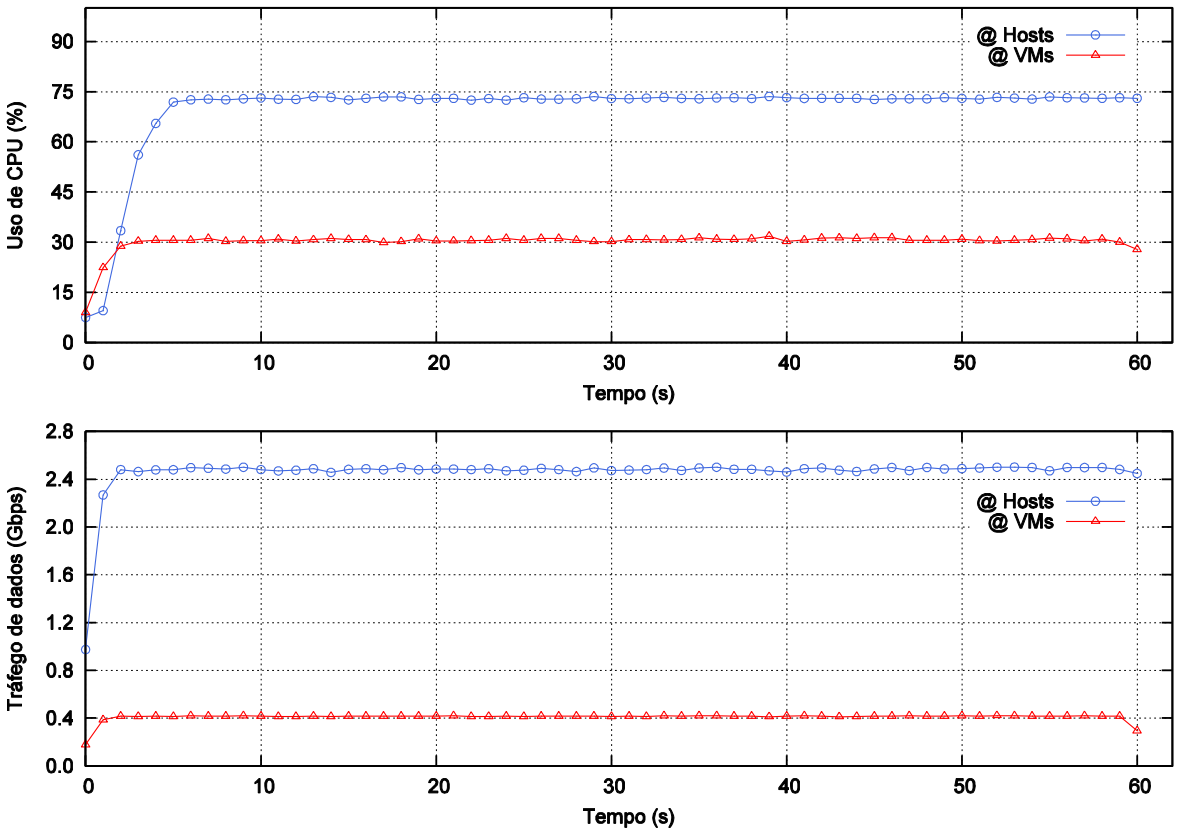


Figura 7.8: Tráfego de dados e uso de CPU registrados durante a comunicação entre VMs no mesmo *Host* e na mesma rede virtual.

O gráfico da parte inferior da Figura 7.8 apresenta a tráfego de dados médio registrado nos experimentos. Observe que, enquanto a taxa média do fluxo entre as VMs foi de aproximadamente *0,4 Gbps*, os *Hosts* registraram uma taxa média *6* vezes maior (aproximadamente *2,4 Gbps*). De forma semelhante, o uso de CPU nos *Hosts* foi de *5* vezes maior (aproximadamente *75%*), quando comparado com o uso de CPU observado pelas VMs (aproximadamente *15%*), conforme o gráfico de uso de CPU da Figura 7.8. Para explicar esse comportamento é necessário entender como o *OpenStack* organiza a rede interna dos *Hosts,* para criar o isolamento entre as redes virtuais, utilizando o mecanismo de VLAN. Para isso, será utilizada a ilustração da Figura 7.9, que foi adaptada da documentação do OpenStack [72] para corresponder ao ambiente experimental deste trabalho.

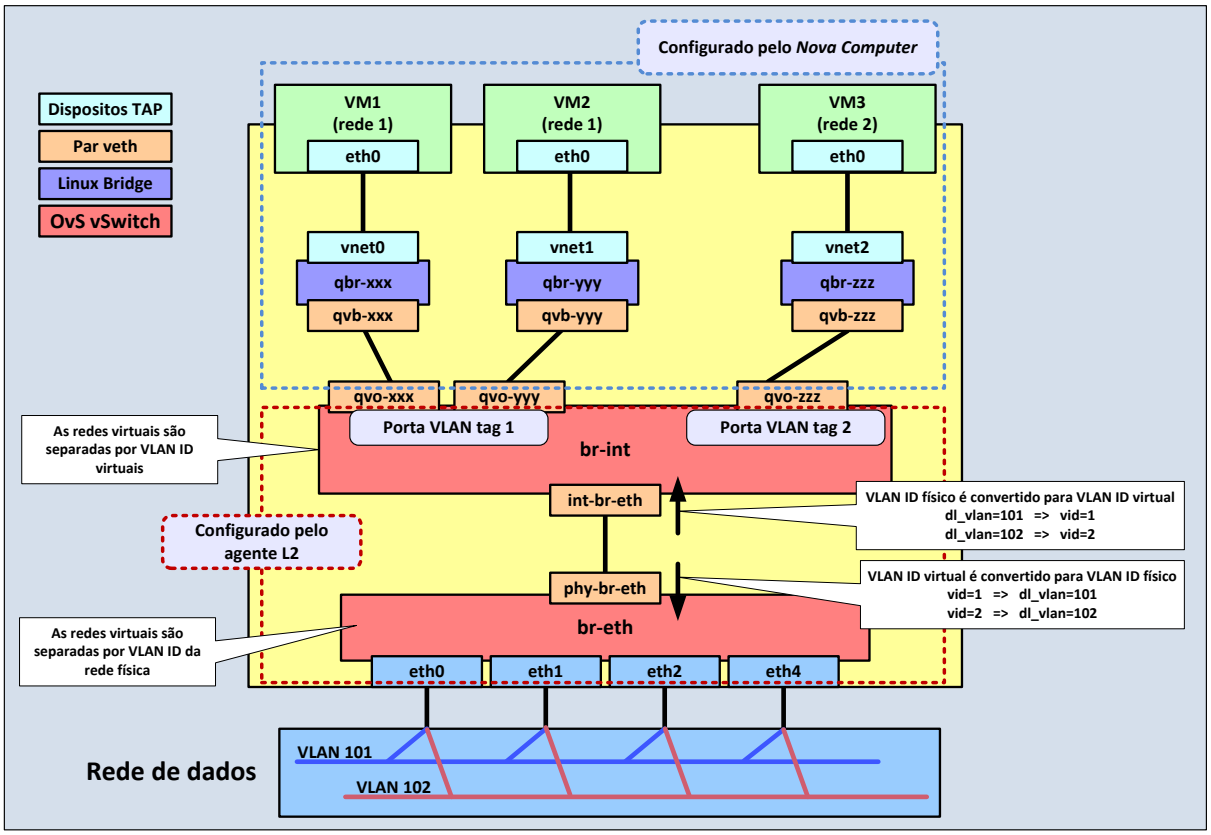


Figura 7.9: Configuração da rede virtual realizada pelo serviços do OpenStack nos *Hosts* para separar as máquinas virtuais por VLAN. Adaptada da documentação do OpenStack [72].

A organização da rede da Figura 7.9 é configurada por dois componentes do OpenStack: agente *L2* do *Neutron* e *Nova Computer*. A conexão com a rede física é de responsabilidade do agente *L2* do *Neutron* (*plugin ML2 e plugin Open vSwitch*), que cria dois *switches* virtuais sobre o *OvS*. O *switch* denominado *br-eth* é responsável: pela interligação

das interfaces físicas de rede; e pela conversão do identificador de VLAN da rede virtual (*vid)* para o identificador de VLAN rede de dados (*dl_vlan)*. O *switch* denominado *br-int* promove a integração entre as redes virtuais criadas por meio dos agentes do *Neutron*. A interligação entre o *br-int* o *br-eth* é realizado pelo par de dispositivos de redes virtuais (do tipo *veth*, *virtual net devices*) *phy-br-int* e *int-br-eth,* que funcionam com um *patch cord* em um *rack* físico, de modo que todo o tráfego que chega a uma ponta é copiado para a outra ponta.

Na parte superior da Figura 7.9 foram ilustradas três VMs para exemplificar como o serviço *Nova Computer* interliga as mesmas ao *switch* de integração *br-int*. Tomando como exemplo a *VM1*, é possível observar que sua interface de rede (*eth0*) é conectada a interface interna *vnet0* (do tipo *TAP*, dispositivo de rede virtual do *kernel*), da *Linux bridge* chamada *qbr-xxx*. Essa *bridge* se conecta ao *switch br-int* pelo par *veth* formado por *qvb-xxx e qvo-xxx*. Então, o *switch br-int* associa à interface *qvo-xxx* o *vid* da rede virtual que *VM1* está conectada. Portanto, cada pacote gerado na *eth0* da *VM1* é copiado, sequencialmente, pelos dispositivos *vnet0, qvb-xxx, qvo-xxx, qvo-yyy, qvb-yyy e vnet1* para chegar a *eth0* da *VM2* (vide Quadro 7.1). Isso explica o aumento em *6x* da taxa média observada nos *nós de computação* em relação às VMs, no gráfico de tráfego de dados da Figura 7.8. Ademais, esse excessivo número de cópias, impacta de forma significativa sobre a CPU dos *nós de computação*, conforme apresentado no gráfico de uso de CPU da Figura 7.8.

Quadro 7.1: Exemplo de arquivo de saída gerado pelo programa bwm-ng apresentando a cópia dos pacotes entre as interfaces da rede interna utilizada pelo OpenStack.

unix_timestamp	iface_name	bits_out/s	bits_in/s	bits_total/s	+11 campos
1417121966	int-br-eth	92.00	179.00	271.00	...
1417121966	qbrdb2f5ef6-da	0.00	0.00	0.00	...
1417121966	br-int	0.00	0.00	0.00	...
1417121966	phy-br-eth	179.00	92.00	271.00	...
1417121966	br-eth	0.00	0.00	0.00	...
1417121966	qvb6b7153e7-d4	51591104.00	1144296.00	52735400.00	...
1417121966	qvo6b7153e7-d4	1144560.00	51591104.00	52735664.00	...
1417121966	eth0	51.00	60.00	111.00	...
1417121966	eth1	0.00	60.00	60.00	...
1417121966	eth2	0.00	0.00	0.00	...
1417121966	eth3	70.00	273.00	343.00	...
1417121966	eth4	120.00	468.00	588.00	...
1417121966	lo	0.00	0.00	0.00	...
1417121966	tap6b7153e7-d4	1145946.00	49462344.00	50608288.00	...
1417121966	qbr6b7153e7-d4	0.00	0.00	0.00	...
1417121966	qvbdb2f5ef6-da	1144872.00	51591124.00	52735996.00	...

1417121966	qvodb2f5ef6-da	51591124.00	1144872.00	52735996.00	...
1417121966	tapdb2f5ef6-da	51591124.00	1144872.00	52735996.00	...

O Quadro 7.1 apresenta um fragmento do arquivo de monitoramento das interfaces de rede dos *nós de computação*, destacando as linhas nas quais os pacotes são copiados de uma interface para a outra, conforme explicado anteriormente. Observe que os valores abaixo de *bits_total/s*, nas linhas destacadas, deveriam ser exatamente os mesmo, porém devido a descartes de pacotes por falta de recursos, algumas linhas apresentam valores diferentes.

7.3.2 Encaminhamento de tráfego entre VMs no mesmo nó de computação, porém em redes virtuais diferentes

No cenário apresentado na Figura 7.10(a), o *nó de computação H1* hospeda a *VM1* e *VM3*, que estão conectadas por redes virtuais diferentes, de modo que a *VM1* está na rede virtual 1 e a *VM3* está na rede virtual 2, fazendo referência a *VM1* e a *VM3* da Figura 7.9. De forma semelhante ao cenário anterior, a *VM1* foi utilizada para gerar tráfego, enquanto a *VM3* foi utilizada para monitorar o uso de CPU e o tráfego de rede em sua interface. Por estarem em redes diferentes, os pacotes gerados pela *VM1* devem ser encaminhados para o roteador (*Router*), que está fisicamente localizado no *controlador da Rede*, para então serem encaminhados para a *VM3*, conforme ilustrado na Figura 7.10(b).

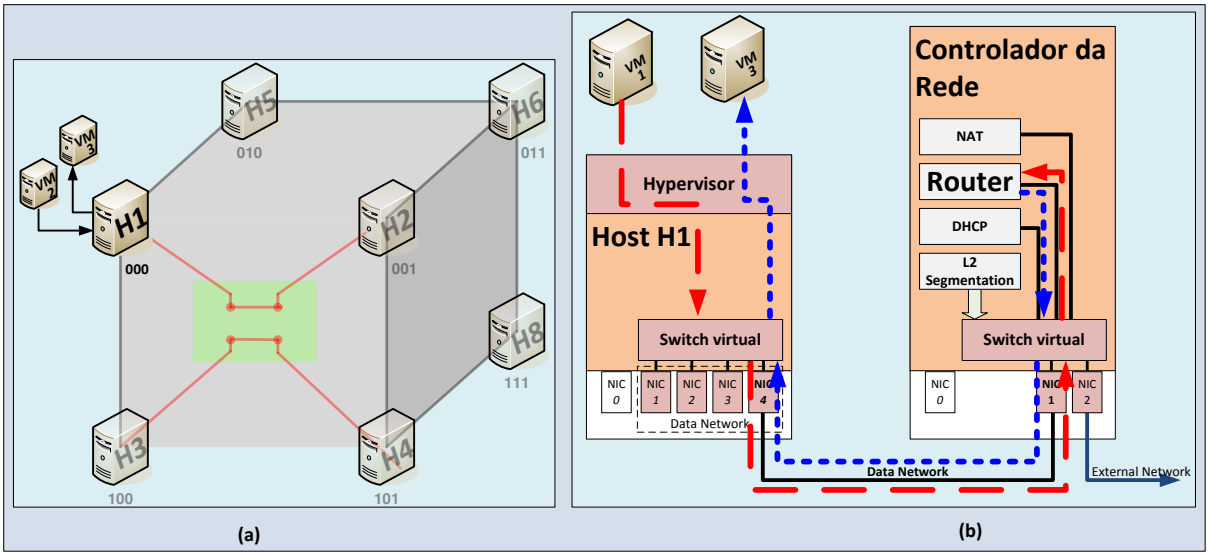


Figura 7.10: Cenário de teste de encaminhamento de tráfego entre VMs no mesmo *Host*, porém em redes virtuais diferentes: (a) Destaque do *Host H1* que hospeda a *VM1* e a*VM3*. (b) O tráfego entre a *VM1,* na rede virtual 1, é roteado pelo *controlador da Rede* para a *VM3,* na rede virtual 2 .

Sobre o cenário da Figura 7.10, realizou-se o mesmo experimento do cenário anterior. Depois de analisados, os dados do monitoramento de *H1* e *VM3* foram plotados nos gráficos ilustrados na Figura 7.11.

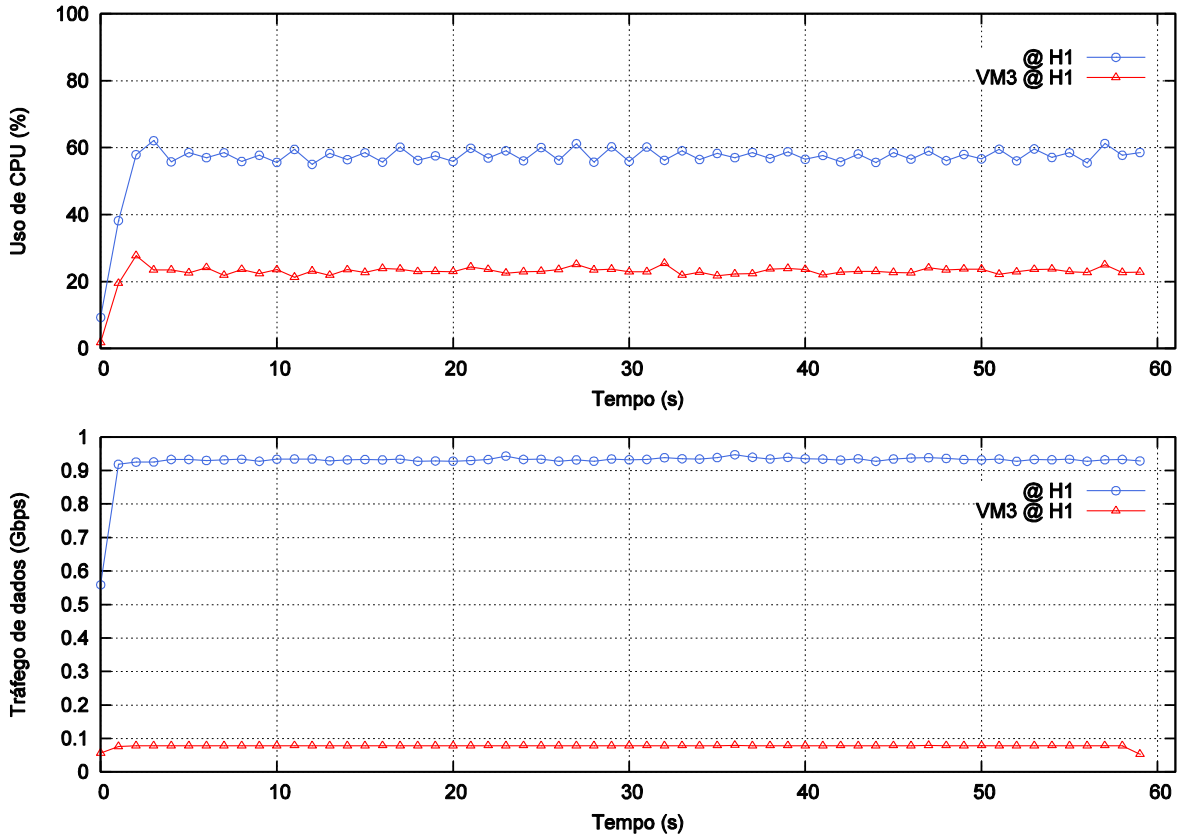


Figura 7.11: Tráfego de dados e uso de CPU registrados durante a comunicação entre VMs no mesmo *Host*, porém em redes virtuais diferentes.

No gráfico de tráfego de dados da Figura 7.11 é possível observar que fluxo visto pela *VM3* está abaixo de *0,1 Gbps*. Essa limitação foi imposta pela velocidade do enlace entre o host *H1* e o *controlador da Rede* que é de *100 Mbps*, em função do *switch* físico utilizado no ambiente experimental. No entanto, o fato mais importante a ser destacado é a proporção entre o tráfego visto por *H1* e em relação do tráfego visto pela *VM3*, que é de *10* vezes!

Mais uma vez, isso de deve a organização da rede interna dos *Hosts* utilizada pelo OpenStack, pois conforme pode ser observado na Figura 7.9, cada pacote que sai da *VM1* é copiado, sequencialmente, pelos dispositivos *vnet0, qvb-xxx, qvo-xxx, int-br-eth, phy-br-eth* para chegar a interface física *eth4* do *Host*. Por outro lado, um pacote que chega à interface física *eth4* do *Host* é copiado, sequencialmente, pelos dispositivos *phy-br-eth, int-br-eth, qvo-zzz, qvb-zzz e vnet2* para atingir a interface *eth0* da *VM3*. Ou seja, são realizadas 5 para sair para rede física mais 5 cópias para voltar da rede física, criando um fluxo maior que *0,9 Gbps*

em *H1*. Como consequência, a CPU de *H1* trabalha próximo a *60%* da sua capacidade para encaminhar o fluxo entre suas VMs, conforme o gráfico de uso de CPU da Figura 7.11.

Conforme demonstrados pelos resultados dos cenários da Fase 2, até este ponto, o modelo adotado pelo OpenStack para organizar a rede interna dos *Hosts* gera uma sobrecarga muito grande sobre os *nós de computação*, prejudicando o desempenho dos fluxos entre as VMs hospedadas nestes *Hosts*.

7.3.3 Encaminhamento de tráfego no diâmetro do Hipercubo

No cenário apresentado na Figura 7.12, o *Host H1* hospedou a *VM1* e o *Host H8* hospedou a *VM2*. Ambas as VMs estão conectadas pela mesma rede virtual. Da mesma forma que no cenário anterior, foi instalado na *VM1* o *script shell gerador* e na *VM2* o *script shell monitor*. Em seguida, cada *script* foi parametrizado com: o horário de início, *30* repetições, *60*s de duração por repetição, além dos endereços IP de cada VM.

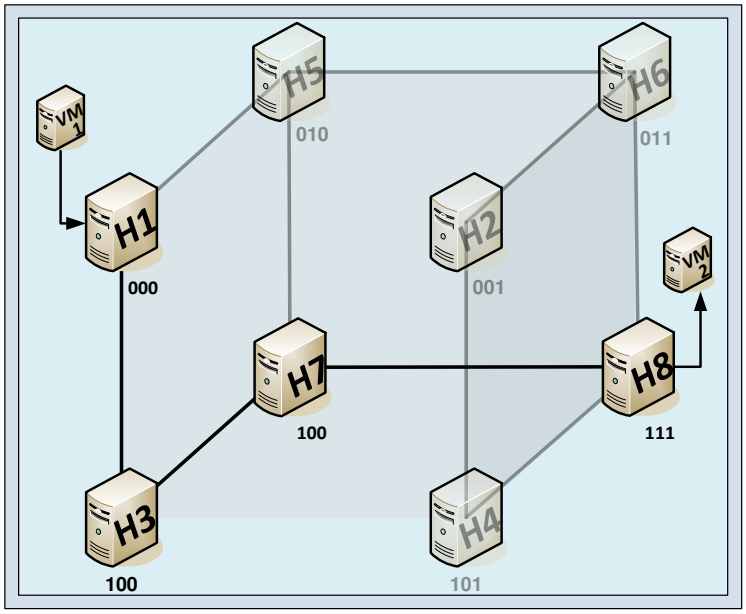


Figura 7.12: Cenário de teste de encaminhamento de tráfego no diâmetro do *Hipercubo* do ambiente experimental. As chaves ópticas foram removidas para destacar os *Hosts H1, H3, H7* e *H8*.

Assim como em outros cenários, foi instalado em cada *nó de computação* o *script shell de monitoramento*. Após a execução do experimento, verificou-se que os pacotes enviados pela *VM1* foram encaminhados pelo caminho *H1* → *H3* → *H7* → *H8*, assim estes *Hosts* foram destacados na Figura 7.12, para finalmente chegarem a *VM2*. Para representar esse cenário, o tráfego de trânsito foi isolado e plotado, juntamente com o uso de CPU, na Figura 7.13.

