UNIVERSIDADE FEDERAL DO ESTADO DO RIO DE JANEIRO
CENTRO DE CIÉNCIAS EXATAS E TECNOLOGIA
PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

Comutação baseada em caminhos: uma solução SDN para problema de migração de máquinas virtuais em *Data Centers*

Gilvan de Almeida Chaves Filho

Orientador

Sidney Cunha de Lucena

RIO DE JANEIRO, RJ - BRASIL
SETEMBRO DE 2015

Comutação baseada em caminhos: uma solução SDN para problema de migração de máquinas virtuais em *Data Centers*

Gilvan de Almeida Chaves Filho

DISSERTAÇÃO APRESENTADA COMO REQUISITO PARCIAL PARA OBTENÇÃO DO TÍTULO DE MESTRE PELO PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA DA UNIVERSIDADE FEDERAL DO ESTADO DO RIO DE JANEIRO (UNIRIO). APROVADA PELA COMISSÃO EXAMINADORA ABAIXO ASSINADA.

Aprovada por:

Sidney Cunha de Lucena, D.Sc. - UNIRIO

Carlos Alberto Vieira Campos, D.Sc. - UNIRIO

Marcelo Gonçalves Rubinstein, D.Sc. - UERJ

Anderson Fernandes Pereira dos Santos, D.Sc. - IME

RIO DE JANEIRO, RJ - BRASIL
SETEMBRO DE 2015

Chaves Filho, Gilvan de Almeida.
C512 Comutação baseada em caminhos: uma solução SDN para problema de migração de máquinas virtuais em *Data Centers* / Gilvan de Almeida Chaves Filho, 2015.
96 f. ; 30 cm + 2 CD-ROM

Orientador: Sidney Cunha de Lucena
Dissertação (Mestrado em Informática) - Universidade Federal do Estado do Rio de Janeiro, Rio de Janeiro, 2015.

1. Centros de processamento de dados. 2. Servidores da web. 3. Softwares. I. Lucena, Sidney Cunha de. II. Universidade Federal do Estado do Rio de Janeiro. Centro de Ciências Exatas e Tecnológicas. Curso de Mestrado em Informática. III. Título.

CDD - 004

Não me lembro da primeira vez que me viste, nem que me abraçaste.

Não me lembro da primeira vez que me pusestes o lápis na mão para escrever meus primeiros rabiscos, quando queria apenas brincar.

Também não me recordo de tua apreensão em me deixar na escola nos primeiros dias, quando a chorar pedia para voltar para casa.

Mas sei que estavas lá a me apoiar quando pensei em fraquejar. Sempre me dizendo: meu filho não desista e não perqueis a fé.

A ti, mãe amada, dedico este trabalho. Tu foste ontem e és hoje minha maior incentivadora.

Agradecimentos

A Deus, por tudo, por estar a meu lado e me abençoar todos os dias de minha vida.

A meus pais, Gilvan e Joana, que se doaram para que eu pudesse chegar onde cheguei.

A Suzan, esposa amada e companheira, que me apoiou em tudo para que eu pudesse me dedicar a este trabalho.

Ao meu amigo Bruno Guerra, que me convidou para participar da seleção deste mestrado.

Ao professor Sidney que, com paciência e olhar aguçado, orientou-me a trilhar o caminho deste trabalho.

Aos colegas Roberto Gerpe e André Barros, pela disponibilidade em sempre tirar minhas dúvidas.

Chaves F., Gilvan de Almeida. **Comutação baseada em caminhos: Uma solução SDN para problema de migração de máquinas virtuais em *Data Centers***. UNIRIO, 2015. 96 páginas. Dissertação de Mestrado. Departamento de Informática Aplicada, UNIRIO.

RESUMO

A adoção da tecnologia de virtualização de servidores nos *Data Centers* permitiu a implementação de diversas funcionalidades de computação em nuvem ao possibilitar a alocação dinâmica de servidores e dos serviços a eles associados. Nesse contexto, surge também a necessidade de se alterar dinamicamente a localização física dos serviços, de maneira que estes se adaptem a novas demandas de tráfego e desempenho. Por este motivo, as máquinas virtuais (VMs) que abrigam tais serviços precisam ser migradas, conforme demanda, para outros servidores físicos. Os *softwares* atuais de virtualização permitem esta migração *on-line* de VMs, que consiste em mudá-las de servidor físico sem que o serviço seja interrompido ou que caiam suas conexões de rede. Para tal, o IP da VM deve ser mantido na migração, o que se torna um problema quando a mesma necessita ocorrer entre redes diferentes. Grande parte das soluções que endereçam tal problema utiliza técnicas de tunelamento, que impactam o tráfego e a escalabilidade da rede. Este trabalho propõe uma solução escalável de comutação de caminhos, chamada PathFlow, que faz uso de rede definida por *software* para possibilitar esta migração de VMs. Os resultados obtidos para o PathFlow mostram a exequibilidade e a efetividade da solução ao permitir a migração *on-line* de máquinas virtuais entre redes distintas com uma significativa redução das tabelas de comutação em relação às soluções tradicionais. Consequentemente, obtém-se uma maior escalabilidade para a mobilidade de VMs em *Data Centers*.

Palavras-chave: *Data Center*, *Cloud Computing*, SDN.

ABSTRACT

The adoption of server virtualization technology in Data Centers has enabled the implementation of some Cloud Computing capabilities by allowing dynamic allocation of servers as well as services associated with them. In this context, there is also the need to dynamically change the physical location of the services, therefore these servers may adapt themselves to new traffic and performance demands. For this reason, the virtual machines (VMs) that support such services need to be migrated by demand to other physical servers. Nowadays virtualization softwares allow for online migration of VMs, which means changing their physical server without services interruptions or network connections beeing lost. The VM IP must remain the same during migration, which becomes a problem when it needs to occur between different subnetworks. Most of solutions that address this problem use tunneling techniques, impacting traffic and network scalability. This paper proposes a scalable path switching solution, called PathFlow, which is a software-defined networking solution that enables VM migration. Obtained results show the feasibility and effectiveness of PathFlow as it enables online migration of virtual machines between different networks, with a significant reduction in switching table sizes over traditional solutions. Consequently, we get greater scalability for mobility of VMs in data centers.

Keywords: Data Center, Cloud Computing, SDN .

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Nomenclaturas

DC	Data Center
EoR	End of Row
FIB	Forwarding Information Base
IP	Internet Protocol
LAN	Local Area Network
LDP	Label Distribution Protocol
LLDP	Link Layer Discovery Protocol
MAC	Media Access Control
MPLS	Multi Protocol Label Switching
OF	OpenFlow
QoS	Quality of Service
SDN	Software Defined Networking
SPF	Shortest Path First
SSL	Secure Sockets Layer
TLS	Transport Layer Security
ToR	Top of Rack
VM	Virtual Machine
VTEP	VXLAN Tunnel Endpoint
VXLAN	Virtual eXtensible Local Area Network
WAN	Wide Area Network

1. Introdução

1.1 Considerações Iniciais

Os *Data Centers* vêm ganhando importância nos últimos anos à medida que aumenta a demanda de infraestrutura para armazenamento de grandes volumes de dados e hospedagem de serviços de aplicação de larga escala. Grandes empresas de tecnologia, como Google, Facebook e Yahoo, utilizam *Data Centers* como repositórios de grandes volumes de dados, hospedagem de aplicações de pesquisas *web* e computação em larga escala[1].

Há um entendimento que o modelo de *Cloud Computing* será o mais utilizado na disponibilização de serviços sob demanda em que há uma alocação dinâmica de recursos computacionais e de rede no *Data Center*. Um exemplo é o Amazon EC2[2], entre outros.

A adoção da tecnologia de virtualização de servidores permitiu endereçar muitos requisitos de *Cloud Computing* ao possibilitar a alocação dinâmica de servidores e serviços associados. Nessa tecnologia, múltiplas máquinas virtuais (*virtual machines* – VMs) são alocadas em um único servidor físico (*host*).

1.2 Motivação

Para que haja um eficiente acesso aos recursos computacionais, é necessário que a alocação das VMs seja disponibilizada o mais próximo dos usuários do serviço, para que esses percebam o menor *delay* possível. Como há uma grande diversidade na localização desses usuários, há a necessidade de se alterar dinamicamente a localização dos servidores para adaptar os serviços às novas demandas de tráfego. Há também a demanda de mudança da localização de VMs para

manutenção parcial do *Data Center*, assim como mudança de um conjunto inteiro de VMs entre *Data Centers*. As VMs, portanto, precisam ser migradas para novas localizações físicas, que podem ser tanto no próprio *Data Center* (*intra DC migration*) quanto entre *Data Centers* distintos (*inter DC migration*).

Os softwares atuais de virtualização (hipervisores) permitem a migração *online* de VMs, que consiste em mudá-las de *host* sem que as VMs sejam desligadas. Isso é feito dinamicamente gravando o estado de memória na origem e incrementalmente transferindo este estado para o *host* destino. Para evitar que caiam as conexões, e consequentemente o serviço, o endereço IP da VM deve ser mantido durante a migração. Isso limita a migração para que esta ocorra apenas dentro da mesma rede local (LAN), ou seja, na mesma sub-rede delimitada pela máscara do endereço IP usado. Uma série de desafios precisam ser tratados para que a mobilidade ocorra entre redes de grande distância (WANs) e até mesmo entre sub-redes no próprio Data Center.

1.3 Problema

Não existe forma simples de migrar VMs entre redes (ou sub-redes) distintas quando o IP é fixo. Uma primeira abordagem sobre mobilidade foi sugerida como adendo à especificação do IPv4[3] para comunicação destinada a dispositivos móveis. Nessa, o tráfego destinado a um dispositivo precisaria ser redirecionado a um ponto de âncora (*home agent*) para depois ser repassado ao destino móvel, e a resposta percorre o caminho inverso. Isso causa uma ineficiente triangulação do tráfego. Além do mais, essa solução poderia ser até tolerada em uma rede IP móvel cujo tráfego é relativamente baixo, mas não para um *Data Center* em que servidores lidam com volumes de tráfego muito maiores.

Uma resolução desse problema de ancoragem é prevista na especificação do IPv6, ao requisitar sinalização entre os nós participantes e dessa forma rastrear o novo endereço IP do nó móvel. Porém, não pode ser exigido, por questões de compatibilidade, que a adoção do IPv6 seja totalmente suportada entre todos os dispositivos do *Data Center*.

A infraestrutura da rede deve suportar a migração de máquinas virtuais entre redes distintas de forma eficiente. Uma grande parte das soluções propostas para endereçar tal problema utilizam técnicas de tunelamento como VXLAN [4], NVGRE[5] e LISP[6], cuja principal ideia é a extensão da LAN da rede origem

para a rede destino através de túneis, inclusive através de WANs. A principal desvantagem dessas soluções é que o tráfego destinado a um *host* móvel precisa passar por um ponto de âncora na rede origem para ser direcionado ao túnel e alcançar o *host* na rede destino, e a resposta percorre o caminho inverso. Isto causa também uma ineficiente triangulação do tráfego que, além de sobrecarregar o túnel, traz severas restrições de escalabilidade.

1.4 Indicação da Proposta de Solução

Este trabalho endereça o problema de migração de VMs entre redes distintas tomando como base recentes trabalhos de virtualização de redes [7] que permitem um *switch* ou uma rede inteira serem logicamente independentes dos dispositivos físicos. Uma rede sobreposta (*overlay*) é uma rede virtual que atua acima da topologia de uma rede física (*underlay*)[7]. Os nós em uma rede sobreposta são conectados através de *links* virtuais que correspondem a caminhos na rede física. A arquitetura de nossa proposta enxerga a rede como uma entidade única formada por elementos de encaminhamento (*forwarding elements* - FEs) que são gerenciados por um controle central (*central control* - CC).

Diante das limitações impostas pela arquitetura tradicional baseada no TCP/IP, este trabalho propõe uma **solução SDN de comutação baseado em caminhos, para o problema de migração de VMs em *Data Centers*** chamado **PathFlow**. O paradigma do SDN (Software Defined Network) propõe uma forma mais ágil de se implantar inovação na rede. O SDN [8] é uma arquitetura emergente onde a funcionalidade de controle da rede (plano de controle) é destacada do *hardware* que realiza a comutação (plano de dados) permitindo a programabilidade da rede. O controle, que era integrado fortemente nos equipamentos individuais de rede, é realizado por controladores SDN (logicamente centralizados), permitindo a abstração da infraestrutura de rede para as aplicações e serviços de rede. Dessa forma, aplicações podem tratar a rede como uma entidade lógica ou virtual.

O PathFlow utiliza como base principal o fato de que o controlador SDN possui uma visão holística da rede: conhece todos os dispositivos de encaminhamento (FE – *Forwarding Elements* - que no nosso caso são *switches* de rede) e ligações entre eles, conseguindo dessa forma criar e manter uma topologia e calcular os menores caminhos (***paths***) de encaminhamento.

Além da topologia, o PathFlow mapeia em quais *switches* estão ligados todos

os *hosts* da rede e relaciona, para cada *switch* origem, qual o caminho (*path*) que deve ser tomado para se alcançar o *switch* destino em que cada *host* está ligado. O *switch* origem então insere um *tag* identificador do caminho no pacote. A comutação em todos os *switches* intermediários se dá por esse *tag* até alcançar o *switch* destino, que o retira e entrega ao *host*.

Dessa forma, cada *switch* origem precisa conhecer apenas o caminho (*path*) para cada um dos *hosts* destinos. Adicionalmente, um *switch* não precisa conhecer todos os caminhos para todos os *hosts* da rede, apenas aqueles para os quais o *host* origem tem interesse de tráfego. Isso reduz o impacto de escalabilidade à medida que há um grande crescimento do número de *hosts* no *Data Center*, sendo essa a principal contribuição deste trabalho. Nos *switches* intermediários, tipicamente *switches* de agregação ou núcleo que possuem poucos *hosts* diretamente ligados, as tabelas de comutação ficam menores, pois recebem do controlador SDN basicamente as informações de comutação dos *paths*.

A ideia de se utilizar comutação baseada em caminhos não é em si uma novidade. Essa técnica é a base do MPLS (*Multi Protocol Label Switching*). A diferença principal é que o MPLS possui uma arquitetura distribuída em que as funções são implementadas no mesmo dispositivo onde também há a função de comutação.

Por se estar sendo proposta uma solução SDN, as funções de controle serão centralizadas e implementadas pelo controlador SDN. Isto é o equivalente à função LDP (*Label Distribution Protocol*) do MPLS, por exemplo, que é responsável pela distribuição e manutenção dos rótulos de comutação, tarefa que pode ser absorvida e gerenciada centralmente pelo controlador SDN.

No caso de um *host* ser movido de um *switch* a outro, por movimentação física ou virtual, no caso de migração de uma VM, o controlador SDN notifica os demais *switches* da rede para alterarem o caminho para os quais devem comutar o pacote de maneira que este alcance o novo *switch* destino. Essa alteração se dá simplesmente alterando o *path* para o destino e pode ser acionada tanto quando o controlador SDN é notificado da nova posição como, adicionalmente, ao receber uma mensagem ou diretiva do hipervisor. Essa sistemática permite a livre movimentação do *host* para qualquer parte da rede sem que seja necessária uma alteração de IP nem que sejam interrompidas as conexões a essa VM.

Outras soluções SDN também endereçam o problema de migração de máquinas

virtuais entre redes distintas, como o VICTOR [9], porém este requer suporte a tabelas de encaminhamento de grande tamanho, levando a problemas de escalabilidade [1].

Os resultados obtidos para o PathFlow mostram a exequibilidade e a efetividade da solução, que será detalhada no decorrer deste trabalho.

1.5 Estrutura da Dissertação

Esta dissertação está organizada conforme descrito a seguir. No Capítulo 2 será abordada a fundamentação teórica dos temas abordados, percorrendo os fundamentos de um *Data Center* e detalhando sua arquitetura, seguindo para as tecnologias de virtualização e continuando com as atuais técnicas de tunelamento. Encerra-se o capítulo falando do paradigma SDN e respectivas soluções para o problema de migração de VMs entre redes. No Capítulo 3 serão revistas as principais topologias de *Data Centers*, para logo em seguida apresentar o PathFlow. Será apresentada sua ideia principal e arquitetura, realizando analogias para melhor entendimento. No Capítulo 4 serão detalhados os testes de validação da proposta e serão apresentadas as limitações da implementação. O Capítulo 5 finaliza este trabalho trazendo a conclusão da dissertação e os trabalhos futuros.

2. Fundamentação

2.1 Importância do *Data Center*

Um *Data Center* é um ambiente onde são interligados servidores (máquinas físicas), equipamentos de *storage* e dispositivos de rede (*switches*, roteadores etc) suportados por sistema de distribuição elétrica e de refrigeração[1]. Grandes empresas de tecnologia da informação, como Google, Facebook e Yahoo, utilizam *Data Centers* para armazenamento, pesquisas na *web* e computação em grande escala[1].

Quando o *hardware* e os sistemas em um *Data Center* proveem aplicações como serviços na Internet, temos o que se chama de computação em nuvem (*cloud computing*)[10]. Quando a nuvem (*cloud*), que é conjunto de *hardware* e *software* do *Data Center*, está disponível para uso do público em geral, temos o que se chama de nuvem pública (*public cloud*). No caso da nuvem ser um *Data Center* interno de uma empresa ou companhia, temos o que se chama de nuvem privada (*private cloud*). O Amazon EC2[2] é um exemplo de nuvem pública, entre outros.

2.2 Arquitetura do *Data Center*

Há um crescente entendimento que o modelo de *Cloud Computing* será o mais utilizado na disponibilização de serviços sob demanda em que há uma alocação dinâmica de recursos computacionais e de rede no *Data Center*. Não obstante à sua importância, a arquitetura do *Data Center* está longe de ser a ideal. Os *Data Centers* utilizam servidores dedicados para executar as aplicações, causando ineficiência na utilização dos recursos computacionais com reflexo nos custos associados[1].

A tecnologia de virtualização de servidores (Ex.: VMWare[11], Xen[12]) ame-

nizou essa ineficiência ao alocar múltiplas máquinas virtuais (VMs) em único servidor físico. Dessa forma, endereça muito dos requisitos de *Cloud Computing* ao permitir o compartilhamento dos recursos computacionais, procurando também manter o isolamento das aplicações entre máquinas virtuais de forma a garantir desempenho.

Na Seção 2.3 será detalhada a virtualização de servidores e o processo de migração *on-line* de máquinas virtuais.

Por outro lado, conforme mencionado na Seção 1.2, para maior eficiência na utilização dos recursos computacionais, é necessário que a alocação das VMs minimize o *delay* no acesso dos usuários. As VMs, portanto, precisam ser migradas para novas localizações físicas, que podem estar no próprio *Data Center* (*intra DC migration*) ou em um *Data Center* diferente (*inter DC migration*). No entanto, as arquiteturas tradicionais de *Data Centers* impõem que uma migração "transparente", ou seja, sem a mudança do endereço IP da VM, ocorra apenas dentro de uma mesma sub-rede, não existindo uma forma simples para se migrar VMs entre redes locais (LANs) distintas quando o IP é fixo. Na Seção 2.4 será revisada a literatura onde serão abordados as principais propostas sobre o tema mobilidade.

2.3 Virtualização e Hipervisores

A virtualização recebeu bastante atenção da comunidade de *Data Center* ao permitir que múltiplas instâncias de sistemas operacionais rodem em uma única máquina física, com alto desempenho, provendo melhor uso dos recursos assim como isolamento, conforme demonstrado em [13].

2.3.1 Migração *on-line* de VMs

A virtualização provê uma série de benefícios, como consolidação de recursos, isolamento de desempenho e migração de serviços transparente ao usuário. Aqui será dada ênfase na migração transparente ao usuário, que permite a mudança da localização da máquina virtual, possibilitando balanceamento de carga, otimização de desempenho e manutenção não disruptiva de *hardware*.

Tecnologias atuais de virtualização, como VMware[11] e XEN[12], permitem a migração *on-line* de máquinas virtuais com tempo de interrupção de serviço não significativo para as aplicações. Isso é feito dinamicamente gravando o estado

de memória na origem e incrementalmente transferindo este estado para o *host* destino. As etapas de migração serão detalhadas na Seção 2.3.2.

Deve ser ressaltado que o requisito principal para que a migração ocorra de forma não disruptiva é que a máquina virtual mantenha o seu IP durante a migração. O endereço IP carrega em si a informação da rede em que o *host* está localizado. Do ponto de vista da arquitetura TCP/IP, isso restringe a migração para que essa somente ocorra na mesma subrede IP. A migração *on-line* de máquinas virtuais é um caso específico do problema de mobilidade IP. Na Seção 2.4 serão apresentadas as soluções já propostas para o problema de mobilidade em redes TCP/IP.

2.3.2 Etapas para a Migração de VMs

O processo de migração de VMs possui algumas etapas, cada qual com suas características em termos de função e tempo de execução.

