CENTRO UNIVERSITÁRIO UNIVATES
CURSO DE TECNOLOGIA EM REDES DE COMPUTADORES

REDES DEFINIDAS POR SOFTWARE: UMA OUTRA ABORDAGEM SOBRE O TRATAMENTO DE FLUXOS EM REDES DE COMPUTADORES

Fábio Angelo Brizzolla

Lajeado, novembro de 2016

Fábio Angelo Brizzolla

REDES DEFINIDAS POR SOFTWARE: UMA OUTRA ABORDAGEM SOBRE O TRATAMENTO DE FLUXOS EM REDES DE COMPUTADORES

Artigo apresentado na disciplina de Seminário Integrador, do Curso de Tecnologia em Redes de Computadores, do Centro Universitário UNIVATES, como parte da exigência para a obtenção do título de Tecnólogo em Redes de Computadores.

Orientador: Prof. Ms. Luis Antônio Schneiders

RESUMO

As redes de computadores são fundamentais para a evolução contínua da computação. No entanto os algoritmos que governam o encaminhamento e roteamento de tráfego permanecem inalterados desde a sua concepção, e novas funcionalidades ou atualizações são apenas somadas a pilha já existente. Em Redes Definidas por Software (*Software Defined Networks* - SDN) esses algoritmos que controlam os fluxos são elevados da camada de dados para camadas mais altas, transformando funcionalidades de rede em aplicações (software) hospedadas em um ponto central de controle e coordenação da rede. Essa nova abordagem proporciona uma nova forma de se tratar os tráfegos de rede, inclusive na demanda de esforço necessária para a sua configuração.

1. INTRODUÇÃO

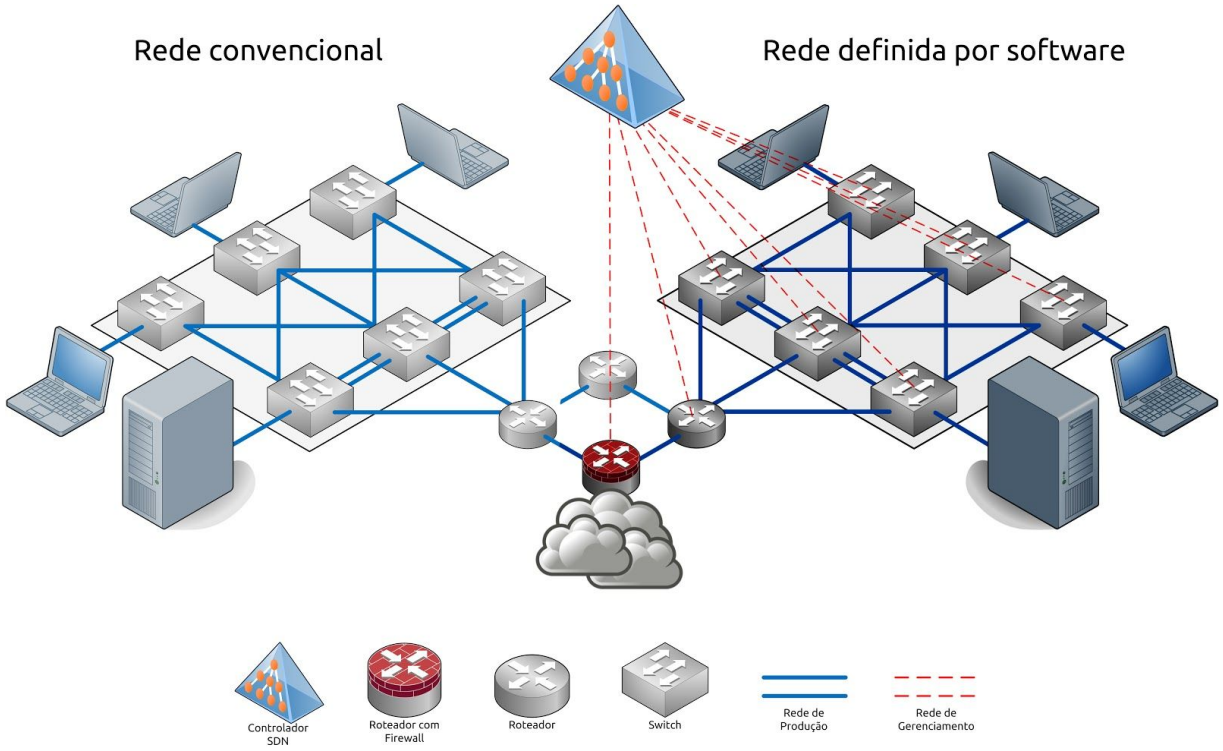
Desde que as redes de computadores passaram a ser vistas como fundamentais e críticas para o funcionamento e evolução da computação, pouca coisa de fato mudou no que tange a forma como se processa e se encaminha pacotes nas redes. Os algoritmos que governam o encaminhamento e roteamento de pacotes permanecem inalterados a medida em que novas funcionalidades são apenas adicionadas à pilha já existente. Em redes definidas por software (SDNs) essas funcionalidades, antes confinada em EPROMs e de domínio privado, é desacoplada do barramento, abstraída na forma de software e elevada às camadas superiores, transformada em uma aplicação instalada em um controlador central permitindo que, finalmente, desde a década de 60, o tráfego e funções de rede possam ser tratados de forma mais dinâmica, flexível, otimizada e (em tese) mais eficiente (MORREALE & ANDERSON, 2014).

Em uma SDN há então a abstração das funcionalidades de baixo-nível do ativo de rede, separando o sistema que toma as decisões (camada de controle - Control Plane) do sistema que encaminha o tráfego para um determinado destino (camada de dados - Data Plane). Entre as camadas de controle e dados existe uma interface (protocolo) e é através dele que um determinado controlador SDN instrui o ativo sobre como o tráfego deve ser manipulado. Entre os protocolos atualmente em uso podemos citar o NETCONF[1], BGP-LS[2] , PCE-P[3], SNMP[4] e muitos outros. No entanto, o protocolo mais amplamente utilizado e que já conta com suporte da maioria dos fabricantes é o OpenFlow, que se encontra atualmente na sua revisão 1.4[5].