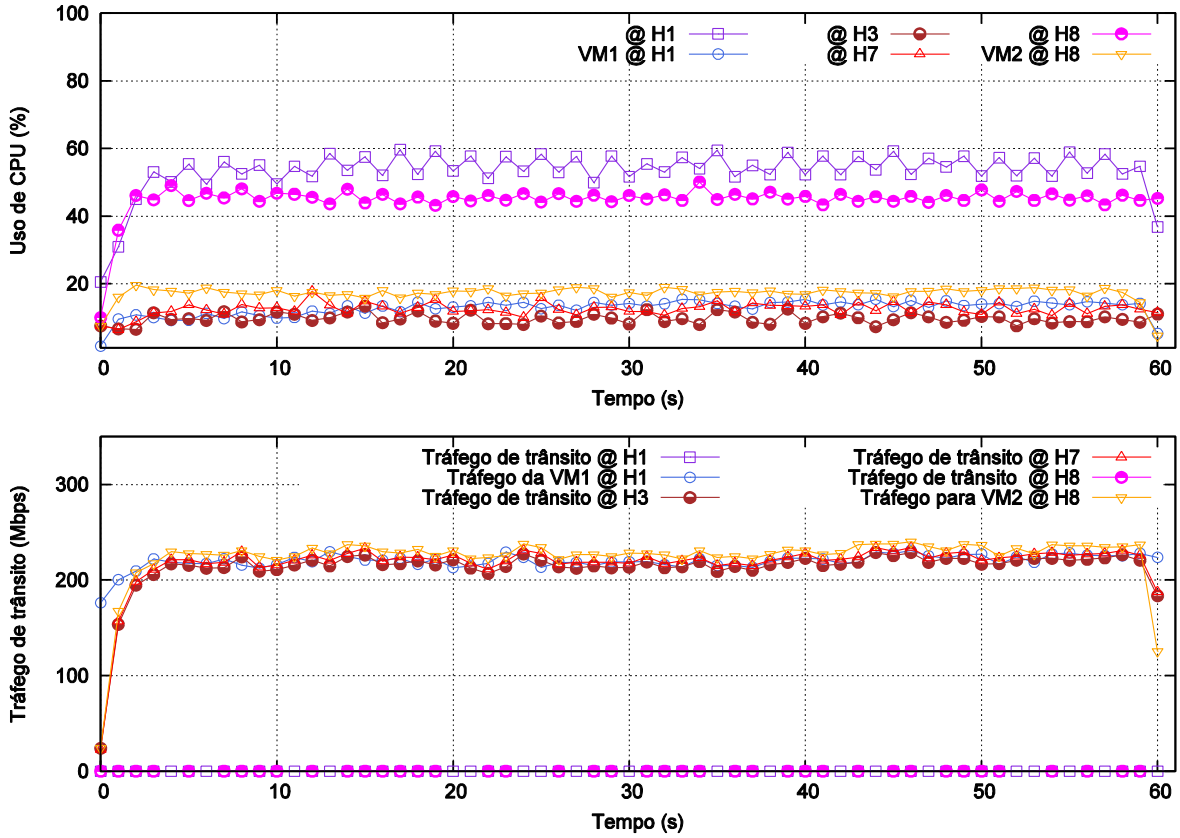


Figura 7.13: Encaminhamento de tráfego de trânsito no diâmetro do *Hipercubo* do ambiente experimental.

Analisando o gráfico de uso de CPU da Figura 7.13, é possível observar que os *Hosts H1* e *H8,* que hospedam a *VM1* e *VM2* respectivamente*,* consumiram mais que o dobro de CPU em relação aos demais *Hosts*. Esse fato está relacionado à cópia de pacotes que ocorre dentro do *Host*, descrita no cenário anterior. Por outro lado, o encaminhamento em *Hipercubo* não gera cópias, pois o *kernel* trabalha com a movimentação de referências para transferir um pacote de uma porta física para outra, utilizando menos recursos de CPU dos *Hosts H3* e *H7* quando comparado aos *Hosts H1* e *H8*. Entretanto, de forma absoluta, o encaminhamento do tráfego de trânsito consome em torno de 15% da CPU dos *Hosts H3* e *H7*, para um fluxo de pouco maior que 200 *Mbps*. Considerando que o consumo de CPU é diretamente proporcional a taxa do fluxo, a utilização da camada óptica reconfigurável poderia transferir o tráfego de trânsito para os nós com baixa carga de CPU durante o tempo de vida do fluxo. Este comportamento será apresentado na *Fase 3*.

7.4 Fase 3 – Caracterização dos impactos da comutação óptica

O objetivo da *Fase 3* foi avaliar como a reconfiguração dos enlaces na camada óptica, podem reduzir o uso dos recursos dos *nós de computação*, em relação ao encaminhamento de tráfego de trânsito. Assim como na *Fase 2*, foram utilizados cenários de tráfego sintetizados, utilizando VMs OpenWrt, para minimizar os recursos necessários com virtualização. Ainda, as comutações ópticas realizada nesta fase foram comandadas pelos *scripts* em tempos pré-definidos.

7.4.1 Impacto do tráfego de trânsito sobre o tráfego interno entre VMs

No cenário ilustrado na Figura 7.14, onde o *Host H1* hospeda a *VMrx*, o *Host H3* hospeda a *VM1* e a *VM2*, e o *Host H4* hospeda a *VMtx*, o seguinte experimento foi realizado. No tempo *0s* (zero) a *VM1* inicia um fluxo de dados para a *VM2*, com a configuração da Figura 7.14(a). Após *10s* a *VMtx* em *H4* inicia um fluxo com a *VMrx* em *H1*, que é encaminhado via *H3*, ilustrado pela linha pontilhada na Figura 7.14(a). Aos *40s* as chaves são comutadas, alternando para a configuração da Figura 7.14(b), onde *H1* e *H4* passam a se comunicar diretamente. Os fluxos são acompanhados por mais 30s. Então todo o processo é finalizado e configuração inicial é restaurada, permitindo a repetição do experimento.

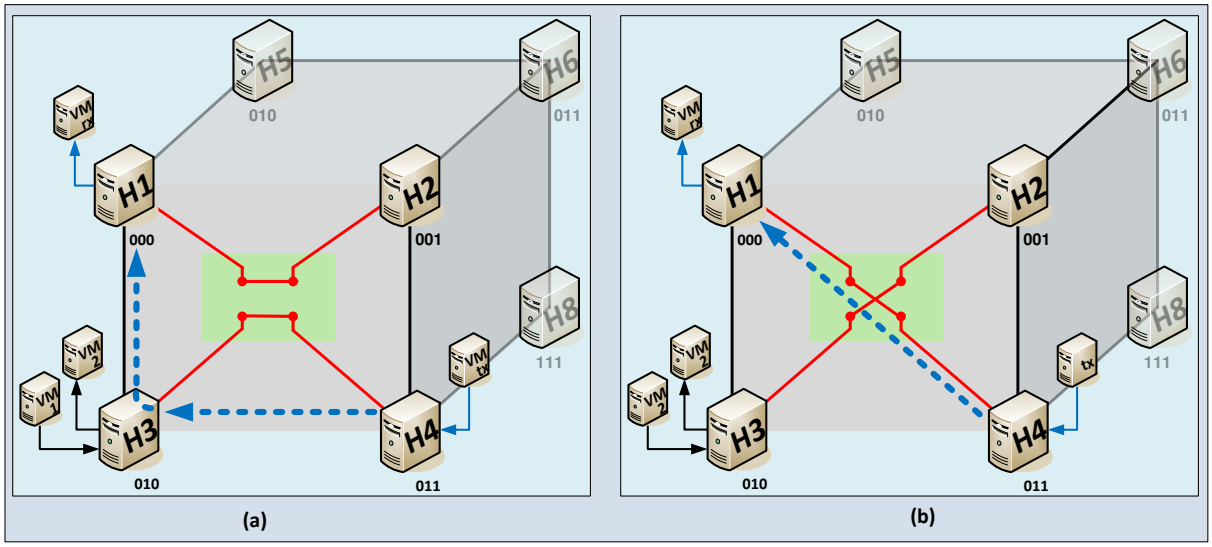


Figura 7.14: Impacto do tráfego de trânsito sobre o tráfego interno entre VMs: (a) chaves em estado *barra* com o tráfego de trânsito passando por *H3*; (b) chaves em estado *cruz* sem tráfego de trânsito.

O experimento relatado acima foi repetido 30 vezes e os dados coletados foram consolidados e plotados nos gráficos da Figura 7.15.

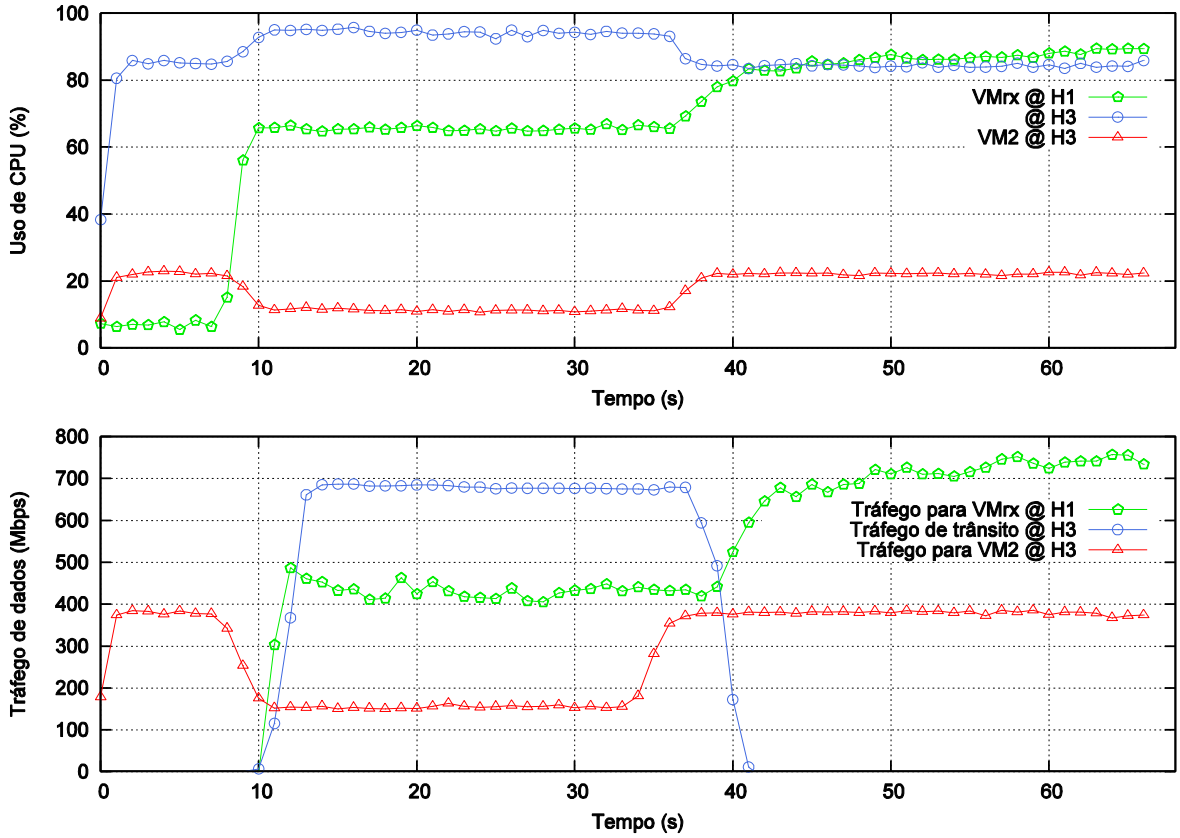


Figura 7.15: Impacto do encaminhamento no trânsito local, após o chaveamento há um alivio no nó intermediário e um aumento de tráfego entre os das pontas.

Conforme pode ser observado nos gráficos da Figura 7.15, nos primeiros *10s*, o consumo de CPU de *H3* está acima de *80%*, devido ao encaminhamento do tráfego interno, recebido por *VM1* a uma taxa próxima de *400 Mbps*. A partir do *10°s*, é possível notar que a CPU de *H3* está trabalhando próximo a *100%* de sua capacidade, devido ao tráfego de trânsito que chega a sua interface a *700 Mbps*. Com isso, o tráfego interno que chega a *VM2* é reduzido para menos de metade, juntamente com o seu uso de CPU. Ainda o tráfego recebido por *VMrx* em *H1* é menor que *700 Mbps*, indicando que *H3* está descartando pacotes. Aos *40s*, a reconfiguração óptica é realizada, de forma que *H3* e *VM2* voltam para seus níveis anteriores de carga. Por outro lado, a *VMrx* passa a receber efetivamente o fluxo de *700 Mbps*, aumentado com isso seu uso de CPU para processar os pacotes extras.

Os resultados descritos acima demonstram que o tráfego de trânsito interfere de forma negativa no tráfego interno. Isso acontece, pois o trafego interno é gerado por *VMs*, que são processos de usuários, portanto têm prioridade menor que o encaminhamento de tráfego que é realizado em nível de *kernel*. Por fim, foi possível observar o ganho que o chaveamento

óptico trouxe para este cenário, reduzindo a carga sobre o nó intermediário (*H3*) e aumentando a taxa do fluxo de dados entre *VMtx* e *VMrx*.

7.4.2 Impacto do tráfego de trânsito sobre os processos de usuários

Para confirmar a hipótese de que o encaminhamento é tratado de forma prioritária pelo sistema operacional, foi criado o cenário de teste ilustrado na Figura 7.16, onde o *Host H1* hospeda a *VMrx* e o *Host H4* hospeda a *VMtx*. O experimento inicia-se na configuração apresenta na Figura 7.16(a), ocupando *H3* com uma carga artificial de 99% de uso de CPU. Após *10s* a *VMtx* em *H4* inicia um fluxo com a *VMrx* em *H1*, que é encaminhado via *H3*, ilustrado pela linha pontilhada na Figura 7.16(a). Aos *40s* as chaves são comutadas, alternando para o cenário da Figura 7.16(b), onde *H1* e *H4* passam a se comunicar diretamente. Os fluxos são acompanhados por mais 30s. Então todo o processo é finalizado e o cenário inicial é restaurado, permitindo a repetição do experimento.

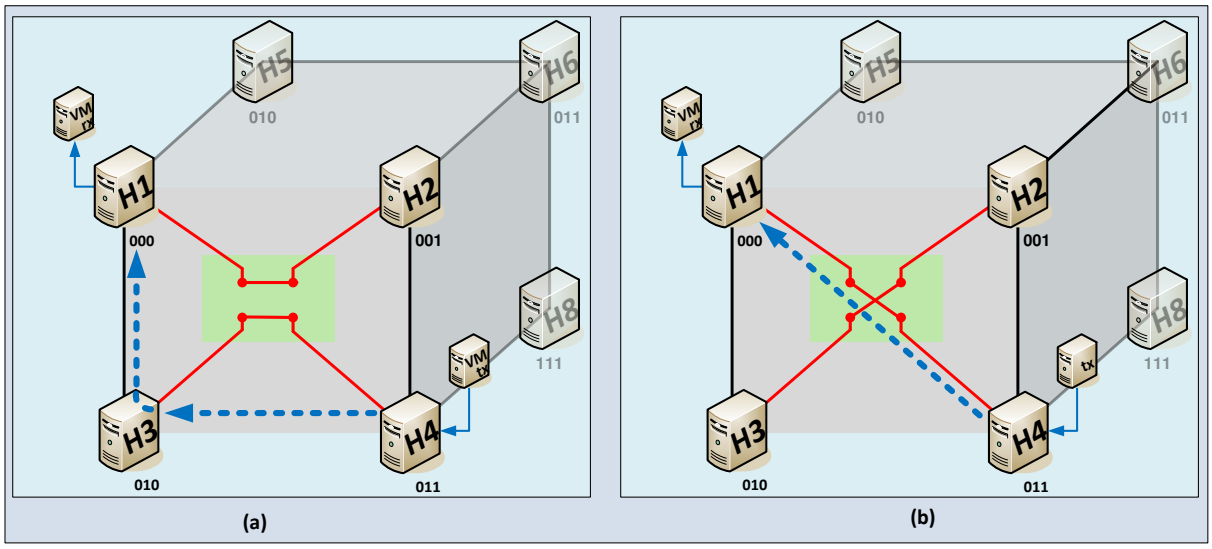


Figura 7.16: Impacto do tráfego de trânsito sobre os processos de usuário: (a) chaves em estado *barra* com tráfego de trânsito passando por *H3*; (b) chaves em estado *cruz* sem tráfego de trânsito.

Trinta repetições do experimento foram conduzidas; e dados mais detalhados do uso de CPU foram coletados. Especificamente, registrou-se o percentual de CPU para processos do usuário ($\%usr$) e para processos de *kernel* do sistema ($\%sys$). Da analise dos dados coletados, foram plotados os gráficos da Figura 7.17.

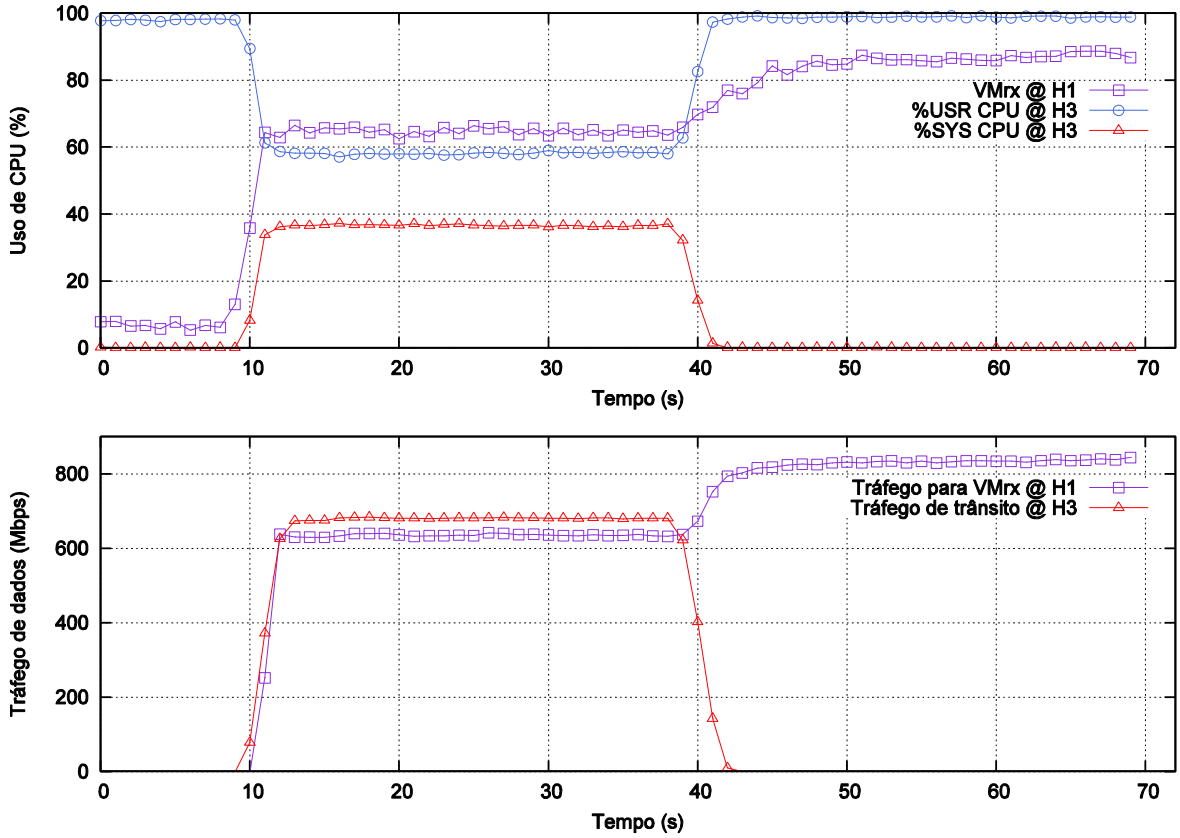


Figura 7.17: Influência do trânsito de trânsito sobre os processos do usuário e reconfiguração da camada óptica.

O primeiro gráfico da Figura 7.17 apresenta o de uso de CPU de *H3* para processos de usuário (*%USR CPU @ H3*) e para processos de *kernel* do sistema (*%SYS CPU @ H3*), além do uso total de CPU de *VMrx (MV @ H1)*. Observe que o experimento inicia com praticamente toda a CPU de *H3* alocada para atender processos de usuário. Aos 10s, *%USR CPU @ H3* é reduzido em torno de *40%*, enquanto *%SYS CPU @ H3* aumenta na mesma proporção. Isso ocorre em função do tráfego de trânsito que *H3* passa a encaminhar, conforme o segundo gráfico da Figura 7.17. Ao mesmo tempo, *VMrx* passa a receber o fluxo encaminhado por *H3*, aumentando o seu uso de CPU para processar essa demanda. Aos *40s*, a reconfiguração óptica é realizada. A CPU de *H3* é novamente dedicada aos processos de usuário. Por sua vez, a taxa do fluxo que chega a *VMrx* é aumentado, exigindo mais CPU para processar esse aumento de demanda. Por fim, comparando a taxa de fluxo vista por *VMrx*, antes e depois do chaveamento, é possível concluir que *H3* limitou essa taxa de fluxo durante o período em que encaminhava o tráfego para a *VMrx*.

Os resultados deste teste confirmam que o sistema operacional prioriza o encaminhamento de dados, em detrimento dos processos do usuário. Segundo Godard [67], o

%sys é o percentual de tempo para atender chamadas do sistema (*kernel*), Porém, não inclui o tempo gasto para atender interrupções de *hardware* ou de *software*, ou seja, praticamente todo *%SYS CPU @ H3* é utilizado para encaminhamento dos dados neste experimento. Assim como no cenário anterior, o chaveamento óptico reduziu a carga sobre o nó intermediário (*H3*) e aumentou a taxa do fluxo observada por *VMrx*.

7.4.3 Impacto do chaveamento na distribuição do tráfego de trânsito

Para demostrar como o chaveamento pode reduzir o tráfego de trânsito total da rede, foi construído o cenário de teste ilustrado na Figura 7.18, onde o *Host H1* hospeda a *VMrx*, *Host H3* hospeda a *VM2* e o *Host H4* hospeda a *VMtx*. O experimento inicia-se na configuração apresenta na Figura 7.18 (a), com a *VMtx* enviando dois fluxos para *H3*, sendo o fluxo menor (linha tracejada) para *VM2* e o fluxo maior (linha pontilhada) para *VMrx* em *H1*. Após *40s* as chaves são comutadas, alternando para o cenário da Figura 7.18(b), de forma que *H1* passa a encaminhar o fluxo menor para *VM2* em *H3*, reduzindo assim o tráfego de trânsito total rede. Os fluxos são acompanhados por mais 30s. Então todo o processo é finalizado e o cenário inicial é restaurado, permitindo a repetição do experimento.