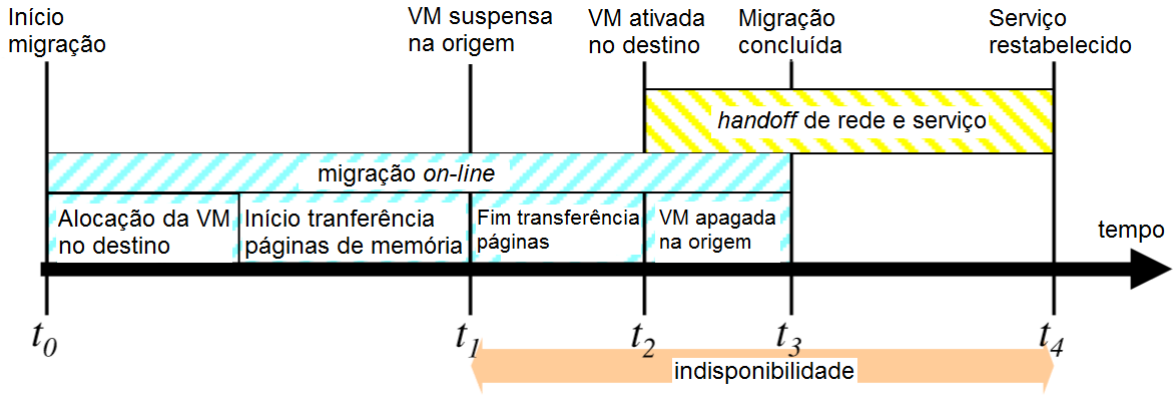

Figura 2.1: Etapas de migracao

A Figura 2.1, adaptada de [14], ilustra as principais etapas que ocorrem durante uma migração *on-line* de VMs. A partir de *t0* a migração começa com o hipervisor destino iniciando a alocação da VM. Inicia-se também a fase de transferência de páginas de memória a partir do hipervisor origem para o hipervisor destino. Durante essa fase, a VM está totalmente operacional na origem e, portanto, o serviço está totalmente disponível para os usuários. Após a fase de transferência inicial, a VM é suspensa na origem (*t1*) e todas as páginas de memória restantes e estado da CPU são transferidos para o hipervisor no destino. Obviamente, estando a VM fora, o serviço também estará indisponível. Após todas as informações necessárias estarem no hipervisor destino, a VM é ativada na máquina destino (*t2*)

enquanto a VM na origem começa a ser descartada, finalizando em *t3*.

Em primeira análise, a indisponibilidade deveria estar relacionada apenas ao tempo em que a VM esteve fora (*t1-t2*) e todos os serviços deveriam ser restabelecidos em *t2*. Porém, a indisponibilidade do serviço não depende somente do estado da VM, mas está relacionada também ao tempo de convergência da rede para identificar a mudança e se ajustar no sentido de restabelecer a conectividade.

O tempo dessa convergência está associado principalmente ao tempo de expiração do *cache* ARP que, em cada *switch*, estabelece a associação "MAC/-porta de saída" nos dispositivos de rede. A expiração de tempo do cache ARP é o principal responsável pela latência causada pelo ***handoff*** de nível 2. *Handoff* é o processo que permite que um nó móvel mude o seu ponto de interligação à rede de um local físico a outro. O estudo do tempo de *handoff* em redes é muito relacionado a redes sem fio (*wireless*), em que um nó móvel migrava de um ponto de acesso (Access Point - AP) para outro.

Na literatura, existem vários métodos para diminuir o tempo de *handoff* e a perda de pacotes associada: [15] [16] [17] [18]. Pode ser destacada a contribuição de [14], que propõe o estabelecimento de um *framework* para sincronização de eventos ou (*triggers*) entre os hipervisores.

2.3.3 Framework para sincronização de eventos entre hipervisores

A proposta destacada em [14] se beneficia do fato de que os hipervisores envolvidos no processo de migração possuem total controle sobre o mesmo (o que não acontece no caso de redes *wireless*) e podem, dessa forma, enviar mensagens ou diretivas para a rede (vista de uma forma genérica) para que essa possa se "preparar"para a migração que está prestes a ocorrer. Neste sentido, algumas mensagens pré-estabelecidas são apresentadas: **MIGRATION START** indica o início da migração; **SOURCE SUSPEND** indica a suspensão da VM na origem (tempo t1); **TARGET RESUME** indica o início da disponibilidade da VM no destino (tempo t2); por fim, **MIGRATION DONE** indica que o processo de migração foi finalizado (tempo t3).

Essas diretivas podem ser enviadas para um controlador central da rede que tenha a capacidade de gerenciar a infraestrutura a ponto de se antecipar e "preparar"a rede para a migração, diminuindo, dessa forma, o tempo de *handoff*. Fazendo-se uma comparação, segue a Figura 2.2, adaptada de [14], indicando a

diminuição do tempo de *handoff* para t5 em comparação com t4 da Fig. 2.1.

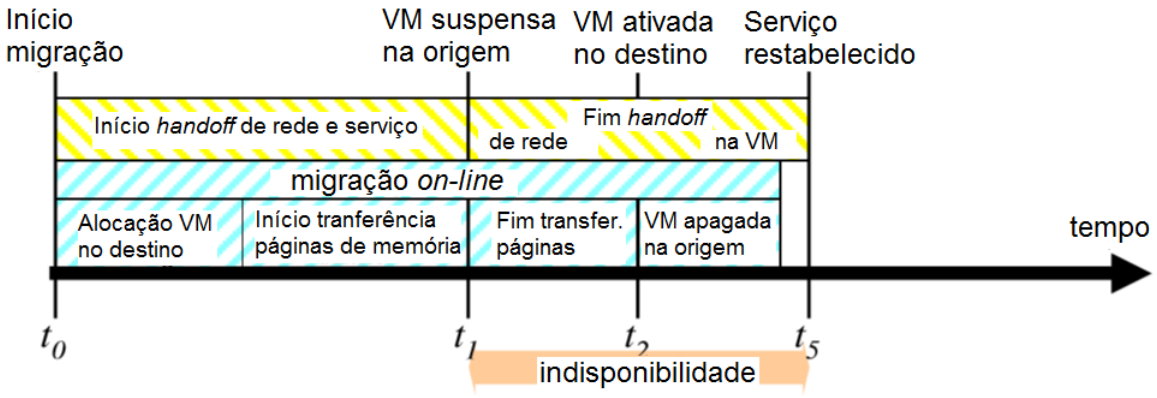

Figura 2.2: Etapas de migracao com diretivas do hipervisor

2.4 Mobilidade em Redes TCP/IP

O tema mobilidade em redes TCP/IP teve seu principal impulso com a popularização das redes móveis, e as primeiras propostas para endereçar esse problema foram testadas utilizando-se essa tecnologia. Pode-se dividir essas propostas baseando-se nas camadas do protocolo TCP/IP onde se propuseram suas soluções. As principais propostas para mobilidade na camada IP serão descritas a seguir, sendo que nas Seções 2.4.2 e 2.4.3 serão enumeradas as propostas que atuam nas camadas acima e abaixo do IP, respectivamente.

2.4.1 Mobilidade IP

Uma das primeiras abordagens sobre mobilidade em redes IP foi normatizada pelo IETF como adendo à especificação do IPv4[3] para comunicação destinada a dispositivos móveis, permitindo ao nó móvel alterar seu ponto de interligação com a rede sem, no entanto, alterar seu endereço IP. Dessa forma, as aplicações poderiam manter suas conexões ativas durante a migração.

Isso é possível permitindo ao nó móvel possuir dois endereços IP: um fixo (*home address*) associado ao endereço da sua rede original (*home network*), e outro temporário baseado no endereço da rede para qual o nó móvel migrou. O protocolo IP móvel realiza uma associação entre os dois endereços através de agentes que rodam no nó móvel (*home agent* - HA) e na rede (*foreign agent* - FA). Quando o nó móvel está em outra rede que não a sua rede original, HA e FA estabelecem

um túnel para comunicação entre o endereço fixo e o temporário.

Essa abordagem causa uma ineficiente triangulação do tráfego, pois o tráfego destinado a um dispositivo precisaria ser redirecionado a um ponto de âncora (*home agent*) para depois ser repassado ao destino móvel, e a resposta percorre o caminho inverso. Essa solução poderia ser até tolerada em uma rede IP móvel cujo tráfego é relativamente baixo, mas não para um *Data Center* em que servidores lidam com volumes de tráfego muito maiores.

Uma resolução desse problema de ancoragem é prevista na especificação do IPv6[19] ao definir um protocolo de rastreamento de nós móveis. Esse protocolo, chamado IPV6 móvel (*mobile IPv6*), permite aos nós móveis migrarem de uma rede para outra sem mudar seu *home address*. Pacotes podem ser roteados ao nó móvel independente de sua localização à rede, através da troca de informações previstas no protocolo. Porém, para que essa proposta seja implementada, todos os nós do *Data Center* precisariam suportar o protocolo IPv6, o que leva a outros problemas, dado que a transição de IPv4 para IPv6 não ocorre em pouco tempo e não se pode exigir que todos os nós de um *Data Center* suportem esse protocolo.

2.4.2 Mobilidade acima da camada IP

Foram também propostos alguns protocolos de suporte à mobilidade na camada de transporte, como o SCTP [20], o TCP-MH [21] e o DCCP [22]. Em geral essas propostas endereçam o problema de mobilidade adicionando semântica de migração para conexões TCP na pilha de protocolo dos *hosts* finais.

O suporte à mobilidade na camada de transporte tenta evitar a complexidade e o custo da solução transparente fornecido pela mobilidade IP. O argumento dado é que nem todas as aplicações requerem o nível de transparência e generalidade fornecida pela mobilidade IP.

Há também o suporte a mobilidade em camadas mais acima (Ex.: Snoeren [23] e MSOCKS [24])

As soluções propostas possuem a desvantagem de necessitarem de alterações nos *hosts* para permitir a funcionalidade de controle ou execução da migração. Isso tende a ser um problema em um *Data Center*, que necessita suportar uma diversidade de tipos de *hosts*.

2.4.3 Mobilidade abaixo da camada IP

A mobilidade também pode ocorrer em camadas abaixo à do IP. Na camada de enlace, a solução da Cisco chamada *Local Area Mobility* (LAM [25]) permite que nós móveis migrem de sua sub-rede origem para outra localidade, dentro da mesma organização, mantendo a conectividade transparente. O Cisco LAM é configurado em um roteador que fica responsável por procurar nós móveis que estão fora de sua rede original. Ao detectar tráfego em sua interface que não corresponda ao prefixo IP e máscara de sua interface - o que indicaria a presença de um *host* que migrou - o roteador instala uma rota específica para esse *host* e a divulga através de seu protocolo de roteamento. Os demais *hosts* da rede origem também podem se comunicar com o *host* móvel dado que, se um roteador conhece uma rota para um *host*, este roteador, ao interceptar uma requisição ARP, pode respondê-la através de *proxy* ARP [26].

Embora não dependa de alteração de *hosts*, técnicas como essa possuem pouca eficácia à medida que uma rede aumenta de tamanho e número de *hosts*, pois para cada nó móvel uma rota específica precisa ser instalada e divulgada, causando problemas de partição das tabelas de roteamento.

2.4.4 Redes Sobrepostas (*Overlay Networks*)

Uma rede sobreposta (*Overlay*) é uma rede virtual que é criada acima de uma rede física (Underlay). Os nós em uma rede sobreposta são interconectados através de *links* virtuais que correspondem a caminhos na rede física. A sobreposição normalmente é implementada na camada de aplicação, porém há várias implementações em camadas inferiores.

As redes sobrepostas não são geograficamente restritas, são flexíveis e adaptáveis à mudanças, assim como facilmente implementadas em comparação a outras redes. Como resultado, redes sobrepostas são utilizadas para implementar novas funcionalidades e resolver limitações da Internet [7].

No contexto de mobilidade, redes sobrepostas são utilizadas para interconexão de redes. Um dispositivo na borda de uma rede (em *hardware* ou *software*) é dinamicamente programado para gerenciar túneis entre os hipervisores e/ou *switches* da rede. O túnel sobreposto geralmente é terminado dentro de um *switch* virtual integrante do hipervisor ou de um dispositivo físico atuando como *gateway* da rede.

Diversas tecnologias de tunelamento são utilizadas para o estabelecimento e controle dos túneis. Podem ser destacados o *Stateless Transport Tunneling* (STT) [27], o *Virtualized Layer 2 Networks* (VXLAN) [4], o *Network Virtualization Using Generic Routing Encapsulation* (NVGRE) [5], o *Locator/ID Separation Protocol* (LISP) [6] e o *Generic Network Virtualization Encapsulation* (GENEVE) [28].

Embora flexíveis, as redes sobrepostas possuem limitações. Anderson et al. [29] pondera que as tecnologias de sobreposição não podem ser consideradas como caminho de implantação para tecnologias disruptivas. Primeiramente, elas são utilizadas para resolver limitações pontuais sem uma visão holística da rede nem interações entre as sobreposições. Em segundo, muitas das sobreposições estão sendo construídas na camada de aplicação, sendo, portanto, incapazes de suportar radicalmente diferentes arquiteturas.

2.4.5 VXLAN

Esta seção descreve o *Virtual eXtensible Local Area Network* (VXLAN). O VXLAN é um padrão IETF para atendimento a requisitos de *Data Centers* virtualizados multi-clientes (*multiple tenants*). Seu esquema e protocolos são endereçados aos requisitos de redes de provedores de *cloud* e *Data Centers* corporativos.

A computação em nuvem requer provisionamento de recursos para ambientes multi-clientes. No caso de nuvem pública, o provedor de *cloud* oferece seus serviços aos seus clientes utilizando a mesma infraestrutura física.

O isolamento de tráfego na rede de cada cliente é realizado através da camada 2 ou 3. No caso da camada 2, as VLANs são geralmente utilizadas. Uma VLAN utiliza um endereçamento de 12 bits para segregar o tráfego em vários domínios de *broadcast*. Esse esquema, em geral, atende bem *Data Centers* corporativos ou de pequeno e médio porte. Porém, para um provedor de *cloud* que precisa comportar um alto número de clientes em uma nuvem, cada um com um conjunto de VLANs e com uso massivo de virtualização, esse limite de 4094 VLANs pode não ser insuficiente.

A mobilidade também é afetada na necessidade de expansão entre ***pods*** de um *Data Center*. Um *pod* consiste em um ou mais *racks* de servidores associados pela conectividade de rede e *storage*. Os *pods* são isolados a nível 3 e, à princípio, não é possível a migração de VMs entre eles. Os clientes de uma nuvem podem inicialmente possuir suas VMs restritas a um *pod* e, à medida de seu crescimento,

requisitarem VMs em outros *pods*. Para que a migração seja possível é necessário realizar uma **extensão em L2** (nível 2) para permitir a livre movimentação de VMs entre os *pods*.

O isolamento através de nível 3 (L3) também não é adequado para o caso de *Data Centers* multi-clientes, pois dois clientes podem necessitar utilizar um mesmo conjunto de endereçamento IP, necessitando de isolamento para evitar sobreposição de roteamento.

O VXLAN endereça tais requisitos ao permitir a extensão L2 através de sobreposição em L3. O VXLAN realiza isso criando túneis entre dois segmentos de redes locais (L2) e encapsulando o tráfego dentro de pacotes UDP para envio roteado pelo túnel.

O VXLAN atribui para cada segmento um endereçamento de 24 bits chamado *VXLAN Network Identifier* (VNI), aumentando para 16M a quantidade de redes que podem coexistir no mesmo domínio administrativo. A Figura 2.3, baseada em [30], ilustra o cabeçalho VXLAN e o formato do pacote UDP ao encapsular o VXLAN.

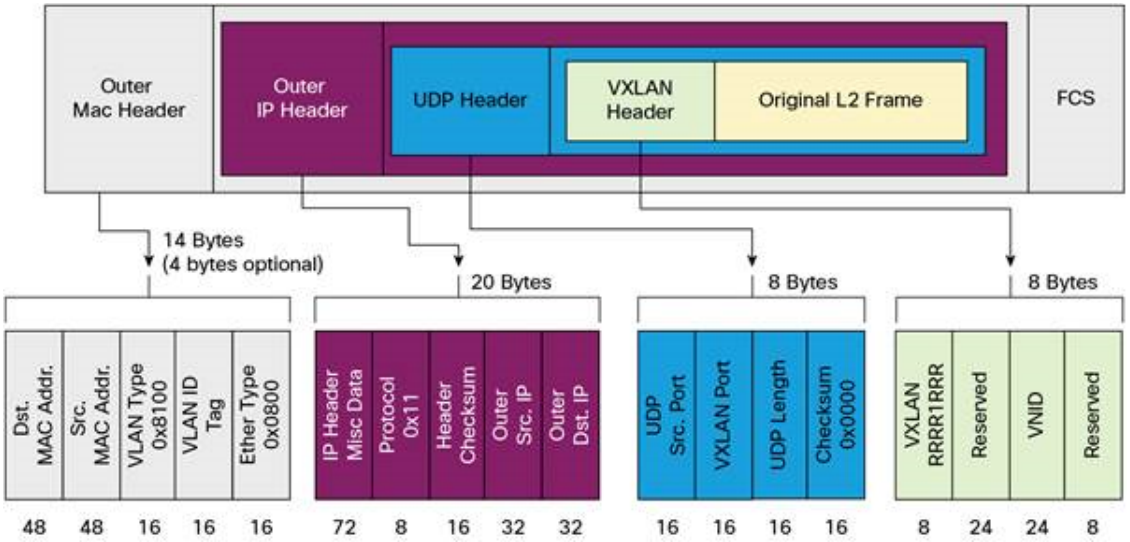

Figura 2.3: Formato do pacote VXLAN

O VXLAN utiliza dispositivos finais de túnel (*VXLAN tunnel endpoints* - VTEP) para mapear os clientes e dispositivos finais e também para a função de encapsulamento e desencapsulamento dos pacotes UDP. Usualmente um VTEP é implementado em um hipervisor em um servidor que hospeda uma ou mais VMs, podendo estabelecer o túnel com outro hipervisor para proceder uma migração *on-line* transparente de VM. O VTEP também pode ser implementado em *switches*

ou servidores físicos e ser implementado em *hardware* ou *software*. O VTEP possui duas interfaces: uma no *switch* do segmento de LAN local e outra na interface IP para o transporte roteado sobre uma rede IP, conforme mostrado na Figura 2.4, adaptada de [30].

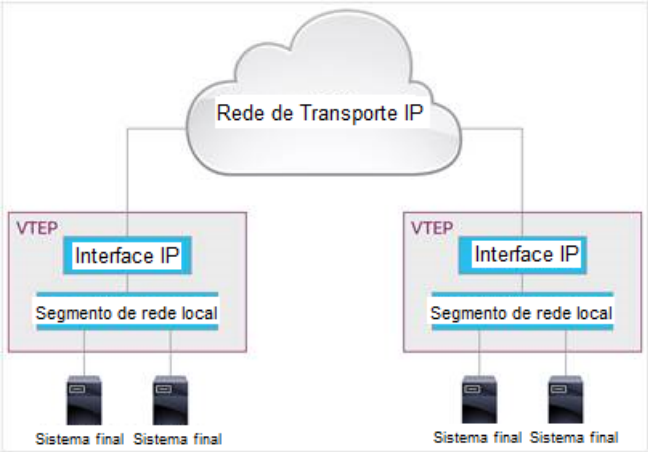

Figura 2.4: VTEP

Não obstante à sua flexibilidade, o VXLAN possui limitações inerentes a redes sobrepostas ao não possuir um visão holística da rede. Nos VTEPs que podem estar espalhados pela infraestrutura, existe o mapeamento entre as VMs e os VNIs (*Virtual Network Interfaces*) a que estão ligados. A implementação em *software* nos hipervisores tende a se tornar um gargalo à medida que aumenta a quantidade de túneis e sua demanda de processamento para encapsulamento e desencapsulamento. Por outro lado, a implantação em *hardware* depende de uma profunda e onerosa atualização dos equipamentos para que esses suportem os VTEPs.

Também não está previsto no padrão VXLAN uma associação com os elementos de roteamento da infraestrutura, possibilitando assim que um pacote seja comutado a um *pod* procurando um *host* que migrou para outro, criando assim um tráfego desnecessário através do túnel e por conta da triangulação.

Em resumo, a tecnologia VXLAN possui diversas características que endereçam os requisitos de computação em nuvem, mas sua implementação deveria estar associada a outros módulos que o complementaria em relação às suas limitações. O PathFlow, que será detalhado no Capítulo 3, possui uma visão holística da rede e prevê suporte ao VXLAN.

Como enfatizado anteriormente, muitas das limitações dos *Data Centers* estão associadas à rigidez e à falta de flexibilidade do TCP/IP. Na Seção 2.5 será revisto

o conceito de SDN e posteriormente serão abordadas as soluções SDN para o problema de mobilidade.

2.5 SDN

Apesar de sua ampla adoção, as redes IP tradicionais são complexas e difíceis de administrar [31]. Para expressar as políticas de redes de alto nível, os operadores de rede precisam configurar cada dispositivo de rede separadamente através de comandos de baixo nível e muitas vezes específicos do fabricante.

Adicionalmente, as redes atuais são verticalmente integradas. O plano de controle (que decide como lidar com o tráfego de rede) e o plano de dados (que encaminha o tráfego de acordo com as decisões tomadas pelo plano de controle) são unificados dentro dos dispositivos de rede, reduzindo a flexibilidade e dificultando a inovação e evolução da infraestrutura de rede

Rede definida por software (*Software-Defined Networking* - SDN) [8, 32] é um paradigma emergente que endereça as limitações de infraestrutura das redes atuais. Inicialmente, ele quebra a integração vertical, separando o plano de controle do plano de dados dos dispositivos de rede, fazendo com que roteadores e *switches* tornem-se simples dispositivos de encaminhamento de tráfego, sendo o controle lógico efetuado por um **controlador** centralizado (ou sistema operacional de rede). Simplifica, dessa forma, a aplicação de políticas, reconfiguração e evolução da rede[33].