[1] NETCONF - RFC6241 <https://tools.ietf.org/html/rfc6241>
[2] BGP-LS - RFC7752 <http://tools.ietf.org/html/rfc7752>
[3] PCE-P - RFC5440 <https://tools.ietf.org/html/rfc5440>
[4] SNMP - RFC3413 <https://tools.ietf.org/html/rfc3413>
[5] OpenFlow Specifications - <https://www.opennetworking.org/ja/sdn-resources-ja/onf-specifications/openflow>

No que tange a disposição ou apresentação física dos elementos de uma SDN em nada se difere de uma rede de computadores convencional. Os elementos físicos do modelo de comunicação de dados estão presentes (ETD, DCE e meio de comunicação) e em nada se diferem de sua apresentação, conforme ilustra o diagrama abaixo:

Figura 1 - Diagrama lógico de uma rede de computadores



Fonte: Produção do próprio autor.

Em uma SDN temos a presença de um elemento adicional, um controlador, membro da topologia e que cuja finalidade é instruir os ativos de rede com relação a destinação e trato dos pacotes de dados entrantes e que compõem o tráfego. As regras são definidas por elementos de software presentes no controlador SDN, como por exemplo a definição de encaminhamento de pacotes no nível 2, que ao invés de se apresentar individualmente em cada *switch* da rede, confinado em EPROM, passa a ser uma aplicação carregada pelo controlador SDN e que cujas regras são distribuídas aos *switches* de acesso, distribuição e/ou core, por meio do protocolo OpenFlow.

O tráfego de gerenciamento pelo qual estas regras são distribuídas é feito através de uma conexão lógica dedicada para essa função. Uma vez instruídos pelo controlador SDN, os dispositivos de rede só necessitarão comunicar-se novamente com o ponto central de gerenciamento em um evento onde os pacotes que compõem um determinado fluxo de dados não são conhecidos, ou seja, não apresentam regras definidas por aplicações ativas no(s) controlador(es), que pode(m) estar implementado(s) de forma distribuída ou centralizada. SDNs implicam numa mudança de paradigma na forma que as redes são desenhadas e

operadas, sendo resultado de uma tendência mundial onde as funcionalidades de hardware são substituídas por implementações em software (HU, 2013).

Por tratar-se de uma tecnologia que ainda permeia com mais intensidade o meio acadêmico, de forma experimental e com poucos casos de uso em funcionamento em ambientes de produção, como por exemplo em grandes corporações como AT&T e Google, este trabalho irá explorar o funcionamento desse novo paradigma na implementação, simulação e pesquisa sobre redes definidas por software, tendo como objeto a comparação do funcionamento entre topologias utilizando o modelo convencional e o modelo definido por software, por meio de ferramentas padrão de mercado e largamente utilizadas no meio acadêmico na pesquisa e desenvolvimento da tecnologia.

2. METODOLOGIA

Serão avaliadas duas topologias de rede: *tree* (árvore) e *full-mesh* (múltiplos caminhos). Sobre as topologia serão feitas comparações dos tráfegos computados no modelo definido por software e no modelo tradicional, utilizando uma bateria de testes padronizada no que se refere as cargas injetadas.

Sobre a topologia *full-mesh* será efetuada uma análise do comportamento da mesma sob uma condição adversa com mútiplos caminhos redundantes (que no modelo tradicional geram problemas como *looping* quando não há um mecanismo de controle - STP ou *Spanning Tree Protocol*), observando as decisões de recálculo de rotas em eventos de corte de links de comunicação entre dispositivos de rede.

Sobre a topologia *tree* será efetuada uma comparação do esforço necessário para a configuração de uma típica rede com *switches* de *Core,* Distribuição e Acesso; segregando *switches* de Acesso como departamentais (impedindo troca de informações entre departamentos) porém permitindo mútuo acesso a estrutura por trás dos *switches* de *Core*.

Os testes serão realizados utilizando o Mininet como ambiente de emulação, o OpenDayLight como controlador SDN e Wireshark como ferramenta de captura a análise de tráfego. Todo o ambiente ficará contido em máquina virtual multiprocessada rodando Ubuntu Server 14.04 LTS de 64-bits.

2.1. MININET

É um emulador de redes, de código aberto[6], capaz de simular uma coleção de *hosts*, *switches*, roteadores e *links* em uma única instância de Kernel de Linux. É largamente

[6] http://mininet.org/

utilizado no teste de SDNs porque permite criar topologias personalizadas (via linguagem Python ou graficamente por meio do Mini Edit), complexas e realistas, tais como aquelas utilizadas em pesquisas sobre BGP na Internet (NADEAU & GRAY, 2013), possibilitando a prototipagem de grandes redes com recursos limitados a de um simples *laptop* (LATZ, HELLER & MCKEOWN, 2010).

2.2. OPENVSWITCH (OVS)

Open vSwitch (OVS) é uma implementação em software de um *switch* distribuído de múltiplas camadas, cujo principal propósito é atuar na camada de comutação de pacotes em ambientes de virtualização em hardware, onde é necessário o suporte a diversos tipos de protocolos e padrões utilizados em redes de computadores. É distribuído gratuitamente sob a licença Apache 2.0 e faz parte do *Kernel* de todas as distribuições Linux desde a versão 3.3.

2.3. OPENDAYLIGHT (ODL)

OpenDayLight (ODL) é um projeto de código-aberto que propõem um controlador SDN modular e flexível, implementado em Java (JVM) e que, por essa razão, pode ser descarregado em qualquer hardware ou sistema operacional de mercado que suporte essa linguagem. Em suas versões mais recentes utiliza o Apache Karaf, um servidor de aplicações Web de código aberto que utiliza as especificações do OSGi versão 5 (Open Specifications Group Initiative) para rodar aplicações no mesmo espaço de endereçamento de memória do próprio controlador. Isso permite que as aplicações ou módulos sejam carregados e descarregados instantaneamente, sem prejuízo aos serviços já em operação.

Sob tutela da Linux Foundation, o projeto conta com a participação de mais de 30 grandes empresas do segmento de redes e telecomunicações[7], como por exemplo Cisco, Intel, NEC, Brocade, AT&T, Avaya, Hitachi, Juniper, Alcatel/Lucent, e muitas outras.