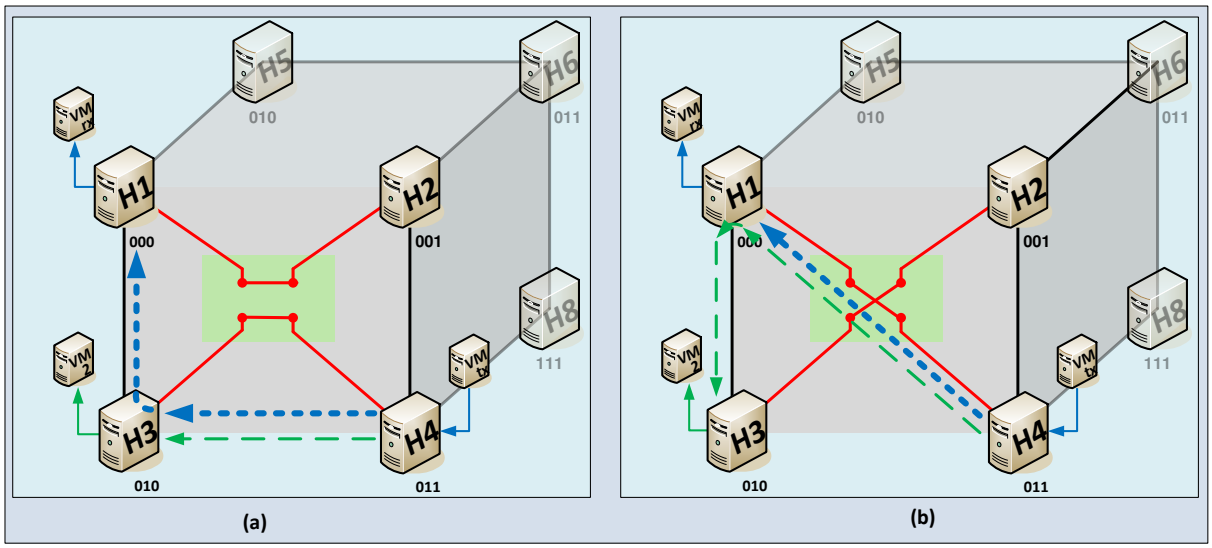


Figura 7.18: Impacto do chaveamento sobre tráfego de trânsito total: (a) chaves em estado *barra* com tráfego de trânsito maior passando por *H3*; (b) chaves em estado *cruz* com tráfego de trânsito menor passando por *H1*.

O experimento foi conduzido com o fluxo maior a *400 Mbps* e o fluxo menor a *100 Mbps*, esses valores foram escolhidos para não saturar totalmente os *Hosts* intermediários. O processo foi repetido trinta vezes e os dados de CPU e tráfego de rede foram coletados e plotados nos gráficos da Figura 7.19.

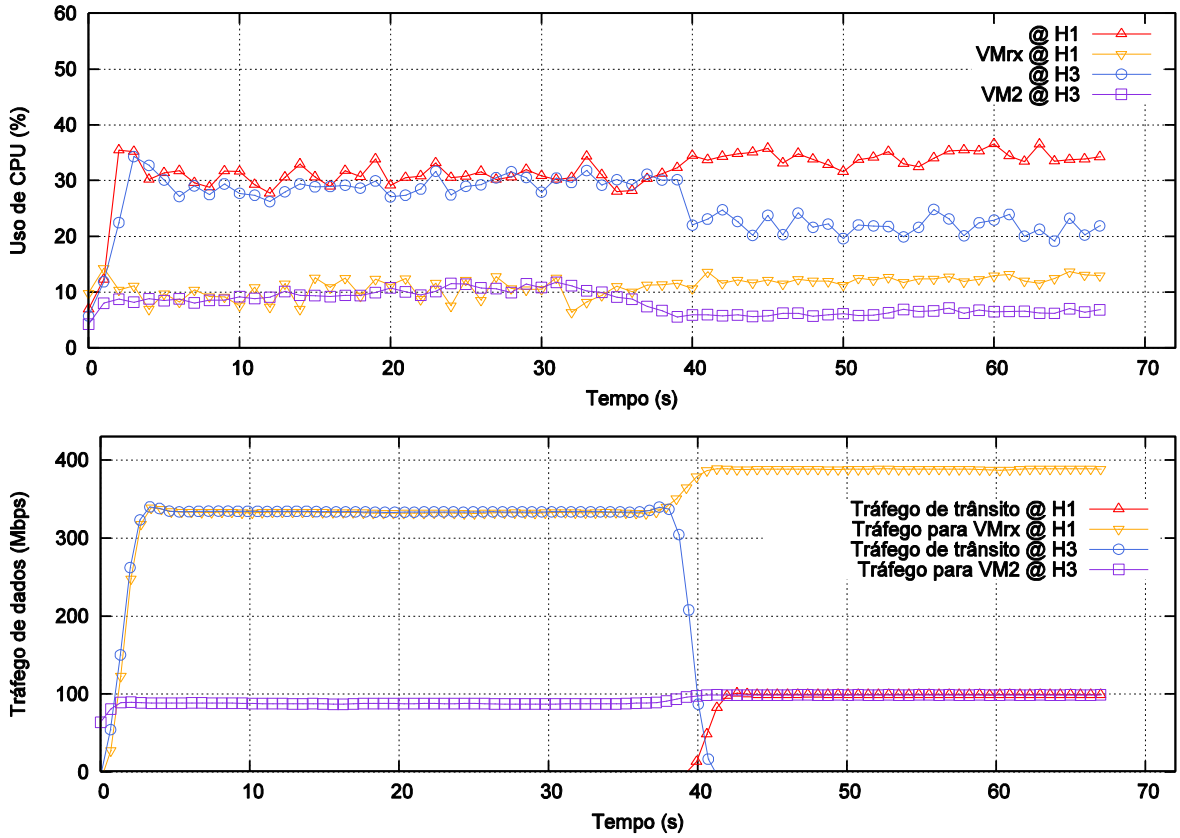


Figura 7.19: Impacto do chaveamento sobre tráfego de trânsito total.

O primeiro gráfico da Figura 7.19 apresenta o de uso de CPU de *H1* e *H3* em torno de 30%, porém *H1* utiliza a CPU apenas para tratar os pacotes encaminhados para a VM que ele hospeda, enquanto *H3* utiliza parte da CPU para encaminhar o tráfego de trânsito de *400 Mbps* para a *VMrx* em *H1*. Porém, *H3* não consegue encaminhar os *400 Mbps* e ainda não entrega *100 Mbps* para *VM2* como mostrado no segundo gráfico da Figura 7.19. Aos *40s*, a reconfiguração óptica é realizada. Com isso, o uso de CPU em *H3* é reduzido para aproximadamente 20%, por não haver mais tráfego de trânsito passando por ele. Por outro lado, ocorre um aumento do uso de CPU em *H1*, que agora precisa encaminhar o tráfego de trânsito de *100 Mbps* para a *VM2* em *H3*. Ao mesmo tempo, as taxas dos fluxos que chegam a *VMrx* e *VM2* são aumentadas para *400 Mbps* e *100 Mbps*, respectivamente.

Os resultados apresentados nesse cenário comprovam que a reconfiguração da camada física pode ser utilizada para redistribuir os fluxos, de modo a reduzir o tráfego de trânsito total sobre a rede, e consequentemente, reduzir o uso médio de CPU para encaminhamento. Ainda é possível que haja um aumento das taxas dos fluxos, devido à redistribuição de carga entre os elementos da rede. Neste ponto é importante destacar que diversos fatores contribuem para a saturação do encaminhamento, levando ao descarte dos pacotes. No ambiente

experimental onde os testes foram executados, o principal fator é o compartilhamento do barramento PCI entre diversas interfaces impede que altas taxas sejam alcançada.

Capítulo 8 – Orquestração Autônoma do Controlador A-SDN

A *Fase 3* do capítulo anterior apresentou diversos cenários de tráfego em que o uso da camada de reconfiguração óptica reduziu o consumo de recursos do *nós de computação* e, em alguns casos, aumentou a eficiência da rede de dados. Naquele capítulo, também foi discutido que o escalonador de VMs do OpenStack, prioriza as máquinas com grande capacidade de memória, em detrimentos ao uso de CPU. Ainda pesa contra o escalonador do OpenStack o fato desse não tomar conhecimento do estado da rede, em especial, da carga dos *nós de computação* para encaminhamento de tráfego. Assim, o uso da arquitetura do OpenStack sobre uma rede de *data center* centrado em servidores, pode gerar um desbalanceamento de carga significativo, principalmente em redes com servidores heterogêneos.

Por outro lado, o *controlador A-SDN* da TRIIIAD utiliza os parâmetros de carga dos *nós de computação* (CPU, memória e tráfego de trânsito) para realizar a orquestração autônoma da arquitetura. Distribuindo de forma mais justa a carga entre os servidores, por meio da reconfiguração dos enlaces ópticos da rede e da migração de máquinas virtuais dos servidores sobrecarregados para os servidores com menos carga. Para demostrar o funcionamento da orquestração autônoma, foram construídos os cenários apresentados ao longo deste capítulo.

8.1 Orquestração com limites fixos de tráfego de trânsito

O cenário ilustrado da Figura 8.1 foi elaborado para mostrar a atuação do *controlador A-SDN* sobre a reconfiguração dos enlaces da camada óptica da rede na tentativa de reduzir o tráfego de trânsito total, redistribuindo os tráfegos de trânsito entre os *nós de computação*. Conforme pode ser observado na Figura 8.1(a), as chaves ópticas estão na posição *barra*,

formando o *Hipercubo* tradicional, onde *Host H1* hospeda a *VM1*, *Host H3* hospeda a *VM3* e o *Host H4* hospeda as *VM4T e VM4R*.

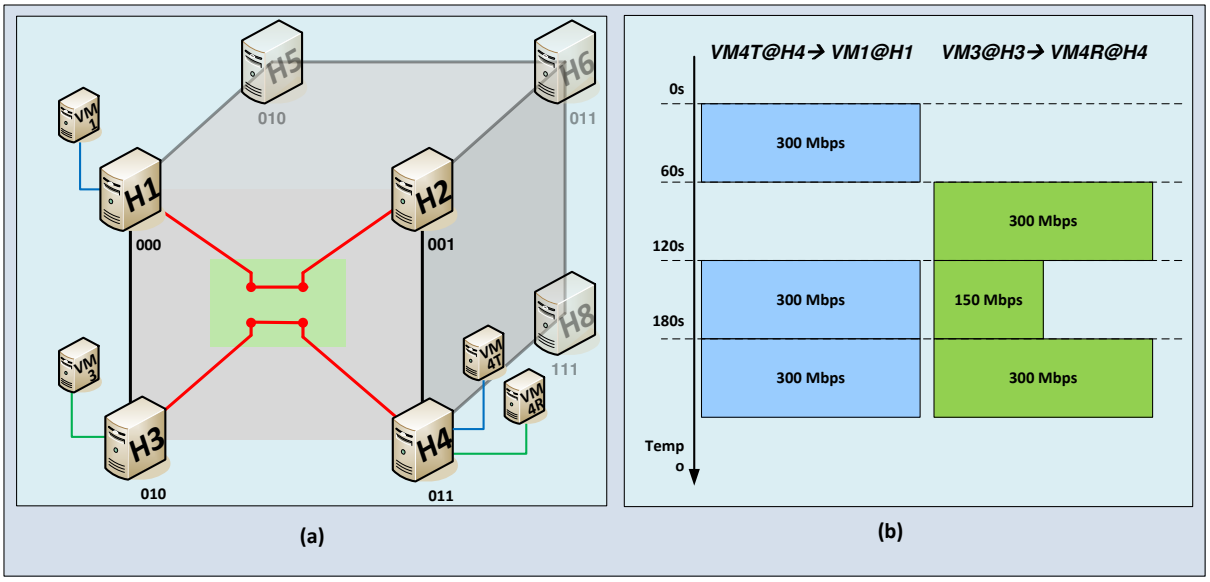


Figura 8.1: Cenário da orquestração com limites fixos de tráfego de trânsito. (a) configuração do *Hipercubo* com as chaves em *barra*; (b) sequência de fluxos utilizada no experimento.

Por outro lado, a Figura 8.1(b) ilustra a sequência temporal de fluxos utilizados durante a condução dos experimentos. De modo a garantir que não haveria migração de máquinas virtuais em função do uso excessivo de CPU, a taxa máxima dos fluxos foi limitada a *300 Mbps*. Outros dois parâmetros definidos para esse teste foram: limite de tráfego de trânsito igual a *200 Mbps*; intervalo de verificação da carga média de trabalho dos nós igual a *30s*.

No inicio do experimento a VM4T em H4 inicia um fluxo a *300 Mbps* durante *60s* para a VM1 em H1. Aos 60s, a VM3 em H3 inicia um fluxo a *300 Mbps* durante *60s* para a VM4R em H4. Aos 120s, dois novos fluxos são iniciados simultaneamente com durante *60s* cada, um partindo da VM4T a *300 Mbps* para a VM1, e outro partindo da VM3 a *150 Mbps* para a VM4R. Aos 180s, mais dois fluxos a *300 Mbps* são iniciados com duração de *180s* cada, um partindo da VM4T para a VM1, e outro partindo da VM3 para a VM4R. O comportamento da topologia foi plotado nos gráficos da Figura 8.2, onde as curvas foram suavizadas para favorecer a visualização dos patamares.

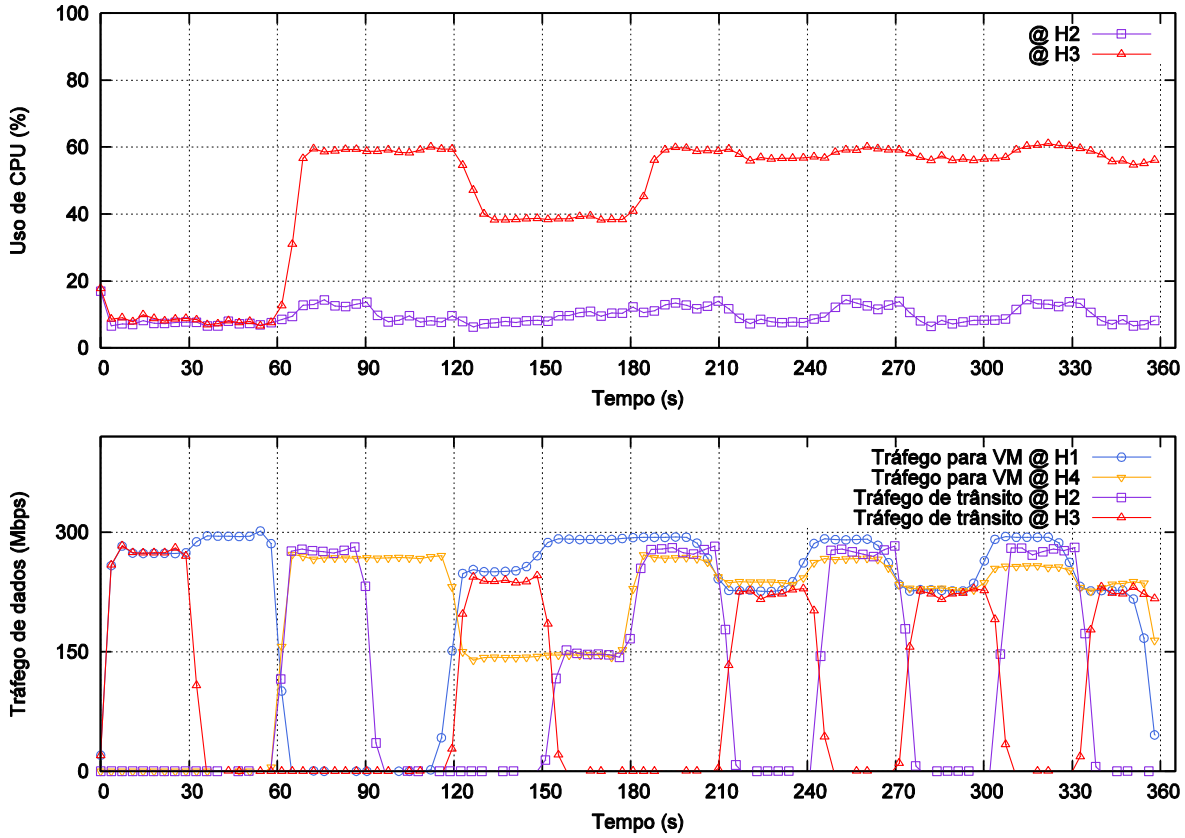


Figura 8.2: Orquestração com limite fixo de tráfego de trânsito de 200 Mbps e intervalo de verificação de carga média de 30s.

Referente ao gráfico de uso de CPU da Figura 8.2, é importante observar que o uso de CPU por *H3* é superior ao uso de CPU por *H2*, devido ao fato de *H3* hospedar a *VM3*, que encaminha os fluxos comentados no paragrafo acima. Assim, a partir de *60s* a CPU de *H3* passa a usar certa de *60%* de sua capacidade, para encaminhar um fluxo a *300 Mbps* da *VM3* para *VM4R* hospedada em *H4*. Observe ainda, que durante o intervalo entre *120s* a *180s*, quando a taxa do fluxo da *VM3* para *VM4R* é de *150 Mbps*, o uso de CPU de *H3* reduz para *40%*. Voltando a atingir *60%* quando o fluxo entre *VM3* e *VM4R* volta a ser encaminhado a *300 Mbps*. Por outro lado, o uso de CPU de *H2* sobre pequenas variações, pois como será mostrado *H2* é utilizado para encaminhar tráfego de trânsito ao logo do experimento.

A fim de facilitar o entendimento do gráfico de tráfego de dados da Figura 8.2, sua explicação será realizada dividindo o tempo do experimento entre os intervalos: 0s a 60s; 61s a 120s; 121s a 180s; e 180s a 360s.

No intervalo 0s a 60s, com as chaves no estado *barra* conforme ilustrado na Figura 8.3(a), o fluxo a *300 Mbps* (linha pontilhada) iniciado por *VMT4* é encaminhado por *H3* até a *VM1*, conforme mostrado no intervalo de 0s a 30s do gráfico tráfego de dados da Figura 8.2.

Esse fluxo excede o limite de tráfego de trânsito, de forma que aos *30s*, quanto ocorre à verificação da carga média, *H3* reporta para o *controlador A-SDN* que seu tráfego de trânsito foi excedido. Por sua vez o *controlador A-SDN* verifica que *H3* faz parte de um plano de chaveamento, assim ele envia uma mensagem ao *controlador das Chaves Ópticas* ordenando a comutação desse plano (conforme explicado na seção 5.6.2). Esta ação reconfigura o cenário de acordo com a Figura 8.3(b), de forma que o fluxo é encaminhado diretamente para o *H1* que hospeda a *VM1*. Observe no gráfico de tráfego de dados da Figura 8.2 que de 30s a 60s não existe nenhum tráfego de trânsito sobre *H3*.

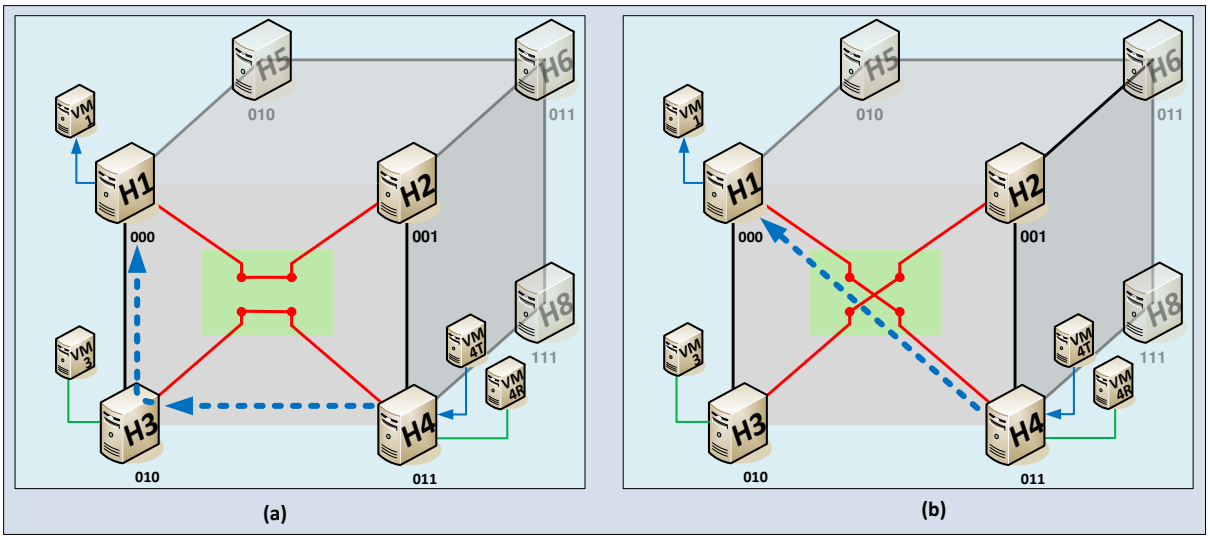


Figura 8.3: Intervalo de 0s a 60s com o fluxo de VM4T para VM1 a 300 Mbps. (a) configuração com as chaves em *barra* com trafego de trânsito a 300 Mbps em *H3*; (b) configuração com as chaves em *cruz* com encaminhamento direto.

No intervalo 61s a 120s, com as chaves no estado *cruz* conforme ilustrado na Figura 8.4(a), o fluxo a *300 Mbps* (linha tracejada) iniciado por *MV3* é encaminhado por *H2* até a *VM4R*, conforme mostrado no intervalo de 60s a 90s do gráfico tráfego de dados da Figura 8.2. Esse fluxo, também, excede o limite de tráfego de trânsito, de forma que aos *90s*, quanto ocorre a verificação da carga média, *H2* reporta para o *controlador A-SDN* que seu tráfego de trânsito foi excedido. O *controlador A-SDN,* de forma semelhante ao paragrafo anterior, ordena a comutação desse plano. Isso leva o cenário para a configuração inicial conforme Figura 8.4 (b). Então o fluxo passa a ser encaminhado diretamente para o *H4* que hospeda a *VM4R*. Observe no gráfico de tráfego de dados da Figura 8.2 que de 90s a 120s não existe nenhum tráfego de trânsito sobre *H2*.

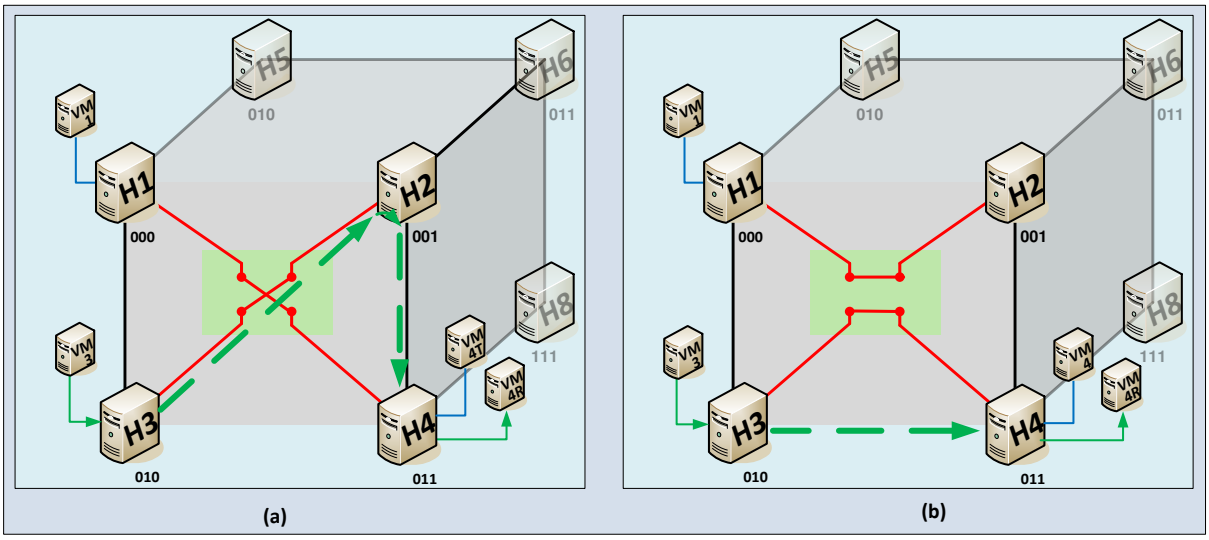


Figura 8.4: Intervalo de 61s a 120s com o fluxo de VM3 para VM4R a 300 Mbps. (a) configuração com as chaves em *cruz* com trafego de trânsito a 300 Mbps em *H2*; (b) configuração com as chaves em *barra* com encaminhamento direto.