Uma consequência importante dos princípios do SDN é a separação de interesses introduzida entre a definição de políticas de rede, sua implementação em *switches* e o encaminhamento de tráfego. Esta separação é o ponto chave para a flexibilidade, quebrando o problema de controle de rede em partes distintas, tornando mais fácil a criação e introdução de novas abstrações de rede, simplificando o gerenciamento e facilitando a evolução e inovação da rede [34].

2.5.1 Arquitetura SDN

A Figura 2.5, adaptada de [34], indica a arquitetura simplificada do SDN. O controlador SDN se comunica com os elementos de encaminhamento de dados através de interfaces *southbound*. Adicionalmente, se comunicam com as camadas superiores através de interfaces *northbound*.

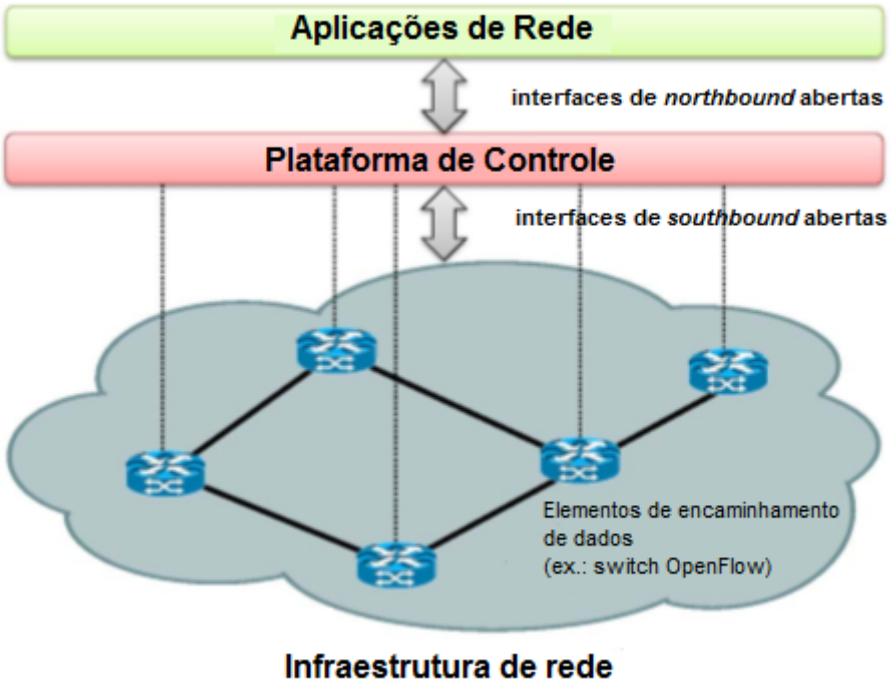

Figura 2.5: Visão simplificada da arquitetura SDN

Na seção seguinte será detalhado o Openflow, que é o principal representante de interface *southbound*.

2.5.2 Openflow

O Openflow foi proposto para permitir inovação por parte de pesquisadores em testar novas formas de comunicação em rede. O Openflow foi proposto para padronizar a comunicação entre um controlador e os *switches* em uma arquitetura SDN [35], definindo também o protocolo dessa comunicação. O Openflow, dessa forma, provê as formas de controlar os dispositivos de rede, que no caso são genericamente denominados *switches* Openflow, sem estar atrelado a comandos específicos do fabricante do *hardware* e sem que os fabricantes precisem expor o código de seus equipamentos [35].

Dessa forma, através do protocolo OpenFlow, os controladores podem programar tabelas de fluxo dos *switches*. As tabela de fluxo são parte integrante do *switch* Openflow e serão detalhadas adiante. O Openflow é mantido pelo *Open Network Foundation* - ONF [36].

Diferentes versões do protocolo OpenFlow foram desenvolvidas. A primeira foi a 0.2, lançada em maio de 2008, que está atualmente obsoleta. A versão

1.0, lançada em dezembro de 2009, é atualmente a mais amplamente difundida e implementada. A proposta deste trabalho é baseada na versão 1.0 por esse motivo. Após a versão 1.0, foram lançadas as versões 1.1, 1.2, 1.3, 1.4 e atualmente se encontra na versão 1.5 [35].

2.5.3 Componentes do *Switch*

Segundo a especificação v1.0.0, os *switches* Openflow possuem os seguintes componentes: uma tabela de fluxos (*flow table*), um controlador Openflow e um canal seguro de comunicação entre os dois utilizando o protocolo Openflow. A Figura 2.6, adaptada de [37], ilustra esses componentes.

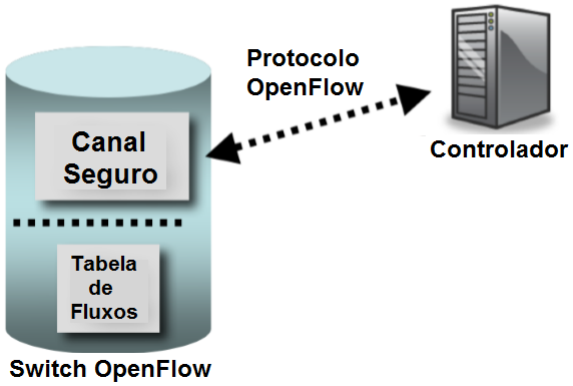

Figura 2.6: Um *switch* Openflow se comunica com o controlador através de uma conexão segura utilizando-se o protocolo Openflow

A tabela de fluxos possui 12 campos que são utilizados para a classificação de um fluxo, uma ação para cada fluxo e contadores. O canal seguro irá realizar a conexão entre o *switch* OpenFlow e o controlador e permitirá que os comandos fluam por ele. O protocolo OpenFlow é o que será utilizado para essa comunicação.

O *switch* OpenFlow, por sua vez, será responsável para atuar da seguinte forma como os fluxos entrantes:

• Caso haja uma entrada equivalente para o fluxo em questão em sua tabela de fluxos, deve realizar a ação associada a este fluxo, como encaminhar o fluxo entrante para uma determinada interface de saída, por exemplo;

• No caso de não haver uma entrada equivalente na tabela de fluxos que corresponda ao pacote entrante, o *switch* deve ser capaz de encapsular o primeiro pacote (ou todos, de acordo com a necessidade) e encaminhá-lo para o controlador

através do canal seguro, que, por sua vez, irá decidir o que fazer com esse fluxo - pode, por exemplo, instalar uma entrada na tabela de fluxos do *switch* para tratar esse novo fluxo; e

• Ou, por fim, o *switch* OpenFlow poderá descartar os pacotes.

2.5.4 Tabela de Fluxos

A tabela de fluxos é parte essencial de um *switch* OpenFlow. Na Tabela 2.5.4, adaptada de [37], são ilustrados os campos de uma entrada na tabela de fluxos.

Tabela 2.1: Entradas da tabela de fluxos

Campos de Cabeçalho	Contadores	Ações

Os **campos de cabeçalho** (*header fields*) definem os campos do pacote entrante que serão comparados. Caso o valor do parâmetro especificado corresponda ao valor da entrada da tabela, a ação definida no campo **Ação** (*action*) será realizada. Cada campo do cabeçalho pode ser ou não definido, o que permite ao *switch* Openflow analisar um ou vários campos do pacote entrante e aplicar as ações correspondentes de acordo com o informado no campo Ação. Essa flexibilidade permite ao pesquisador moldar o comportamento do *switch* para que ele possa atuar como um *switch*, roteador ou outro dispositivo de rede qualquer. Na Tabela 2.2, baseada em [37], estão ilustrados os campos de cabeçalho possíveis de serem comparados na versão 1.0.0.

Tabela 2.2: Campos do pacote utilizados para correspondência com entradas de fluxos

Ingress Port	Ether Src	Ether dst	Ether type	VLAN id	VLAN prio-rity	IP src	IP dst	IP proto	IP ToS bits	TCP/ UDP src port	TCP/ UDP dst port

O campo **contadores**, por sua vez, são incrementados cada vez que é encontrada uma correspondência à linha do fluxo. Existem quatro tipos diferentes: os contadores por tabela (*table*), por fluxo (*flow*), por porta (*port*) e por fila (*queue*). Estes contadores podem ser utilizados em mensagens de estatísticas definidas pelo OpenFlow. A Tabela 2.3, baseada em [37], ilustra a lista de contadores.

Quanto às ações que deverão ser tomadas com os pacotes, existem ações obrigatórias que todos os *switches* OpenFlow devem implementar e outras opcionais.

Tabela 2.3: Lista de contadores

Counter	Bits
Per Table	
Active Entries	32
Packet Lookups	64
Packet Matches	64
Per Flow	
Received Packets	64
Received Bytes	64
Duration (seconds)	32
Duration (nanoseconds)	32
Per Port	
Received Packets	64
Transmitted Packets	64
Received Bytes	64
Transmitted Bytes	64
Receive Drops	64
Transmit Drops	64
Receive Errors	64
Transmit Errors	64
Receive Frame Alignment Errors	64
Receive Overrun Errors	64
Receive CRC Errors	64
Collisions	64
Per Queue	
Transmit Packets	64
Transmit Bytes	64
Transmit Overrun Errors	64

Ao se conectar ao controlador, o *switch* deve informar quais ações opcionais o mesmo implementa.

Cada entrada na tabela de fluxos é associada a uma ou mais ações. Se para uma determinada entrada na tabela não houver uma ação especificada, os pacotes desse fluxo serão descartados.

Abaixo seguem os tipos de ações:

1. Encaminhamento
   (a) Obrigatório
      - ALL - Envia o pacote para todas as interfaces, exceto a interface de entrada;
      - CONTROLLER - Encapsula e envia o pacote para o controlador;
      - LOCAL - Envia o pacote para a pilha de rede local;
      - TABLE - Realiza ações na tabela de fluxos; e
      - IN_PORT - Envia o pacote para a porta de entrada.
   (b) Opcional

- NORMAL - Processa o pacote utilizando um encaminhamento tradicional; e
- FLOOD - Inunda a rede com o pacote, sem incluir a interface de entrada, levando em consideração o *Spanning Tree*.

2. Enfileirar (opcional) - Encaminha o pacote através de uma fila relacionada a uma porta;
3. Descartar (obrigatória);
4. Modificar campo (opcional):
   - Setar Vlan ID;
   - Setar Vlan Priority;
   - Separar o cabeçalho da Vlan;
   - Modificar o endereço MAC (*Media Access Control*) de origem;
   - Modificar o endereço MAC de destino;
   - Modificar o endereço IP de origem;
   - **Modificar o TOS**;
   - Modificar a porta de transporte de origem; e
   - Modificar a porta de transporte de destino.

O item **Modificar o TOS** foi destacado, pois esse recurso foi utilizado para implementação da primeira versão do PathFlow, conforme será detalhado na Seção 3.2.

O fluxograma referente a entrada de um pacote em um *switch* OpenFlow é detalhado na Figura 2.7, baseada em [38].

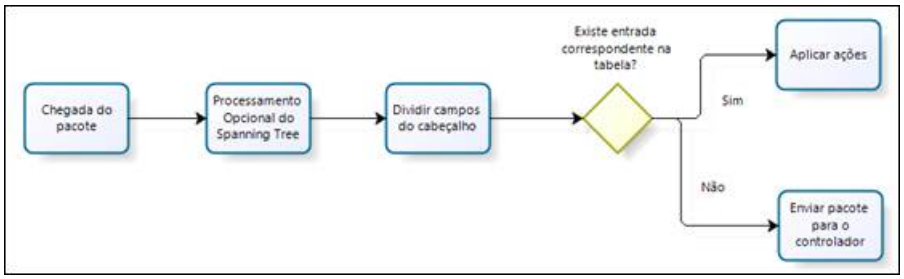

Figura 2.7: Fluxograma do processamento dos pacote

Deve ser enfatizado que, assim como (nos roteadores) rotas para sub-redes mais específicas têm prioridade, entradas na tabela de fluxos que especifiquem exatamente o fluxo entrante, ou seja, que não apresentem campos coringa (identificados por *), terão maior prioridade.

2.5.5 Canal seguro

Outra parte essencial da especificação de um *switch* OpenFlow é o estabelecimento de um canal seguro com o controlador. Através desse canal, o controlador irá configurar e gerenciar o *switch* OpenFlow.

Embora uma conexão TCP seja permitida entre o *switch* e o controlador, é recomendado o uso de SSL/TLS com o objetivo de garantir confidencialidade em toda comunicação. É importante ressaltar que o OpenFlow não requer uma conexão física direta entre o *switch* e o controlador, podendo utilizar redes já em produção para efetuar a comunicação. Daí a importância de um canal seguro.

O protocolo OpenFlow suporta três tipos diferentes de mensagens:

- Controlador-Switch - Geradas pelo controlador para gerenciar e inspecionar o estado de um *switch*;
- Assíncronas - Geradas pelo *switch* para atualizar o controlador sobre eventos da rede e mudanças no estado do *switch*;
- Simétricas - Podem ser geradas tanto pelo controlador quanto pelo *switch*, sendo enviadas sem solicitação.

Abaixo, seguem os sub-tipos de cada:

1. Controlador-Switch
    - Características (Features) - O controlador requisita as características do *switch* e esse deve responder com as características suportadas;
    - Configuração (Configuration) - Usado para configurar ou solicitar configurações do *switch*;
    - Modificação de estado (Modify-State) - Usado para adicionar, deletar e modificar a tabela de fluxos e para setar propriedades nas portas do *switch*;

- Leitura de estado (Read-State) - Coleta estatísticas;
- Envio de pacote (Send-Packet) - Utilizado para enviar pacotes por uma determinada porta do *switch*; e
- Barreira (Barrier) - Usado para garantir que as dependências foram atendidas ou para receber notificações de operações finalizadas.

2. Assíncrona

   - Entrada de pacotes (Packet-In) - Utilizado quando fluxos não classificados entram no *switch*;
   - Remoção de fluxo (Flow-Removed) - Mensagem enviada para o controlador quando um fluxo é removido da tabela, seja por *Idle Timeout*, *Hard Timeout* ou por uma mensagem de modificação da tabela de fluxos que delete a entrada em questão;
   - Estado da porta (Port-Status) - Mensagem enviada para o controlador sempre que há mudanças nas configurações de portas; e
   - Erro (Error) - Notificações de erros.

3. Simétrica

   - Hello - Mensagens trocadas entre o controlador e o *switch* quando uma conexão é estabelecida;
   - Echo - Mensagens usadas para identificação de latência, largura de banda e existência de conectividade; e
   - Vendor - Proveem uma forma padrão para os *switches* OpenFlow oferecerem funcionalidades adicionais.

No estabelecimento de uma comunicação OpenFlow, o controlador e o *switch* devem enviar imediatamente uma mensagem Hello (OFPT_HELLO) contendo a mais alta versão do protocolo OpenFlow suportada pelos dispositivos. Ao receber a mensagem, o dispositivo deve escolher a menor versão do OpenFlow entre a que foi enviada e recebida. Se as versões forem compatíveis, a comunicação tem continuidade. Caso não sejam, uma mensagem de erro é gerada (OFPT_ERROR) e a conexão é encerrada.

É interessante informar também que, caso haja alguma perda de conexão entre o *switch* e o controlador, o *switch* tentará se conectar ao controlador *back-up*, caso exista. Caso também falhe essa conexão, o *switch* entrará em modo de **emergência**,

modo esse que utilizará apenas as entradas na tabela de fluxos marcadas com um bit de emergência. Todas as outras entradas serão apagadas.

Para modificar as tabelas de fluxos dos *switches*, o controlador poderá gerar cinco tipos de mensagens diferentes, como ilustrado na Figura 2.8, baseada em [37].

```
enum ofp_flow_mod_command {
    OFPFC_ADD,              /* New flow. */
    OFPFC_MODIFY,           /* Modify all matching flows. */
    OFPFC_MODIFY_STRICT,    /* Modify entry strictly matching wildcards */
    OFPFC_DELETE,           /* Delete all matching flows. */
    OFPFC_DELETE_STRICT    /* Strictly match wildcards and priority. */
};
```

Figura 2.8: Tipos de mensagens para modificação da tabela de fluxos

As mensagens ADD adicionarão entradas na tabela de fluxos, as mensagens do tipo MODIFY modificarão entradas já existentes e as mensagens do tipo DELETE apagarão entradas na tabela de fluxo. As mensagens OFPFC_MODIFY_STRICT e OFPFC_DELETE_STRICT correspondem a entradas na tabela de fluxos idênticas às especificadas.

Além das mensagens OFPFC_DELETE e OFPFC_DELETE_STRICT, as entradas na tabela de fluxos podem ser removidas pelos parâmetros *idle_timeout* ou *hard_timeout*. A entrada será apagada caso não sejam verificados pacotes correspondentes a uma determinada entrada na tabela durante o tempo indicado em *idle_timeout* segundos, ou caso transcorra o tempo indicado em *hard_timeout* segundos após a inserção dessa entrada na tabela.

2.5.6 Demais versões do Openflow

Até então, o OpenFlow 1.0.0 foi apresentado com mais detalhes, considerando que o mesmo é a versão mais utilizada e difundida. Entretanto, muitas mudanças foram realizadas desde o lançamento da versão 1.0.0 até a versão 1.4. Abaixo, seguem algumas características das demais versões:

OpenFlow 1.1

Na especificação do OpenFlow 1.1, os *switches* OpenFlow contêm diversas tabelas de fluxos e uma tabela de grupo, ao invés de uma única tabela de fluxos

como na versão 1.0.0. A Figura 2.9, adaptada de [39], ilustra os componentes do *switch* OpenFlow 1.1 em que estão ilustradas as diversas tabelas de fluxos, assim como a tabela de grupo.

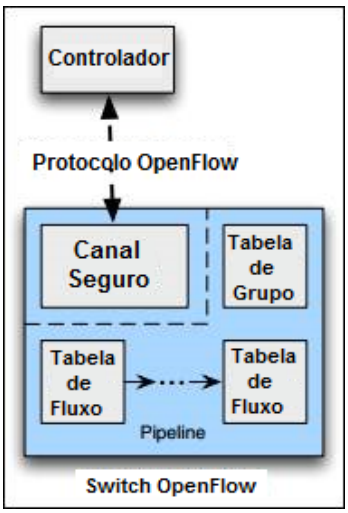

Figura 2.9: Componentes de um switch Openflow 1.1

Adicionalmente, a Figura 2.10, baseada em [38], ilustra como é o processamento de pacotes ao entrarem em um *switch* OpenFlow 1.1.

Um fluxo de pacotes, ao entrar no *switch* OpenFlow, poderá ser verificado por várias tabelas de fluxos, para que diversas ações diferentes sejam realizadas. A tabela de grupo é um tipo especial de tabela projetada para executar ações que sejam comuns para diversos fluxos.

Além disso, na versão 1.1.0, três novos campos de cabeçalho foram incluídos: Metadata (que pode ser usado para passar informações entre as tabelas no *switch*), MPLS label e MPLS traffic class. Maiores detalhes podem ser encontrados na especificação OpenFlow 1.1.0 em [39].

OpenFlow 1.2

O OpenFlow 1.2 foi lançado em dezembro de 2011 e uma das principais implementações foi o suporte ao protocolo IPv6, incluindo novos campos de cabeçalho nas tabelas de fluxos.

Adicionalmente, passou a suportar a possibilidade de os *switches* se conectarem a mais de um controlador ao mesmo tempo. Ao se conectar a vários controladores, aumenta-se a segurança, já que o *switch* pode continuar a operar normalmente mesmo se um controlador ou a conexão ficar indisponível [35].

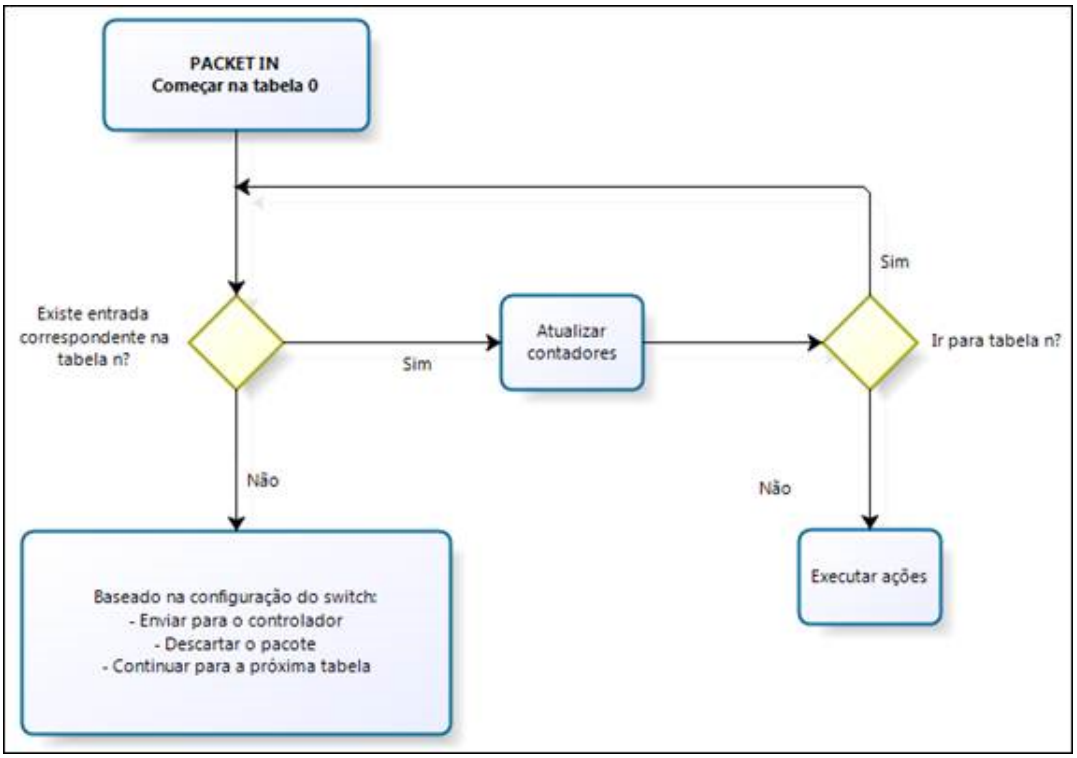

Figura 2.10: Fluxograma detalhando o fluxo de pacotes do switch Openflow 1.1

Maiores detalhes podem ser encontrados em [40].