2.4. OPENFLOW (OF)

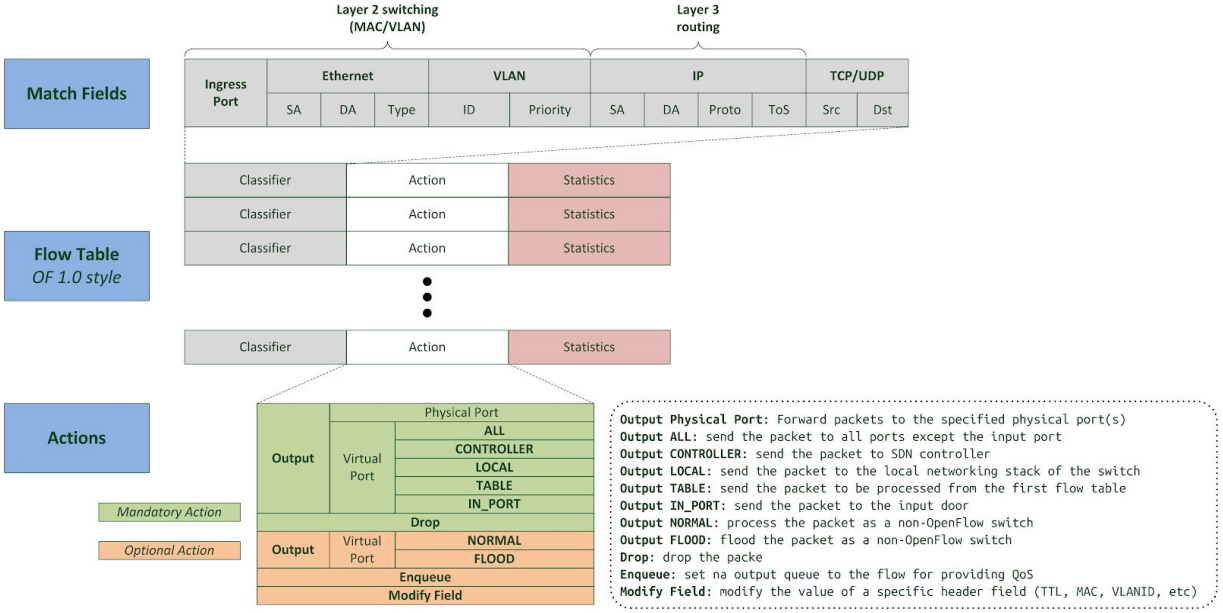
O OpenFlow é um conjunto de protocolos + API, desenvolvidos como parte de uma pesquisa realizada na Universidade de Stanford, cuja ideia central era permitir a substituição completa dos protocolos de camadas L2 e L3 em *swicthes* e roteadores, em uma abordagem referida como "*clean slate proposition*" onde a camada de controle seria completamente centralizada (NADEAU & GRAY, 2013).

No OpenFlow todo o cabeçalho do pacote (campos de nível 2 e nível 3) fica disponível para ações de filtragem e modificação, permitindo um comportamento similar ao de um

[7] https://www.opendaylight.org/membership.

roteador, estendendo as possibilidades por conta do número de ações possíveis (que podem estar ou não mascaradas/disponíveis no dispositivo de rede), como mostra a Figura 3, onde se apresenta, inclusive, a opção de uma ação do tipo *Modify Field*, a qual permite manipular campos do cabeçalho. "Não há na indústria um protocolo equivalente padronizado e com as vantagens e possibilidades do OpenFlow, características que o apontam como um forte substituto para mecanismos de pesquisa e encaminhamento baseados em políticas de roteamento (*Policy Based Routing - PBR*)" (NADEAU & GRAY, 2013).

Figura 3 - Estrutura de uma entrada de fluxo, contendo as instruções, contadores e regras constituídas por campos como porta, MAC, tipo de serviço, entre outros.



Fonte: Do autor, adaptado de NADEAU & GRAY, et. al., 2013.

Por conta da versatilidade do OpenFlow e a possibilidade de estender à *switches* funcionalidades que se assemelham a de roteadores, é comum surgirem questões sobre ao que de fato o protocolo se refere: comutação ou roteamento? Na verdade nem um e nem outro: o protocolo OpenFlow trata de fluxos, que são rajadas de pacotes. Tradicionalmente se tem na camada de controle de um *switch* a lógica para associação de endereços MAC com suas portas, alguma forma de implementação do RSTP (*Rapid Spanning Tree Protocol*), entre outros; e na camada de controle de um roteador temos algoritmos de roteamento distribuído como OSPF e BGP. Quando inserimos um controlador central e elevamos as funções de rede para elementos de software instalados nele, a diferença entre um *switch* e um roteador não é mais tão clara[8]. Por essa razão muitos trabalhos acadêmicos sobre OpenFlow tratam todos os dispositivos de uma SDN como "*OpenFlow Nodes"*, e o que acaba distinguindo um do outro

[8] FERRO, Greg. **OpenFlow and SDN: Is it Routing or Switching?** Disponível em <http://etherealmind.com/openflow-software-defined-networking-routing-or-switching/>. Acessado em: 02 abr. 2016.

é apenas a camada (L2, L3 ou L4 do modelo OSI) em que está desempenhando algum controle sobre os fluxos em trânsito.

3. IMPLEMENTAÇÃO

A implementação do ambiente não abordará em detalhe a instalação do sistema operacional e se deterá na instalação, configuração e execução dos elementos de software necessários para o desenvolvimento dos objetivos propostos neste artigo, principalmente do controlador SDN (OpenDayLight) e da ferramenta de emulação/simulação de topologias de rede (Mininet).

3.1. DISPONIBILIZANDO O MININET

Há três formas de se obter uma instalação do Mininet: a) *download* de uma VM pronta [9] com todas as dependências e configurações previamente instaladas; b) efetuar uma instalação nativa compilando o o código fonte[10] ou; c) fazer a instalação a partir de pacotes pré-compilados[11] para o sistema operacional escolhido. Foi efetuada a escolha pela compilação do pacote por questões de controle do ambiente como um todo, permitindo uma maior seletividade dos componentes e dependências instaladas no sistema operacional.

Primeiramente é necessário efetuar o clone do repositório do Mininet para a máquina local, selecionar a versão (*branch*) desejada e fazer o *checkout* antes de executar o *script* de instalação:

```
$ git clone git://github.com/mininet/mininet
$ cd mininet
$ git tag
$ git checkout -b 2.2.1 2.2.1
$ cd ..
$ sudo ./mininet/util/install.sh -nfv
```

Esse método de instalação a partir do repositório (GitHub) permite que o Mininet seja facilmente atualizado, posteriormente, realizando um *fetch/checkout/pull* da nova versão:

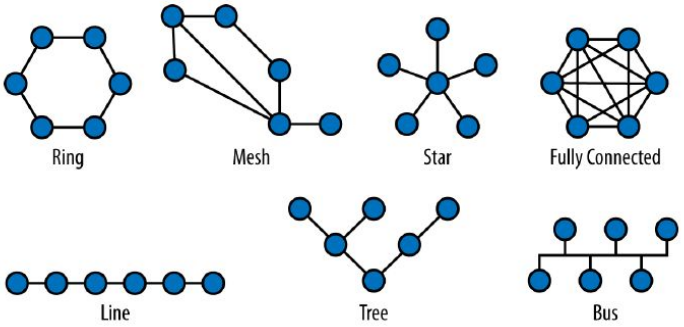
```
$ cd mininet
$ git fetch && git checkout master
$ git pull
$ sudo make install
```

Conforme descrito na seção 2 (metodologia), serão instanciadas 2 topologias diferentes que serão utilizadas como base para os testes comparativos entre o modelo convencional e o modelo proposto por uma SDN. A Figura 4 ilustra as topologias clássicas de

[9] http://mininet.org/download/#option-1-mininet-vm-installation-easy-recommended
[10] http://mininet.org/download/#option-2-native-installation-from-source
[11] http://mininet.org/download/#option-3-installation-from-packages

rede, dos quais serão instanciadas as do tipo *Tree* e *Full-Mesh* que serão submetidas aos testes:

Figura 4 - Topologias clássicas de rede



Fonte: NADEAU & GRAY, et. al., 2013.