No intervalo 121s a 180s, com as chaves no estado *barra* conforme a Figura 8.5(a), o fluxo a *150 Mbps* (linha tracejada) iniciado por *MV3* é encaminhado diretamente para *H4* que hospeda a *VM4R*. Ao mesmo tempo, o fluxo a *300 Mbps* (linha pontilhada) iniciado por *VMT4* é encaminhado por *H3* até a *VM1*, conforme mostrado no intervalo de 120s a 150s do gráfico tráfego de dados da Figura 8.2. O segundo fluxo excede o limite de tráfego de trânsito, de forma que aos *150s*, quanto ocorre à verificação da carga média, *H3* reporta para o *controlador A-SDN* que seu tráfego de trânsito foi excedido. Então, o *controlador A-SDN* ordena a comutação deste plano. Isso leva o cenário para a configuração com a chave em *cruz* conforme Figura 8.5(b). Essa modificação da topologia redistribui as cargas de forma que *H2* passa a encaminhar o fluxo entre *MV3* e *VM4R*. Porém como esse fluxo não excele o limite de tráfego de trânsito, *H2* não reporta nenhum evento para o controlador. Importante notar que houve uma redução total do tráfego de trânsito da rede de *300 Mbps* para *150 Mbps*. Observe no gráfico de tráfego de dados da Figura 8.2 que de 150s a 180s não existe nenhum tráfego de trânsito sobre *H3*, porém existe um tráfego de trânsito de *150 Mbps* sobre *H2*.

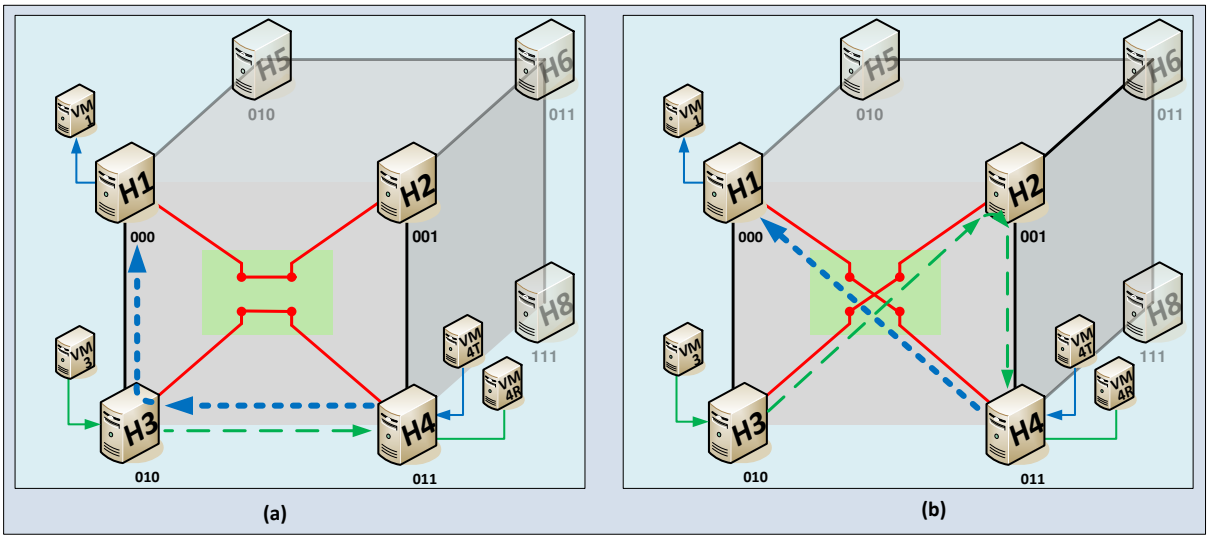


Figura 8.5: Intervalo de 121s a 180s com os fluxos de VM4T para VM1 a 300 Mbps; e de VM3 para VM4R a 150 Mbps. (a) configuração com as chaves em *barra* com trafego de trânsito a 300 Mbps em *H3*; (b) configuração com as chaves em *cruz* com trafego de trânsito a 150 Mbps em *H2*.

No intervalo 181s a 360s, com as chaves no estado *cruz* conforme a Figura 8.6(a), o fluxo a *300 Mbps* (linha pontilhada) iniciado por *VMT4* é encaminhado diretamente para *H1* que hospeda a *VM1*. Ao mesmo tempo, o fluxo a *300 Mbps* (linha tracejada) iniciado por *MV3* é encaminhado por *H2* até a *VM4R*, conforme mostrado no intervalo de 180s a 210s do gráfico tráfego de dados da Figura 8.2. O segundo fluxo excede o limite de tráfego de trânsito, de forma que aos *210s*, quanto ocorre à verificação da carga média, *H2* reporta para o *controlador A-SDN* que seu tráfego de trânsito foi excedido. O *controlador A-SDN* de forma semelhante aos casos anteriores ordena a comutação desse plano. Isso leva o cenário para a configuração com a chave em *barra* conforme Figura 8.6(b). Essa modificação da topologia redistribui as cargas de forma que *H3* passa a encaminhar o fluxo (linha pontilhada) entre *MV4T* e *VM1*. Como esse fluxo excele o limite de tráfego de trânsito, *H3* reporta este evento para o *controlador A-SDN*, aos 240s, que vai comutar o plano de chaveamento de *H3*, voltando à configuração do cenário inicial desse intervalo, conforme a Figura 8.6(a). De fato, o *controlador A-SDN* irá comutar entre as configurações da Figura 8.6, até que um dos fluxos termine. Esse comportamento está registrado no intervalo de *210s* a *360s* do gráfico de tráfego de dados da Figura 8.2, onde o tráfego de trânsito é distribuído em tempos iguais entre *H2* e *H3*.

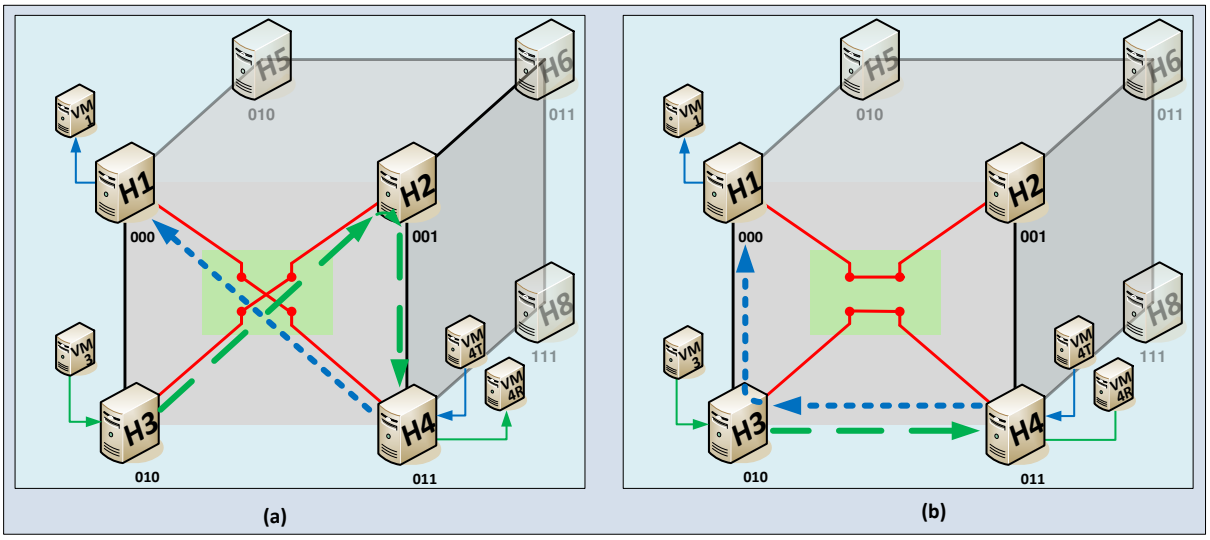


Figura 8.6: Intervalo de 180s a 360s com os fluxos de VM4T para VM1 a 300 Mbps; e de VM3 para VM4R a 300 Mbps. (a) configuração com as chaves em *cruz* com trafego de trânsito a 300 Mbps em *H2*; (b) configuração com as chaves em *barra* com trafego de trânsito a 300 Mbps em *H3*.

Conforme apresentado, mesmo não eliminando o tráfego de trânsito por completo, o *controlador A-SDN* promove uma redução desse tráfego de trânsito, ou no mínimo, uma distribuição no tempo do tráfego de trânsito entre os *nós de computação*. No entanto, deve-se notar que o *Host H3* está sendo penalizado, devido sua alta carga de CPU. Observe que em alguns momentos ele não é capaz de enviar todo o tráfego de trânsito, por exemplo, no intervalo de *210s* a *240s* do tráfego de dados da Figura 8.2.

8.2 Orquestração com limite adaptativo de tráfego de trânsito em função do uso de CPU

No cenário anterior, a reconfiguração dos enlaces físicos foi realizada observando-se exclusivamente o tráfego de trânsito sobre o *Host*. No entanto, essa abordagem gera um desbalanceamento em relação às cargas de CPU, ao redistribuir um fluxo de um *Host* com baixa carga de CPU, para outro com alta carga de CPU, como foi o caso do *Host H3*.

Neste cenário, será demostrado como o *controlador A-SDN* pode melhorar a distribuição de carga, tanto de tráfego de trânsito, quanto de CPU, utilizando o modelo de limite adaptativo de tráfego de trânsito em função do uso de CPU, ilustrado na Figura 8.7, para definir quando o chaveamento deve ocorrer.

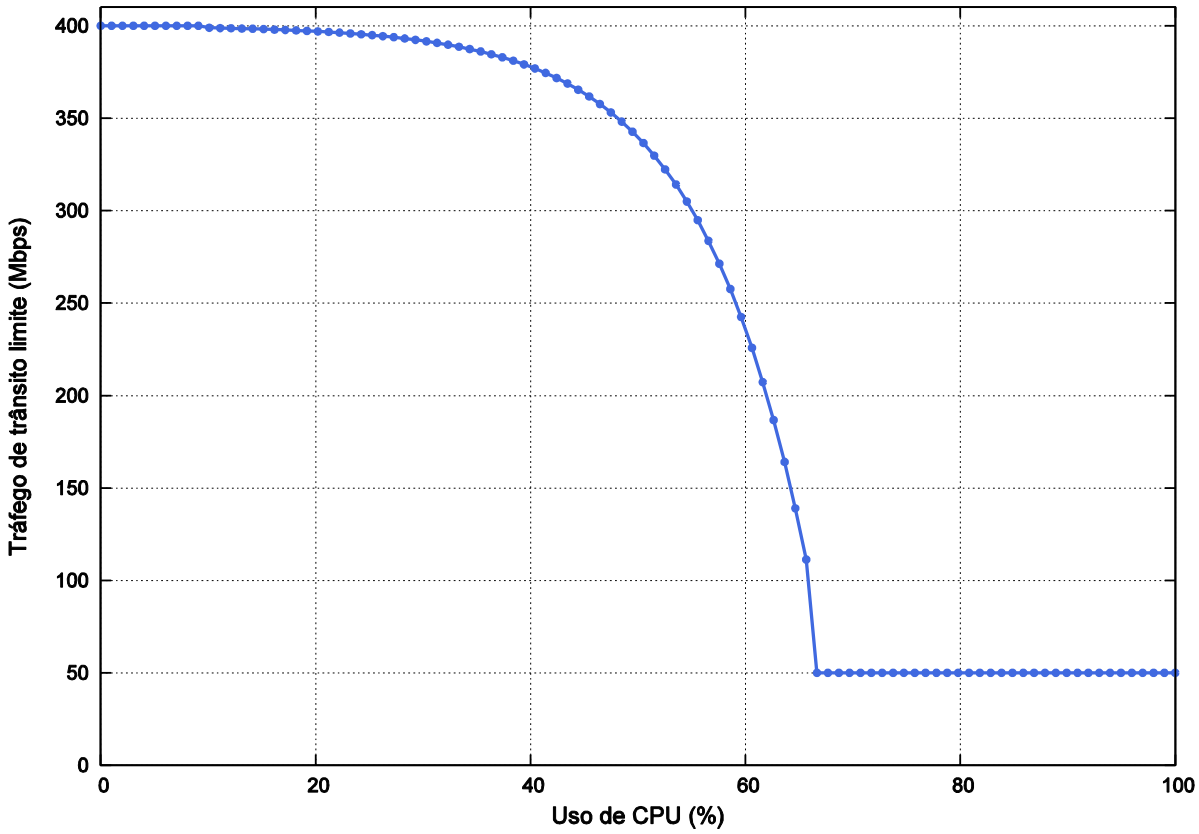


Figura 8.7: Modelo de limite adaptativo em função da carga de CPU.

Conforme o gráfico da Figura 8.7, o limite de tráfego de trânsito vai decaindo à medida que a CPU do *Host* vai sendo mais exigida. Nesse experimento, foi definido o valor máximo de *400 Mbps*, para uso de CPU entre 0% e 10%. Para o próximo intervalo de 11% a 65%, o limite de tráfego de trânsito é calculado pela Equação (2), que subtrai do limite máximo a exponencial do $\%CPU$, modulado pelo par de constantes $k1$ e $k2$. A função exponencial foi escolhida para permitir um decaimento leve para valores de CPU abaixo de 50% e um decaimento acentuado acima desse percentual, até 65% de uso de CPU, quanto o valor do limite de tráfego de trânsito volta a ser constante. Nesse experimento este limite foi definido em *50 Mbps*. É importante observar que os limites mínimos e máximos, podem ser ajustados para atender as necessidades especiais de uma determinada aplicação.

$$LT = Limite_{Maximo} - \exp\left(\frac{\%CPU\ x\ k1}{k2}\right) \quad \textbf{(2)}$$

Para fins de comparação, foi utilizado o mesmo cenário da orquestração com limite fixo, ilustrado da Figura 8.1. Os resultados obtidos utilizando o limite de tráfego de trânsito adaptativo estão plotados nos gráficos da Figura 8.8, onde as curvas foram suavizadas para favorecer a visualização dos patamares.

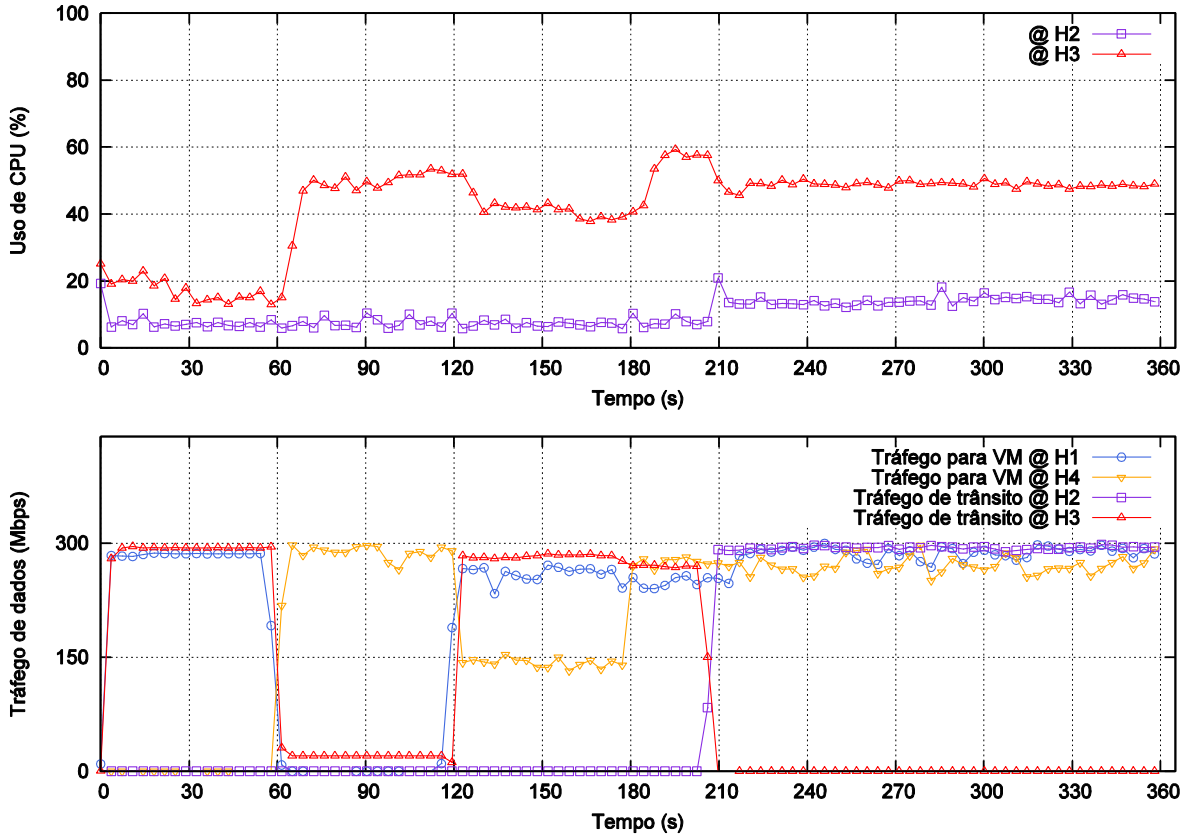


Figura 8.8: Orquestração com limite adaptativo de tráfego de trânsito em função da carga de CPU e intervalo de verificação de carga média de 30s.

No intervalo 0s a 60s, com as chaves no estado *barra* conforme ilustrado na Figura 8.9, o fluxo a *300 Mbps* (linha pontilhada) iniciado por *VMT4* é encaminhado por *H3* até a *VM1*. Nesse intervalo, *H3* está com 20% de uso de CPU e *300 Mbps* de tráfego de trânsito, como mostra o gráfico de uso de CPU e tráfego de trânsito da Figura 8.8, respectivamente. Observe que o tráfego de trânsito em *H3* está abaixo do limite, para sua carga de CPU, de acordo com a Figura 8.7. Portanto, o *controlador A-SDN* não realizou nenhuma ação.

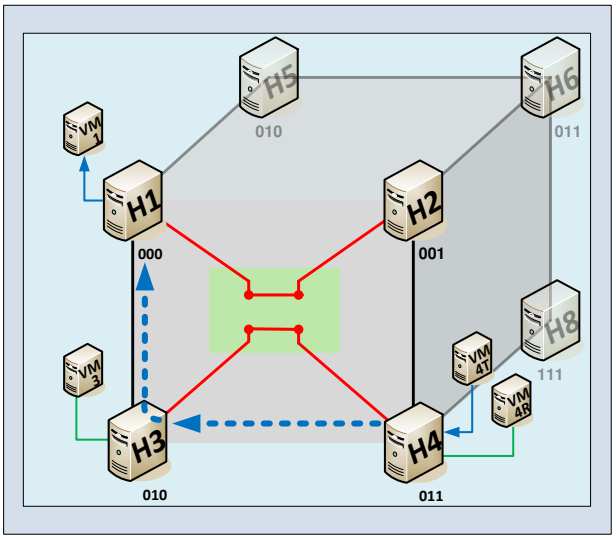


Figura 8.9: Intervalo de 0s a 60s com o fluxo de VM4T para VM1 a 300 Mbps. Configuração com as chaves em *barra* com trafego de trânsito a 300 Mbps em *H3*.

No intervalo 61s a 120s, as chaves continuam no estado *barra* conforme ilustrado na Figura 7.12, o fluxo a *300 Mbps* (linha pontilhada) iniciado por *VM3* é encaminhado diretamente para *VM4,* conforme mostrado no gráfico tráfego de dados da Figura 8.7. Observe que o uso de CPU de *H3* aumentou para aproximadamente 50%, conforme o gráfico de uso de CPU Figura 8.8. Esse aumento ocorreu devido à demanda de geração de tráfego da *VM3* hospedada neste *Host*. Como não há nenhum tráfego de trânsito, o *controlador A-SDN* não realizou nenhuma ação.

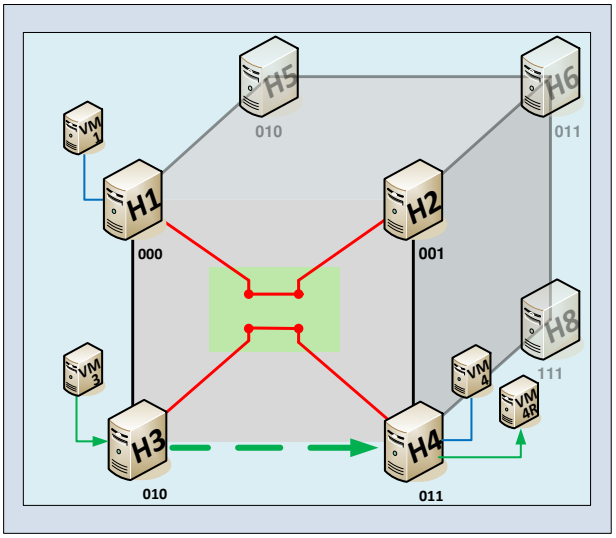


Figura 8.10: Intervalo de 61s a 120s com o fluxo de VM3 para VM4R a 300 Mbps. Configuração com as chaves em *barra* com encaminhamento direto.

No intervalo 121s a 180s, as chaves continuam no estado *barra* conforme ilustrado na Figura 8.11, o fluxo a *150 Mbps* (linha tracejada) iniciado por *MV3* é encaminhado

diretamente para *H4* que hospeda a *VM4R*. Ao mesmo tempo, o fluxo a *300 Mbps* (linha pontilhada) iniciado por *VMT4* é encaminhado por *H3* até a *VM1*, conforme mostrado no gráfico tráfego de dados da Figura 8.7. Observe que o uso de CPU de *H3* reduziu para 40%, conforme o gráfico de uso de CPU Figura 8.8. Essa redução ocorreu devido à diminuição, de *300 Mbps* para *150 Mbps,* da taxa de geração de tráfego da *VM3* hospedada neste *Host*. Observe, novamente, que o tráfego de trânsito em *H3* está abaixo do limite, para sua carga de CPU, de acordo com a Figura 8.7. Portanto, o *controlador A-SDN* não realizou nenhuma ação.

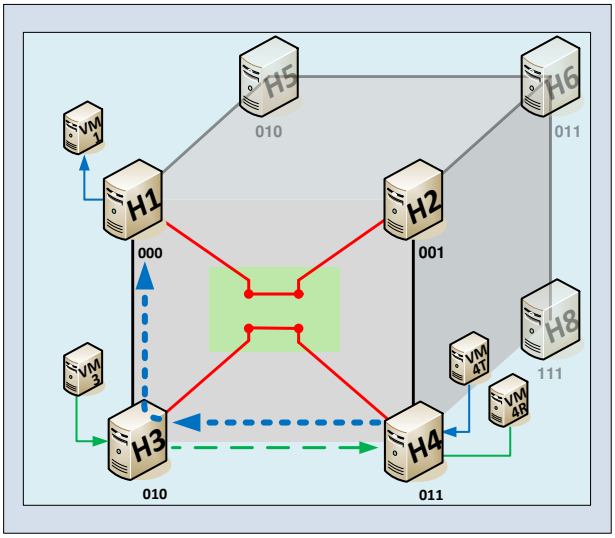


Figura 8.11: Intervalo de 121s a 180s com os fluxos de VM4T para VM1 a 300 Mbps; e de VM3 para VM4R a 150 Mbps. Configuração com as chaves em *barra* com trafego de trânsito a 300 Mbps em *H3* e com encaminhamento direto para VM4R.