OpenFlow 1.3

A versão 1.3.0 do OpenFlow foi lançada em junho de 2012 e passou a suportar o controle de taxas de pacotes através de medidores de fluxos, introduzindo a tabela de medições Meter Table.

A tabela de medições por fluxo permite ao OpenFlow implementar simples operações de QoS, como limitação de taxas de transmissão, e pode ser combinada com as filas por porta para criar políticas de QoS mais complexas.

Adicionalmente, a versão 1.3 permite ao *switch* criar conexões auxiliares para o controlador, ajudando a melhorar o desempenho de processamento do *switch*, explorando o paralelismo presente na maioria dos *switches*.

OpenFlow 1.4

A versão 1.4 do OpenFlow introduziu o suporte a portas ópticas. Além disso, os conceitos de *Eviction* e *Vacancy Events* foram introduzidos para evitar que as tabelas de fluxos fiquem cheias, dado que as mesmas têm capacidades finitas.

Nas especificações anteriores, quando uma tabela de fluxos enchia, novos fluxos não eram inseridos nas tabelas e uma mensagem de erro era enviada para o controlador. Porém, a ocorrência de tal situação era problemática e poderia causar interrupções no serviço. O *Eviction* adiciona um mecanismo que permite ao *switch* apagar automaticamente as entradas nas tabelas de fluxos que tenham menor importância. Já o *Vacancy events* permite ao controlador configurar um limiar que, ao ser atingido, faz o *switch* enviar mensagens de alerta para o controlador. Esses recursos permitem que o controlador possa reagir com antecedência, evitando que as tabelas de fluxos fiquem cheias.

As subseções acima deram apenas uma visão global sobre as características das versões do OpenFlow posteriores à 1.0.0. Maiores detalhes e outras características podem ser encontradas nas especificações técnicas, citadas na bibliografia.

2.5.7 Controladores

Como informado anteriormente, o controlador é parte essencial nas redes definidas por *software*. São os controladores que irão definir a lógica de encaminhamento através das regras que irão configurar nos *switches* OpenFlow e, dessa forma, indicar o comportamento de toda a rede.

Atualmente, existem alguns controladores disponíveis para uso, como NOX [41], POX [42], Maestro [43], Trema [44] e Beacon [45], entre outros. A Tabela 2.4, baseada em [46], indica alguns controladores existentes, informando também qual a linguagem de sua implementação e se são baseados em *software* livre, assim como o desenvolvedor.

2.5.8 POX

Aqui será destacado o controlador POX. O POX é derivado do NOX clássico, que foi um dos primeiros controladores Openflow. O POX é desenvolvido em Python e apresenta em sua distribuição diversos componentes reusáveis e bibliotecas interessantes, como as seguintes informadas:

- **openflow.discovery**: Utiliza mensagens LLDP (*Link Layer Discovery Protocol*) para descobrir a conectividade entre os *switches* com o objetivo de mapear a topologia da rede. O mesmo cria eventos (os quais podem ser "escutados") quando um *link* fica *up* ou *down*. Esse componente foi essencial para a aplicação desenvolvida;

Tabela 2.4: Alguns controladores Openflow existentes

Controlador	Implementação	Software Livre	Desenvolvedor
POX	Python	Sim	Nicira
NOX	Python/C++	Sim	Nicira
MUL	C	Sim	Kulcloud
Maestro	Java	Sim	Universidade Rice
Trema	Ruby/C	Sim	NEC
Beacon	Java	Sim	Stanford
Jaxon	Java	Sim	Desenvolvedores Independentes
Helios	C	Não	NEC
Floodlight	Java	Sim	BigSwitch
SNAC	C++	Não	Nicira
Ryu	Python	Sim	NTT, grupo OSRG
NodeFlow	JavaScript	Sim	Desenvolvedores Independentes
ovs-controller	C	Sim	Desenvolvedores Independentes
Flowvisor	C	Sim	Stanford/Nicira
RouteFlow	C++	Sim	CPqD

- **openflow.spanning_tree**: Esse componente utiliza o **discovery** para construir uma visão da topologia da rede e monta uma árvore *spanning tree* para desabilitar a inundação (*flooding*) nas portas dos *switches* que não estão na árvore;
- **pox.lib.revent**: Biblioteca do POX que implementa eventos, sendo que todos os eventos do POX são instâncias de sub-classes da classe *Event*;
- **pox.lib.recoco**: Biblioteca do POX que implementa *threads* e *timers*; e
- **pox.lib.addresses**: Biblioteca utilizada para lidar com endereços IP e Ethernet (MAC).

2.6 Mobilidade em Redes definidas por *Software*

O tema mobilidade também foi endereçado por iniciativas baseadas em SDN. Nas seções seguintes serão abordadas suas contribuições, características e limitações.

2.6.1 VICTOR

O VICTOR (*Virtually Clustered Open Router*) [9] é uma arquitetura de rede que endereça o problema de migração de máquinas entre diversas redes permitindo

a migração de VMs mantendo seus endereços IP originais. A ideia principal do VICTOR, como mostrado na Figura 2.11, baseada em [9], é criar um *cluster* de elementos de encaminhamento (*Forwarding Elements* - FE), sendo esses dispositivos L3, que servem como placas (*line cards*) com múltiplas portas virtuais de um único roteador virtualizado. Dessa forma, a agregação de FEs realiza o encaminhamento de dados para o tráfego de uma rede. Os FEs são distribuídos ao longo de várias redes, o que ajuda a suportar a migração de máquinas virtuais através dessas redes.

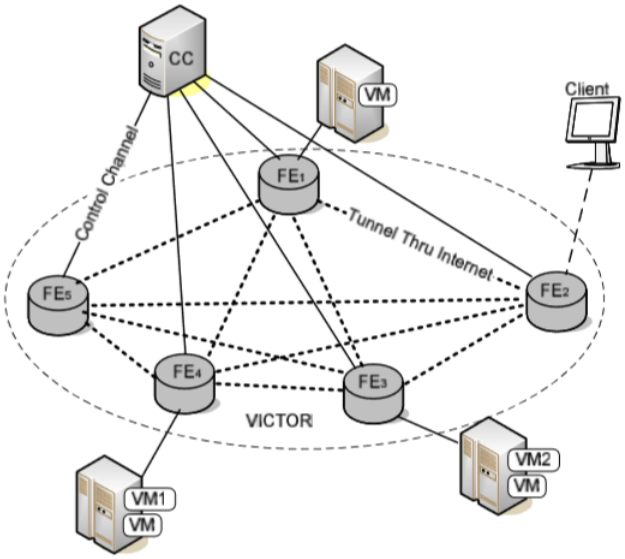

Figura 2.11: Arquitetura do Victor

O plano de controle é suportado por um ou vários controladores centralizados (*Central Controllers* - CC). A VM é implantada em um servidor conectado a apenas um FE de borda. O CC mantém uma tabela de topologia que especifica a conectividade entre FEs, e uma tabela de endereço que relaciona a conectividade de cada VM à FE no qual o servidor que hospeda a VM é conectada. O CC calcula a melhor rota de cada FE para as VMs e divulga essa informação entre FEs, que utilizam essas tabelas de roteamento para encaminhar os pacotes.

A principal limitação do VICTOR é que ele requer suporte a bases de informação de encaminhamento (*Forwarding Information Base* - FIBs) de grande tamanho, levando a preocupações quanto à escalabilidade da solução [1].

3. Proposta de Solução

3.1 Topologias de um *Data Center*

Um *Data Center* é um ambiente onde são interligados servidores (máquinas físicas), equipamentos de *storage* e dispositivos de rede (*switches*, roteadores e cabeamento) suportados por sistema de distribuição elétrica e de refrigeração [1].

A rede de um *Data Center* é a infraestrutura de comunicação usada no *Data Center*. É composta pela topologia de rede, equipamentos de roteamento e *switching*, e pelo conjunto de seus protocolos, como IP e Ethernet [1]. A seguir, será apresentada uma topologia convencional utilizada em *Data Centers* e outras topologias recentemente propostas.

A Figura 3.1, baseada em [1], ilustra uma topologia de rede convencional em três camadas. A camada de acesso (*access layer* - AS) é formada por *switches* topo de *rack* (*Top of Rack* - ToR) que são os *switches* nos quais são ligados os servidores em cada *rack* do *Data Center*. Esses *switches* são ligados aos *switches* de fim de fila (*end-of-row* - EoR), que formam a camada de agregação (*aggregation layer*) ou distribuição. A terceira e mais superior camada é a de núcleo (*Core layer*) e é formada por roteadores ou *switches* de núcleo (*CORE routers*, *CORE Switches*). Na camada de núcleo geralmente é realizado o roteamento da rede do *Data Center*.

Cada ToR é ligado a mais de um *switch* de distribuição para fins de redundância, e os *switches* de agregação fazem a interconexão dos *switches* ToR aos roteadores ou *switches* de núcleo que, por sua vez, têm saída para a Internet.

Essa topologia convencional em três camadas é adequada aos casos de *Data Centers* tradicionais baseados primordialmente em servidores de *rack*. Porém, o advento massivo da virtualização adicionou um nível a mais de comutação, além disso muitas das funções de rede (*switching*, *firewall*, controladores e balancea-

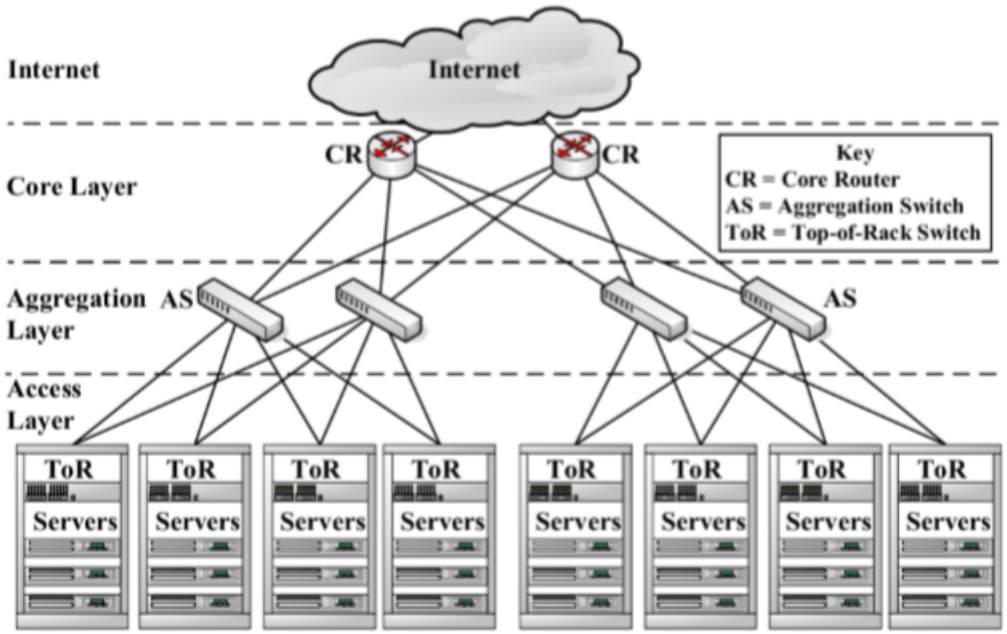

Figura 3.1: Topologia Convencional de um *Data Center*

mento de carga) são implementadas em *software* em um servidor físico. Na Figura 3.2, baseada em [47], é ilustrado como exemplo um ***switch* virtual**.

O *switch* virtual é geralmente um *plugin* integrante do hipervisor. As máquinas virtuais possuem **placas de rede ethernet virtuais** que se conectam ao *switch* virtual. O hipervisor, por sua vez, se conecta ao *switch* físico da infraestrutura de rede (normalmente o *switch* ToR) através da placa de rede do servidor onde o hipervisor se encontra. A virtualização adiciona, portanto, uma nova dimensão de rede.

À medida que se concentram mais servidores virtuais, começam a ser requisitadas maiores larguras de banda para atender o tráfego agregado de várias máquinas virtuais que precisam sair pela mesma conexão ethernet. Grande parte desse tráfego se concentra apenas entre os *switches* topo de *rack* e de agregação, o que levou ao desenvolvimento de novas topologias mais adequadas ao atendimento a esses novos requisitos.

A topologia **Clos** [48] foi proposta para endereçar o problema de tráfego agregado entre camadas. Sua ideia consiste em dividir a rede do *Data Center* em vários estágios de *switches*. Cada *switch* de um estágio deve se conectar a todos os outros *switches* dos estágios subsequentes. Dessa forma, existe uma total interconexão entre estágios subsequentes. A Figura 3.3, baseada em [1], exemplifica

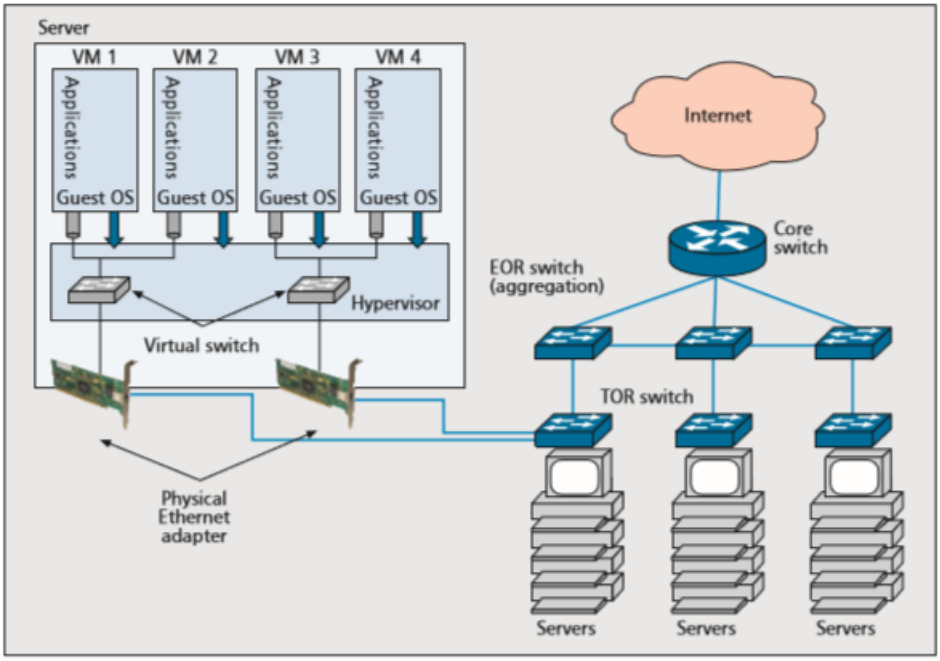

Figura 3.2: Arquitetura de Rede com virtualização

uma topologia Clos de três estágios.

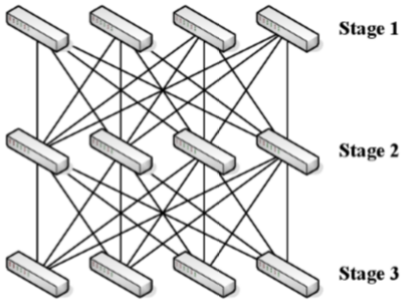

Figura 3.3: Topologia Clos de três estágios

Outro modelo de interconexão é o ***fat-tree*** [49]. O *fat-tree* é um caso especial da topologia Clos. Na Figura 3.4, baseada em [1], está ilustrado uma topologia *fat-tree* que é dividida em k **pods** (*pieces of Data Center* - "pedaços"de *Data Center*) e cada um possui duas camadas: agregação (*aggregation*) e acesso (*edge*), cada camada com k/2 *switches*. Na camada de núcleo (*core*), cada k/2 *switches* possuem uma porta ligada a um dos *switches* de agregação de cada pod, com os demais k/2 *switches* possuindo portas ligadas aos demais *switches* de agregação de cada pod. Cada um dos *switches* de acesso da camada inferior é diretamente conectado a k/2 *hosts*. E cada uma das k/2 portas restantes são conectadas a k/2 portas da camada de agregação da hierarquia [50].

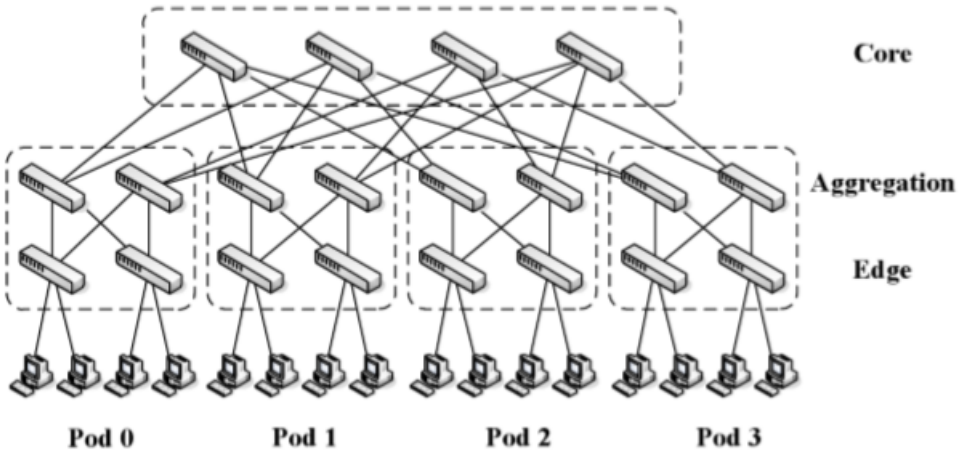

Figura 3.4: Topologia *Fat Tree* com 4 *pods*

Embora, as topologias Clos e *fat-tree* possibilitem a criação de topologias multi-camadas, é muito comum a adoção de *switches* topo de *rack* menores e com maior densidade de portas ligados a *switches* modulares de alta capacidade criando uma rede *flat* densamente conectada com apenas duas camadas. A essa arquitetura é dado o nome de *spine and leaf*.

Em uma arquitetura *spine and leaf* cada um dos *switches* na função *leaf* deve ser interconectado a todos os *switches* da função *spine*. O tráfego de cada *leaf* para os *spines* é agregado utilizando-se ECMP (*Equal cost multipath*).

Essa arquitetura permite um aumento escalável do tráfego, com a adição de mais *switches spine* e suas interligações aos *switches leafs*. Dessa forma, a arquitetura *spine and leaf* permite um aumento de *throughput* de forma uniforme e previsível [51].

A Figura 3.5, baseada em [52], ilustra uma implementação de uma rede *spine and leaf* em que 16 *switches leafs*, cada um com até 48 portas para servidores, se interconectam a 4 *spines* com uma taxa de *oversubscription* relativamente baixa (3:1). Essa arquitetura pode escalar para algo entre 1000 a 2000 [52].

Não obstante a sua complexidade, os *Data Centers* normalmente têm topologia simples e hierárquicas onde uma pequena quantidade de saltos é suficiente para um quadro atravessar toda a rede. E essa topologia normalmente é definida em projeto e possui pouca alteração em comparação a uma WAN.

Dessa forma, um controlador SDN consegue calcular facilmente todos os caminhos que interligam os *switches*, sendo essa quantidade bem menor em relação

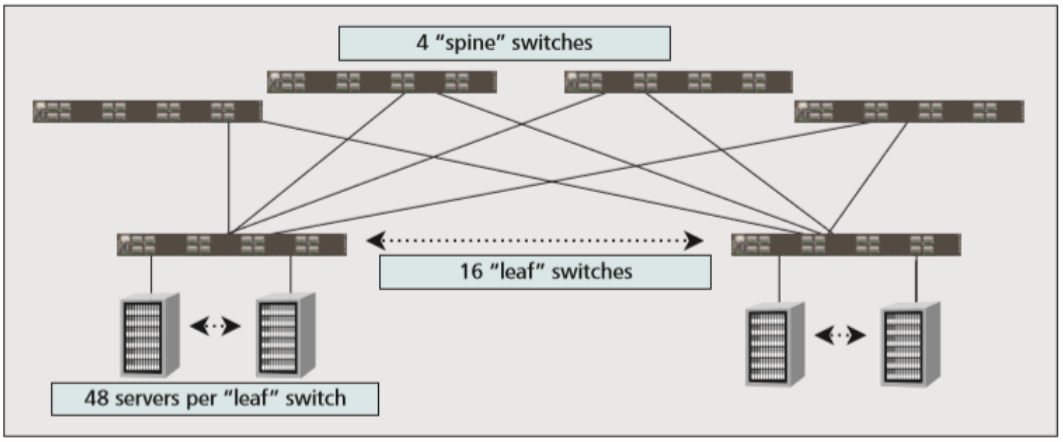

Figura 3.5: *Spine and Leaf*

à quantidade de MACs totais que os equipamentos de núcleo precisam manter. Como há um controle da topologia do *Data Center*, há como limitar em projeto a topologia e consequentemente a quantidade de caminhos totais.