Respectivamente para cada uma das topologias alvo serão usados os seguintes comandos e parâmetros:

```
$ mn --topo linear,3 --mac --switch ovsk
$ mn --topo tree,3 --mac --switch ovsk
$ mn --topo mytopo --custom fullmesh.py --mac --switch ovsk
```

Para os testes onde a topologia utilizará o modelo SDN, será especificado o parâmetro **`--controller remote`**, que instrui os *switches* virtuais a se comunicarem através da porta 6633 com um controlador SDN externo ao domínio de execução do Mininet, utilizando o endereço de *loopback* (por padrão) ou especificando um endereço IP e porta, como por exemplo **`--controller=remote,ip=<endereço>,port=<porta>`**. Os parâmetros **`--mac`** e **`--switch ovsk`** são usados para especificar o uso simplificado de endereços MAC (exe.: *switch* S1 terá MAC 00:00:00:00:00:01) e informar que os *switches* instanciados devem ser do tipo Open Virtual Switch Kernel (OVSK), respectivamente.

O teste realizado no modelo convencional, os *switches* que compõem estas topologias devem ser configurados para operarem fazendo o encaminhamento simples, em modo *Ethernet Bridge* , isto é, operando como *switches* L2 convencionais; para isso utiliza-se o parâmetro **`set-fail-mode`** em todos os switches da topologia com a configuração **`standalone`**:

```
$ for i in {1..N}; do ovs-vsctl set-fail-mode s$i standalone; done
```

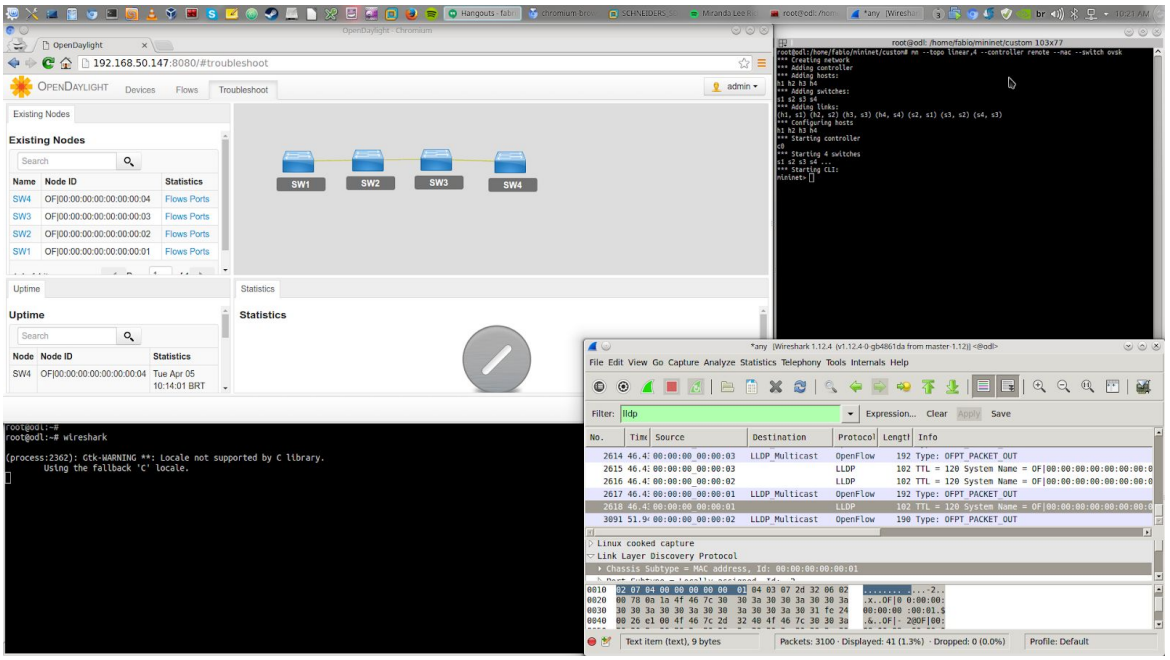
Com relação as topologias, por se tratarem de disposições personalizadas de dispositivos de rede, a construção das mesmas foi feita utilizando Python e os códigos utilizados são os que seguem:

```
# CODIGO TOPOLOGIA FULLMESH

from mininet.topo import Topo

class MyTopo( Topo ):
  def __init__( self ):
    Topo.__init__( self )

  # Adiciona swicthes
  S1 = self.addSwitch( 's1' )
  S2 = self.addSwitch( 's2' )
  S3 = self.addSwitch( 's3' )
  S4 = self.addSwitch( 's4' )
  S5 = self.addSwitch( 's5' )
  S6 = self.addSwitch( 's6' )
  S7 = self.addSwitch( 's7' )
  S8 = self.addSwitch( 's8' )
  S9 = self.addSwitch( 's9' )
  S10 = self.addSwitch( 's10' )
  S11 = self.addSwitch( 's11' )
  S12 = self.addSwitch( 's12' )

  # Adiciona hosts
  H1 = self.addHost( 'h1' )
  H2 = self.addHost( 'h2' )
  H3 = self.addHost( 'h3' )
  H4 = self.addHost( 'h4' )

  # Agrupador de switches
  SwLanA = (S1,S2,S3,S4,S5,S6)
  SwLanB = (S7,S8,S9,S10,S11,S12)

  # Cria Mesh na LAN A
  for index in range ( 0 , len(SwLanA)):
    for index2 in range ( index+1 , len (SwLanA)):
      self.addLink(SwLanA[index] , SwLanA[index2])

  # Cria Mesh na LAN B
  for index in range ( 0 , len(SwLanB)):
    for index2 in range ( index+1 , len (SwLanB)):
      self.addLink(SwLanB[index] , SwLanB[index2])

  # Conecta hosts aos switches e interliga as Meshes
  self.addLink(H1, S1)
  self.addLink(H2, S2)
  self.addLink(H3, S11)
  self.addLink(H4, S12)
  self.addLink(S4, S8)

topos = { 'mytopo': ( lambda: MyTopo() ) }
```

```
# CODIGO TOPOLOGIA ÁRVORE

from mininet.topo import Topo

class MyTopo( Topo ):
  def __init__( self ):
    Topo.__init__( self )

  # Adiciona swicthes
  S1 = self.addSwitch( 's1' )
  S2 = self.addSwitch( 's2' )
  S3 = self.addSwitch( 's3' )
  S4 = self.addSwitch( 's4' )
  S5 = self.addSwitch( 's5' )
  S6 = self.addSwitch( 's6' )
  S7 = self.addSwitch( 's7' )

  # Adiciona hosts
  H1 = self.addHost( 'h1' )
  H2 = self.addHost( 'h2' )
  H3 = self.addHost( 'h3' )
  H4 = self.addHost( 'h4' )
  H5 = self.addHost( 'h5' )
  H6 = self.addHost( 'h6' )
  H7 = self.addHost( 'h7' )
  H8 = self.addHost( 'h8' )
  H9 = self.addHost( 'h9' )
  H10 = self.addHost( 'h10' )

  # Conecta switches e hosts
  self.addLink(H3, S4)
  self.addLink(H4, S4)
  self.addLink(H5, S6)
  self.addLink(H6, S6)
  self.addLink(H7, S7)
  self.addLink(H8, S7)
  self.addLink(S2, S1)
  self.addLink(S5, S1)
  self.addLink(S3, S2)
  self.addLink(S4, S2)
  self.addLink(S6, S5)
  self.addLink(S7, S5)
  self.addLink(H1, S3)
  self.addLink(H2, S3)
  self.addLink(H9, S1)
  self.addLink(H10, S1)

topos = { 'mytopo': ( lambda:
MyTopo() ) }
```