No intervalo 181s a 360s, as chaves continuam no estado *barra* conforme ilustrado na Figura 8.12(a), o fluxo a *300 Mbps* (linha pontilhada) iniciado por *VMT4* é encaminhado por *H3* até a *VM1*. Ao mesmo tempo, o fluxo a *300 Mbps* (linha tracejada) iniciado por *MV3* encaminhado diretamente para *VM4R*, conforme mostrado no intervalo de 180s a 210s do gráfico tráfego de dados da Figura 8.7. Essas demandas elevam o uso de CPU de *H3* para valores próximos a 60%, conforme o gráfico de uso de CPU Figura 8.8. Deste modo, a combinação das cargas em *H3* ultrapassa o limite tráfego de trânsito, de acordo com a Figura 8.7. Então, *H3* reporta sua carga para o *controlador A-SDN*, aos 210s, que reconfigura o cenário de acordo com Figura 8.12 (b). Desta forma, o fluxo entre *VM4T* e *VM1* é encaminhado diretamente, enquanto o fluxo entre *VM3* e *VM4R* é encaminhado via *H2*. Esta ação redistribui a carga de CPU, reduzindo em *H3* e aumenta em *H2*. Ao mesmo tempo, melhorando a qualidade do tráfego entre *VM4T* e *VM1*, conforme pode ser observado, a

partir de 210s, no gráfico de uso de CPU e de tráfego de trânsito da Figura 8.8, respectivamente.

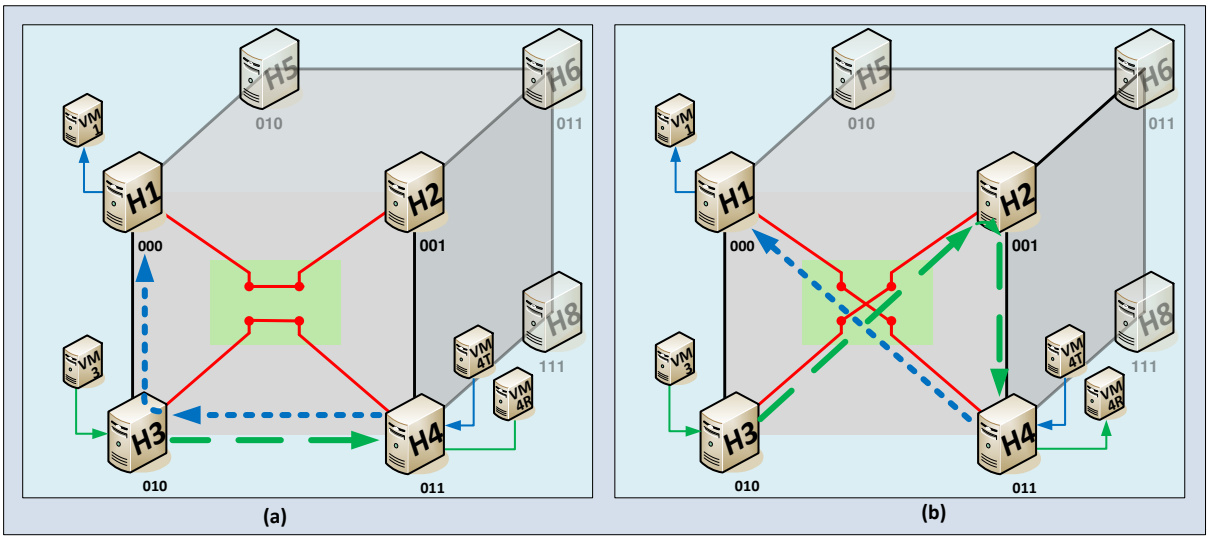


Figura 8.12: Intervalo de 180s a 360s com os fluxos de VM4T para VM1 a 300 Mbps; e de VM3 para VM4R a 300 Mbps. (a) configuração com as chaves em *barra* com trafego de trânsito a 300 Mbps em *H3*; (b) configuração com as chaves em *cruz* com trafego de trânsito a 300 Mbps em *H2*.

Os resultados desse cenário comprovam que o *controlador A-SDN*, utilizando um modelo de limites adaptativo, permite redistribuir, além do tráfego de trânsito, as cargas de CPU. Ainda, como os valores de carga são expressos em percentuais, a distribuição das cargas em um ambiente heterogêneo é realizada de forma proporcional aos recursos de cada *nó de computação*, ou seja, de forma mais justa.

8.3 Orquestração das máquinas virtuais em função do uso de CPU e memória

Para demonstrar como orquestração autônoma, promovida pelo *controlador A-SDN*, atua em relação às cargas de CPU e memória dos *nós de computação*, foi elaborado um cenário, onde cada *Host* hospedou três VMs OpenWrt, e realizado dois experimentos. O primeiro utilizou uma carga artificial de *99%* de CPU, enquanto segundo utilizou uma carga artificial de 65% de memória física. Em ambos os experimentos a carga artificial foi alocada sequencialmente nos *Hosts* a cada *30s*. Porém, em cada *Host* a carga permanecia ativa por *35s* para garantir que a média, no intervalo de verificação da carga, fosse acima do limite estabelecido. Neste cenário, o limite utilizado foi de 70% da capacidade, tanto para CPU quanto para memória, porém esse valor pode ser ajustado conforme a necessidade do ambiente.

Durante os experimentos, o *script shell de monitoramento* de cada *nó de computação* registrou as cargas de CPU e memória. Os resultados do primeiro experimento (carregamento de CPU) foram plotados nos gráficos da Figura 8.13, enquanto os dados do segundo experimento (carregamento de memória) foram plotados nos gráficos da Figura 8.14.

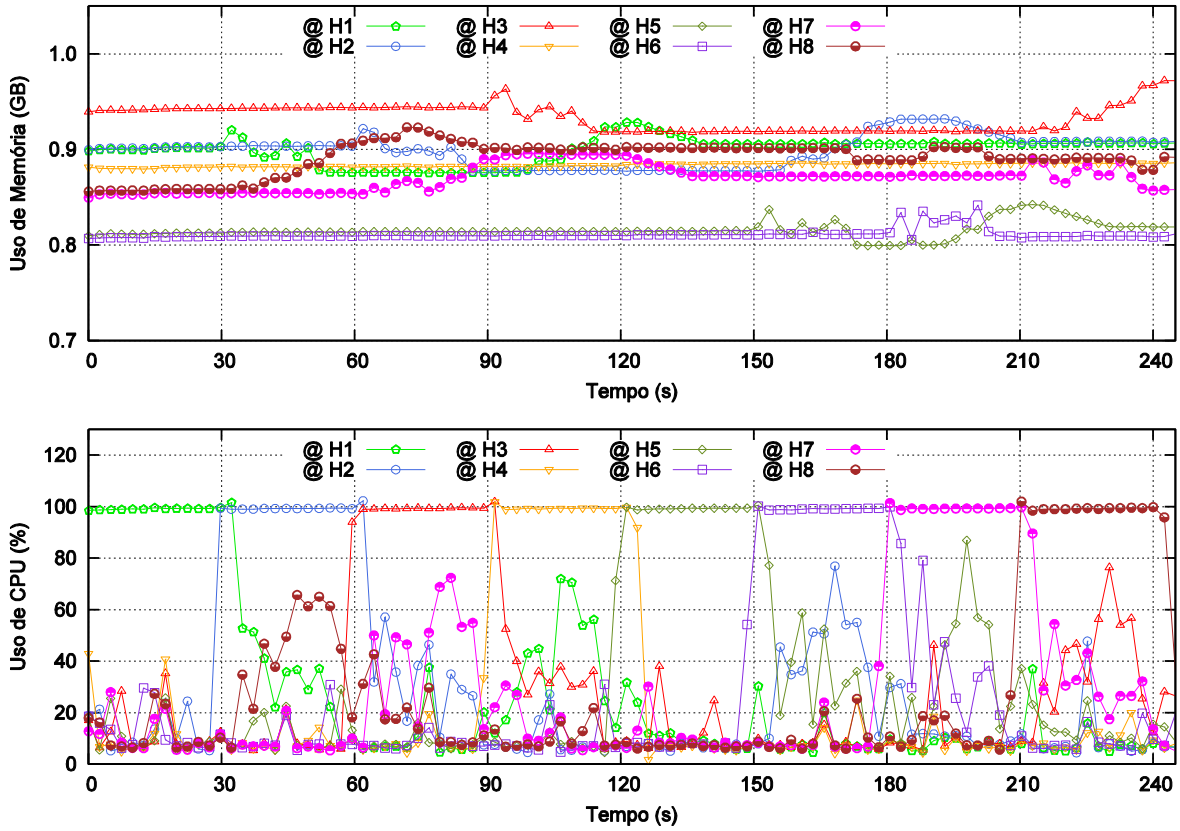


Figura 8.13: Orquestração das máquinas virtuais em função do uso de CPU

No intervalo de *0s* a *29s* do gráfico de uso de CPU da Figura 8.13 é possível observar que o *Host H1* está utilizando praticamente *100%* de sua CPU, enquanto os demais *Hosts* estão com baixa utilização de CPU. Nesse mesmo intervalo, o gráfico de uso de memória não apresenta variação significa em nenhum *Host*. No início do segundo intervalo, que vai de *30s* a *59s*, é a vez de *H2* receber a carga de CPU, elevando a utilização para praticamente *100%*. Ainda nesse intervalo, é possível observar que a carga sobre *H1* é removida. Porém, o evento mais importante é a redução da utilização de memória de *H1*, e o aumento da utilização de memória em *H8*, conforme ilustrado no gráfico de uso de memória da Figura 8.13.

Esse evento de movimentação de memória ocorre pelo fato de *H1* reportar para o *controlador A-SDN* que sua carga média de CPU, dos últimos *30s*, excedeu o limite estabelecido. Ao receber esse tipo de evento, o *controlador A-SDN* busca por um *Host* com

recursos disponíveis para receber uma VM, na tentativa de aliviar a carga sobre *H1*. No cenário desse experimento, o controlador encontra diversos *Hosts* capazes a receber uma VM. Dessa forma, ele ordena os candidatos, nesse caso, por CPU a fim de escolher o *Host* com mais recursos disponíveis. Então, o controlador envia uma resposta para *H1* ordenando a migração de uma VM específica para o *Host H8*, conforme destacado no fragmento do arquivo de *log* de *H1,* contido no Quadro 8.1. Os demais intervalos dos gráficos da Figura 8.13, apresentam uma situação semelhante envolvendo outros *Hosts*. No entanto, é importante notar que as cargas são distribuídas de forma a manter o uso dos recursos, proporcionalmente, iguais entre os *Hosts*.

Quadro 8.1: Fragmento do arquivo de *log* de H1, que mostra o evento de limite de CPU excedido e a ordem vinda do *controlador A-SDN* para migrar uma VM específica.

```
##########      Workload in node-000 at 17:44:42      ##########
CPU workload:
sys       : 2.16 %
total     : 99.14 %
usr       : 96.79 %
guest     : 0.07 %

MEM workload:
memory_used     : 48.17 %

BWM workload:
int_traf_in         : 0.00 Mbps
int_traf_out        : 0.00 Mbps
nw_traf_in          : 0.03 Mbps
nw_traf_out         : 0.00 Mbps
tr_traf    : 0.00 Mbps
vm_traf_in          : 0.00 Mbps
vm_traf_out         : 0.02 Mbps
Starting new HTTP connection (1): 192.168.1.201

##########      Report high workload state to A-SDN Controller      ##########
Workload sent to SDN controller: {"mem": 48.17, "bwm": 0.0, "hostname": "node-000",
"compute_node_id": 2, "cpu": 99.14}
A-SDN Controller response: {u'dsthost': u'node-111', u'task': u'migrate', u'dstcompute_node_id': u'5', u'li-
vemigration': u'True', u'instance': u'd5681a8b-95ee-4712-bcbe-17f15183a4e7', u'response': u'ok'}
Executing Task: migrate
```

No fragmento do arquivo de *log* mostrado no Quadro 8.1, é possível observar os percentuais de utilização de CPU para: tarefas do sistema operacional (*sys*), tarefas do usuário (*usr*), tarefas das VMs (*guest*) e o uso total (*total*). Ainda é registrado o uso percentual do uso de memória (*memory_used*), bem com o tráfego de rede (*BWM workload*) dividido entre: tráfego da rede interna (*int_traf_in e int_traf_out*), tráfego nas interfaces físicas (*nw_traf_in*

e nw_traf_out), tráfego de trânsito (*tr_traf*) e o tráfego das portas das VMs (*vm_traf_in e vm_traf_out*). Por fim, a notificação de carga excedida para o *controlador A-SDN*, com os percentuais das cargas e identificação do *Host*. E a resposta do controlador com: a tarefa (*task*), o *Host* de destino (*dsthost*), o tipo de migração (*livemigration*) e a VM a ser migrada (*instance*).

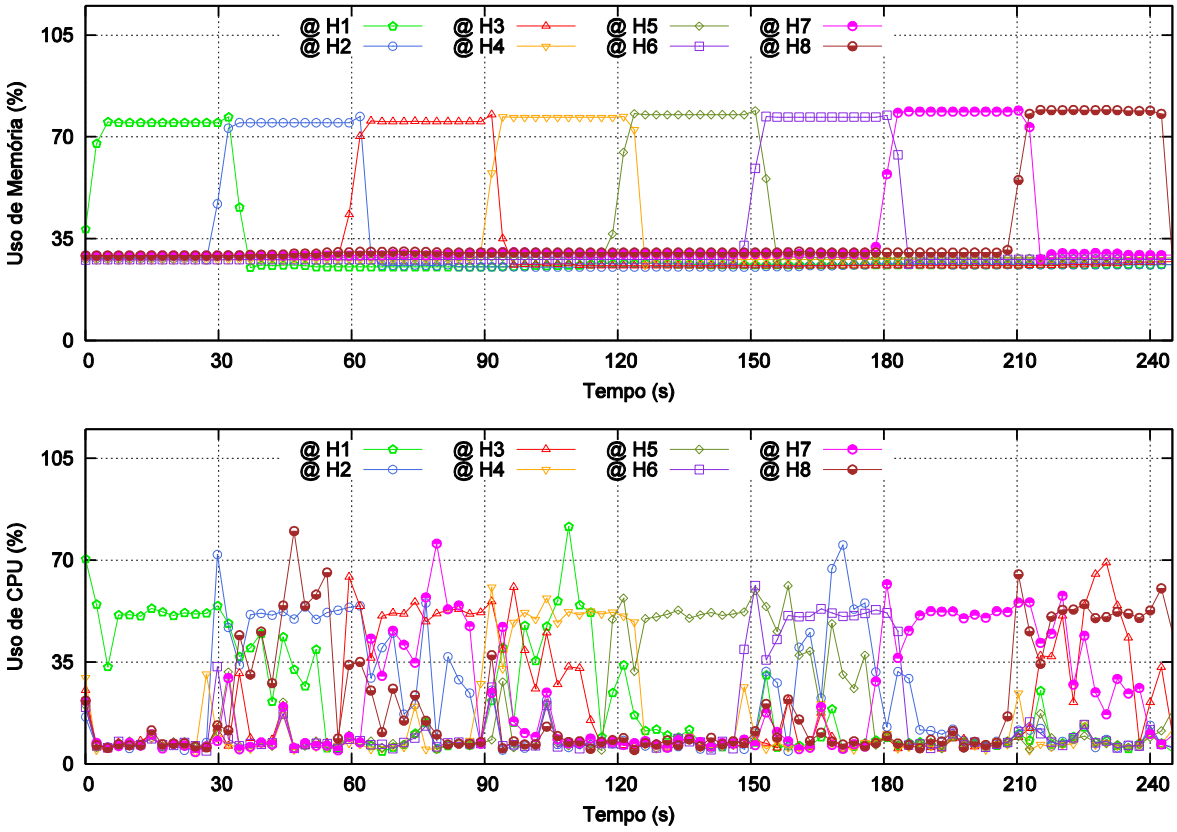


Figura 8.14: Orquestração das máquinas virtuais em função do uso de memória

A orquestração com base na utilização de memória ocorre de forma análoga à orquestração com base utilização de CPU. Por isso, sua explanação será descrita de forma sucinta. Ainda, para uma observação mais precisa, o uso de memória foi apresentado em valores percentuais no gráfico superior da Figura 8.14.

Observe que no intervalo de *0s* a *29s*, o uso de memória em *H1* é maior que *70%*, enquanto nos demais *Hosts* o uso de memória não ultrapassa *35%*. Observe ainda, que apenas a CPU de *H1* apresentou utilização elevada, em função do processo de alocação de memória. Já no segundo intervalo, de *30s* a *59s*, o processo de alocação memória sobre *H2* elevou seu uso de CPU. Por sua vez, houve um aumento do uso de CPU em *H8*, para receber a VM migrada de *H1*, conforme o gráfico de uso de CPU da Figura 8.14. Essa migração ocorreu em

reposta ao evento reportado por *H1* ao *controlador A-SDN*, informando que sua carga média de memória, dos últimos *30s*, excedeu o limite estabelecido. Os demais intervalos dos gráficos da Figura 8.14, apresentam uma situação semelhante envolvendo outros *Hosts*.

8.4 Estabilização da plataforma para cenários de alta carga

Este último cenário apresenta uma situação hipotética, onde todos os *nós de computação* da rede estão com cargas acima do limite, conforme apresentado na Figura 8.15.

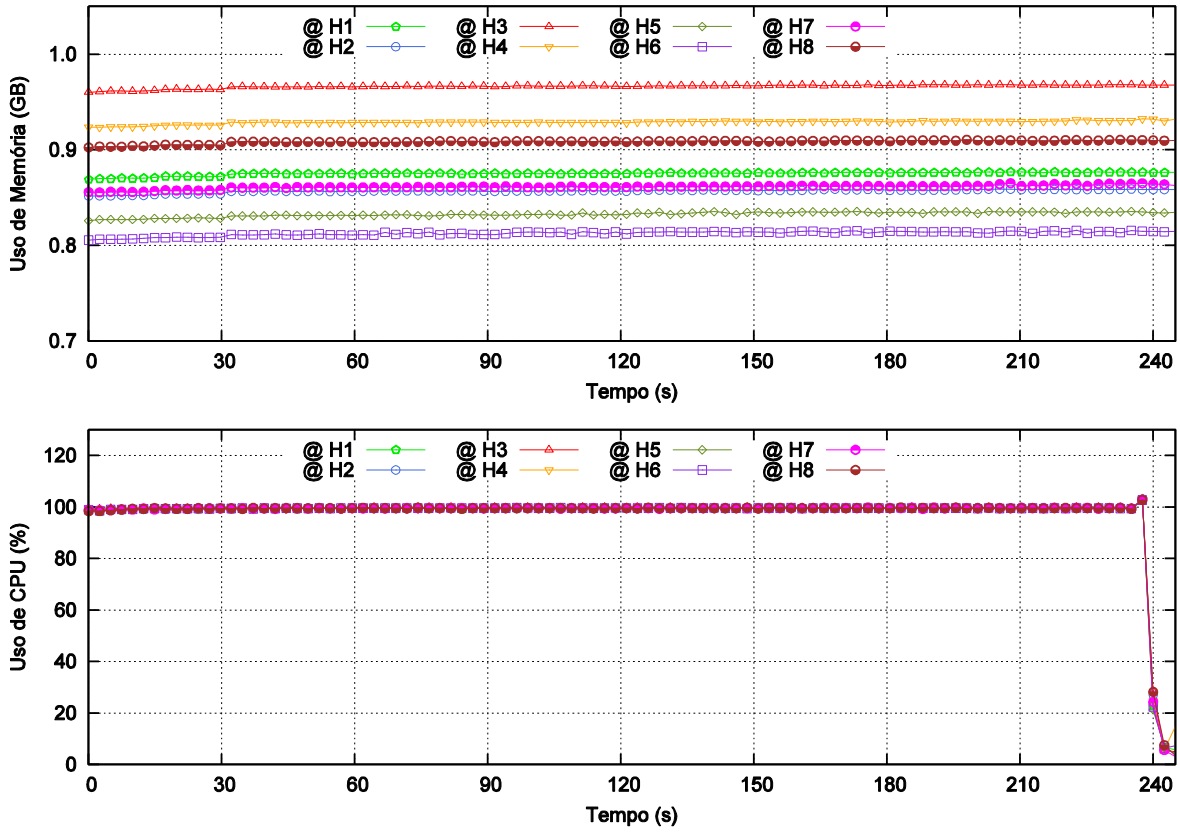


Figura 8.15: Estabilização da plataforma para cenário hipotético de sobrecarga de CPU em todos os *nós de computação*.

Observe que todos os *Hosts* da estão com *100%* de carga de CPU, conforme o gráfico de uso de CPU da Figura 8.15. Desta forma, cada *Host* reporta para o controlador, a cada *30s*, que sua carga média de CPU, neste intervalo de verificação da carga, excedeu o limite estabelecido. No entanto, nenhuma alteração significativa é observada no gráfico de uso de memória da Figura 8.15. Indicando que o controlador não realizou nenhuma operação. De fato, não há nenhum *Host* com recursos disponíveis para receber uma VM. Esse mecanismo impede que uma VM seja migrada de um *Host* para o outro, em cenários de sobrecarga como este, evitando um regime de instabilidade na arquitetura.

Para entender como o *controlador A-SDN* reage diante dos eventos de sobrecarga reportados pelos *Hosts*, o algoritmo que busca o *Host* mais adequado para receber a uma VM, em outras palavras, o escalonar do *controlador A-SDN* é apresentado no Quadro 8.2, e discutido em seguida.

Quadro 8.2: Algoritmo do escalonador do *controlador A-SDN*.

```
 1:  function Escalonador (prioridade, origem)
 2:    if prioridade == "cpu" then
 3:      listaOrdenada = ordenaNósPorCpu()
 4:      for destino in listaOrdenada
 5:          if (destino.cpu < origem.cpu) and (destino.cpu < limitMaximoCpu)
 6:              if destino.memória < limitMaximoMemória
 7:                  return destino
 8:              end if
 9:          else
10:              return
11:          end if
12:       end for
13:    end if
14:    if prioridade == " memória" then
 ...        lógica semelhante a utilizada para escolher por CPU
25:    end if
26:    return
27:  end function
```

Observe no Quadro 8.2 que o algoritmo do escalonador recebe dois parâmetros: a prioridade de ordenação e o nó de origem. Se a prioridade de ordenação for por CPU, o algoritmo utiliza uma função para ordenar os *nós de computação* por CPU de forma crescente e atribui a variável *listaOrdenada*. Assim, os nós desta lista são testados, a fim de encontrar algum nó que atenda simultaneamente os seguintes critérios: estar com carga de CPU abaixo do nó de origem e abaixo do limite máximo de carga de CPU (linha 5); estar com carga de memória abaixo do limite máximo de carga de memória (linha 6). Se os critérios da linha 5 não forem atendidos, o algoritmo não precisa testar nenhum outro nó, pois todos os nós restante na lista estarão com carga de CPU maior. No entanto, se o critério da linha 6 não foi atendido, ainda é possível encontrar um nó que atenda aos critérios, na lista. Caso algum nó atenda os critérios da linha 5 e 6, o algoritmo retorna este nó, caso contrário, nenhum nó é retornado. Se a prioridade de ordenação for por memória, o algoritmo executa uma lógica semelhante a descria acima.