3.2 PathFlow

O PathFlow utiliza como base principal o fato de que o controlador SDN possui uma visão holística da rede, ou seja, conhece todos os elementos de encaminhamento (FE – *Forwarding Elements* - que no nosso caso são *switches* de rede) e ligações entre eles, conseguindo dessa forma criar e manter uma topologia e calcular os menores caminhos (*paths*) de encaminhamento. É importante garantir que os *switches* virtuais também façam parte da rede e sejam gerenciados pelo controlador SDN.

Além da topologia, o PathFlow mapeia em quais *switches* estão ligados todos os *hosts* da rede e relaciona, para cada *switch* origem, qual o caminho (*path*) que deve ser tomado para se alcançar o *switch* destino em que cada *host* está ligado. O *switch* origem então insere um ***tag*** identificador do caminho no pacote. A comutação em todos os *switches* intermediários se dá por esse *tag* até alcançar o *switch* destino, que o retira e entrega ao *host*.

O quadro Ethernet ao chegar ao *switch* SDN, seja físico ou virtual, é alterado e é incluído o rótulo do caminho que ele precisa pegar para chegar ao destino, conforme a Figura 3.6. Dessa forma, cada *switch* origem precisa conhecer apenas o caminho (*path*) para cada um dos *hosts* destinos. Adicionalmente, um *switch* não

precisa conhecer todos os caminhos para todos os *hosts* da rede, apenas aqueles para os quais o *host* origem tem interesse de tráfego. Isso reduz o impacto de escalabilidade à medida que há um grande crescimento do número de *hosts* no *Data Center*. Nos *switches* intermediários, tipicamente *switches* de agregação ou núcleo que possuem poucos *hosts* diretamente ligados, as tabelas de comutação ficam menores, pois recebem do controlador SDN basicamente as informações de comutação dos *paths*.

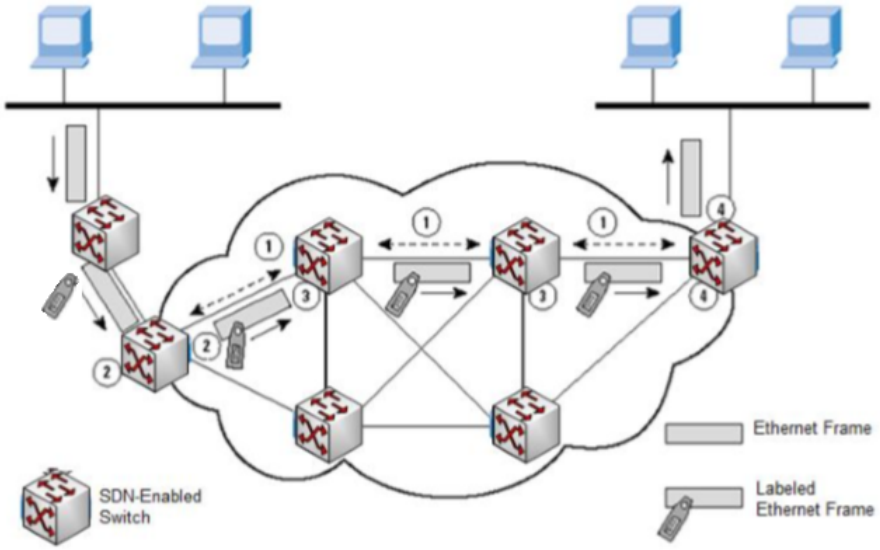

Figura 3.6: Comutação baseada em caminhos

A ideia de se utilizar comutação baseada em caminhos não é em si uma novidade. Essa técnica é a base do MPLS (*Multi Protocol Label Switching*). A diferença principal é que o MPLS possui uma arquitetura distribuída em que as funções são implementadas no mesmo dispositivo onde também há a função de comutação. Por se estar propondo uma solução SDN, as funções de controle serão centralizadas e implementadas pelo controlador SDN. O equivalente à função LDP (*Label Distribution Protocol*) do MPLS, por exemplo, que é responsável pela distribuição e manutenção dos rótulos de comutação, é gerenciado centralmente pelo controlador SDN.

No caso de um *host* ser movido de um *switch* a outro, por movimentação física ou virtual no caso de migração de uma VM, o controlador SDN notifica os demais *switches* da rede para alterarem o caminho para os quais deve comutar o pacote para alcançar novo *switch* destino. Essa alteração se dá simplesmente alterando o *path* para o destino e pode ser acionada tanto quando o controlador SDN é notificado da nova posição como, adicionalmente, recebendo uma mensagem ou

diretiva do hipervisor.

O PathFlow permite, dessa forma, que o host seja alcançado em qualquer ponto da infraestrutura da rede gerenciada pelo sistema independentemente de sua posição. Diferentemente de uma rede tradicional em que as informações das subredes são descentralizadas e estão gravadas nos switches da infraestrutura, o PathFlow gerencia as informações de forma centralizada. Essa sistemática permite a livre movimentação do *host* para qualquer parte da rede, independentemente da subrede a que pertença, sem que seja necessária alteração de IP nem que sejam interrompidas as conexões a esse *host*.

3.2.1 Comutação baseada em caminhos

Está sendo proposta uma **solução SDN de comutação baseada em caminhos**, para a migração de VMs em Data Centers, que ocorre quando há necessidade de migração de VMs entre redes distintas. A Figura 3.7 ilustra a diferença que existe do comportamento de aprendizagem em um *switch* ethernet tradicional e um *switch* de comutação baseada em caminhos.

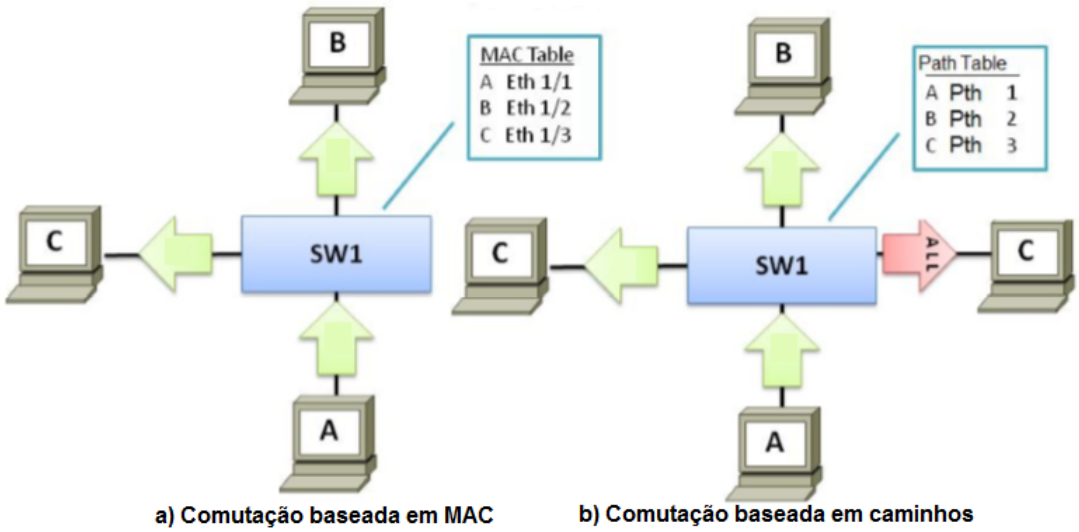

Figura 3.7: Comportamento de aprendizagem

O *switch* Ethernet tradicional (a) cria e mantém a tabela MAC que relaciona para cada MAC conhecido pelo *switch* uma porta de saída. O aprendizado inicial dos MACs se dá por inspeção dos quadros que transpassam o *switch*, que verifica o campo MAC de origem de um tráfego entrante e o relaciona à porta de onde foi recebido esse tráfego. O processo de aprendizagem é realizado por cada *switch*

de forma isolada.

Diferentemente, um *switch* de comutação baseado em caminhos (b) relaciona cada MAC a um caminho que deve ser tomado para comutar o pacote. Em uma abordagem SDN, o processo de aprendizagem é regido centralmente por um controlador SDN que gerencia todos os *switches* da infraestrutura de rede.

Uma vantagem da comutação baseada em caminhos é a possibilidade de agregação de rotas que ocorre quando dois ou mais *hosts* se encontram no mesmo *switch* destino. Desta forma, apenas uma entrada na tabela de comutação em *switches* intermediários é suficiente para comutação dos dados que utilizam o mesmo caminho.

3.2.2 Operação do *Switch*

A Figura 3.8 ilustra a operação do *switch* em uma solução PathFlow. A partir do *start-up* da rede, os *switches* encaminham ao controlador o *status* de suas conexões e dos *hosts* diretamente ligados, para que esse crie o mapeamento de toda a rede. Por outro lado, o controlador envia mensagens para a configuração da tabela de fluxo de cada *switch* para que esse possa descobrir como comutar uma pacote entrante. O controlador cria e mantém um conjunto de tabelas que refletem o estado atual de todos os *switches* e de toda infraestrutura de rede. Na Seção 3.2.3 será detalhada a função de cada tabela e dos processos que as mantém.

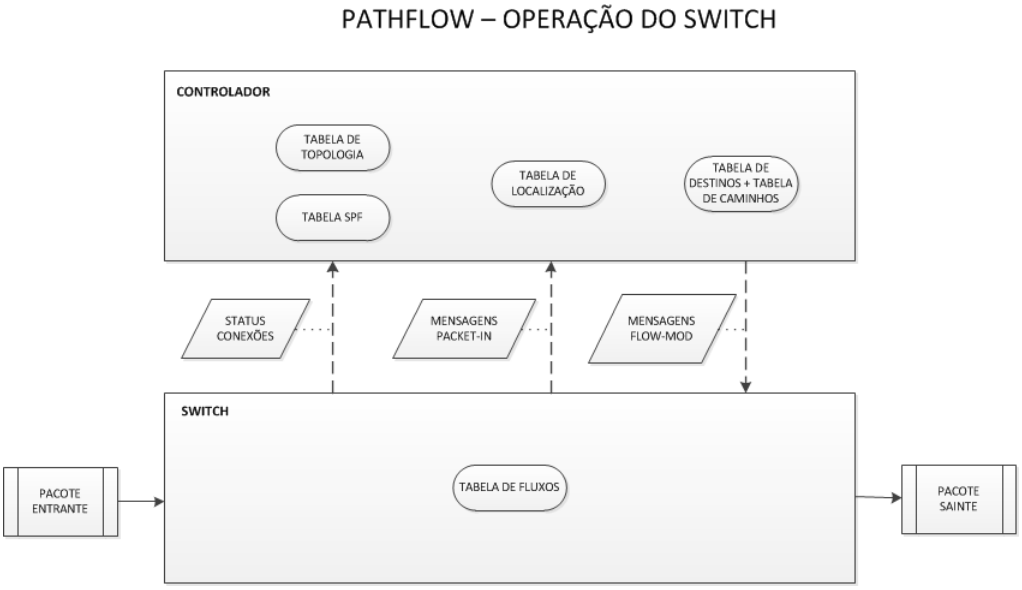

Figura 3.8: Operação do Switch

Em relação a *switch* ethernet tradicional, o aprendizado inicial não é modificado. Os *switches* de borda (ToR), que recebem diretamente os servidores, podem, através da monitoração do tráfego inicial ou através de ARP gratuitos, conhecer quais MACs estão diretamente ligados em suas portas. A diferença é que, em se tratando de uma infraestrutura SDN, essa informação deve ser repassada ao controlador SDN para que esse atualize o mapeamento da rede. Disso, segue a operação inicial dos *switches*:

1. O controlador SDN, no *start-up* dos *switches*, monta uma tabela de topologia (*topology table*) dos *switches* e calcula os caminhos (*paths*) para que todos os *switches* possam alcançar todos os demais.
   (a) Os caminhos são calculados através de algoritmo de *shortest path first* (SPF), baseado na velocidade dos *links* ou quantidade de saltos;
   (b) Os *paths* são unidirecionais; e
   (c) Aos *paths* são atribuídos ***tags***, que pode ser de 6 ou 12 bits a depender da implementação.
2. Os *switches*, por sua vez, começam a montar sua tabela de destinos (*destination table*) relacionando os seus MACs locais – máquinas diretamente ligadas – às suas portas de saída.
   (a) A descoberta dos *hosts* locais pode ocorrer tanto com a verificação do tráfego entrante nas portas como pela emissão de ARPs gratuitos (gratuitous ARP); e
   (b) Todos os *switches* devem informar ao controlador SDN o conteúdo dessa tabela para que esse crie a tabela de localização (*localization table*) geral da rede, relacionando em qual *switch* e porta estão associados todos os MACs da rede.
3. Quaisquer alterações na tabela de destinos precisa ser notificada ao controlador SDN para que esse atualize a tabela de localização.
4. Pode haver na tabela de destinos mais de um caminho para o mesmo destino. Nesse caso, são instaladas as múltiplas entradas e é habilitado o balanceamento de carga.
5. Concomitamente, o controlador SDN informa aos *switches* os caminhos (*paths*) para os demais *switches*. Os *switches* criam uma tabela de caminhos (*path table*) associando os caminhos com a porta de saída.

(a) Em algumas implementações, *destination table* e *path table* podem ser unificadas.

Após a operação inicial, segue o procedimento em caso de migração de algum host. Primeiramente, existem duas formas de atualização da localização do *host*: autônoma e *triggered*:

1. A atualização autônoma ocorre quando o controlador SDN "percebe" a alteração de um *host* devido à atualização da *localization table* que é informada por outro *switch*. Nesse caso:
   (a) O controlador SDN atualiza sua *localization table*; e
   (b) Envia uma atualização para que todos os *switches* façam a troca em sua *destination table* e *path table* para o novo caminho, o que faz a comutação ser instantaneamente alterada.
2. A atualização *triggered*, diferentemente da autônoma, ocorre quando o controlador é avisado de uma mudança - por exemplo, após ter sido recebida uma diretiva do hipervisor. Todos os demais procedimentos são iguais aos da atualização autônoma.

Essa proposta permite atender o requisito principal de migração de *hosts* entre redes distintas, mantendo-se o endereço IP e as conexões a ele. Ao mesmo tempo, propicia uma comutação que permita escalabilidade da rede.

3.2.3 Arquitetura do PathFlow

Como informado anteriormente, o controlador é parte essencial nas redes definidas por *software*. São os controladores que definem a lógica de encaminhamento através das regras que irão configurar nos *switches* e, dessa forma, indicam o comportamento de toda a rede. A arquitetura do PathFlow está detalhada no diagrama em blocos ilustrado na Figura 3.9.

O controlador PathFlow possui cinco processos e mantém internamente cinco tabelas. A Tabela 3.1 detalha suas características, assim como entradas e saídas de cada processo. Os processos indicados com * foram baseados no código *l2_multi.py* [53].

3.3 Implementação do PathFlow

O PathFlow v1.0 foi desenvolvido em Python como componente da controladora POX (Seção 2.5.8). O PathFlow v1.0 é compatível com a versão 1.0 do Openflow assim como a própria controladora POX.

O PathFlow possui estrutura modular, como ilustrado na Seção 3.2.3, sendo dividido em processos possuindo entradas e saídas bem definidas. Essa arquitetura possibilita maior flexibilidade no caso de evolução da solução, pois um processo específico poderá ser substituído ou atualizado sem prejuízo aos demais.

O PathFlow utiliza outros componentes da controladora POX. O componente **openflow.discovery** é utilizado pelo processo **Criar e manter tabela de topologia**, para o recebimento de mensagens LLDP dos *switches* da infraestrutura e criação da tabela de topologia. Esse processo também utiliza o componente **openflow.spanning_tree** para contrução de árvore spanning-tree livre de *loops*. O componente **pox.lib.revent** é utilizado por vários processos para "escutar"eventos dos *switches*, como **Packet-In**, **PortStatus** e **ConnectionUp**.

O código-fonte do PathFlow v1.0 é parte integrante desta dissertação. Está integralmente disponibilizado no Anexo A.

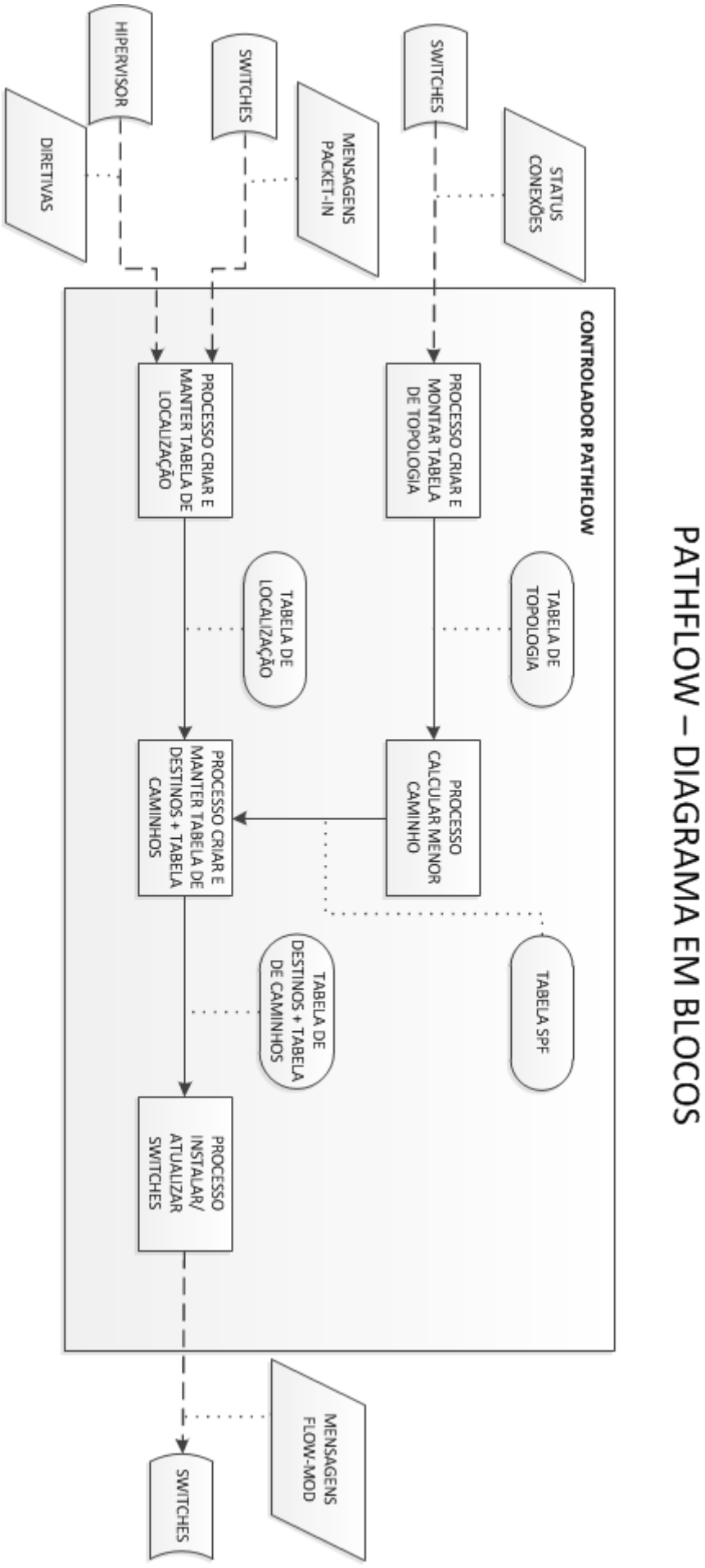

Figura 3.9: Diagrama em blocos do PathFlow

Tabela 3.1: Detalhamento dos processos do PathFlow

PathFlow - Detalhamento dos Processos			
Processo	Detalhamento	Entrada	Saída
Criar e manter tabela de topologia *	Através de informações LLDP consegue criar um grafo de rede detalhando através de quais portas os *switches* estão interligados	Status de conexões do *switch*	Tabela de topologia
Calcular menor caminho *	Para cada *switch*, calcula o menor caminho (*path*) até os demais destinos através de um algoritmo de menor caminho primeiro (SPF)	Tabela de topologia	Tabela SPF
Criar e manter tabela de localização *	Cria a tabela de localização de todos os dispositivos de rede associando *switch* e porta - opcionalmente pode receber diretivas de hipervisores	Mensagens Packet-in + diretivas do hipervisor (optativo)	Tabela de localização
Criar e manter tabela de destinos + tabela de caminhos	Recebe a tabela SPF e tabela de localização e cria as tabelas de destino e de caminhos	Tabela SPF + tabela de localização	Tabela de destinos + tabela de caminhos
Instalar/atualizar *switches*	Recebe as tabelas de destino e de caminhos e gera mensagens de configuração das tabelas do *switch*	Tabela de destinos + tabela de caminhos	Mensagens de modificação de fluxo (flow-mod)

4. Análise Experimental

4.1 Avaliação da Solução

O PathFlow foi avaliado em duas etapas: inicialmente foi verificada a efetividade do sistema em permitir uma migração de *hosts* entre redes distintas sem troca de IP ou perda de conexões. Posteriormente, foi avaliada a eficácia do sistema analisando o parâmetro **tamanho de tabelas de comutação** ao realizar um aumento gradual da quantidade de *hosts* suportados, analisando assim a escalabilidade do sistema. O PathFlow foi comparado com um *Data Center* convencional baseado em TCP/IP e a uma implementação de SDN tradicional.

4.2 Ambiente de emulação

O PathFlow foi avaliado em um ambiente emulado através da ferramenta Mininet [54]. O Mininet foi o primeiro sistema de emulação que propiciou uma forma simples de avaliação de protocolos e aplicações SDN. O Mininet cria uma rede virtual, utiliza *kernel* real de estações, *switches* virtuais e códigos de aplicação, podendo rodar em um servidor físico ou máquina virtual. Um dos pontos principais do Mininet é a prototipação de *switches* Openflow virtuais, provendo a mesma semântica de *switches* baseados em *hardware*. O Mininet suporta o *Open vSwitch* (OVS) [55], que implementa o plano de dados no *kernel* e o plano de controle como processo de trabalho em *software*.