Uma vez instanciada, a autodescoberta dos dispositivos de rede é feita por meio de anúncios (multicast) emitidos pelos dispositivos de rede, utilizando protocolo LLDP (*Link Layer Discovery Protocol* - IEEE 802.1AB). A autodescoberta dos *hosts* conectados aos dispositivos de rede se dará apenas depois de inicializada alguma transferência de pacotes, como por exemplo via **`ping`** ou **`pingall`** do Mininet, que força uma troca de "*echos*" entre todos os *hosts* virtuais da topologia.

Figura 5 - Autodescoberta via LLDP da topologia de *switches*, em linha, com captura dos referidos pacotes pelo Wireshark.



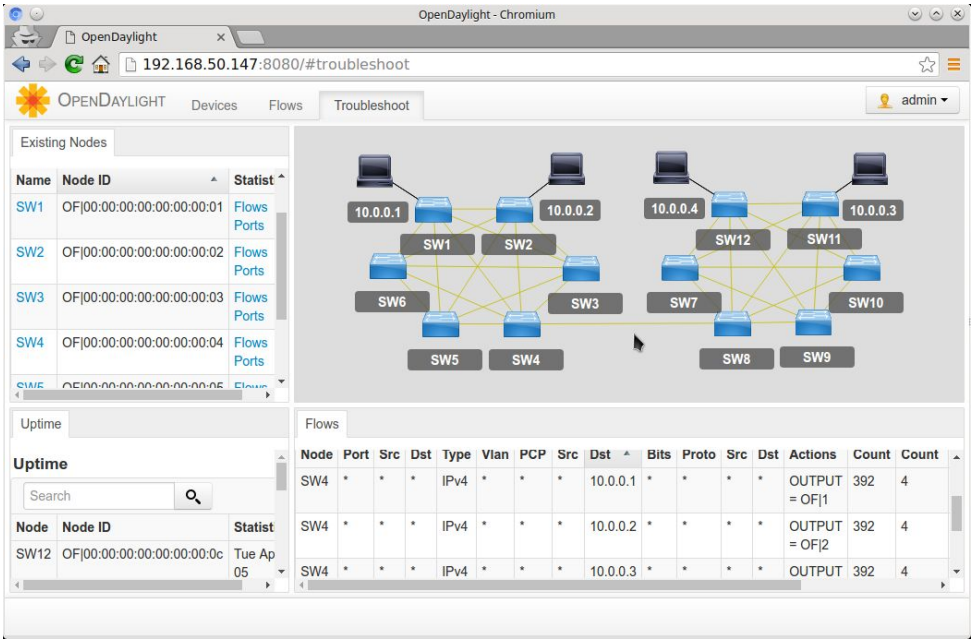
Fonte: Produção do próprio autor.

O estado dos *links* entre os dispositivos de rede e *hosts* pode ser mapeado e manipulado diretamente via terminal do Mininet através dos comandos **`ports`**, **`links`** e **`link`**, conforme mostra o exemplo que segue. Primeiramente instanciamos uma topologia para testes, neste exemplo usaremos a topologia *Full-Mesh*:

```
$ mn --topo mytopo --custom fullmesh.py --controller remote --mac
```

A topologia instanciada possui duas redes de *switches* em *Full-Mesh*, que são interligadas entre si pelos *switches* S4 e S8, conforme pode ser evidenciado pela autodescoberta da topologia ilustrada pelo ODL:

Figura 6 - Autodescoberta das duas topologias *Full-Mesh*, interligadas por meio dos *switches* S4 e S8.



Fonte: Produção do próprio autor.