Na maioria das vezes, o escalonador encontrará o nó desejado no início da *listaOrdenada*. Desta forma, o maior esforço computacional fica por conta da ordenação da lista. Neste ponto é importante ressaltar que, as cargas de trabalho dos *nós de computação* estão armazenadas na base de dados compartilhada, mantida por um sistema gerenciador de banco de dados (SGBD). Assim, a função de ordenação do escalonador se resume a uma consulta ao banco de dados, solicitando as médias de carga dos nós, no intervalo de tempo desejado (*30s* neste trabalho), ordenadas pelo critério recebido como parâmetro. Supondo uma arquitetura com *10000 nós de computação*, supondo ainda que sejam armazenadas, a cada *1s*, as cargas de CPU e memória de cada nó. Serão registrados *600000* eventos na base de dados, no intervalo de *30s*. A princípio, este número de eventos aparenta ser demasiadamente grande para ordenar em tempo hábil. No entanto, os SGBD são programas especializados em lidar com enormes volumes de dados. Para isso, eles utilizam algoritmos otimizados de agregação (média, soma, etc.) e ordenação de valores, além de armazenarem as informações de forma a agilizar a busca ao máximo. No exemplo da Figura 8.16, o servidor *MySql* do ambiente experimental realizou a agregação de *1085030* registros em apenas *0.88s*, para calcular as médias de CPU do usuário, das VMs e total. Por fim, pode-se atribuir um *tempo de validade* para a uma ordenação, de forma que o *controlador A-SDN* utilize essa ordenação para analisar os eventos reportados pelos *Hosts,* durante o *tempo de validade*, sem a necessidade de uma nova consulta a base de dados.

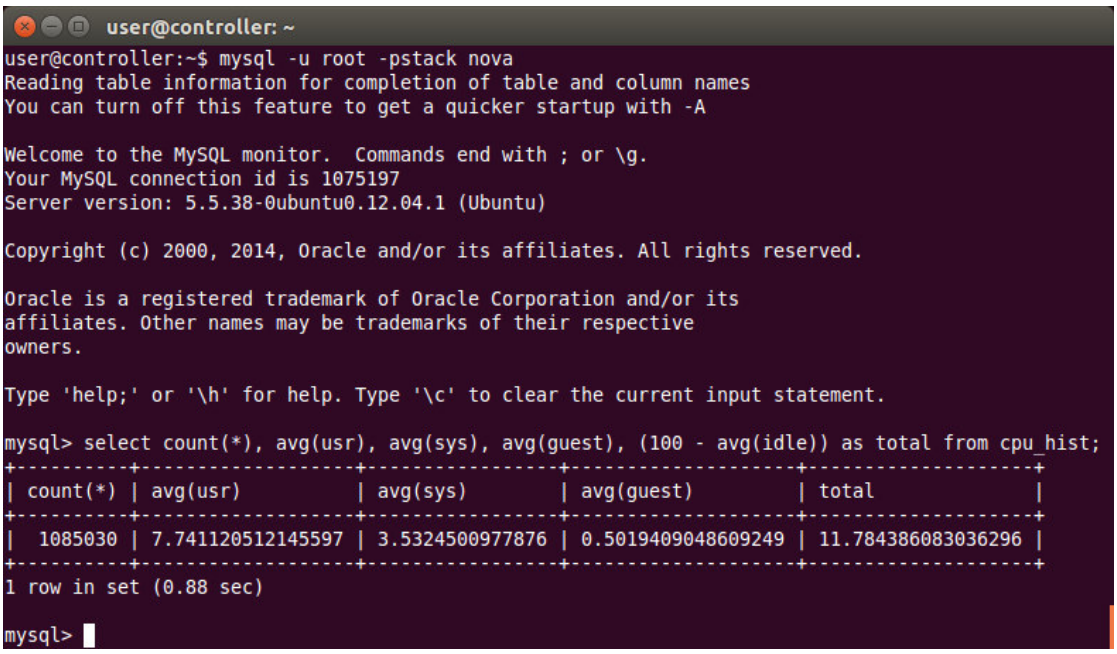


Figura 8.16: Tempo de agregação de mais de 1 milhão de registros de uso de CPU pelo o servidor MySql 5.5.38, utilizado no ambiente experimental para suportar a base de dados compartilhada.

Capítulo 9 – Conclusão

O capítulo de conclusão discute as contribuições deste trabalho e indica os trabalhos futuros a serem investigados.

9.1 Twin Datacenter Interconnection Topology

A primeira contribuição deste trabalho propõe o uso de uma topologia física baseada na classe de grafos 2-geodesicamente-conexo denominada *Grafos Gêmeos* em redes de *data center* centrado em servidores, a qual foi chamada de *Twin Datacenter Interconnection Topology*. Ao comparar a topologia *Twin* com as topologias: *Fat-Tree*, *DCell*, *BCube* e *Hipercubo*, em relação aos aspectos de: custo, escalabilidade, tolerância da falhas, resiliência e desempenho. Os resultados demonstraram que a topologia *Twin*:

- Beneficiam-se do fato dos *Grafos Gêmeos* utilizarem o menor número possível de enlaces para interligar os nós, com no mínimo duas geodésicas entre os pares de nós não adjacentes. Isso reduz o custo de capital (CAPEX) e também o custo operacional (OPEX), pois naturalmente menos enlaces consomem menos energia. Portanto, a topologia *Twin* apresenta uma relação custo-benefício ótima em relação a topologia de rede do *data center*;
- Beneficiam-se dos processos de crescimento e união dos *Grafos Gêmeos* que garante escalabilidade com granularidade a partir de uma unidade. Assim, é teoricamente possível encontrar uma topologia *Twin* que atenda uma rede de *data center* com qualquer quantidade de servidores. Ainda, esta rede pode ser incrementada com um ou mais servidores; ou unida à outra rede (*Twin*) com apenas quatro novos enlaces, gerando uma terceira rede *Twin;*

- Beneficiam-se do fato dos *Grafos Gêmeos* serem 2-geodesicamente-conexo, isto é, apresentar dois caminhos disjuntos por nó para cada par de nós não adjacentes. Este fato confere tolerância a falhas e alta resiliência. De forma que, se a falha ocorrer em um nó ou um enlace, o tráfego ainda pode ser encaminhado pelo melhor caminho, em relação ao número de saltos.

Por outro lado, algumas implicações práticas e ainda não resolvidas, inviabilizam o uso dos *Twin* em redes de *data center* de grande escala, conforme lista do abaixo:

i. A dificuldade de encontrar uma topologia que atenda aos requisitos do projeto, devido a complexidade computacional de gerar todas as possiblidades para então escolher a que melhor se adequa; No entanto, um algoritmo que otimize este processo pode ser desenvolvido, com base em filtros aplicados durante o processo de crescimento;

ii. Do melhor do nosso conhecimento, não existe um algoritmo de roteamento específico para esta topologia. Apesar dos resultados com o ECMP apresentarem uma boa distribuição, um algoritmo leve de roteamento, que pudesse ser implementado em nível de *kernel*, poderia reduzir o uso de CPU em nós intermediários;

iii. A dificuldade de implantar a comutação óptica distribuída para reconfigurar os enlaces físicos, sem prejuízo as propriedades da topologia.

Neste contexto, concluiu-se que a topologia *Twin* apresentam grande potencial para reduzir custos e garantir as propriedades fundamentais das redes de *data center*. No entanto, alguns aspectos ainda precisam ser tratados para viabilizar sua aplicação em redes de grande porte. Devido a limitação *ii* conflitar com hipótese 2 e a limitação *iii* conflitar com a hipótese 4. A solução de compromisso limitou o desenvolvimento do restante do trabalho à utilização do *Hipercubo*.

9.2 TRIIIAD

A segunda contribuição deste trabalho apresenta o projeto, a implementação e a avaliação da *TRIIIAD*; uma arquitetura autonômica composta por três camadas horizontais e um plano vertical de controle, gerência e orquestração; desenvolvida sobre os seguintes pilares:

i. Uma camada de virtualização baseada no conceito de nuvem. Isto permitiu o compartilhamento dos recursos físicos e compatibilizou a *TRIIIAD* com as tecnologias de redes amplamente difundidas, tais como: IPv4 e *Ethernet*.

ii. Uma camada de encaminhamento baseada em uma rede de *data center* centrado em servidores e conectada por uma topologia *Hipercubo*. Isto permitiu o implementação de um algoritmo de roteamento: leve e baseado na operação XOR para escolher o menor caminho, apenas com o conhecimento local do nó; em nível de *kernel* utilizando o módulo *datapath* do *OvS;* sobre uma arquitetura de uso geral *x86* como uma *função virtual de rede;* e configurado pelo *controlador SDN;*

iii. Uma camada de hibrida reconfigurável baseada no *Cross-Braced Hypercube*. Isso permitiu a "*opticalização*" do processo de comutação distribuído e gradual da seguinte forma: chaves ópticas 2x2 foram distribuídas na camada física, criando planos de chaveamentos capazes de reduzir o número médio de saltos no encaminhamento de pacotes, colaborando para redução do tráfego de trânsito.

iv. Um plano de controle, gerência e orquestração, que atua de forma vertical na arquitetura, alinhando o funcionamento das três camadas e mantendo-as agnósticas entre si. Este plano foi viabilizado por um *controlador SDN* aumentado, que se integrou à dinâmica da orquestração, de forma a manter a consistência entre as informações da rede e as decisões tomadas na camada de virtualização.

Os resultados dos experimentos sobre a plataforma demostraram que: o pilar *ii* valida a hipótese 2; o pilar *iii* valida a hipótese 4; os pilares *ii* e *iv* validam a hipótese 3; por fim, o pilar *iv* conferiu a arquitetura as habilidades exigidas para provar a hipótese 5.

Ainda com base nos resultados foi possível concluir que: (*i)* a integração do SDN ao processo de orquestração da nuvem foi fundamental para garantir a estabilidade da plataforma; (*ii)* é possível implementar um mecanismo de roteamento/encaminhamento leve e eficiente em nível de *kernel,* sobre o *Open vSwitch,* com menos de 700 linhas de código, controlado por SDN; (*iii)* o escalonador, ao se basear no uso dos recursos dos servidores (CPU, memória e tráfego de trânsito), promove um balanceamento de carga justo e

proporcional à capacidade de cada servidor da rede; (*iv)* a orquestração da topologia baseada em limites adaptativos de carga dos servidores físicos contribui para reduzir as migrações de VM e as comutações da camada híbrida de reconfiguração; (*v)* embora a proposta desta tese esteja focada em redes de *data center* centrado em servidores, é perfeitamente possível aplicar as ideias apresentada aqui em outras topologias, inclusive em *data center* centrado em redes; (*vi)* por outro lado, os resultados também mostraram que embora o OpenStack seja uma arquitetura muito interessante, ela precisa evoluir para reduzir a sobrecarga gerada com as cópias de pacotes, a fim de aliviar o consumo de CPU dos servidores físicos.

9.3 Trabalhos futuros

Durante o desenvolvimento deste trabalho foram identificadas diversas oportunidades e necessidades de trabalhos que podem ser futuramente investigadas, tais como:

- Desenvolver os algoritmos: (*i)* de escolha de topologias *Twin* para atender os requisitos dos projetos de redes de grande escala; (*ii*) de roteamento para otimizar o uso das geodésicas entre os pares de nós da topologia. Estes avanços viabilizariam o uso das topologias *Twin* em rede de grande escala.
- Adicionar ao escalonador do A-SDN políticas de agrupamento de VMs com base em seus tráfegos, de forma a minimizar o tráfego de trânsito total na rede.
- Reduzir o número de cópias de pacotes da rede interna criada pelo OpenStack;
- Investigar os possíveis benefícios de comutadores elétricos 2x2, para possibilitar a comutação em nível de fluxos de dados, além a comutação de todo o circuito;
- Modificar o algoritmo de encaminhando para promover a engenharia de tráfegos sobre os múltiplos caminhos da topologia *Hipercubo.* Por exemplo, com prioridades para escolha dos caminhos de acordo com as necessidades. Assim, tráfegos tolerantes a latência poderiam ser encaminhados por caminhos maiores.

Publicações diretamente relacionadas ao tema

[A] Hybrid Reconfiguration for Upgrading Datacenter Interconnection Topology

Abstract *— This paper proposes the introduction of 2x2 magneto- optical switches as a reconfigurable physical layer with no link loss to provide higher throughput, and simultaneously, reduce CPU packet forwarding load in server-centric datacenters.*

G. L. Vassoler, F. R. de Souza, P. P. P. Filho, M. R. N. Ribeiro, and F. R. De Souza, "Hybrid reconfiguration for upgrading datacenter interconnection topology," in IEEE Photonics Conference 2012, 2012, no. Ccm, pp. 782–783.

[B] Twin Datacenter Interconnection Topology

Abstract *— A new twin-graph-based interconnection topology for server-centric networks exploits graph theory to cost-effectively improve network scalability and resilience. A comparison of twin interconnection topologies and other datacenter topologies shows that the former have a lower link cost, scale with fine granularity, and are efficient, fault tolerant, and resilient, because they enable traffic balancing under both normal and faulty operating conditions.*

G. L. Vassoler, M. H. . Paiva, M. R. N. Ribeiro, and M. E. V. Segatto, "Twin Datacenter Interconnection Topology," IEEE Micro, vol. 34, pp. 8–17, 2014.

[C] FlexForward: Enabling an SDN manageable forwarding engine in Open vSwitch

Abstract *— This paper presents FlexForward, an approach for dynamically manage SDN data plane forwarding mechanisms. This feature leverages SDN flexibility in two aspects: (i) it provides a new management method, in which topological infrastructure*

requirements are satisfied by the most appropriate forwarding mechanism, residing in each managed network element; (ii) further decoupling between control and data planes in SDN by offering tableless forwarding support. As proof of concept, an implementation with different forwarding methods is carried out in Open vSwitch. Results show that FlexForward is able to achieve seamless switchover between forwarding methods. It also allows efficient tableless forwarding implementations, improving by up to 31% in latency, and 90% in throughput, when compared to regular OpenFlow.

R. D. Vencioneck, G. Vassoler, M. Martinello, M. R. N. Ribeiro, and C. Marcondes, “FlexForward: Enabling an SDN manageable forwarding engine in Open vSwitch,” in 10th International Conference on Network and Service Management (CNSM) and Workshop, 2014, pp. 296–299.

Referências

[1] I. Drago, M. Mellia, M. M Munafo, A. Sperotto, R. Sadre, and A. Pras, "Inside dropbox: understanding personal cloud storage services," in *the 2012 ACM conference*, 2012, p. 481.

[2] Google, "Google Cloud Print," 2015. [Online]. Available: http://www.google.com/cloudprint/learn/.

[3] M. Weiser, "Ubiquitous Computing," *Computer (Long. Beach. Calif).*, vol. 26, no. 10, pp. 71–72, 1993.

[4] P. Mell and T. Grance, "The NIST Definition of Cloud Computing Recommendations of the National Institute of Standards and Technology," 2011.

[5] V. Sarathy, P. Narayan, and R. Mikkilineni, "Next Generation Cloud Computing Architecture: Enabling Real-Time Dynamism for Shared Distributed Physical Infrastructure," *2010 19th IEEE Int. Work. Enabling Technol. Infrastructures Collab. Enterp.*, pp. 48–53, 2010.

[6] K. Wu, J. Xiao, and L. Ni, "Rethinking the architecture design of data center networks," *Front. Comput. Sci.*, vol. 6, no. 5, pp. 596–603, 2012.

[7] F. L. Verdi, C. E. Rothenberg, R. Pasquini, and M. F. Magalhães, "Novas Arquiteturas de Data Center para Cloud Computing," in *XXVIII Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos-Gramado-RS*, 2010.

[8] G. F. Pfister, "An Introduction to the InfiniBand Architecture," *High Perform. Mass Storage Parallel {I/O} Technol. Appl.*, no. 42, pp. 617–632, 2001.

[9] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, and J. N. Seizovic, "Myrinet: a gigabit-per-second local area network," *IEEE Micro*, vol. 15, no. 1, pp. 29–36, 1995.

[10] Top 500.org, "Statistics: Interconnect Family," 2015. [Online]. Available: http://www.top500.org/statistics/list/. [Accessed: 31-Mar-2015].

[11] M. Al-Fares, A. Loukissas, and A. Vahdat, “A scalable, commodity data center network architecture,” in *Proceedings of the ACM SIGCOMM 2008 conference on Data communication - SIGCOMM ’08*, 2008, pp. 63–74.

[12] R. Niranjan Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, and A. Vahdat, “PortLand: A Scalable Fault-Tolerant Layer 2 Data Center Network Fabric,” in *Proceedings of the ACM SIGCOMM 2009 conference on Data communication - SIGCOMM ’09*, 2009, p. 39.

[13] A. Greenberg, J. R. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. A. Maltz, P. Patel, and S. Sengupta, “VL2 : A Scalable and Flexible Data Center Network,” pp. 51–62, 2009.

[14] C. Guo, H. Wu, K. Tan, L. Shi, Y. Zhang, and S. Lu, “Dcell: a scalable and fault-tolerant network structure for data centers,” *ACM SIGCOMM Comput. Commun. Rev.*, pp. 75–86, 2008.

[15] C. Guo, G. Lu, D. Li, H. Wu, X. Zhang, Y. Shi, C. Tian, Y. Zhang, and S. Lu, “BCube: a high performance, server-centric network architecture for modular data centers,” *ACM SIGCOMM Comput. Commun. Rev.*, vol. 39, no. 4, pp. 63–74, Aug. 2009.

[16] C. Hong, L. Popa, and P. B. Godfrey, “Jellyfish : Networking Data Centers , Randomly,” in *NSDI*, 2012.

[17] Y. Saad and M. H. Schultz, “Data communication in hypercubes,” *J. Parallel Distrib. Comput.*, vol. 6, no. 1, pp. 115–135, Feb. 1989.

[18] H. Farhady, H. Lee, and A. Nakao, “Software-Defined Networking: A survey,” *Comput. Networks*, vol. 81, pp. 79–95, 2015.

[19] N. Mckeown, T. Anderson, L. Peterson, J. Rexford, S. Shenker, and S. Louis, “OpenFlow : Enabling Innovation in Campus Networks,” 2008.

[20] U. Hoelzle, “Openflow@ google,” *Open Netw. Summit*, 2012.

[21] Facebook, “Introducing ‘6-pack’: the first open hardware modular switch,” 2015. [Online]. Available: https://code.facebook.com/posts/717010588413497/introducing-6-pack-the-first-open-hardware-modular-switch/. [Accessed: 13-Apr-2015].

[22] European Telecommunications Standards Institute, “Network Functions Virtualisation,” 2012.

[23] G. Wang, D. G. Andersen, M. Kaminsky, K. Papagiannaki, T. S. E. Ng, M. Kozuch, and M. Ryan, “c-Through: Part-time Optics in Data Centers,” *ACM SIGCOMM Comput. Commun. Rev.*, vol. 40, no. 4, p. 327, Aug. 2010.

[24] N. Farrington, G. Porter, S. Radhakrishnan, H. H. Bazzaz, V. Subramanya, Y. Fainman, G. Papen, and A. Vahdat, “Helios: a hybrid electrical/optical switch architecture for modular data centers,” *ACM SIGCOMM Comput. Commun. Rev.*, vol. 40, no. 4, p. 339, Aug. 2010.

[25] M. F. Bari, R. Boutaba, R. Esteves, L. Z. Granville, M. Podlesny, M. G. Rabbani, Q. Zhang, and M. F. Zhani, "Data center network virtualization: A survey," *IEEE Commun. Surv. Tutorials*, vol. 15, no. 2, pp. 909–928, 2013.

[26] G. L. Vassoler, M. H. . Paiva, M. R. N. Ribeiro, and M. E. V. Segatto, "Twin Datacenter Interconnection Topology," *IEEE Micro*, vol. 34, pp. 8–17, 2014.

[27] A. Greenberg, J. Hamilton, D. A. Maltz, and P. Patel, "The Cost of a Cloud : Research Problems in Data Center Networks," vol. 39, no. 1, pp. 68–73, 2009.

[28] L. Shalev, J. Satran, E. Borovik, and M. Ben-yehuda, "IsoStack – Highly Efficient Network Processing on Dedicated Cores," in *USENIX Annual Technical Conference*, 2010.

[29] S. Azodolmolky, P. Wieder, and R. Yahyapour, "Cloud computing networking: Challenges and opportunities for innovations," *IEEE Commun. Mag.*, vol. 51, no. 7, pp. 54–62, 2013.

[30] S. Ghorbani, C. Schlesinger, M. Monaco, E. Keller, M. Caesar, J. Rexford, and D. Walker, "Transparent, Live Migration of a Software-Defined Network," *Proc. ACM Symp. Cloud Comput. - SOCC '14*, pp. 1–14, 2014.

[31] B. Boughzala, R. Ben Ali, M. Lemay, Y. Lemieux, and O. Cherkaoui, "OpenFlow supporting inter-domain virtual machine migration," *2011 Eighth Int. Conf. Wirel. Opt. Commun. Networks*, pp. 1–7, May 2011.

[32] Q. Li, J. Huai, J. Li, T. Wo, and M. Wen, "HyperMIP: Hypervisor controlled mobile IP for virtual machinem live migration across networks," in *Proceedings of IEEE International Symposium on High Assurance Systems Engineering*, 2008, pp. 80–88.

[33] M. H. M. Paiva, G. Caporossi, and M. E. V. Segatto, "Twin Graphs for OTN Physical Topology Design," *Les Cah. du GERAD*, vol. 48, pp. 1–12, 2013.

[34] M. Casado, T. Koponen, S. Shenker, and A. Tootoonchian, "Fabric: a retrospective on evolving SDN," in *Proceedings of the first workshop on Hot topics in software defined networks - HotSDN '12*, 2012, p. 85.

[35] H. Cui, D. Rasooly, M. Ribeiro, and L. Kazovsky, "Optically Cross-Braced Hypercube: a Reconfigurable Physical Layer for Interconnects and Server-Centric Datacenters," *Opt. Fiber Commun. Conf.*, vol. 1, p. OW3J.1, 2012.

[36] Ryu SDN Framework Community, "WHAT'S RYU?," 2015. [Online]. Available: http://osrg.github.io/ryu/. [Accessed: 31-Mar-2015].

[37] OpenStack.org, "OpenStack: The Open Source Cloud Operating System," 2015. [Online]. Available: https://www.openstack.org/software/. [Accessed: 31-Mar-2015].

[38] Etsi and J. Quittek, "GS NFV-MAN 001 - V1.1.1 - Network Functions Virtualisation (NFV); Management and Orchestration," vol. 1, pp. 1–184, 2014.

[39] A. Farley and A. Proskurowski, “Minimum self-repairing graphs,” *Graphs Comb.*, vol. 13, no. 4, pp. 345–352, 1997.

[40] L. Popa, S. Ratnasamy, G. Iannaccone, A. Krishnamurthy, and I. Stoica, “A cost comparison of datacenter network architectures,” in *Proceedings of the 6th International COnference on - Co-NEXT ’10*, 2010, p. 1.

[41] R. R. Kompella, K. Levchenko, A. C. Snoeren, and G. Varghese, “Every microsecond counts: tracking fine-grain latencies with a lossy difference aggregator,” *ACM SIGCOMM Comput. Commun. Rev.*, vol. 39, no. 4, p. 255, Aug. 2009.

[42] C. Rotsos, N. Sarrar, and S. Uhlig, “Oflops: An open framework for openflow switch evaluation,” *Passiv. Act. Meas.*, pp. 85–95, 2012.

[43] M. Abu Sharkh, M. Jammal, A. Shami, and A. Ouda, “Resource allocation in a network-based cloud computing environment: design challenges,” *IEEE Commun. Mag.*, vol. 51, no. 11, pp. 46–52, Nov. 2013.

[44] Operators Network, “Network Functions Virtualization: An Introduction, Benefits, Enablers, Challenges and Call for Action,” *SDN OpenFlow SDN OpenFlow World Congr.*, 2012.

[45] B. Han, V. Gopalakrishnan, L. Ji, and S. Lee, “Network function virtualization: Challenges and opportunities for innovations,” *IEEE Commun. Mag.*, vol. 53, no. Vm, pp. 90–97, 2015.