O ambiente de emulado foi executado em um *laptop* com sistema operacional Windows®8 e processador Intel®i7 64 bits com 8,00 GB de RAM. Acima do sistema operacional foi instalado um VMWare®Workstation 10 em que foram criadas duas máquinas virtuais com Ubuntu 14.04 LTS: a primeira para execução do Mininet 2.2.1 associado ao OVS 2.3.1 e outra para execução exclusiva do con-

trolador Pox 0.2.0.

4.3 Etapa de avaliação da efetividade do sistema

A primeira etapa foi analisar a funcionalidade de migração do sistema. Os testes de mobilidade foram realizados fazendo a desconexão de estações virtuais no Mininet, reproduzindo o equivalente a um *handoff* (Seção 2.4) de *hosts* ligados aos *switches* OVS. Os testes de migração de VMs são aproximações que envolvem um *handoff*, dado que esse envolve um conjunto de troca de sinalizações entre estado anterior e posterior para que as conexões se mantenham após a migração.

No momento do *handoff*, o *switch* origem envia ao controlador uma mensagem de *portstatus-event.deleted*, e o *switch* destino envia mensagem de *portstatus-event.added*. O PathFlow atua no sentido de alterar as tabelas de localização, de destinos e caminho, e cria novos *paths* para que o *host* móvel possa ser alcançado pelos demais (e vice-versa). Esse teste corresponde a uma migração autônoma (Seção 3.2.2), que ocorre quando o controlador ”percebe”uma movimentação através da monitoração de sua infraestrutura sem participação de uma entidade externa.

Para os testes foi utilizada a topologia constante na Figura 4.1. Trata-se de uma topologia com um núcleo (Switch3) e três *switches* como acesso (Switch1, Switch2 e Switch4).

Inicialmente foi testado o recurso de *handoff* com a utilização de *hosts* em uma mesma sub-rede. Na Tabela 4.1 está ilustrada a tabela de localização antes da migração. A tabela de localização é única e interna ao controlador. Relaciona para cada MAC conhecido uma tupla Switch/Porta. O PathFlow monta essa tabela à partir da monitoração dos fluxos recebidos através de mensagens **Packet-In**.

As Tabelas 4.2 e 4.3 mostram a integração da tabela de destinos com a tabela de caminhos em um típico *switch* de acesso com *hosts* diretamente ligados às suas portas. Essas tabelas são estratificadas por *switch* e indicam qual deve ser a decisão de encaminhamento do *switch*. A partir dessa tabela, o PathFlow encaminha mensagens de **Flow Mod** do **OF** para que o *switch* propriamente dito crie sua tabela de fluxos. Nota-se que, nos casos em que um *host* está diretamente ligado ao *switch*, a comutação é direcionada à porta de saída sem adição de nenhum *tag*, e, caso seja recebido um pacote com *tag* de outro *switch*, esse deve ser retirado.

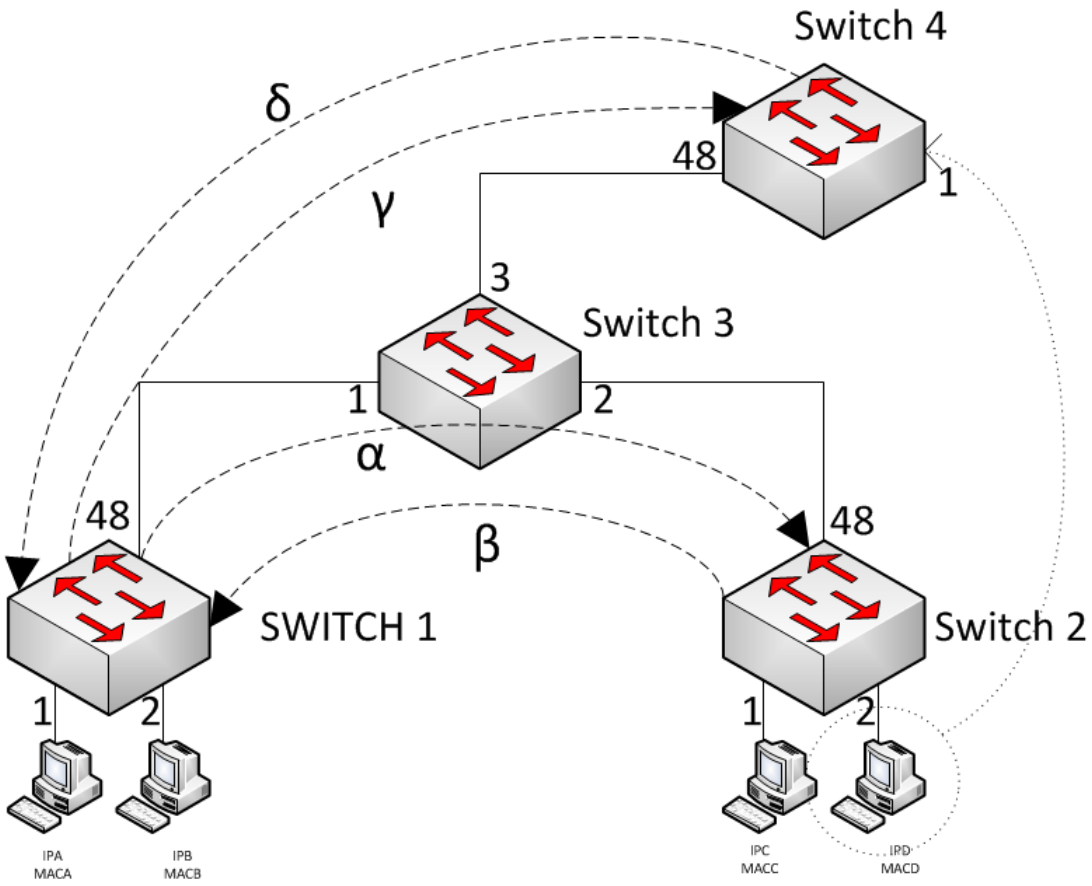

Figura 4.1: Avaliação da eficácia do sistema

Nos casos de *hosts* externos, a comutação é realizada encaminhando para porta de saída - nesse caso, trata-se do *uplink* do *switch* - adicionando o *tag* indicado.

A tabela de destinos e caminhos do SWITCH3 é ilustrada na Tabela 4.4. Um *switch* de camada de núcleo ou agregação, cuja principal função é a interligação de *switches*, tipicamente não possui *hosts* diretamente ligados. A comutação é efetuada pela inspeção do *tag* do pacote recebido, que o mantém e comuta à porta de saída. É nessa camada que a agregação de caminhos se torna importante, à medida que um mesmo *path* pode ser utilizado para alcançar múltiplos *hosts* que estejam ligados ao mesmo *switch*.

Um adendo importante é sobre os *tags* que aqui estão genericamente indicados com as letras gregas α e β. A versão atual do PathFlow utiliza o campo **TOS** do IP por conta de limitações da versão do OpenFlow 1.0.0, que serão detalhadas na Seção 4.5.

Será indicado o comportamento depois da migração. O *hand-off* ocorreu com o *host* **D** que migrou do Switch2 para o Switch4. Foi utilizado o *script* **mo-**

Tabela 4.1: Tabela de localização - antes da migração

Tabela de Localização		
MAC	SWITCH	Porta
MacA	Switch1	Port1
MacB	Switch1	Port2
MacC	Switch2	Port1
MacD	Switch2	Port2

Tabela 4.2: Tabela de Destinos + Tabela de Caminhos - Switch 1 (antes)

Tabela de Destinos + Tabela de Caminhos - Switch 1		
Destino	Porta de Saída	Tag
MacA	Port1	-
MacB	Port2	-
MacC	Port48	α
MacD	Port48	α

bility.py, pertencente ao pacote Mininet, para esse teste. Esse *script* utiliza extensões do *switch* OVS para desassociar/associar (*detach/attach*) *hosts* às suas portas. A migração ocorreu no caso autônomo, em que o Controlador "percebe"a migração através de inspeção de mensagem de **PortStatus-event.deleted** e **Portstatus-event.added** vindas de um *host*, nesse caso, originalmente associado ao Switch3 e que "aparece"no Switch4.

O PathFlow atualiza sua tabela de localização, que fica de acordo com a Tabela 4.5, comunicando a **todos** os *switches* da infraestrutura que apaguem seus registros sobre o MAC de destino (MACD). Isso é realizado com o envio de mensagens OF **flow-mod** aos *switches*. E, após algum tráfego direcionado a esse *host*, novos caminhos (*paths*) são criados. A tabelas de destinos e caminhos dos *switches* fica de acordo com as Tabelas 4.6, 4.7 e 4.8. Apenas foram enfatizados os caminhos entre o Switch1 e o Switch2, assim como entre o Switch1 e Switch4.

O teste foi realizado efetuando-se um **ping** contínuo com intervalo de 0,01 segundos de todas as máquinas para a estação móvel, assim como da estação móvel para outra estação. A migração ocorreu tendo sido perdidos cinco pacotes, o que significa que a **migração ocorreu em apenas 0,05 segundos**, aproximadamente. Um dos pontos que pode ser destacado é que, inicialmente, poderiam ser apagados os registros em relação ao MACD apenas nos *switches* afetados na migração, porém os testes demostraram haver um menor tempo de convergência ao se apagar seus registros em todos os *switches* da infraestrutura.

O PathFlow possibilita a livre movimentação de uma máquina virtual para

Tabela 4.3: Tabela de Destinos + Tabela de Caminhos - Switch 2 (antes)

Tabela de Destinos + Tabela de Caminhos - Switch 2		
Destino	Porta de Saída	Tag
MacA	Port48	β
MacB	Port48	β
MacC	Port1	-
MacD	Port2	-

Tabela 4.4: Tabela de Destinos + Tabela de Caminhos - Switch 3 (antes)

Tabela de Destinos + Tabela de Caminhos - Switch 3		
Destino	Porta de Saída	Tag
α	Port2	α
β	Port1	β

quaisquer pontos da rede gerenciada pelo sistema. Isso é possível porque as informações que possibilitam a segregação em sub-redes - tipicamente VLANs em DCs TCP/IP - não estão registradas de forma autônoma e pulverizada nos *switches* da infraestrutura e sim centralizados no controlador SDN.

A segregação lógica L2 é prevista no PathFlow ao se adicionar a informação de sub-rede na tabela de localização. Dessa forma, caminhos (*paths*) para comunicações em unicast somente seriam criados caso os dois *hosts* estejam na mesma sub-rede. Por questões de compatibilidade, poderiam ser adicionadas o tag VLAN ou VXLAN, conforme Tabela 4.9. Nesse caso, um roteador seria necessário para a comutação entre as duas sub-redes, precisando possuir uma interface (ou sub-interface) pertencente a cada uma das sub-redes. A comunicação em *broadcast* também é restrita fazendo com que essa somente ocorra em portas pertencentes à mesma sub-rede.

Uma das principais diferenças do PathFlow em relação ao VICTOR [9], nesse ponto, é que esse utiliza tabelas de localização relacionando a tupla Switch+Porta com o IP, ao invés de MAC como o PathFlow. Uma vantagem da utilização do MAC é que a descoberta pode ocorrer sem precisar escalonar ao nível de IP. Além disso, não seria possível descobrir o IP em comunicações que não fossem relacionados a esse protocolo.

Tabela 4.5: Tabela de localização - depois da migração

Tabela de Localização		
MAC	SWITCH	Porta
MacA	Switch1	Port1
MacB	Switch1	Port2
MacC	Switch2	Port1
MacD	**Switch4**	**Port1**

Tabela 4.6: Tabela de Destinos + Tabela de Caminhos - Switch 1 (depois)

Tabela de Destinos + Tabela de Caminhos - Switch 1		
Destino	Porta de Saída	Tag
MacA	Port1	-
MacB	Port2	-
MacC	Port48	α
MacD	**Port48**	γ

4.4 Etapa de avaliação da escalabilidade do sistema

Um dos principais diferenciais do PathFlow é a capacidade de agregar caminhos, para que dois ou mais *hosts* que tenham interesse de tráfego possam utilizar um mesmo *path* para o destino, fazendo com que haja uma economia na quantidade de entradas nos *switches* de core/distribuição que podem realizar o encaminhamento baseado somente nos *tags* do fluxo.

Na etapa da avaliação descrita adiante, para uma mesma topologia será registrada a quantidade de entradas na tabela de comutação para três diferentes arquiteturas: um *Data Center* convencional TCP/IP baseado em comutação de pacotes; um *Data Center* SDN tradicional baseado na comutação por fluxos, de acordo com a arquitetura de Victor [9] e o PathFlow. É necessário ser destacado que, enquanto um *Data Center* convencional realiza sua comutação baseada nas entradas em sua base de informações de encaminhamento (*Forwarding Information Base* - FIB), os *Data Centers* SDN e PathFlow fazem a comutação baseada na quantidade de entradas em suas tabelas de fluxo (*Flow Table*), e o que se deseja é analisar a quantidade de entradas seja em FIB ou *Flow Table*. Para o devido entendimento e unificação de nomenclatura, a expressão quantidade de fluxos representará a quantidade de entradas, seja em uma base de informações de encaminhamento ou em uma tabela de fluxos.

Foi utilizada a arquitetura *spine and leaf* (Seção 3.1) para avaliação das tabelas de comutação. Foram utilizados dois *switches* na função de *spine* e quatro *switches*

Tabela 4.7: Tabela de Destinos + Tabela de Caminhos - Switch 4 (depois)

Tabela de Destinos + Tabela de Caminhos - Switch 4		
Destino	Porta de Saída	Tag
MacA	Port48	δ
MacB	Port48	δ
MacD	**Port1**	**-**

Tabela 4.8: Tabela de Destinos + Tabela de Caminhos - Switch 3 (depois)

Tabela de Destinos + Tabela de Caminhos - Switch 3		
Destino	Porta de Saída	Tag
α	Port2	α
β	Port1	β
γ	**Port3**	γ
δ	**Port1**	δ

na função *leaf*, interconectados conforme a Figura 4.2.

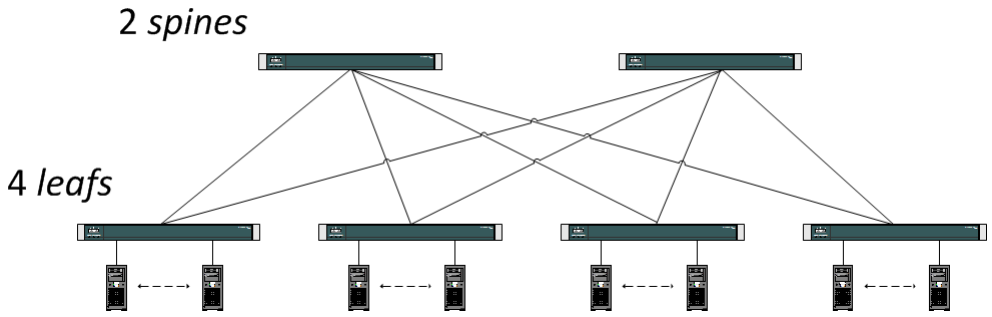

Figura 4.2: Topologia Spine and Leaf utilizada na avaliação de tamanho de tabelas

Os testes foram executados alterando-se a quantidade de *hosts* por *leaf*, variando de um *hosts/leaf* (portanto, quatro *hosts* ao total) até 48 *hosts/leaf* (total de 192 *hosts*). Foi utilizada uma alteração do *script* **mobility.py** para permitir alterar a quantidade de *hosts*, e executado o comando **pingall** do Mininet, que faz com que todos os *hosts* realizem **ping** a todos os demais. Foi registrada a quantidade de entradas totais do sistema e também estratificado por camada, sendo a camada de Núcleo equivalente aos dois *switches spines* e a camada de acesso equivalente aos quatro *switches leafs*. Foram desconsideradas as entradas de fluxo geradas por IPs de gerência do sistema, somente foram consideradas as entradas de comunicação entre os *hosts*.

Na Tabela 4.10, assim como no gráfico da Figura 4.3, estão registrados os dados obtidos para a quantidade de fluxos gerais do sistema. Já era esperada uma grande diferença de uma solução SDN para uma solução de *Data Center* conven-

Tabela 4.9: Tabela de localização - Com suporte a VLAN ou VXLAN

Tabela de Localização			
MAC	SWITCH	Porta	Sub-Rede
MacA	Switch1	Port1	VLAN1
MacB	Switch1	Port2	VXLAN2
MacC	Switch2	Port1	VLAN1
MacD	Switch2	Port2	VXLAN2

cional baseada em TCP/IP, pois uma solução SDN tradicional gera uma entrada para cada fluxo de comunicação - que nesse caso se restringiu a um simples **ping** - ocasionando um crescimento praticamente exponencial. Porém, o PathFlow consegue diminuir a quantidade total de fluxos por conta da agregação de caminhos que faz com que, a partir de 16 *hosts*, já se consegue ter uma quantidade menor de fluxos se comparado ao *Data Center* TCP/IP, que possui um crescimento praticamente linear em relação à quantidade de *hosts*.

Tabela 4.10: Fluxos Totais por número de *hosts*

Fluxos Totais			
Qtd. *hosts*	*Data Center* TCP/IP	SDN	PathFlow
4	20	36	28
8	40	152	44
16	80	624	76
32	150	2528	140
64	320	12226	268
128	640	40832	524
192	950	110400	780

Fazendo-se uma análise por camada, no Núcleo da infraestrutura, explicitado na Tabela (4.11) e no gráfico da Figura 4.4, também é evidente o comportamento exponencial do SDN. Da mesma forma, fica evidente a agregação do PathFlow, que possui uma quantidade de fluxos fixa baseada apenas na topologia da rede. Enquanto isso, é evidenciada a linearidade do *Data Center* TCP/IP, cuja tabela depende apenas da quantidade total de *hosts*.

Em relação à camada de Acesso, ilustrada na Tabela 4.12 e no gráfico da Figura 4.5, verifica-se a relação linear do número de fluxos à quantidade de *hosts* da infraestrutura, comparando-se o PathFlow ao *Data Center* TCP/IP.

Analisando-se os dados obtidos, é notado que, embora o PathFlow seja uma solução SDN e realize suas comutações baseado em fluxos, não acompanha o crescimento exponencial de uma solução SDN tradicional e possui sua escalabilidade

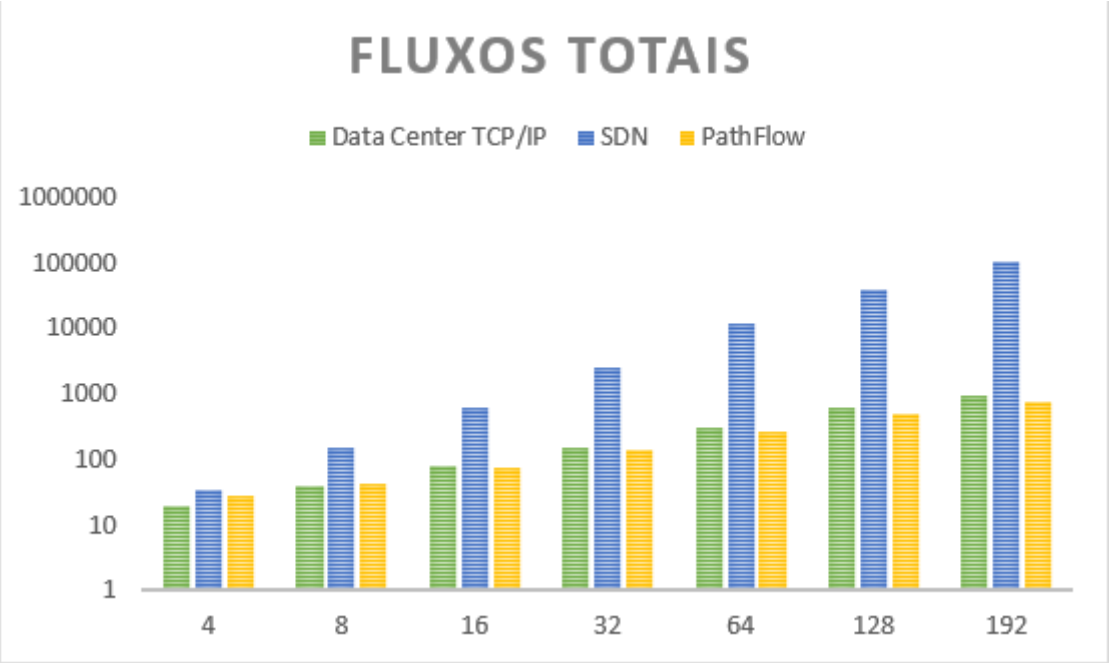

Figura 4.3: Fluxos Totais

Tabela 4.11: Fluxos Totais no Núcleo por número de *hosts*

Fluxos Totais - Núcleo			
Qtd. *hosts*	*Data Center* Tradicional	SDN Tradicional	PathFlow
4	4	12	12
8	8	48	12
16	16	192	12
32	32	768	12
64	64	2562	12
128	128	12288	12
192	192	23068	12

compatível com a de um *Data Center* tradicional TCP/IP.

4.5 Limitações da implementação

O PathFlow foi desenvolvido para que sua instalação fosse transparente em uma infraestrutura SDN *ready*, em que todos os *switches* suportam o protocolo Openflow (OF). Não há necessidade de alterações no *firmware* dos equipamentos da infraestrutura de rede.