As duas redes *Full-Mesh* foram então desconectadas alterando o estado do *link* entre os *switches* S4 e S8 para "*down*" e um teste de conectividade feito via **`pingall`**. Observa-se que na tabela de fluxos do *switch* S1 não foram criadas automaticamente as regras para os *hosts* (10.0.0.3 e 10.0.0.4) da rede vizinha:

```
mininet> link s4 s8 down
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2 X X
h2 -> h1 X X
h3 -> X X h4
h4 -> X X h3
*** Results: 66% dropped (4/12 received)

$ ovs-ofctl dump-flows s1
(...)
cookie=0x0, duration=1294.001s, table=0, n_packets=4, n_bytes=392, idle_age=532,
priority=1,ip,nw_dst=10.0.0.2 actions=output:1
cookie=0x0, duration=1294.033s, table=0, n_packets=8, n_bytes=784, idle_age=531,
priority=1,ip,nw_dst=10.0.0.1 actions=mod_dl_dst:00:00:00:00:00:01,output:6
```

O link entre as duas redes então foi restabelecido e um novo teste de conectividade realizado. Neste momento a aplicação **`l2switch`** carregada no controlador automaticamente calculou a rota mais curta para que os *hosts* de uma rede *Mesh* pudessem alcançar a rede vizinha e sem qualquer implicação de problemas por *looping* entre *switches*:

```
mininet> link s4 s8 up
mininet> pingall
*** Ping: testing ping reachability
```

```
h1 -> h2 h3 h4
h2 -> h1 h3 h4
h3 -> h1 h2 h4
h4 -> h1 h2 h3
*** Results: 0% dropped (12/12 received)

$ ovs-ofctl dump-flows s1
(...)
cookie=0x0, duration=1615.278s, table=0, n_packets=6, n_bytes=588, idle_age=3,
priority=1,ip,nw_dst=10.0.0.2 actions=output:1
cookie=0x0, duration=6.336s, table=0, n_packets=2, n_bytes=196, idle_age=3,
priority=1,ip,nw_dst=10.0.0.4 actions=output:3
cookie=0x0, duration=6.328s, table=0, n_packets=2, n_bytes=196, idle_age=3,
priority=1,ip,nw_dst=10.0.0.3 actions=output:3
cookie=0x0, duration=1615.31s, table=0, n_packets=14, n_bytes=1372, idle_age=3,
priority=1,ip,nw_dst=10.0.0.1 actions=mod_dl_dst:00:00:00:00:00:01,output:6
```

Observa-se que a porta designada para alcançar os *hosts* 10.0.0.3 e 10.0.0.4 foi a porta 3 do *switch* S1. Uma rápida verificação da rota designada pela aplicação **`l2switch`** evidencia que este é de fato o caminho mais curto entre os *hosts*, passando pelos *switches* S4 (porta 6) e S8 (portas 4 e 5):

```
mininet> links
(...)
s1-eth3<->s4-eth1 (OK OK)
(...)
s4-eth6<->s8-eth6 (OK OK)
(...)
s8-eth5<->s12-eth2 (OK OK)
s8-eth4<->s11-eth2 (OK OK)
(...)
mininet>

$ ovs-ofctl dump-flows s4
(...)
cookie=0x0, duration=711.456s, table=0, n_packets=4, n_bytes=392, idle_age=708,
priority=1,ip,nw_dst=10.0.0.4 actions=output:6
cookie=0x0, duration=711.448s, table=0, n_packets=4, n_bytes=392, idle_age=708,
priority=1,ip,nw_dst=10.0.0.3 actions=output:6
(...)

$ ovs-ofctl dump-flows s8
(...)
cookie=0x0, duration=2416.002s, table=0, n_packets=8, n_bytes=784, idle_age=813,
priority=1,ip,nw_dst=10.0.0.4 actions=output:5
(...)
cookie=0x0, duration=2425.987s, table=0, n_packets=8, n_bytes=784, idle_age=813,
priority=1,ip,nw_dst=10.0.0.3 actions=output:4
```

No teste utilizando a topologia em árvore, com *switches* de múltiplas funções (Core, Distribuição e Acesso), foi avaliado o esforço necessário para uma segmentação da rede onde cada *switch* de acesso representaria uma LAN isolada, não podendo haver comunicação direta entre LANs mas que fôsse possível, para todas as LANs, comunicarem-se com a infraestrutura disponibilizada no *Core*.

No modelo tradicional, uma das formas de se atingir esse objetivo seria utilizando VLANs (uma para cada *switch* de Acesso) e permitir a comunicação com a estrutura de *Core* utilizando *trunks* (permitindo o tráfego de todas as VLANs até o *Core* da rede. Neste caso seria necessária a distribuição de configurações para todos os *switches*, individualmente, de acordo com a sua função/posição dentro da topologia. Por exemplo, nos *switches* de acesso seria preciso gerar configurações para as portas físicas habilitando-as para operar no padrão 802.1Q (VLAN Tagging[12]), operando com IDs individuais; gerar configurações de *trunks* nos *switches* de Distribuição e por fim, mais um *trunk* nos *switches* de *Core*. Traduzindo isto para uma linguagem padrão, como o Cisco IOS, o volume total de interações com todos os *switches* e o volume de comandos submetidos poderia ser resumido conforme segue:

```
# CONFIGURACAO SWITCH CORE-1
interface range GigabitEthernet0/0-23
 description SERVERS
 switchport access vlan 101
 switchport mode access
!
interface range GigabitEthernet1/0-1
 description DOWN-TO-DIST
 switchport mode trunk
!

# CONFIGURACAO SWITCH DISTRIBUTION-1
interface range GigabitEthernet0/1-12
 description CORE-X-ACCESS
 switchport mode trunk
!

# CONFIGURACAO SWITCH DISTRIBUTION-2
interface range GigabitEthernet0/1-12
 description CORE-X-ACCESS
 switchport mode trunk
!


# CONFIGURACAO SWITCH ACCESS-LAN1
interface range GigabitEthernet0/0-23
 description FINANCEIRO
 switchport access vlan 1001
 switchport mode access
!
interface range GigabitEthernet1/0-1
 description UP-TO-DIST
 switchport mode trunk
!
```

```
# CONFIGURACAO SWITCH ACCESS-LAN2
interface range GigabitEthernet0/0-23
 description DIRETORIA
 switchport access vlan 1002
 switchport mode access
!
interface range GigabitEthernet1/0-1
 description UP-TO-DIST
 switchport mode trunk
!


# CONFIGURACAO SWITCH ACCESS-LAN3
interface range GigabitEthernet0/0-23
 description COMERCIAL
 switchport access vlan 1003
 switchport mode access
!
interface range GigabitEthernet1/0-1
 description UP-TO-DIST
 switchport mode trunk
!


# CONFIGURACAO SWITCH ACCESS-LAN4
interface range GigabitEthernet0/0-23
 description JURIDICO
 switchport access vlan 1004
 switchport mode access
!
interface range GigabitEthernet1/0-1
 description UP-TO-DIST
 switchport mode trunk
!
```

No bloco de configurações necessários para cada *switch* membro da topologia, podemos evidenciar a necessidade de se especificar informações como intervalo de portas (`interface range`), identificação da rede virtual (`switchport access vlan <número>`) e modo de operação dos intervalos de portas (`switchport mode <trunk/access>`). Essas informações são

[12] http://www.ieee802.org/1/pages/802.1Q-2014.html

fundamentais e o mínimo necessário para se chegar ao objetivo de segmentação da topologia. Nota-se também o cuidado e atenção necessários na identificação das portas de *trunk* que compõem o *backbone* dessa topologia, bem como as identificações individuais de cada intervalo de portas de cada *switch* de Acesso.

