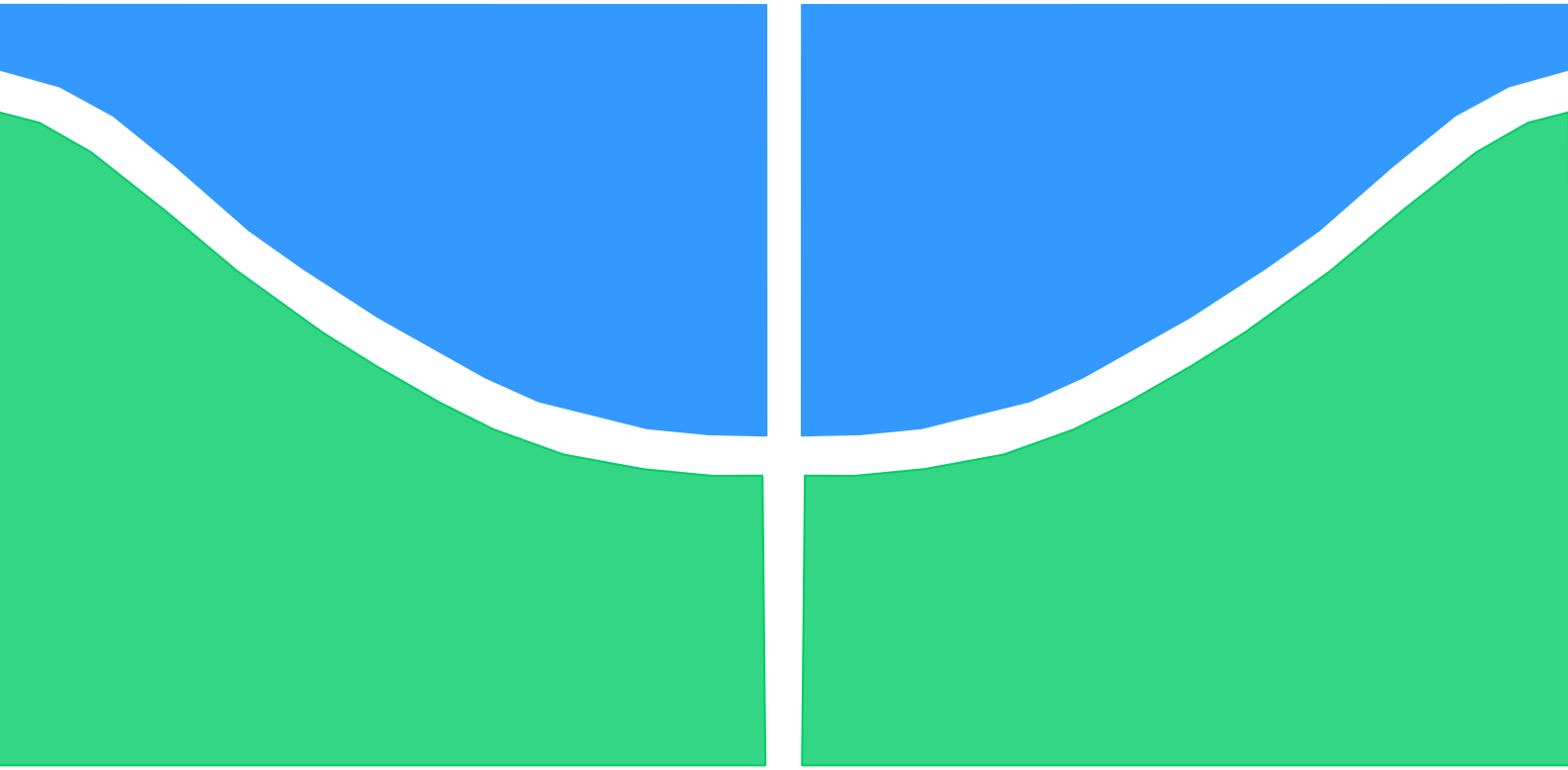


TRABALHO DE GRADUAÇÃO

DESCOBERTA DE PADRÕES DE FLUXO EM REDES DEFINIDAS POR SOFTWARE (SDN) PARA OTIMIZAÇÃO DO USO EFETIVO DO CANAL DE CONTROLE

Igor Cezar da Silva

Brasília, dezembro de 2017

UNIVERSIDADE DE BRASILIA

FACULDADE DE TECNOLOGIA

UNIVERSIDADE DE BRASILIA
Faculdade de Tecnologia

TRABALHO DE GRADUAÇÃO

DESCOBERTA DE PADRÕES DE FLUXO EM REDES DEFINIDAS POR SOFTWARE (SDN) PARA OTIMIZAÇÃO DO USO EFETIVO DO CANAL DE CONTROLE

Igor Cezar da Silva

Relatório submetido ao Departamento de Engenharia Elétrica como requisito parcial para obtenção do grau de Engenheiro de Redes de Comunicação

Banca Examinadora

Prof. William Ferreira Giozza, ENE/UnB
Orientador

Prof. Valério Aymore Martins, ENE/UnB
Co-orientador

Prof. Georges Daniel A. Nze, ENE/UnB
Examinador interno

Dedicatória

Primeiramente a Deus, que nunca me abandonou e nunca me abandonará, e sem o qual não estaria aqui. A minha família, pelo apoio e incentivo. A minha noiva, pela ajuda, suporte e compreensão. E a todos em busca do conhecimento por esse novo mundo de Redes Definidas por Software.

Igor Cezar da Silva

RESUMO

A abordagem centralizada da lógica da rede, utilizada pelo protocolo OpenFlow, vem sendo discutida principalmente sobre a ocorrência de sobrecarga no canal de controle. Em relação à alternativa usual de distribuir a lógica da rede sobre os dispositivos de encaminhamento, existe uma maior preocupação nessa abordagem centralizada para evitar a sobrecarga de mensagens no controlador. O protocolo Openflow especifica apenas quais e como são as mensagens de controle, porém não define melhores práticas de utilização dessas mensagens para o controle e monitoramento da rede, assim sem nenhum compromisso com o desempenho global da rede. Entende-se, então, que é necessária a análise dos reais impactos das mensagens de controle e monitoramento geradas pelo protocolo OpenFlow. O ponto focal dessa análise é a classificação do nível de impacto das mensagens na sobrecarga do canal de controle em uma rede baseada em SDN (*Software Defined Networking*) e OpenFlow. O projeto em