O PathFlow v.1.0 é um complemento do controlador POX e esse, por sua vez, somente suporta o Openflow 1.0.0 (atualmente). A versão 1.0.0 do OpenFlow

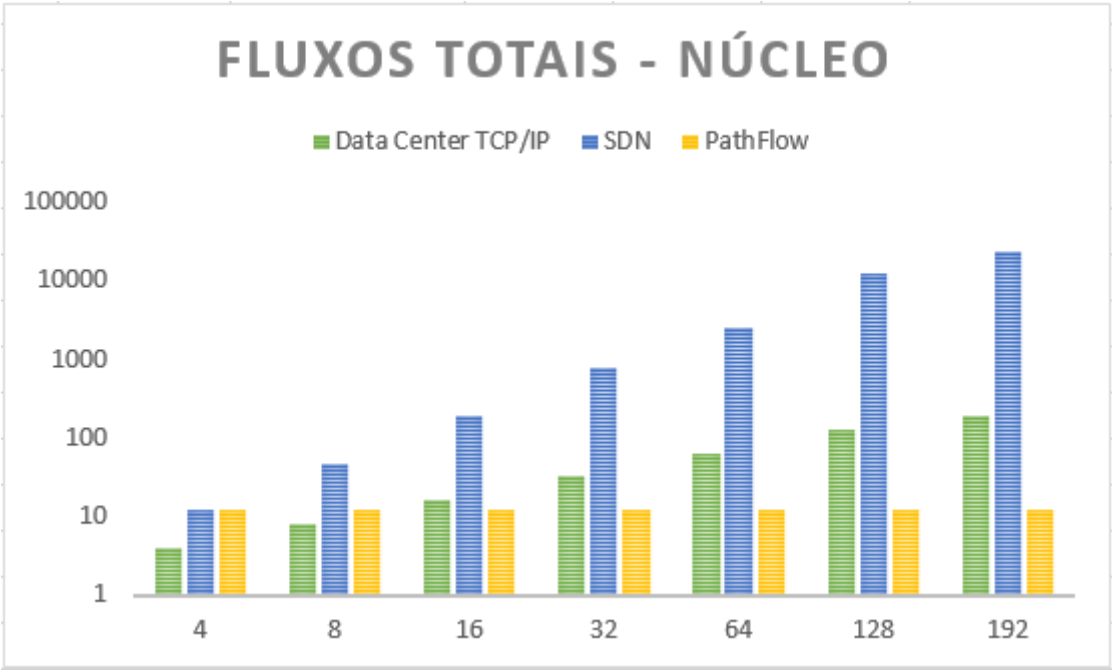

Figura 4.4: Fluxos Totais - Núcleo

Tabela 4.12: Fluxos Totais no Acesso por número de *hosts*

Fluxos Totais - Acesso			
Qtd. *hosts*	*Data Center* TCP/IP	SDN	PathFlow
4	16	24	16
8	32	104	32
16	64	432	64
32	128	1760	128
64	256	9664	256
128	512	28544	512
192	768	87332	768

é atualmente a versão mais difundida e implementada, porém possui algumas limitações:

- **O OF 1.0.0 não suporta MPLS**. A utilização de ***tags*** MPLS seria uma escolha adequada para a implementação dos *tags* do PathFlow. Por conta disso, foi utilizado o campo *Type of Service* (TOS) do IP como alternativa. Em contraposição ao *tag* MPLS que possui 20 bits, o campo TOS somente possui 8 sendo 2 reservados, restando **64 possibilidades** para serem utilizadas como indicação **de *paths* da rede**.

  Poderia também ser desenvolvida uma compatibilização para que o PathFlow pudesse aceitar *tags* MPLS vindos da WAN, assim como o contrário, o

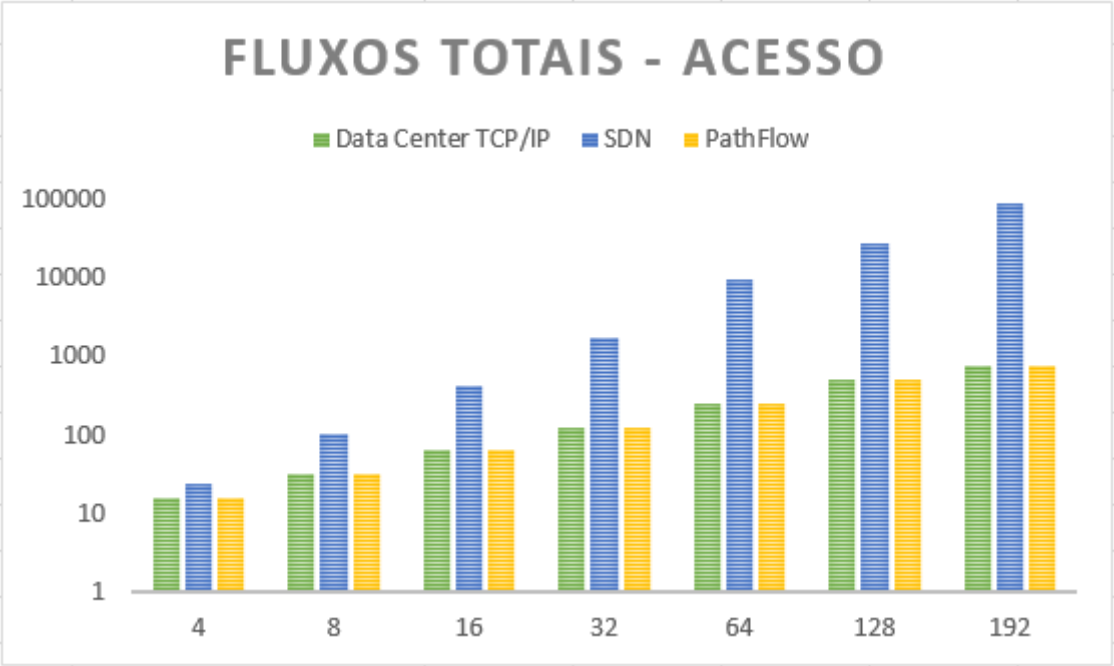

Figura 4.5: Fluxos Totais - Acesso

MPLS transportar um *tag* PathFlow de volta à WAN, permitindo principalmente que o PathFlow pudesse interagir com outro controlador PathFlow em uma rede remota.

O MPLS somente é suportado pelo OF a partir da versão 1.1.0.

- **O OF 1.0.0 não verifica correspondência em múltiplas tabelas**. Um dos comportamentos indesejados do PathFlow é que podem ser criados fluxos nos *switches* intermediários caso um pacote *en-passant* chegue nesse *switch* sem ter um *tag* definido. O *switch* consulta o controlador que instala um caminho para ele. Isso cria uma duplicidade de informações, pois, para um mesmo fluxo, existirão dois caminhos criados, um com *tag* e outro sem.

  Uma das soluções para esse caso é que o controlador criasse pelo menos dois níveis de tabelas, um mais prioritário em que seriam criados os *paths* locais - para os *hosts* diretamente ligados ao *switch* - e outro em que ficariam os fluxos com *tag*, que somente seria consultado caso não houvesse correspondência na tabela prioritária. Isso faria que, em caso de instalação em duplicidade, um dos fluxos seria expirado pelo tempo de **FLOW_IDLE_TIMEOUT**.

  O recurso de um fluxo de pacotes, ao entrar no *switch* OpenFlow, poder passar por várias tabelas de fluxos, para que diversas ações diferentes sejam realizadas, somente está disponível a partir da versão 1.1.0 do OF.

É preciso ressaltar que o suporte ao OF 1.0.0 é uma importante característica do PathFlow, dado que essa é, ainda hoje, a versão mais difundida do Openflow. As limitações que foram observadas não se referem à ideia principal do sistema e sim à implementação de sua versão 1.0, que está associada ao OF 1.0.0. A eliminação desta limitação pode ser equacionada numa futura atualização da solução para versões superiores do Openflow.

5. Conclusão e Trabalhos Futuros

Este trabalho endereça o problema de migração *on-line* de VMs entre redes distintas em *Data Centers* tomando como base recentes trabalhos de virtualização de redes. Propõe-se para tal uma **solução SDN de comutação baseada em caminhos** chamado **PathFlow**.

O PathFlow utiliza como base principal o fato de que o controlador SDN possui uma visão holística da rede: conhece todos os dispositivos de encaminhamento (FE – *Forwarding Elements* - que no nosso caso são *switches* de rede) e ligações entre eles, conseguindo dessa forma criar e manter uma topologia e calcular os menores caminhos (***paths***) de encaminhamento.

O PathFlow foi avaliado em duas etapas: inicialmente foi verificada a efetividade do sistema em permitir uma migração de *hosts* entre redes distintas sem troca de IP ou perda de conexões. O teste foi realizado efetuando-se um **ping** contínuo com intervalo de 0,01 segundos de todas as máquinas para a estação móvel, assim como da estação móvel para outra estação. A migração ocorreu tendo sido perdidos cinco pacotes, o que significa que a **migração ocorreu em apenas 0,05 segundos**. A migração ocorreu no caso autônomo, em que o Controlador "percebe"a migração através de inspeção de mensagem de **PortStatus-event.deleted** e **Portstatus-event.added**.

Posteriormente, foi avaliada a eficácia do sistema analisando o parâmetro **tamanho de tabelas de comutação** ao realizar um aumento gradual da quantidade de *hosts* suportados, analisando assim a escalabilidade do sistema. O PathFlow foi comparado com um *Data Center* convencional baseado em TCP/IP e a uma implementação de SDN tradicional. Foi utilizada a arquitetura *spine and leaf* (Seção 3.1) para avaliação das tabelas de comutação.

O PathFlow possibilita a livre movimentação de uma máquina virtual para

quaisquer pontos da rede gerenciada pelo sistema. Isso é possível porque as informações que possibilitam a segregação em sub-redes não estão registradas de forma autônoma e pulverizada nos *switches* da infraestrtura e sim centralizados no controlador SDN.

O PathFlow possui a capacidade de agregar caminhos em um mesmo *path*, permitindo dessa forma que o crescimento das tabelas de comutação seja linear com o crescimento de *hosts* na camada de acesso, e dependente apenas da topologia do *Data Center* no núcleo. Portanto, embora o PathFlow seja uma solução SDN e realize suas comutações baseado em fluxos, não acompanha o crescimento exponencial de uma solução SDN tradicional e possui sua escalabilidade compatível com a de um *Data Center* tradicional TCP/IP.

Como sugestões de trabalhos futuros, indica-se a utilização de controladoras com suporte ao Openflow 1.1.0 ou acima, a partir do qual podem ser implementadas **tags** de maior capacidade compatíveis com o MPLS, assim como suporte a correspondência em múltiplas tabelas para evitar a duplicidade de entradas de comutação.

Referências Bibliográficas

[1] BARI, M. et al. Data center network virtualization: A survey. *Communications Surveys Tutorials, IEEE*, v. 15, n. 2, p. 909–928, Second 2013. ISSN 1553-877X.

[2] AMAZON. *Amazon EC2*. 2015. `http://aws.amazon.com/ec2/`. [Online; acessado em 01-julho-2015].

[3] PERKINS, C. Ietf rfc 3344: Ip mobility support for ipv4. *http://www. ietf. org/rfc/rfc3344. txt*, 2002.

[4] SRIDHAR, T. et al. Vxlan: a framework for overlaying virtualized layer 2 networks over layer 3 networks. *draft-mahalingamdutt-dcops-vxlan-04*, 2013.

[5] SRIDHARAN, M. et al. Nvgre: Network virtualization using generic routing encapsulation. *IETF draft*, 2011.

[6] FARINACCI, D. et al. *LISP Control Plane for Network Virtualization Overlays*. 2013. Online; acessado em 15-julho-2015. Disponível em: <`https://tools.ietf.org/html/draft-maino-nvo3-lisp-cp-03`>.

[7] CHOWDHURY, N. M. K.; BOUTABA, R. A survey of network virtualization. *Computer Networks*, v. 54, n. 5, p. 862 – 876, 2010. ISSN 1389-1286. Disponível em: <http://www.sciencedirect.com/science/article/pii/S1389128609003387>.

[8] MCKEOWN, N. *How SDN will shape networking*. 2011. Online; acessado em 22-fevereiro-2015. Disponível em: <`https://www.youtube.com/watch?v=c9-K5O_qYgA`>.

[9] HAO, F. et al. Enhancing dynamic cloud-based services using network virtualization. In: *Proceedings of the 1st ACM Workshop on Virtualized Infrastructure Systems and Architectures*. New York, NY, USA: ACM,

2009. (VISA '09), p. 37–44. ISBN 978-1-60558-595-6. Disponível em: <http://doi.acm.org/10.1145/1592648.1592655>.

[10] ARMBRUST, M. et al. A view of cloud computing. *Commun. ACM*, ACM, New York, NY, USA, v. 53, n. 4, p. 50–58, abr. 2010. ISSN 0001-0782. Disponível em: <http://doi.acm.org/10.1145/1721654.1721672>.

[11] VMWARE. *VMware*. 2015. `http://www.vmware.com/`. [Online; acessado em 12-julho-2015].

[12] XEN. *XEN*. 2015. `http://www.xenproject.org/`. [Online; acessado em 15-julho-2015].

[13] BARHAM, P. et al. Xen and the art of virtualization. *SIGOPS Oper. Syst. Rev.*, ACM, New York, NY, USA, v. 37, n. 5, p. 164–177, out. 2003. ISSN 0163-5980. Disponível em: <http://doi.acm.org/10.1145/1165389.945462>.

[14] SILVERA, E. et al. Ip mobility to support live migration of virtual machines across subnets. In: *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*. New York, NY, USA: ACM, 2009. (SYSTOR '09), p. 13:1–13:10. ISBN 978-1-60558-623-6. Disponível em: <http://doi.acm.org/10.1145/1534530.1534548>.

[15] GUSTAFSSON, E. Mobile ip regional registration. *Internet draft*, IETF, 2001. Disponível em: <http://ci.nii.ac.jp/naid/10010195843/en/>.

[16] MALKI, K. E. Ietf rfc 4881: Low-latency handoffs in mobile ipv4. *https://tools.ietf.org/html/rfc4881*, 2007.

[17] KOODLI, R. Ietf rfc 4068: Fast handovers for mobile ipv6. *https://www.ietf.org/rfc/rfc4068.txt*, 2005.

[18] SHARMA, S.; ZHU, N.; CHIUEH, T. cker. Low-latency mobile ip handoff for infrastructure-mode wireless lans. *Selected Areas in Communications, IEEE Journal on*, v. 22, n. 4, p. 643–652, May 2004. ISSN 0733-8716.

[19] JOHNSON, D. Ietf rfc 3775: Mobility support in ipv6. *https://www.ietf.org/rfc/rfc3775.txt*, 2004.

[20] STEWART, R. Ietf rfc 4960: Stream control transmission protocol. *https://tools.ietf.org/html/rfc4960*, 2007.

[21] MATSUMOTO, A. et al. Tcp multi-home options. *draft-arifumi-tcp-mh-00. txt*, *IETF Internet draft*, 2003.

[22] FLOYD, S.; HANDLEY, M.; KOHLER, E. Ietf rfc 4340: Datagram congestion control protocol (dccp). *https://tools.ietf.org/html/rfc4340*, 2006.

[23] SNOEREN, A. C. *A session-based approach to Internet mobility*. Tese (Doutorado) — PhD Thesis, Massachusetts Institute of Technology, 2002.

[24] MALTZ, D.; BHAGWAT, P. et al. Msocks: An architecture for transport layer mobility. In: IEEE. *INFOCOM'98. Seventeenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*. [S.l.], 1998. v. 3, p. 1037–1045.

[25] CISCO. *Cisco IOS Local-Area Mobility*. 2015. `http://www.cisco.com/en/US/products/ps9390/products_white_paper09186a00800a3ca5.shtml`. [Online; acessado em 02-março-2015].

[26] CARL-MITCHELL, S.; QUARTERMAN, J. S. Ietf rfc 1027: Using arp to implement transparent subnet gateways. *https://tools.ietf.org/html/rfc1027*, 1987.

[27] DAVIE, B.; GROSS, J. A stateless transport tunneling protocol for network virtualization (stt). *draft-davie-stt-04*, 2013.

[28] GROSS, J. et al. Geneve: Generic network virtualization encapsulation. *Internet Engineering Task Force, Internet Draft*, 2014.

[29] ANDERSON, T. et al. Overcoming the internet impasse through virtualization. *Computer*, IEEE Computer Society, Los Alamitos, CA, USA, v. 38, n. 4, p. 34–41, 2005. ISSN 0018-9162.

[30] CISCO. *VXLAN Overview: Cisco Nexus 9000 Series Switches*. 2015. `http://www.cisco.com/c/en/us/products/collateral/switches/nexus-9000-series-switches/white-paper-c11-729383.html`. [Online; acessado em 25-julho-2015].

[31] BENSON, T.; AKELLA, A.; MALTZ, D. A. Unraveling the complexity of network management. In: *NSDI*. [S.l.: s.n.], 2009. p. 335–348.

[32] SHENKER, S. et al. The future of networking, and the past of protocols. *Open Networking Summit*, v. 20, 2011.

[33] KIM, H.; FEAMSTER, N. Improving network management with software defined networking. *Communications Magazine, IEEE*, IEEE, v. 51, n. 2, p. 114–119, 2013.

[34] KREUTZ, D. et al. Software-defined networking: A comprehensive survey. *Proceedings of the IEEE*, v. 103, n. 1, p. 14–76, Jan 2015. ISSN 0018-9219.

[35] LARA, A.; KOLASANI, A.; RAMAMURTHY, B. Network innovation using openflow: A survey. *Communications Surveys Tutorials, IEEE*, v. 16, n. 1, p. 493–512, First 2014. ISSN 1553-877X.

[36] ONF. *Open Network Foundation*. 2015. `https://www.opennetworking.org/`. [Online; acessado em 15-abril-2015].

[37] SPECIFICATION, O. S. *Version 1.0. 0 (Wire Protocol 0x01)*. [S.l.]: December, 2009.

[38] TELECO. *Tutoriais: Redes Definidas por SW I: OpenFlow*. 2015. `http://www.teleco.com.br/tutoriais/tutorialsw1/pagina_4.asp`. [Online; acessado em 12-agosto-2015].

[39] SPECIFICATION, O. S. *Version 1.1. 0 Implemented (Wire Protocol 0x02)*.

[40] SPECIFICATION, O. S. *Version 1.2 (Wire Protocol 0x03)*.

[41] GUDE, N. et al. Nox: Towards an operating system for networks. *SIGCOMM Comput. Commun. Rev.*, ACM, New York, NY, USA, v. 38, n. 3, p. 105–110, jul. 2008. ISSN 0146-4833. Disponível em: <http://doi.acm.org/10.1145/1384609.1384625>.

[42] POX. *About POX*. `http://www.noxrepo.org/pox/about-pox/`. [Online; acessado em 15-fevereiro-2015].

[43] NG, E. *Maestro: A System for Scalable OpenFlow Control*. [S.l.], 2010.

[44] `http://trema.github.io/trema/`. [Online; acessado em 27-maio-2015].

[45] ERICKSON, D. The beacon openflow controller. In: *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*. New York, NY, USA: ACM, 2013. (HotSDN '13), p. 13–18. ISBN 978-1-4503-2178-5. Disponível em: <http://doi.acm.org/10.1145/2491185.2491189>.

[46] NUNES, B. et al. A survey of software-defined networking: Past, present, and future of programmable networks. *Communications Surveys Tutorials, IEEE*, v. 16, n. 3, p. 1617–1634, Third 2014. ISSN 1553-877X.

[47] AZODOLMOLKY, S.; WIEDER, P.; YAHYAPOUR, R. Cloud computing networking: challenges and opportunities for innovations. *Communications Magazine, IEEE*, v. 51, n. 7, p. 54–62, July 2013. ISSN 0163-6804.

[48] DALLY, W. J.; TOWLES, B. P. *Principles and practices of interconnection networks*. [S.l.]: Elsevier, 2004.

[49] LEISERSON, C. Fat-trees: Universal networks for hardware-efficient supercomputing. *Computers, IEEE Transactions on*, C-34, n. 10, p. 892–901, Oct 1985. ISSN 0018-9340.

[50] AL-FARES, M.; LOUKISSAS, A.; VAHDAT, A. A scalable, commodity data center network architecture. *SIGCOMM Comput. Commun. Rev.*, ACM, New York, NY, USA, v. 38, n. 4, p. 63–74, ago. 2008. ISSN 0146-4833. Disponível em: <http://doi.acm.org/10.1145/1402946.1402967>.

[51] GREENBERG, A. et al. The cost of a cloud: Research problems in data center networks. *SIGCOMM Comput. Commun. Rev.*, ACM, New York, NY, USA, v. 39, n. 1, p. 68–73, dez. 2008. ISSN 0146-4833. Disponível em: <http://doi.acm.org/10.1145/1496091.1496103>.

[52] DECUSATIS, C. S.; CARRANZA, A.; DECUSATIS, C. Communication within clouds: open standards and proprietary protocols for data center networking. *Communications Magazine, IEEE*, v. 50, n. 9, p. 26–33, September 2012. ISSN 0163-6804.

[53] MCCAULEY, J. *l2_multi.py*. 2012. `https://github.com/CPqD/RouteFlow/blob/master/pox/pox/forwarding/l2_multi.py`. [Online; acessado em 12-setembro-2014].

[54] LANTZ, B.; HELLER, B.; MCKEOWN, N. A network in a laptop: Rapid prototyping for software-defined networks. In: *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*. New York, NY, USA: ACM, 2010. (Hotnets-IX), p. 19:1–19:6. ISBN 978-1-4503-0409-2. Disponível em: <http://doi.acm.org/10.1145/1868447.1868466>.