Em uma SDN, utilizando a mesma topologia e objetivos de segmentação da rede, o esforço aplicado muda consideravelmente: 1) a autodescoberta da topologia via LLDP, desempenhada pela Função de Rede Virtualizada (NFV) **`l2switch`**, automaticamente se encarrega de mapear as portas designadas e suas vizinhanças; 2) a segmentação das LANs será feita (na estratégia adotada neste exemplo) apenas na camada de Distribuição composta por dois *switches*, dispensando a necessidade de configurações individualizadas para os sete *switches* da topologia. Inicialmente, logo após a NFV **`l2switch`** mapear toda a topologia, podemos evidenciar que todos os *hosts* são capazes de se comunicar entre si (via **`pingall`**):

```
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2 h3 h4 h5 h6 h7 h8 h9 h10
h2 -> h1 h3 h4 h5 h6 h7 h8 h9 h10
h3 -> h1 h2 h4 h5 h6 h7 h8 h9 h10
h4 -> h1 h2 h3 h5 h6 h7 h8 h9 h10
h5 -> h1 h2 h3 h4 h6 h7 h8 h9 h10
h6 -> h1 h2 h3 h4 h5 h7 h8 h9 h10
h7 -> h1 h2 h3 h4 h5 h6 h8 h9 h10
h8 -> h1 h2 h3 h4 h5 h6 h7 h9 h10
h9 -> h1 h2 h3 h4 h5 h6 h7 h8 h10
h10 -> h1 h2 h3 h4 h5 h6 h7 h8 h9
*** Results: 0% dropped (90/90 received)
```

Foi então construída uma linha de código para distribuição de regras de fluxo nos *switches* S2 e S5, relacionando endereços de origem (**`$src`**) e destino (**`$dst`**), mais uma ação do tipo **`drop`** (descartar pacote). O resultado final foi uma única linha de comando contendo um laço de repetição triplo conforme segue:

```
for switch in s2 s5; do for src in {1..8}; do for dst in {1..8}; do ovs-ofctl add-flow
$switch priority=233,dl_type=0x0800,nw_src=10.0.0.$src,nw_dst=10.0.0.$dst,action=drop;
done; done; done
```

A matriz resultante do **`pingall`**, depois de aplicadas as regras pelo comando acima, ficou da seguinte maneira:

```
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2 X X X X X h9 h10
h2 -> h1 X X X X X h9 h10
h3 -> X X h4 X X X h9 h10
h4 -> X X h3 X X X h9 h10
h5 -> X X X X h6 X h9 h10
h6 -> X X X X h5 X h9 h10
h7 -> X X X X X X h8 h9 h10
h8 -> X X X X X X h7 h9 h10
h9 -> h1 h2 h3 h4 h5 h6 h7 h8 h10
h10 -> h1 h2 h3 h4 h5 h6 h7 h8 h9
*** Results: 53% dropped (42/90 received)
```

É importante ressaltar aqui, neste exemplo de configuração aplicada a esta SDN, que não foi necessária a aplicação de 802.1Q (VLAN Tagging), a configuração do modo de operação das interfaces ou sequer ter conhecimento dos intervalos de portas a serem utilizados em cada equipamento de rede. Toda segmentação da rede foi executada a partir da mesma console e utilizando, na prática, apenas uma única linha de comando em Shell.

A lógica utilizada neste exemplo poderia ser mais refinada: se analisarmos a execução do laço de repetição que injeta as regras de descarte de pacotes com base nos endereços de origem e destino, temos algumas regras de fluxo redundantes ou que nunca terão uma ocorrência com a qual associar. Como exemplo podemos citar a primeira interação do laço de repetição, que iniciará pelo *switch* S2 injetando uma regra de *drop* para pacotes com origem no IP 10.0.0.1 e destino para o próprio IP 10.0.0.1, 10.0.0.2 até o 10.0.0.8 - como a regra foi aplicada nos *switches* da camada de Distribuição, a regra para bloquear a comunicação entre os *hosts* 10.0.0.1 e 10.0.0.2 não chega a ser acionada. Na prática o laço de repetição, se aplicado na camada de Acesso, faria o bloqueio total da comunicação entre todos os *hosts* da topologia. Uma "sanitização" das regras aplicadas nos *switches* de Distribuição poderia ser posteriormente efetuada utilizando Shellscript, em um laço de repetição para remoção das regras utilizando o comando **`ovs-ofctl del-flows`**:

```
for x in {1..8}; do ovs-ofctl del-flows s5 dl_type=0x0800,nw_src=10.0.0.$x,nw_dst=10.0.0.$x;
done
```

Após aplicar o comando acima o número de regras instaladas no *switch* S5 reduziu de 64 para 56 (exatamente 8 regras onde origem = destino). Scripts mais elaborados, utilizando condicionais, podem ser usados para a eliminação de determinadas regras em *switches* específicos.

3.2. DISPONIBILIZANDO O OPENDAYLIGHT

Conforme já fora descrito na seção 2.3, o OpenDayLight é um controlador escrito em Java e sua implementação utilizando apenas um nó (sem redundância por meio de *clustering*) é bastante simples. Por conta de mudanças recentes ainda em desenvolvimento na interface gráfica do controlador (de ODL DLUX para Cisco NeXt), foi feita a opção pela utilização de um *release* anterior, porém mais amigável e que fosse capaz de desenhar as topologias sem falhas de renderização. Primeiramente é necessário instalar o Java, sendo a forma mais simples e eficiente de fazê-lo por meio do pacote instalador disponível no repositório do sistema operacional:

```
$ sudo apt-get install oracle-java7-installer
$ sudo update-alternatives --config java
$ sudo echo "JAVA_HOME=/usr/lib/jvm/java-7-oracle" >> /etc/environment
$ sudo source /etc/environment
```