[46] G. L. Vassoler, F. R. de Souza, P. P. P. Filho, M. R. N. Ribeiro, and F. R. De Souza, “Hybrid reconfiguration for upgrading datacenter interconnection topology,” in *IEEE Photonics Conference 2012*, 2012, no. Ccm, pp. 782–783.

[47] R. Entringer, D. Jackson, and P. Slater, “Geodetic connectivity of graphs,” *IEEE Trans. Circuits Syst.*, vol. 24, no. 8, pp. 460–463, Aug. 1977.

[48] N. Dukkipati and N. McKeown, “Why flow-completion time is the right metric for congestion control,” *ACM SIGCOMM Comput. Commun. Rev.*, vol. 36, no. 1, p. 59, Jan. 2006.

[49] B. Lantz, B. Heller, and N. McKeown, “A network in a laptop: Rapid Prototyping for Software-Defined Networks,” in *Proceedings of the Ninth ACM SIGCOMM Workshop on Hot Topics in Networks - Hotnets ’10*, 2010, pp. 1–6.

[50] RFC 2328, “OSPF Version 2,” 1998. [Online]. Available: https://tools.ietf.org/html/rfc2328. [Accessed: 31-Mar-2015].

[51] RFC 2991, “Multipath Issues in Unicast and Multicast Next-Hop Selection,” 2000. [Online]. Available: http://tools.ietf.org/html/rfc2991. [Accessed: 31-Mar-2015].

[52] R. 2992, “Analysis of an Equal-Cost Multi-Path Algorithm,” 2000. [Online]. Available: https://tools.ietf.org/html/rfc2992. [Accessed: 31-Mar-2015].

[53] Iperf.fr, “What is Iperf?,” 2015. [Online]. Available: http://iperf.fr/. [Accessed: 31-Mar-2015].

[54] J.-M. Chang and C.-W. Ho, “The recognition of geodetically connected graphs,” *Inf. Process. Lett.*, vol. 65, no. 2, pp. 81–88, Jan. 1998.

[55] M. H. M. Paiva, “Aplicações de teoria (espectral) de grafos no projeto e análise de topologias físicas para redes ópticas,” 2012.

[56] J. Plesník, “Towards minimum k-geodetically connected graphs,” *Networks*, vol. 41, no. 2, pp. 73–82, Mar. 2003.

[57] C. Wang, “MCube: A high performance and fault-tolerant network architecture for data centers,” *2010 Int. Conf. Comput. Des. Appl.*, vol. 5, no. Iccda, pp. V5–423–V5–427, Jun. 2010.

[58] FDK, “2X2 Optical Switch C-band : YS-1200-155.” 2004.

[59] VMware Inc, “VMware ESXi Overview,” 2015. [Online]. Available: http://www.vmware.com/br/products/esxi-and-esx/overview.html. [Accessed: 26-Mar-2015].

[60] BeagleBoard.org, “BeagleBone Black,” 2015. [Online]. Available: http://beagleboard.org/BLACK. [Accessed: 26-Mar-2015].

[61] Canonical Ltd, “O que é o Ubuntu?,” 2015. [Online]. Available: http://ubuntu-br.org/ubuntu. [Accessed: 31-Mar-2015].

[62] OASIS, “AMQP - Advanced Message Queuing Protocol,” 2015. [Online]. Available: https://www.amqp.org/. [Accessed: 31-Mar-2015].

[63] Openvswitch.org, “What is Open vSwitch?,” 2015. [Online]. Available: http://openvswitch.org/. [Accessed: 31-Mar-2015].

[64] ARMhf.com, “Linux for ARMhf devices,” 2015. [Online]. Available: http://www.armhf.com/download/. [Accessed: 31-Mar-2015].

[65] Python Software Foundation, “Welcome to Python.org,” 2015. [Online]. Available: https://www.python.org/. [Accessed: 31-Mar-2015].

[66] R. T. Fielding, “Architectural Styles and the Design of Network-based Software Architectures,” University of California, Irvine, 2000.

[67] S. Godard, “SYSSTAT: System performance tools for the Linux OS,” 2015. [Online]. Available: https://scholar.google.com.br/scholar?q=sysstat&hl=pt-BR&as_sdt=0,5#0. [Accessed: 18-Apr-2015].

[68] Volker Gropp, “bwm-ng (Bandwidth Monitor NG),” 2015. [Online]. Available: http://www.gropp.org/?id=projects&sub=bwm-ng. [Accessed: 31-Mar-2015].

[69] D. Armstrong and K. Djemame, “Performance issues in clouds: An evaluation of virtual image propagation and I/O paravirtualization,” *Comput. J.*, 2011.

[70] M. Dhingra, J. Lakshmi, and S. Nandy, “Resource usage monitoring in clouds,” *Proc. 2012 ACM/IEEE …*, 2012.

[71] OpenWrt.org, “OpenWrt Wireless Fredom,” 2015. [Online]. Available: https://openwrt.org/. [Accessed: 31-Mar-2015].

[72] O. Foundation and OpenStack.com, “OpenStack Compute Administration Guide,” *OpenStack.com*, no. OpenStack, 2013.

[73] M. Martinello, M. R. N. Ribeiro, R. E. Z. De Oliveira, and R. De Angelis Vitoi, “Keyflow: A prototype for evolving SDN toward core network fabrics,” *IEEE Netw.*, vol. 28, no. April, pp. 12–19, 2014.

[74] J. Engelhardt, “The Netlink protocol: Mysteries Uncovered,” *Inai.De*. pp. 1–12, 2010.

Apêndice A – Implementação do mecanismo de roteamento em Hipercubo no Open vSwitch

O *Open vSwitch* – OvS [63] é um *switch* em *software*, de código aberto, com suporte as principais protocolos e interfaces de gerenciamento, tais como: *OpenFlow*, NetFlow, sFlow, IPFIX, RSPAN, CLI, LACP, 802.1ag, entre outros. Sua utilização inclui o suporte para ambientes de virtualização, por exemplo, XenServer 6.0, Qemu/KVM e VirtualBox; o gerenciamento de interfaces físicas de *hardware* dedicado para encaminhamento; até a integração a projetos de gerenciamento de sistemas virtuais, a exemplo, o OpenStack. Ainda, ele é o mecanismo de encaminhamento (*kernel datapath*) do sistema operacional Linux, incorporado a partir da versão 3.3 do *kernel*. Esta grande gama de aplicações torna o *Open vSwitch* uma ferramenta poderosa[4], tanto para a implementação de cenários que utilizem redes tradicionais, como para cenários SDN, ao exportar os métodos para acesso remoto com controle do tráfego, via protocolo *OpenFlow*.

A.1 Visão geral da implementação

Para habilitar o encaminhamento em *Hipercubo*, alteramos o componente do *Open vSwitch*, conhecido como *datapath*, que faz parte do *kernel* do sistema operacional Linux. Desta forma, o encaminhamento em *Hipercubo* é tratado com prioridade máxima pelo sistema. Especificamente, nossa implementação foi realizada sobre o código do *Open vSwitch* 2.30 LTS, por se tratar de uma versão de suporte de longo prazo, e implementar os mais recentes protocolos.

A codificação foi planejada de forma a interferir, minimamente, no fluxo normal de pacotes do *datapath*, e ao mesmo tempo possibilitar a implementação de outros métodos de

[4] https://github.com/openvswitch/ovs/blob/master/WHY-OVS.md

encaminhamentos, como é o caso do Keyflow [73], que já se encontra implementado. De fato, nossa implementação permite que o operador da rede selecione um entre os seguintes métodos de encaminhamento: encaminhamento padrão do OvS, encaminhamento via KeyFlow e encaminhamento em XOR (*Hipercubo*).

Para permitir a seleção do método de encaminhamento desejado, foram criados dois grupos de mensagens. O primeiro grupo foi adicionado ao protocolo *OpenFlow* 1.0, por meio de extensões das mensagens Nicira. Já no segundo grupo foram criadas mensagens utilizando o protocolo Netlink [74], que promove a comunicação entre nível de usuário e o *kernel* do Linux. A Figura A. 1 ilustra a comunicação, via protocolo *OpenFlow*, entre o *controlador SDN* e o vSwitch, que é o processo do *switch* virtual do *Open vSwitch*, executado em nível de usuário. Também, apresenta a comunicação entre o vSwitch e o *datapath* do *Open vSwitch* que é realizada via protocolo Netlink.

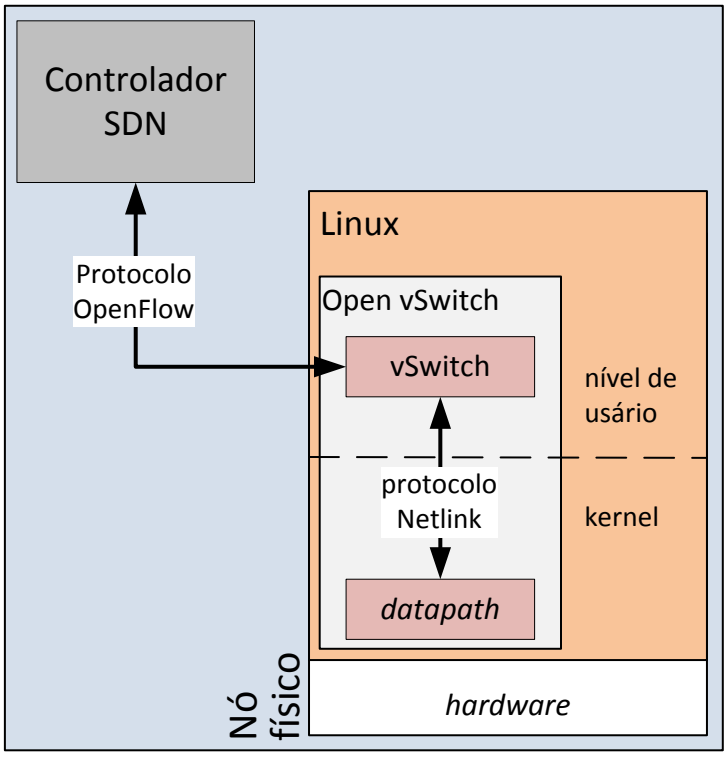


Figura A. 1: Detalhe da comunicação entre o controlador *OpenFlow* e o OvS *datapath*

A.2 Configuração dos nós físicos para formarem o *Hipercubo*

Neste tópico iremos abordar, exclusivamente, a configuração dos nós físicos para realizar o encaminhamento de pacotes via XOR. Nosso ponto de partida será o diagrama de

sequência, com as mensagens utilizadas configurar o *Hipercubo* e habilitar o encaminhamento XOR.

O processo de envio das mensagens, inicia-se logo após o controlador *OpenFlow* identificar os nós físicos que fazem parte do *Hipercubo*. Neste momento o controlador envia, para cada nó físico, uma mensagem tipo *nx_action_set_hypercube_node*, informando o *grau do Hipercubo*, o *endereço deste nó físico dentro do Hipercubo, o nome da interface virtual* (porta_VMs), que interliga a *bridge br-eth*, com as máquinas virtuais deste nó físico e por último, *o nome da interface física* que conecta este nó ao *controlador da Rede* da arquitetura OpenStack (porta_networkcontroler*)*. Em seguida, o controlador *OpenFlow*, envia *N* mensagens do tipo *nx_action_set_hipercube_neighbor,* para cada nó físico, onde *N* representa o número de vizinhos deste nó. Cada mensagem contém o endereço de *Hipercubo do vizinho*, o *nome e o número da interface física* que o vizinho está conectado (porta_name e porta_no, respectivamente). A Figura A. 2 ilustra o diagrama de mensagens entre o controlador *OpenFlow* e os componentes vSwitch e *datapath* do OvS. Conforme explicado anteriormente, o controlador OpenFlow envia as mensagem o para o módulo vSwitch, via protocolo *OpenFlow*. Por sua vez, este módulo, simplesmente, repassa as informações para o *datapath,* via protocolo Netlink, por meio das mensagens *ffw_hypercube_node* e *ffw_hypercube_neighbor*.

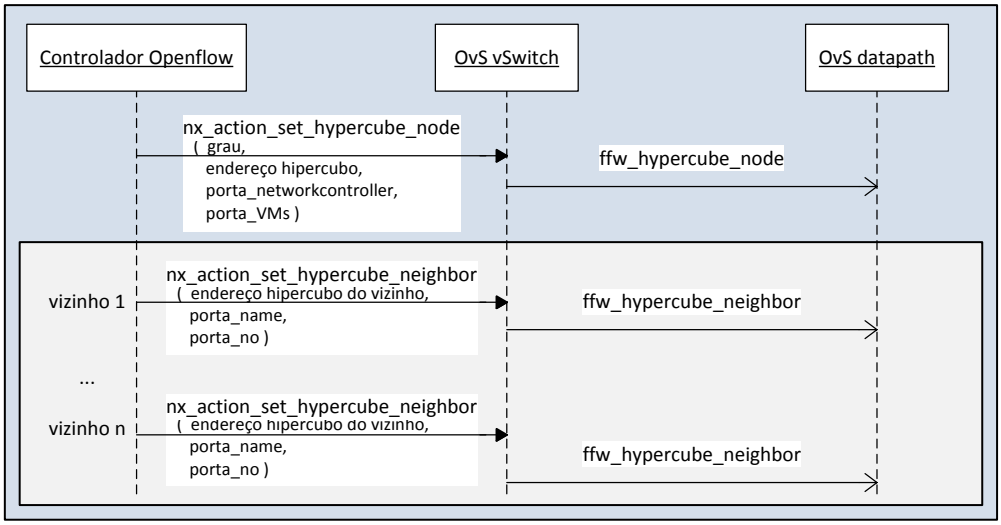


Figura A. 2: Diagrama de sequência das mensagens de configuração do *Hipercubo*

Para que as mensagens *ffw_hypercube_node* e *ffw_hypercube_neighbor* fossem entregues ao *datapath*, que é um módulo do *kernel*, uma nova família de mensagens denominada *dp_ffw_genl_family* do tipo *generic netlink* foi criada.

As mensagens *ffw_hypercube_node* e *ffw_hypercube_neighbor* são roteadas, pelo barramento Netlink, para a função *ovs_ffw_cmd_hypercube (struct sk_buff *skb, struct genl_info *info)* que foi adicionada ao módulo *datapath*. Nesta função cada mensagem é decodificada de acordo com os parâmetros que ela carrega. Se a mensagem for do tipo *ffw_hypercube_node,* uma estrutura chamada *dp_ffw_hypercube_node* é preenchida com os seguintes parâmetros:

- *address*: Representa o endereço de *Hipercubo* deste nó;
- *degree*: Representa o grau do *Hipercubo*;
- *neighbors_count*: Representa o número de vizinhos ativos. Seu valor inicial é zero;
- *odp_vm_port_no*: Representa o número da porta, neste *datapath,* que endereça as máquinas virtuais. Este valor é preenchido com base no nome da porta, passado como parâmetro pelo controlador *OpenFlow* e convertido pela função interna *ovs_vport_locate*;
- *odp_networkcontroler:* Representa o número da porta, neste *datapath,* para o *controlador da Rede* da arquitetura OpenStack. Este valor, também, é preenchido com base no nome da porta, passado como parâmetro pelo controlador *OpenFlow* e convertido pela função interna *ovs_vport_locate*;
- *ffw_hypercube_neighbor **neighbors*: Representa a lista de vizinhos deste nó*;*

Adicionalmente, a variável chamada *dp_ffw.method* é preenchida com o valor FFW_HYPERCUBE, que indica para o módulo *datapath* o método de encaminhamento que deve ser utilizado para encaminhamento dos pacotes. Por fim, a variável chamada *max_hypercube_addr* é preenchida definida com o maior endereço que pode pertencer a este *Hipercubo* usando a fórmula 2^{grau}.

Por outro lado, se a mensagem for do tipo *ffw_hypercube_ neighbor*, uma posição do vetor de vizinhos da estrutura *dp_ffw_hypercube_node* é preenchida com o endereço do vizinho e o número da porta, neste *datapath,* para este vizinho. Além disso, o campo *neighbors_count* é incrementado, informando o novo valor de vizinhos ativos. Para garantir a integridade das informações o seguinte conjunto de testes é realizado:

- Verificar se nó do *Hipercubo* já foi configurado na estrutura *dp_ffw_hypercube_node;*
- Verificar se o vizinho já existe na lista de vizinhos deste nó;
- Verificar o número máximo de vizinhos baseado no grau do *Hipercubo*;
- Verificar a existência das portas informadas pelo controlador *OpenFlow*;
- Verificar se é necessário liberar a memória alocada. Isso ocorre no caso de novas mensagens de configuração.

Se a configuração do nó físico foi realizada com sucesso, o mecanismo de encaminhamento em *Hipercubo* está habilitado e pronto para tratar os pacotes que chegarem às interfaces da máquina que fazem parte do *Hipercubo*.

A.2.1 Tratamento de pacotes pelo mecanismo de encaminhamento em Hipercubo

Para interferir o mínimo possível no fluxo de tratamento normal dos pacotes no módulo *datapath*, adicionamos *quatro* linhas de código, no início da função *ovs_dp_process_received_packet(),* conforme o fluxograma ilustrado na Figura A. 3. O primeiro passo é verificar se o valor da variável *dp_ffw.method* é maior que zero. Caso a resposta seja positiva, o pacote é direcionado a função *ffw_execute()* para ser verificado qual o método de encaminhamento está habilitado. Ao final do processamento, o valor retornado pela função *ffw_execute()* é analisado, caso ele seja igual à zero, significa que o pacote foi encaminhado pelo mecanismo de encaminhamento habilitado, finalizando o tratamento do pacote. Nos casos em que o pacote segue o fluxo padrão de tratamento, a função *ovs_dp_process_received_packet()* executa algumas verificações no pacote, e encaminha o mesmo para a função *ovs_dp_process_packet_with_key()*.

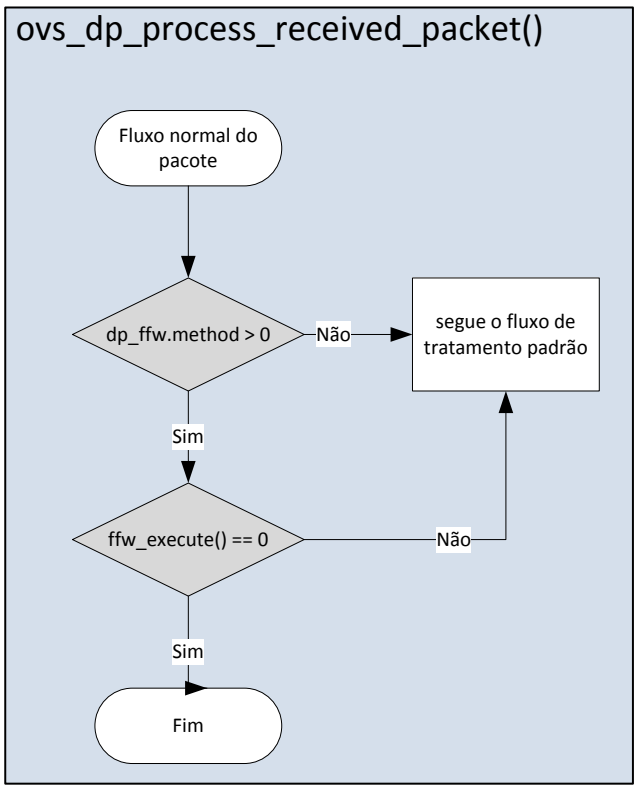


Figura A. 3: Fluxograma apresentando a verificação do método de encaminhamento a ser utilizado no *datapath*

Os pacotes direcionados a função *ffw_execute()* são processados de acordo com o fluxograma apresentado na Figura A. 4. O primeiro passo é extrair os quatro primeiros octetos do endereço *Ethernet* de destino e copiá-los para a variável local *ffw_code*. O valor desta variável irá representar: o endereço de *Hipercubo* deste nó, caso da variável *dp_ffw.method* seja igual à FFW_HIPERCUBE; ou a chave global para o cálculo do KeyFlow, caso o valor de *dp_ffw.method* seja igual à FFW_KEYFLOW. Na hipótese do valor da variável *dp_ffw.method* não conter um dos métodos de encaminhamento conhecido, o código de erro *-1* é retornado e o pacote irá seguir o fluxo padrão do OvS. Nos demais casos, o pacote será processado pela função de encaminhamento correspondente ao valor da variável *dp_ffw.method*.

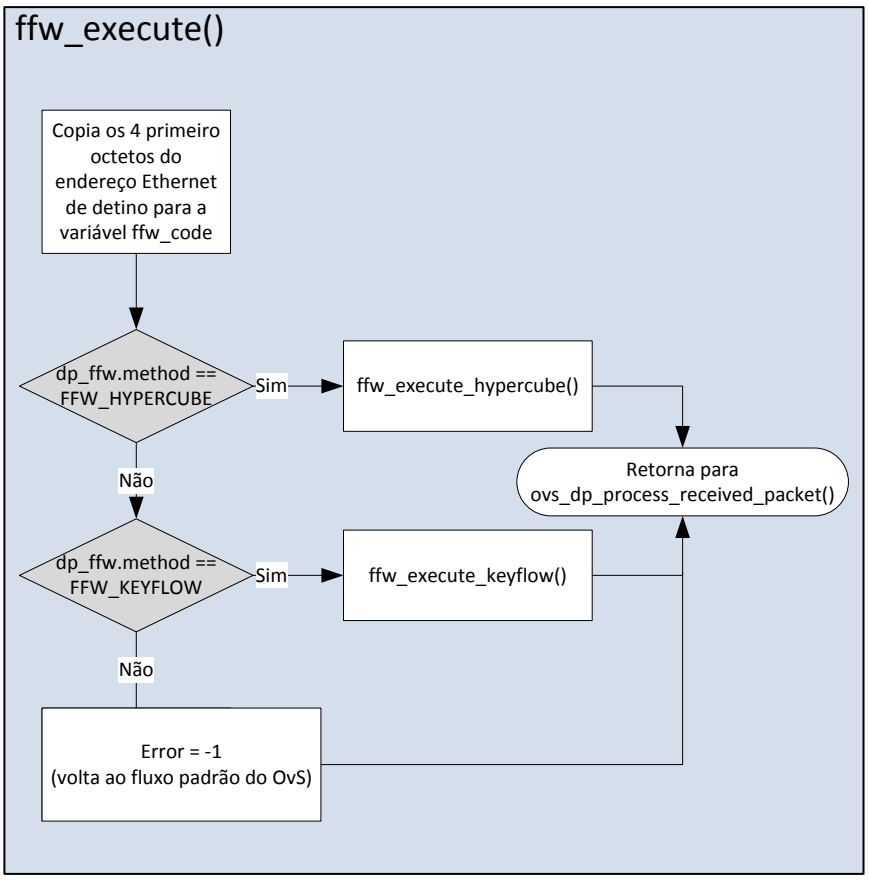


Figura A. 4: Fluxograma da função *ffw_execute()*

No contexto deste trabalho iremos abordar apenas o tratamento dos pacotes pela função *ffw_execute_hypercube()*, que é ilustrado pelo fluxograma apresentado na Figura A. 5. Neste ponto, é importante destacar que o OvS *datapath* trata os pacotes que transitam por todas as *bridges* virtuais criadas pelo usuário, no contexto deste trabalho cada *nó de computação* possui duas *bridges, br-int* e *br-eth*. Desta forma o primeiro teste verifica se o pacote em questão está chegando a *bridge* de dados (*br-eth*). Caso afirmativo, verifica-se se o

pacote é LLDP. SE sim, o pacote é descartado. Caso contrário, é chegado se o pacote possui *tag* de VLAN. Isto é necessário, pois o OvS utiliza um *ID* para isolar as redes virtuais, que é diferente do *segmentation ID* utilizado pelos *switches* físicos. Assim uma conversão se faz necessária, caso a *tag* de VLAN esteja presente. O próximo passo é verificar se o pacote é de *broadcast*. Neste caso estamos interessados em pacotes do tipo ARP, os quais devem ser encaminhados para o controlador *OpenFlow*. Isso é feito, retornando o pacote para o fluxo normal do OvS, que será submetido a uma regra *OpenFlow* pré-instalada para capturar este tipo de pacote. Os demais pacotes de *broadcast* são encaminhamento para controlador de Rede. Isso é feito para tratar os pacotes do tipo DHCP, pois o *controlador da Rede* possui um servidor de DHCP, conforme mencionado, para cada rede virtual instanciada. Pacotes cujo valor dos quatro primeiros octetos seja numericamente maior que o espaço de endereçamento do *Hipercubo* formado pelos *nós de computação*, também são enviados para o controlador de rede, por exemplo, pacotes destinados a Internet. Porém, se a origem destes pacotes for o controlador de rede, eles são descartados. É importante destacar que os tratamentos realizados até este ponto foram necessários para atender as peculiaridades da arquitetura proposta. Para uma abordagem diferente, uma nova função deverá ser escrita para atender a nova realidade.