[55] PFAFF, B. et al. Extending networking into the virtualization layer. In: *Hotnets*. [S.l.: s.n.], 2009.

A. Código Fonte do PathFlow v1.0

```
# Copyright 2015 Gilvan de Almeida Chaves Filho
#
# Licensed under the Apache License, Version 2.0 (the "
    License");
# you may not use this file except in compliance with the
    License.
# You may obtain a copy of the License at:
#
#      http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing
    , software
# distributed under the License is distributed on an "AS IS
    " BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either
    express or implied.
# See the License for the specific language governing
    permissions and
# limitations under the License.

"""
PathFlow.

Depends on openflow.discovery
Works with openflow.spanning_tree
"""
```

```
from pox.core import core
import pox.openflow.libopenflow_01 as of
from pox.lib.revent import *
from pox.lib.recoco import Timer
from collections import defaultdict
from pox.openflow.discovery import Discovery
from pox.lib.util import dpid_to_str
import time
import random
from pox.lib.addresses import IPAddr, EthAddr

log = core.getLogger()

# Adjacency map.  [sw1][sw2] -> port from sw1 to sw2
adjacency = defaultdict(lambda:defaultdict(lambda:None))

# Switches we know of.  [dpid] -> Switch
switches = {}

# ethaddr -> (switch, port)
localization_table = {}

# [sw1][sw2] -> (distance, intermediate)
path_map = defaultdict(lambda:defaultdict(lambda:(None,None
    )))

# [sw1] [sw2] -> (tag)
spf_table = {}

# Waiting path.  (dpid,xid)->WaitingPath
waiting_paths = {}

# A Mac that is moving moving
moving_mac = None

# Time to not flood in seconds
FLOOD_HOLDDOWN = 5
```

```
# Flow timeouts
FLOW_IDLE_TIMEOUT = 10
FLOW_HARD_TIMEOUT = 30

# How long is allowable to set up a path?
PATH_SETUP_TIME = 4


def _calc_paths ():
  """
  Essentially Floyd-Warshall algorithm
  """

  def dump ():
    for i in sws:
      for j in sws:
        a = path_map[i][j][0]
        #a = adjacency[i][j]
        if a is None: a = "*"
        print a,
      print

  sws = switches.values()
  path_map.clear()
  for k in sws:
    for j,port in adjacency[k].iteritems():
      if port is None: continue
      path_map[k][j] = (1,None)
    path_map[k][k] = (0,None) # distance, intermediate

  #dump()

  for k in sws:
    for i in sws:
      for j in sws:
        if path_map[i][k][0] is not None:
```

```
            if path_map[k][j][0] is not None:
              # i -> k -> j exists
              ikj_dist = path_map[i][k][0]+path_map[k][j][0]
              if path_map[i][j][0] is None or ikj_dist <
                 path_map[i][j][0]:
                # i -> k -> j is better than existing
                path_map[i][j] = (ikj_dist, k)

  #print "--------------------"
  #dump()


def _get_raw_path (src, dst):
  """
  Get a raw path (just a list of nodes to traverse)
  """
  if len(path_map) == 0: _calc_paths()
  if src is dst:
    # We're here!
    return []
  if path_map[src][dst][0] is None:
    return None
  intermediate = path_map[src][dst][1]
  if intermediate is None:
    # Directly connected
    return []
  return _get_raw_path(src, intermediate) + [intermediate]
     + \
         _get_raw_path(intermediate, dst)


def _check_path (p):
  """
  Make sure that a path is actually a string of nodes with
     connected ports

  returns True if path is valid
```

```
  """
  for a,b in zip(p[:-1],p[1:]):
    if adjacency[a[0]][b[0]] != a[2]:
      return False
    if adjacency[b[0]][a[0]] != b[1]:
      return False
  return True


def _get_path (src, dst, first_port, final_port):
  """
  Gets a cooked path -- a list of (node,in_port,out_port)
  """
  # Start with a raw path...
  if src == dst:
    path = [src]
  else:
    path = _get_raw_path(src, dst)
    if path is None: return None
    path = [src] + path + [dst]

  # Now add the ports
  r = []
  in_port = first_port
  for s1,s2 in zip(path[:-1],path[1:]):
    out_port = adjacency[s1][s2]
    r.append((s1,in_port,out_port))
    in_port = adjacency[s2][s1]
  r.append((dst,in_port,final_port))

  assert _check_path(r), "Illegal path!"

  return r

def _get_tag (src, dst):
    """
    Gets a tag between a Source Switch and a Destination
```

```
        Switch
    """

    tag = spf_table.get((src,dst))

    possible_tags = [40, 44, 48, 52, 56, 60, 64, 68, 72,
        76, 80, 84, 88, 92, 96, 100]    # TODO improve
        multiples of TOS from 40 to 256

    if tag == None:                          # There's no tag
        associated to that src -> dst
         if src == dst:
             tag = None
         else:                                          #TODO
            improve this feature
             tag = random.choice(possible_tags)
             possible_tags.remove(tag)
             spf_table[(src,dst)] = tag

    return tag


class WaitingPath (object):
  """
  A path which is waiting for its path to be established
  """
  def __init__ (self, path, packet):
    """
    xids is a sequence of (dpid,xid)
    first_switch is the DPID where the packet came from
    packet is something that can be sent in a packet_out
    """
    self.expires_at = time.time() + PATH_SETUP_TIME
    self.path = path
    self.first_switch = path[0][0].dpid
    self.xids = set()
    self.packet = packet
```

```
    if len(waiting_paths) > 1000:
      WaitingPath.expire_waiting_paths()

  def add_xid (self, dpid, xid):
    self.xids.add((dpid,xid))
    waiting_paths[(dpid,xid)] = self

  @property
  def is_expired (self):
    return time.time() >= self.expires_at

  def notify (self, event):
    """
    Called when a barrier has been received
    """
    self.xids.discard((event.dpid,event.xid))
    if len(self.xids) == 0:
      # Done!
      if self.packet:
        log.debug("Sending delayed packet out %s"
                  % (dpid_to_str(self.first_switch),))
        msg = of.ofp_packet_out(data=self.packet,
            action=of.ofp_action_output(port=of.OFPP_TABLE)
              )
        core.openflow.sendToDPID(self.first_switch, msg)

      core.l2_multi.raiseEvent(PathInstalled(self.path))


  @staticmethod
  def expire_waiting_paths ():
    packets = set(waiting_paths.values())
    killed = 0
    for p in packets:
      if p.is_expired:
        killed += 1
```

```
        for entry in p.xids:
          waiting_paths.pop(entry, None)
    if killed:
      log.error("%i paths failed to install" % (killed,))


class PathInstalled (Event):
  """
  Fired when a path is installed
  """
  def __init__ (self, path):
    Event.__init__(self)
    self.path = path


class Switch (EventMixin):
  def __init__ (self):
    self.connection = None
    self.ports = None
    self.dpid = None
    self._listeners = None
    self._connected_at = None

  def __repr__ (self):
    return dpid_to_str(self.dpid)

  def _install (self, switch, in_port, out_port, match, buf
      = None):
    msg = of.ofp_flow_mod()
    msg.match = match
    msg.match.in_port = in_port
    msg.idle_timeout = FLOW_IDLE_TIMEOUT
    msg.hard_timeout = FLOW_HARD_TIMEOUT
    msg.actions.append(of.ofp_action_output(port = out_port
       ))
    msg.buffer_id = buf
    switch.connection.send(msg)
```

```
def _install_pathflow_without_tag(self, switch, out_port,
    match, buf = None ):
    msg = of.ofp_flow_mod()
    msg.match.dl_dst = match.dl_dst
    msg.idle_timeout = FLOW_IDLE_TIMEOUT
    msg.hard_timeout = FLOW_HARD_TIMEOUT
    msg.actions.append(of.ofp_action_output(port =
       out_port))
    msg.buffer_id = buf
    switch.connection.send(msg)

def _install_pathflow_push_tag (self, switch, tag,
   out_port, match, buf = None):
    msg = of.ofp_flow_mod()
    msg.match.dl_dst = match.dl_dst
    msg.idle_timeout = FLOW_IDLE_TIMEOUT
    msg.hard_timeout = FLOW_HARD_TIMEOUT
    msg.actions.append(of.ofp_action_nw_tos(nw_tos=tag))
    msg.actions.append(of.ofp_action_output(port =
       out_port))
    msg.buffer_id = buf
    switch.connection.send(msg)

def _install_pathflow_just_tag (self, switch, tag,
   out_port, match, buf = None):
    msg = of.ofp_flow_mod()
    msg.match.dl_type = match.dl_type
                                        # todo isso eh
       primordial
    msg.match.nw_tos = tag
    msg.idle_timeout = FLOW_IDLE_TIMEOUT
    msg.hard_timeout = FLOW_HARD_TIMEOUT
    msg.actions.append(of.ofp_action_output(port =
       out_port))
    msg.buffer_id = buf
    switch.connection.send(msg)
```

```
def _install_pathflow_pop_tag(self, switch, out_port,
    match, buf = None):
      msg = of.ofp_flow_mod()
      msg.match.dl_dst = match.dl_dst
      msg.idle_timeout = FLOW_IDLE_TIMEOUT
      msg.hard_timeout = FLOW_HARD_TIMEOUT
      msg.actions.append(of.ofp_action_nw_tos(nw_tos=0))
                              # TODO Improve this to pop the
         tag
      msg.actions.append(of.ofp_action_output(port =
         out_port))
      msg.buffer_id = buf
      switch.connection.send(msg)

def _remove_pathflow (self, mac):
                                                  # TODO
    isso eh temporario
      msg = of.ofp_flow_mod()
      msg.match.dl_dst = mac
      msg.command = of.OFPFC_DELETE

      for connection in core.openflow.connections:
          connection.send(msg)


def _install_path (self, p, match, packet_in=None):
   wp = WaitingPath(p, packet_in)
   for sw,in_port,out_port in p:
      self._install(sw, in_port, out_port, match)
      msg = of.ofp_barrier_request()
      sw.connection.send(msg)
      wp.add_xid(sw.dpid,msg.xid)

def _install_path_pathflow(self, tag, p, match, packet_in
    =None ):
```

```
    wp = WaitingPath(p, packet_in)
                                        # TODO Revisar

    dst_sw = localization_table.get(match.dl_dst)[0]

    if len(p) == 1:
                                                    # "
       local" traffic
        for sw,in_port,out_port in p:
            self._install_pathflow_without_tag(sw, out_port
               , match)
            msg = of.ofp_barrier_request()
            sw.connection.send(msg)
            wp.add_xid(sw.dpid,msg.xid)
    if len(p) >=2:
        for sw,in_port,out_port in p:
            if sw == self:
                self._install_pathflow_push_tag(sw,tag,
                   out_port,match)
                msg = of.ofp_barrier_request()
                sw.connection.send(msg)
                wp.add_xid(sw.dpid,msg.xid)
            if (sw != self) and (sw != dst_sw):
                if (match.dl_type == 0x800):
                    self._install_pathflow_just_tag(sw,tag,
                       out_port,match)
                    msg = of.ofp_barrier_request()
                    sw.connection.send(msg)
                    wp.add_xid(sw.dpid,msg.xid)
                else:
                    self._install(sw, in_port, out_port,
                       match)
                    msg = of.ofp_barrier_request()
                    sw.connection.send(msg)
                    wp.add_xid(sw.dpid,msg.xid)
            if sw == dst_sw:
                self._install_pathflow_pop_tag(sw,out_port,
```

```
                match)
            msg = of.ofp_barrier_request()
            sw.connection.send(msg)
            wp.add_xid(sw.dpid,msg.xid)


  def install_path (self, dst_sw, last_port, match, event):
    """
    Attempts to install a path between this switch and some
        destination
    """
    p = _get_path(self, dst_sw, event.port, last_port)

    if p is None:
      log.warning("Can't get from %s to %s", match.dl_src,
          match.dl_dst)

      import pox.lib.packet as pkt

      if (match.dl_type == pkt.ethernet.IP_TYPE and
          event.parsed.find('ipv4')):
        # It's IP -- let's send a destination unreachable
        log.debug("Dest unreachable (%s -> %s)",
                  match.dl_src, match.dl_dst)

        from pox.lib.addresses import EthAddr
        e = pkt.ethernet()
        e.src = EthAddr(dpid_to_str(self.dpid)) #FIXME: Hmm
            ...
        e.dst = match.dl_src
        e.type = e.IP_TYPE
        ipp = pkt.ipv4()
        ipp.protocol = ipp.ICMP_PROTOCOL
        ipp.srcip = match.nw_dst #FIXME: Ridiculous
        ipp.dstip = match.nw_src
        icmp = pkt.icmp()
        icmp.type = pkt.ICMP.TYPE_DEST_UNREACH
```

```
        icmp.code = pkt.ICMP.CODE_UNREACH_HOST
        orig_ip = event.parsed.find('ipv4')

        d = orig_ip.pack()
        d = d[:orig_ip.hl * 4 + 8]
        import struct
        d = struct.pack("!HH", 0,0) + d #FIXME: MTU
        icmp.payload = d
        ipp.payload = icmp
        e.payload = ipp
        msg = of.ofp_packet_out()
        msg.actions.append(of.ofp_action_output(port =
            event.port))
        msg.data = e.pack()
        self.connection.send(msg)

      return

    tag = _get_tag(self, dst_sw)

    self._install_path_pathflow(tag, p, match, event.ofp)

    '''log.debug("Installing path for %s -> %s %04x (%i
       hops)",          #TODO atencao esse eh o passo
       principal do software principal
        match.dl_src, match.dl_dst, match.dl_type, len(p))

    # We have a path -- install it
    self._install_path(p, match, event.ofp)

    # Now reverse it and install it backwards
    # (we'll just assume that will work)
    p = [(sw,out_port,in_port) for sw,in_port,out_port in p
       ]
    self._install_path(p, match.flip())'''
```

```
  def _handle_PacketIn (self, event):
    def flood ():
      """ Floods the packet """
      if self.is_holding_down:
        log.warning("Not flooding — holddown active")
      msg = of.ofp_packet_out()
      # OFPP_FLOOD is optional; some switches may need
         OFPP_ALL
      msg.actions.append(of.ofp_action_output(port = of.
         OFPP_FLOOD))
      msg.buffer_id = event.ofp.buffer_id
      msg.in_port = event.port
      self.connection.send(msg)

    def drop ():
      # Kill the buffer
      if event.ofp.buffer_id is not None:
        msg = of.ofp_packet_out()
        msg.buffer_id = event.ofp.buffer_id
        event.ofp.buffer_id = None # Mark is dead
        msg.in_port = event.port
        self.connection.send(msg)

    packet = event.parsed

    loc = (self, event.port) # Place we saw this ethaddr
    oldloc = localization_table.get(packet.src) # Place we
       last saw this ethaddr

    if packet.effective_ethertype == packet.LLDP_TYPE:
      drop()
      return

    if oldloc is None:
      if packet.src.is_multicast == False:
        localization_table[packet.src] = loc # Learn
           position for ethaddr
```

```
      log.debug("Learned %s at %s.%i", packet.src, loc
        [0], loc[1])
    elif oldloc != loc:
      # ethaddr seen at different place!
      if core.openflow_discovery.is_edge_port(loc[0].dpid,
       loc[1]):
        # New place is another "plain" port (probably)
        log.debug("%s moved from %s.%i to %s.%i?", packet.
          src,
                  dpid_to_str(oldloc[0].dpid), oldloc[1],
                  dpid_to_str(   loc[0].dpid),    loc[1])
        if packet.src.is_multicast == False:
          localization_table[packet.src] = loc # Learn
            position for ethaddr
          log.debug("Learned %s at %s.%i", packet.src, loc
            [0], loc[1])
      elif packet.dst.is_multicast == False:
        # New place is a switch-to-switch port!
        # Hopefully, this is a packet we're flooding
          because we didn't
        # know the destination, and not because it's
          somehow not on a
        # path that we expect it to be on.
        # If spanning_tree is running, we might check that
          this port is
        # on the spanning tree (it should be).
        if packet.dst in localization_table:
          # Unfortunately, we know the destination.  It's
            possible that
          # we learned it while it was in flight, but it's
            also possible
          # that something has gone wrong.
          log.warning("Packet from %s to known destination
            %s arrived "
                      "at %s.%i without flow", packet.src,
                        packet.dst,
                      dpid_to_str(self.dpid), event.port)
```

```
    if packet.dst.is_multicast:
      log.debug("Flood multicast from %s", packet.src)
      flood()
    else:
      if packet.dst not in localization_table:
        log.debug("%s unknown -- flooding" % (packet.dst,))
        flood()
      else:
        dest = localization_table[packet.dst]
        match = of.ofp_match.from_packet(packet)
        self.install_path(dest[0], dest[1], match, event)

  def _handle_PortStatus (self, event):


    if event.deleted == True:
                                                              #
      TODO isso eh temporario
       # Some Host could have gone down

       lValue = (self, event.port)
       print ('lvalue:', lValue)
       mac = [key for key, value in localization_table.
          iteritems() if value == lValue][0]
       print ('mac:',mac)
       self._remove_pathflow(mac)
       global moving_mac
       moving_mac = mac
       print ('moving mac dentro de event deleted:',
          moving_mac)

    if event.added == True:

      # TODO isso eh temporario
       global moving_mac
```

```
            mac = moving_mac

            localization_table[mac] = (self, event.port)



  def disconnect (self):
    if self.connection is not None:
      log.debug("Disconnect %s" % (self.connection,))
      self.connection.removeListeners(self._listeners)
      self.connection = None
      self._listeners = None

  def connect (self, connection):
    if self.dpid is None:
      self.dpid = connection.dpid
    assert self.dpid == connection.dpid
    if self.ports is None:
      self.ports = connection.features.ports
    self.disconnect()
    log.debug("Connect %s" % (connection,))
    self.connection = connection
    self._listeners = self.listenTo(connection)
    self._connected_at = time.time()

  @property
  def is_holding_down (self):
    if self._connected_at is None: return True
    if time.time() - self._connected_at > FLOOD_HOLDDOWN:
      return False
    return True

  def _handle_ConnectionDown (self, event):
    self.disconnect()


class l2_multi (EventMixin):
```

```
_eventMixin_events = set([
  PathInstalled,
])

def __init__ (self):
  # Listen to dependencies
  def startup ():
    core.openflow.addListeners(self, priority=0)
    core.openflow_discovery.addListeners(self)
  core.call_when_ready(startup, ('openflow','
     openflow_discovery'))

def _handle_LinkEvent (self, event):
  def flip (link):
    return Discovery.Link(link[2],link[3], link[0],link
       [1])

  l = event.link
  sw1 = switches[l.dpid1]
  sw2 = switches[l.dpid2]

  # Invalidate all flows and path info.
  # For link adds, this makes sure that if a new link
     leads to an
  # improved path, we use it.
  # For link removals, this makes sure that we don't use
     a
  # path that may have been broken.
  #NOTE: This could be radically improved! (e.g., not *
     ALL* paths break)
  clear = of.ofp_flow_mod(command=of.OFPFC_DELETE)
  for sw in switches.itervalues():
    if sw.connection is None: continue
    sw.connection.send(clear)
  path_map.clear()
```

```
if event.removed:
  # This link no longer okay
  if sw2 in adjacency[sw1]: del adjacency[sw1][sw2]
  if sw1 in adjacency[sw2]: del adjacency[sw2][sw1]

  # But maybe there's another way to connect these...
  for ll in core.openflow_discovery.adjacency:
    if ll.dpid1 == l.dpid1 and ll.dpid2 == l.dpid2:
      if flip(ll) in core.openflow_discovery.adjacency:
        # Yup, link goes both ways
        adjacency[sw1][sw2] = ll.port1
        adjacency[sw2][sw1] = ll.port2
        # Fixed -- new link chosen to connect these
        break
else:
  # If we already consider these nodes connected, we
     can
  # ignore this link up.
  # Otherwise, we might be interested...
  if adjacency[sw1][sw2] is None:
    # These previously weren't connected.  If the link
    # exists in both directions, we consider them
       connected now.
    if flip(l) in core.openflow_discovery.adjacency:
      # Yup, link goes both ways -- connected!
      adjacency[sw1][sw2] = l.port1
      adjacency[sw2][sw1] = l.port2

  # If we have learned a MAC on this port which we now
     know to
  # be connected to a switch, unlearn it.
  bad_macs = set()
  for mac,(sw,port) in localization_table.iteritems():
    if sw is sw1 and port == l.port1: bad_macs.add(mac)
    if sw is sw2 and port == l.port2: bad_macs.add(mac)
  for mac in bad_macs:
    log.debug("Unlearned %s", mac)
```

```
            del localization_table[mac]

    def _handle_ConnectionUp (self, event):
        sw = switches.get(event.dpid)
        if sw is None:
            # New switch
            sw = Switch()
            switches[event.dpid] = sw
            sw.connect(event.connection)
        else:
            sw.connect(event.connection)

    def _handle_BarrierIn (self, event):
        wp = waiting_paths.pop((event.dpid,event.xid), None)
        if not wp:
            #log.info("No waiting packet %s,%s", event.dpid,
                event.xid)
            return
        #log.debug("Notify waiting packet %s,%s", event.dpid,
            event.xid)
        wp.notify(event)


def launch ():
    core.registerNew(l2_multi)

    timeout = min(max(PATH_SETUP_TIME, 5) * 2, 15)
    Timer(timeout, WaitingPath.expire_waiting_paths,
        recurring=True)
```