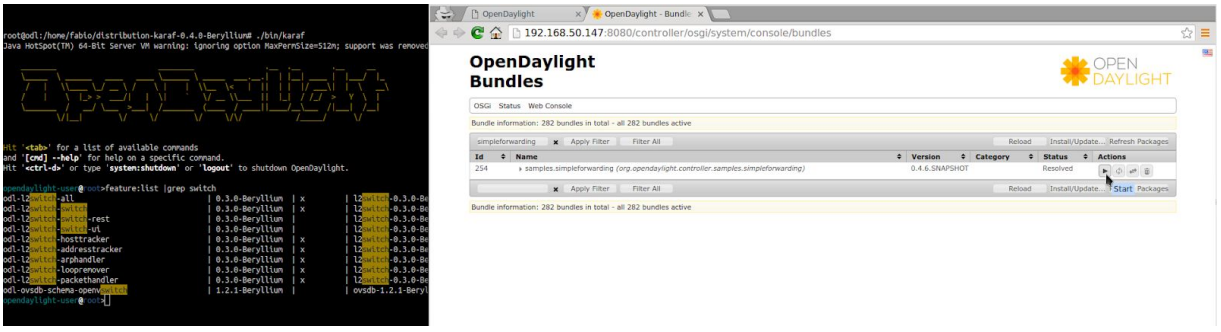
Efetuar o download do release do controlador em questão, descompactar e executá-lo:

```
$ wget
https://nexus.opendaylight.org/content/groups/public/org/opendaylight/integration/distribution-karaf/0.2.3-Helium-SR3/distribution-karaf-0.2.3-Helium-SR3.zip
$ unzip distribution-karaf-0.2.3-Helium-SR3.zip
$ cd opendaylight
$ sudo ./run.sh
```

Feito isso o controlador passará a escutar na porta 6633 por anúncios (*multicast*) emitidos por dispositivos de rede e responderá (ou não) com a atribuição de regras utilizando OpenFlow; isso dependerá de como o controlador SDN está configurado para atender aos anúncios daquele tipo de dispositivo, que podem ser de duas formas: 1) **reativa** - as entradas de fluxo na *FlowTable* são criadas apenas quando o *switch* recebe um pacote no qual não possua uma regra de fluxo corresponde para processamento ou ; 2) **proativa** - as *FlowTables* dos *switches* são atualizadas antes de uma ocorrência de pacote desconhecido, poupando o dispositivo do *delay* de configuração da regra e retenção do pacote até o recebimento da definição, mitigando um possível problema de escalabilidade da solução (MORREALE & ANDERSON, 2014).

Com relação as funcionalidades e aplicações de rede, por padrão, o ODL inicializa sem nenhuma função ativa. Uma lista de aplicações disponíveis para ativação pode ser consultada tanto via terminal como por meio da interface Web:

Figura 7 - Listagem de aplicações via CLI (esquerda) ou GUI (direita).



Fonte: Produção do próprio autor.

Via GUI a ativação/desativação pode ser feita simplesmente com um clique; já via CLI pode ser usado o comando **`feature:install`** e especificando todas funcionalidades que se deseja habilitar. Nos testes foi habilitada a interface DLUX e a aplicação de *switching*[13]. O comando **`feature:list`** pode ser usado para listar todas as funcionalidades ativas passando o parâmetro "**`-i`**"; a ausência desse parâmetro irá listar todas as funcionalidades disponíveis incluindo as inativas:

```
opendaylight@root> feature:install odl-dlux-all odl-l2switch-all
opendaylight@root> feature:list -i
```

[13] https://wiki.opendaylight.org/view/L2_Switch:Main

A relação completa de funcionalidades e links para os códigos fontes das aplicações disponíveis pode ser consultada no site oficial do ODL[14].

4. CONSIDERAÇÕES FINAIS

Não há no mercado até o momento, entre os grandes fabricantes de soluções para telecomunicações, um formato amigável e consumível desta tecnologia para uso geral: *switches* compatíveis com o protocolo OpenFlow estão restritos ainda a muito poucos modelos e parece não haver, pelo menos entre os grandes fabricantes, um interesse tão imediato para que essa tecnologia se popularize. Alguns poucos fabricantes, como por exemplo a Mikrotik, disponibilizam atualização para o seu RouterOS adicionando para qualquer linha ou modelo de seus produtos o suporte ao protocolo OpenFlow, via uma simples instalação de pacote[15].

Nas experimentações com o protocolo OpenFlow onde fora utilizado ShellScript para a interação direta com os ativos de rede, fora do controlador SDN, isso foi assim feito por conta da inexistência de uma NFV ou aplicação para o controlador utilizado que permitisse imputar de maneira rápida, simples e direta as regras de fluxos. Nos manuais para desenvolvedores do controlador utilizado, a injeção de regras de fluxo se dá via linguagem XML[16] utilizando como interface uma REST API aberta na porta 8080 ou 8181 (dependendo da versão utilizada). A interação direta com o protocolo e os ativos de rede se mostrou esclarecedora no que diz respeito a facilidade e esforço reduzidos para a configuração de redes e definições de políticas de encaminhamento de pacotes.

No que diz respeito a introdução de um controlador responsável pela centralização de todas as regras e funcionalidades dos ativos de rede, isso se mostrou uma solução interessante, ao passo que logicamente pode prolongar a utilização dos equipamentos (novas funcionalidades ou suporte a novas tecnologias passam a ser abstrações na forma de softwares instalados no controlador). Questões relacionadas a escalabilidade, desempenho e segurança do controlador e do modelo de operação vêm sendo exaustivamente explorados no meio acadêmico e suportados pela comunidade de software livre e por alguns segmentos da indústria, e ao que tudo indica são o próximo passo na evolução natural das redes de computadores como as conhecemos.

[14] https://wiki.opendaylight.org/view/Project_list
[15] http://wiki.mikrotik.com/wiki/Manual:System/Packages
[16] https://wiki.opendaylight.org/view/OpenDaylight_OpenFlow_Plugin:End_to_End_Flows

5. REFERENCIAL TEÓRICO

NADEAU, Thomas D.; GRAY, Ken. **SDN: Software Defined Networks**. 1. ed. 7 ago 2013. O'Reilly Media Inc. E-book.

HU, Fei. **Network Innovation through OpenFlow and SDN: Principles and Design**. 1. ed. 7 dez 2013. CRC Press. E-Book.

MORREALE, Patricia A; ANDERSON, James M. **Software Defined Networking: Design and Deployment**. 1. ed. 03 jul 2014. CRC Press. E-Book.

Open Networking Foundation. **OpenFlow Switch Specifications 1.4.0.** Disponível em <https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.4.0.pdf>. Acessado em: 10 mar. 2016.

OpenDayLight Project. **OpenDayLight: Installing, running and usage.** Disponível em <https://wiki.opendaylight.org/view/Main_Page>. Acessado em: 25 jan. 2016.

LANTZ, Bob; HELLER, Brandon; MCKEOWN, Nick. **A Network in a Laptop: Rapid Prototyping for Software-Defined Networks**. Disponível em <http://conferences.sigcomm.org/hotnets/2010/papers/a19-lantz.pdf>. Acessado em: 21 mai. 2016.