Por fim, os pacotes que pertencem ao espaço de endereçamento do *Hipercubo* podem ser encaminhados para uma VM local, caso o endereço de destino seja o nó que ele se encontre, ou roteado via XOR para o vizinho mais próximo do seu destino. Este processamento irá se repetir em outros nós até que o pacote alcance o seu destino.

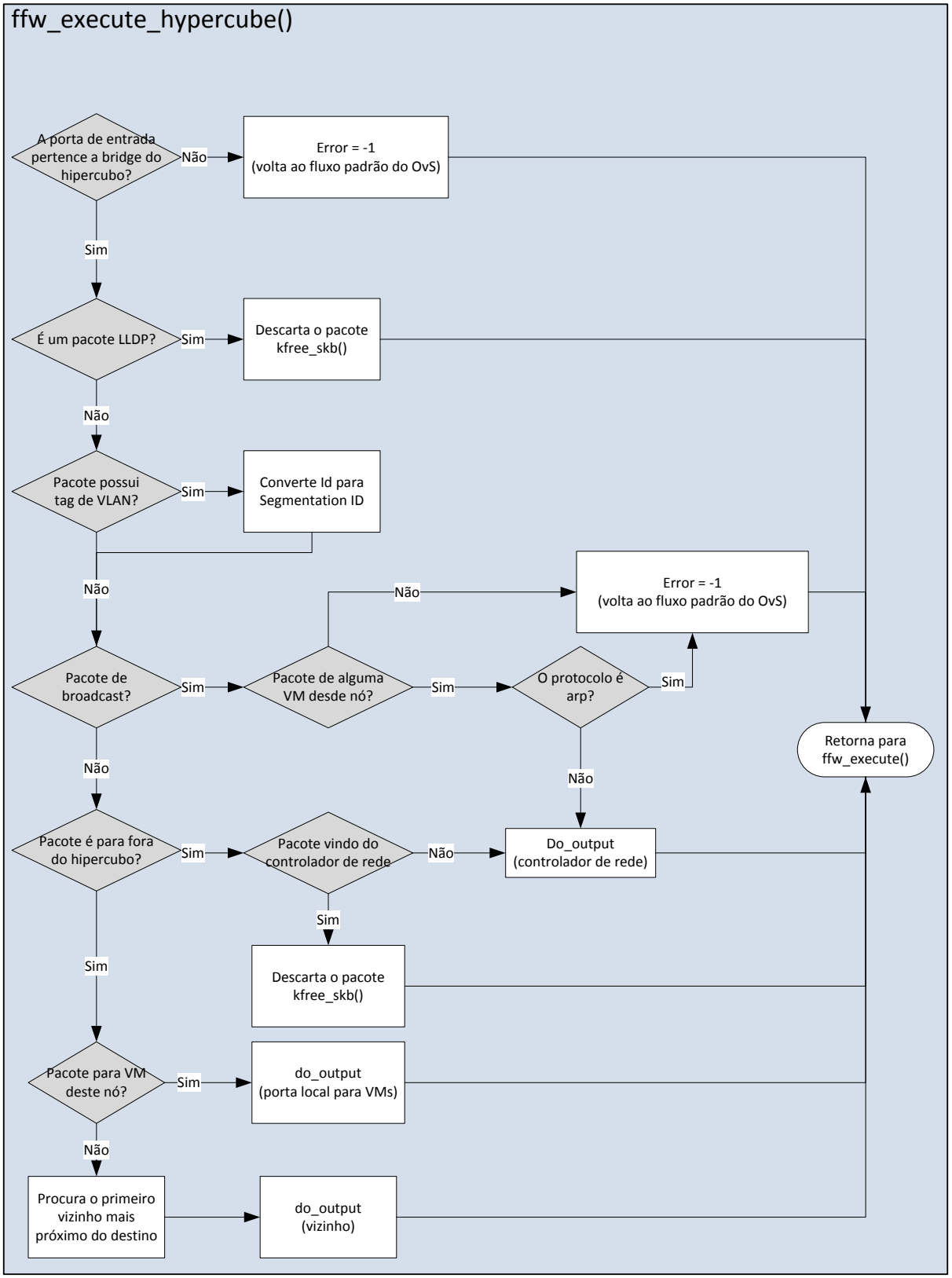
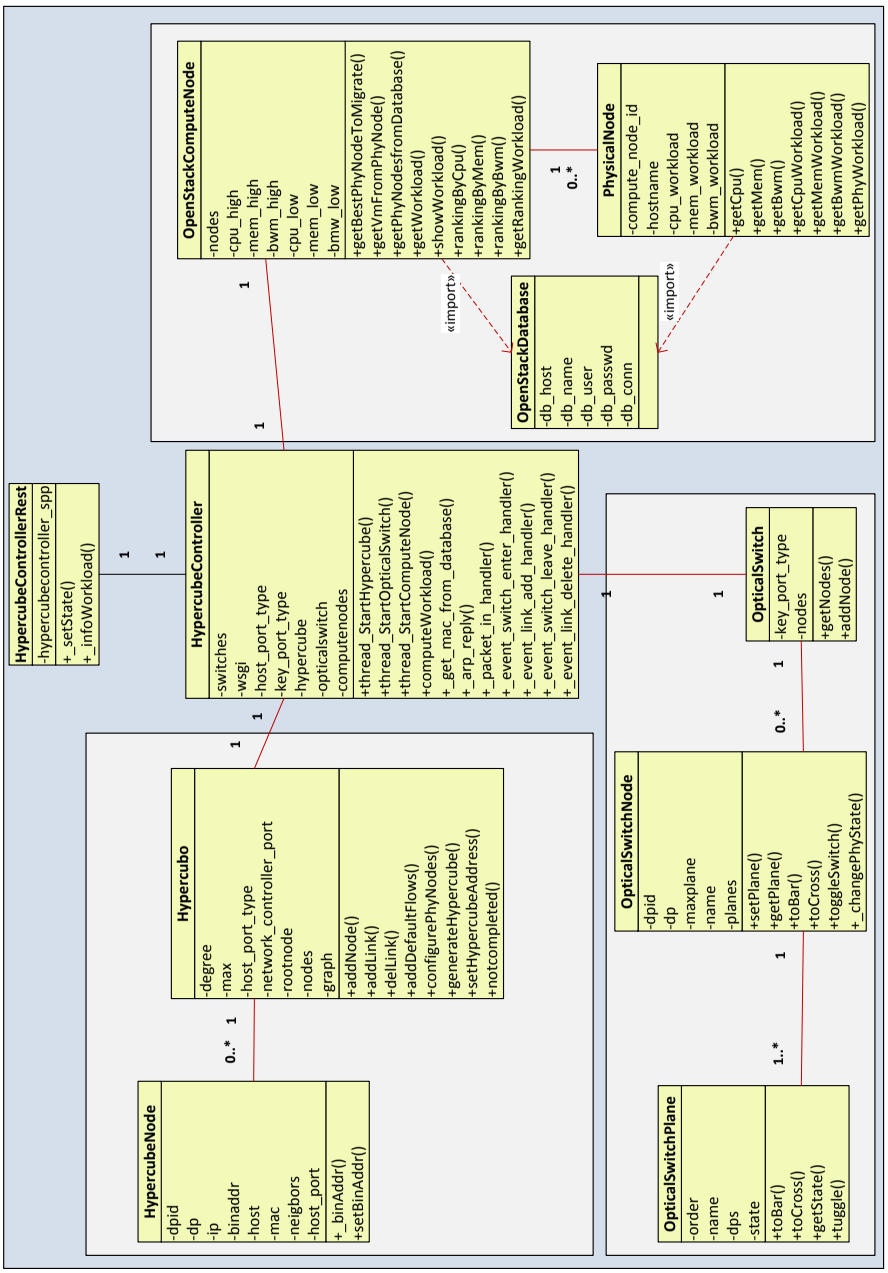


Figura A. 5: Fluxograma da função *ffw_execute_hypercube()*

Apêndice B – Diagrama de classes do software do controlador A-SDN



Apêndice C – OpenStack

OpenStack é uma arquitetura para *computação nas nuvens* de código aberto, que foi projetada como um ambiente de Infraestrutura como Serviço (IaaS), de forma a prover escalabilidade e elasticidade nuvens públicas e privadas [72]. Talvez o maior diferencial entre o OpenStack e outras plataformas abertas para a computação em nuvem seja o número de empresas envolvidas. Atualmente mais de 200 companhias de *hardware, software* e serviços apoiam o seu desenvolvimento.

O desenvolvimento do OpenStack iniciou em meados de 2010, quanto a NASA e a Rackspace Hosting firmaram uma parceria para criar um serviço de computação em nuvem que pudesse ser instalado em *hardware* de prateleira. Assim, essas empresas combinaram seus sistemas (Nebula, da NASA, e Cloud Files, da Rackspace) para criar um código inicial. Para que o sistema pudesse evoluir de forma rápida, foi criado o evento *OpenStack Design Summit* para discutir os detalhes e metas do OpenStack. Ainda, foi criado um calendário de lançamento de versões a cada seis meses. Porém, um evento em particular trouxe grande visibilidade para a plataforma. Em 2011, a Canonical incorporou o OpenStack a distribuição Linux Ubuntu, facilitando sua instalação. Em seguida, outras distribuições Linux como a Red Hat e Debian, fizeram o mesmo, popularizando o OpenStack entre seus usuários [37].

C.1 Visão geral do OpenStack

O OpenStack permite controlar um grande conjunto de servidores, unidades de armazenamento e recursos de rede em todo o *data center*. Para isso, ele integra um conjunto de serviços, que podem ser gerenciados por uma única interface web, conforme apresentado na Figura C. 1.

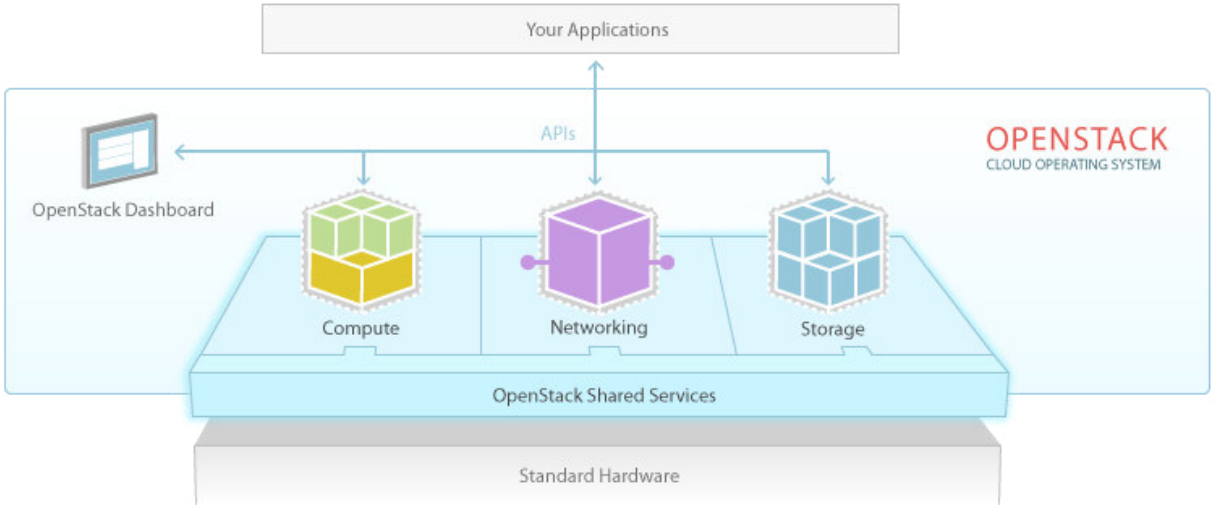


Figura C. 1: Componentes do OpenStack gerenciados pela OpenStack Dashboard.

Fonte: www.openstack.org

Observe no diagrama da Figura C. 1 que as camadas relacionadas a aplicações e administração de acesso estão no topo da arquitetura. Os recursos para computação, redes e armazenamento estão logo abaixo. Por sua vez, os serviços compartilhados do OpenStack aparecem sobre o *hardware* que irá dar vida a este ambiente. Observe ainda que os componentes de *hardware* são identificados como padrão, de forma que não há amarrações a um servidor, aplicação ou componente de rede em particular.

C.1.1 Principais componentes

Atualmente o OpenStack possui sete componentes (projetos) principais sendo eles: *Compute, Dashboard, Identity, Image, Networking.*

- *Compute*: provê o serviço de servidores virtuais sobre demanda. Implementações comerciais como Rackspace e HP foram criadas sobre o a versão do serviço Compute denominada *Nova* e são utilizadas internamente em companhias como Mercado Livre e NASA.
- *Dashboard*: provê uma interface web modular para todos os serviços *OpenStack*. Esta interface permite que o operador facilmente lance uma instância de máquina virtual, associe um endereço IP, defina os controles de acesso, etc.
- *Identity*: provê o serviço de autenticação e autorização para todos os serviços do *OpenStack*. Este componente também fornece um serviço de catálogo de serviços para uma nuvem *OpenStack*.

- *Image*: provê um catálogo e um repositório para imagens de discos virtuais. Estas imagens de discos virtuais são comumente usadas pelo serviço *Compute*. Mesmo este serviço sendo tecnicamente opcional, uma nuvem de qualquer tamanho pode requerê-lo.
- *Networking*: é um componente que provê a conectividade de rede como um serviço entre a interface do dispositivo e o outro serviço *OpenStack* (normalmente *Nova*). O serviço trabalha de forma a permitir que os usuários criem suas próprias redes e então associe interfaces de rede para elas. O componente *OpenStack Network* tem uma arquitetura "*plugavel*" capaz de suportar as tecnologias de redes mais populares.
- *Block Storage*: prove o serviço de bloco de armazenamento persistente para as VMs hospedadas.
- *Object Store*: permite que o usuário armazene e recupere arquivos, porém não montam diretórios como um sistema de arquivos.

C.1.2 Nova: a implementação do serviço Compute

Este é o componente mais complexo e distribuído do *OpenStack*. Assim, vários processos cooperam entre si para executar as atividades do usuário. A lista a seguir apresente os componentes de acordo com as áreas funcionais:

API

- *nova-api*: fornece a interface pela qual o usuário final envia e recebe comandos para o serviço *Compute*. Este serviço suporta, de fato, as API´s do *OpenStack Compute*, EC2 da Amazon, além de uma API especial para administração. Assim, este serviço inicia a maior parte das atividades de orquestração, como executar uma instância, e ainda assegura o cumprimento de algumas políticas.
- *nova-api-metadata*: recebe as requisições de metadados das instâncias. Normalmente é utilizada quando se usa multi-nós com o *nova-network* instalado.

Computing Core

- *nova-compute*: é o processo que funciona como um *daemon* o qual crie e destrói instâncias de máquinas virtuais via API de um *hypervisor*, tais como: XenAPI para XenServer, libvirt para KVM ou QEMU, VMwareAPI para VMware, etc. O

daemon do *nova-compute* processa ações de uma fila e produz uma série de comandos em nível de sistema para executar as ações, ao mesmo tempo em que atualiza os estados em sua base de dados.

- *nova-schedule*: é o processo que toma uma máquina virtual de uma fila e determinada onde ela deverá ser executada. Em outras palavras, qual a máquina física que irá executar a máquina virtual.
- *nova-conductor*: é o processo que trabalha como mediador entre o *nova-compute* e a base de dados. Seu objetivo é eliminar todo acesso direto para a base de dados da nuvem feito pelo processo *nova-compute*. O módulo *nova-conductor* escala horizontalmente, porém deve ser implementado nos mesmos nodos que executam o serviço *nova-compute*.

Networking for VMs

- *nova-network*: similar ao *nova-compute* é um processo que funciona como um *daemon.* Ele recebe as requisições de uma fila e então executa ações para realizar as tarefas de manipulação de rede, por exemplo, mudar regras no *iptables*. Suas funcionalidades estão sendo migradas para o módulo *OpenStack Networking* (codinome *Neutron*).
- *Nova-dhcpbridge*: é um *script* que rastreia os IP associados e os armazena em uma base de dados utilizando o *dhcp-script* do *dnsmasq*. Esta funcionalidade também está sendo migrada para o módulo *OpenStack Networking*.

C.1.3 Distribuição dos serviços no ambiente

A Figura C. 2 apresenta a divisão dos serviços OpenStack por tipo de nós. Observe que o controlador (*controller node*) é o responsável por executar a maioria dos serviços, porém vários deles são serviços opcionais. O nó de rede (*network node*) é encarregado de garantir a funcionalidade das redes virtuais executando serviços como DHCP e roteamento de pacote entre as máquinas. Por fim, os nós computação (*compute node*) representam os servidores (por vezes milhares) que suportam as máquinas virtuais no ambiente em nuvem.

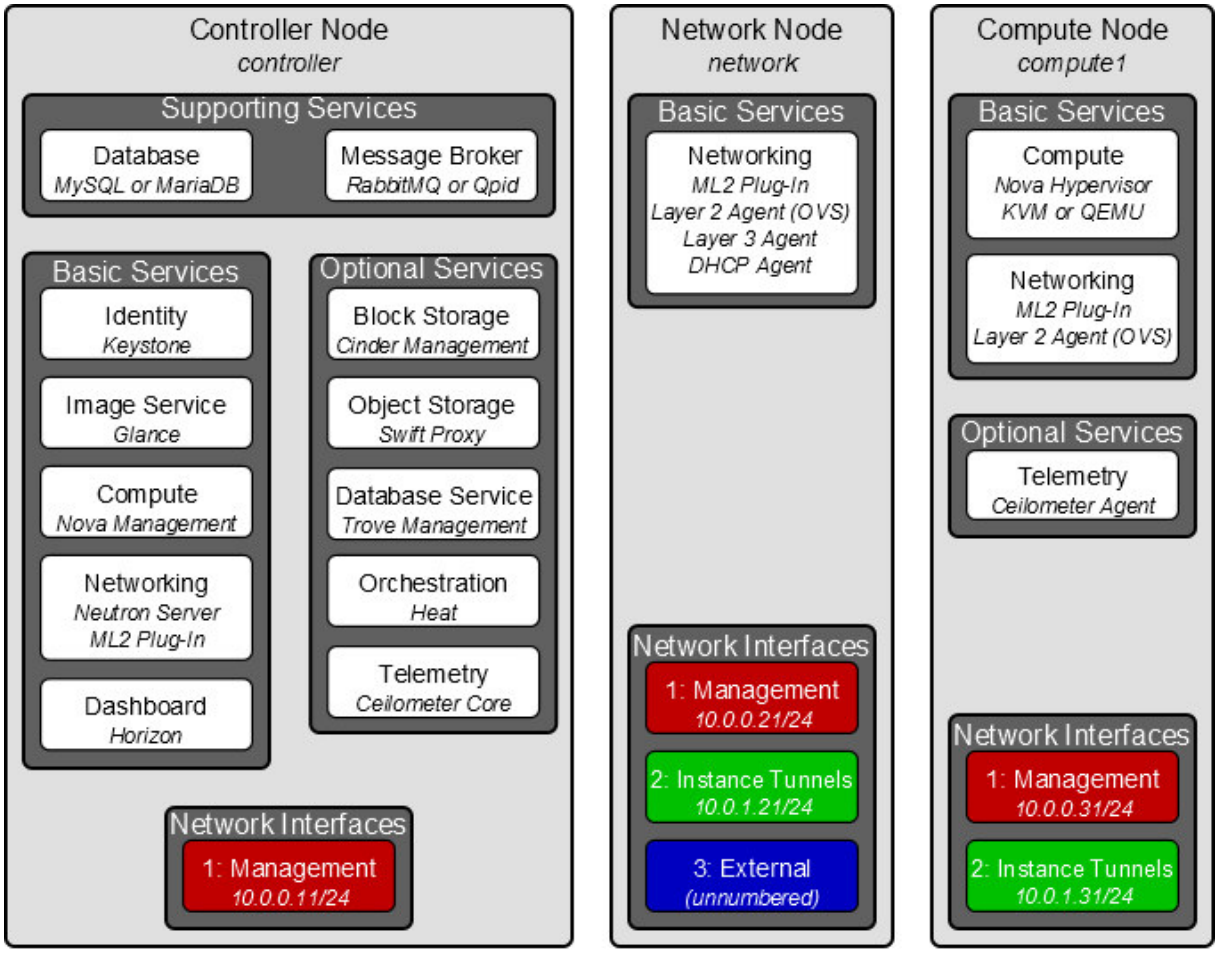


Figura C. 2: Distribuição dos serviços por tipo de nós

Fonte: www.openstack.org

C.2 A flexibilidade e suas implicações

Devido ao amplo suporte em níveis de *hardware*, *software* e serviços oferecidos, o OpenStack possibilita a criação de nuvens para diferentes tipos de aplicações, deste uma aplicação proprietária, até um ambiente público, com o fornecimento de serviços de todos os tamanhos e classes. Dado que seu código fonte é livre, o OpenStack permite criar ambientes altamente personalizados com preços atrativos.

Por outro lado, esta flexibilidade pode se tornar um grande problema, pois ela envolve a mentalidade do "faça você mesmo", para reconhecer e implementar as características e funções que não existem ou que não estão tão de acordo com a necessidade da empresa. Assim, utilizar o OpenStack exige um investimento significativo de tempo e esforço. Como exemplos dessas dificuldades, pode-se citar relatório da Samsung informando que a implantação do OpenStack é "complexa e inclinada a erros". Ademais, as atualizações de versão, nem sempre garantem retro-compatibilidade, criando situações desafiadoras que

podem envolver a recriação e uma nova projeção dos elementos que compõem o OpenStack nas implementações em funcionamento.

Isso cria um cenário de compromisso entre as empresas que desejam utilizar o OpenStack, pois elas devem estar preparadas para dedicarem recursos para projeto e programação da arquitetura OpenStack. Ainda, é possível que a implantação de um ambiente OpenStack, totalmente funcional, demore meses ou mesmo anos para ser alcançada. Por fim, a manutenção do ambiente, também, exige um grupo especializado de funcionários com profundos conhecimentos sobre o OpenStack. No entanto, o retorno financeiro pode ser muito vantajoso, pois utilizar *hardware* convencionais consome uma fração dos recursos necessários para aquisição de alternativas proprietárias para computação em nuvem.