questão irá realizar uma análise quantitativa do tráfego de controle e monitoramento em redes OpenFlow e, assim, fornecer subsídios para construção de soluções no uso efetivo do canal de controle, levando a uma melhor compreensão do impacto real da utilização do protocolo OpenFlow. Essa compreensão é essencial para uma melhor projeção de novos sistemas de gerenciamento de redes baseadas no paradigma SDN. Para a obtenção dos resultados, será realizada uma experimentação considerando aspectos de instalação de regras, utilizando o controlador POX, além de estatísticas de fluxos instalados e diferentes frequências de monitoramento. Adicionalmente, é de intenção desse trabalho realizar tal análise sob diferentes cenários e diferentes frequências de monitoramento, com variações das topologias utilizadas e de configurações específicas do controlador, bem como identificar como se pode evitar a sobrecarga dos modelos pelo excesso de monitoramento, e o impacto na taxa de transferência e na quantidade de mensagens geradas no canal de controle.

Palavras-chave: OpenFlow, SDN, Redes Definidas por Software, POX, sobrecarga do canal de controle, monitoramento, Mininet

ABSTRACT

The centralized approach of the network logic, used by the OpenFlow protocol, has been discussed mainly on the occurrence of overhead in the control channel. Regarding the usual alternative of distributing network logic over routing devices, there is a greater concern in this centralized approach to avoid message overhead in the controller. The Openflow protocol specifies only which and how the control messages are, but does not define the best practices for the use of these messages for network control and monitoring, without any commitment to the overall performance of the network. It is understood, then, that it is necessary to analyze the real impacts of control and monitoring messages generated by the OpenFlow protocol. The focal point of this analysis is the classification of the message impact level in the control channel overhead in a network based on Software Defined Networking (SDN) and OpenFlow. The project in question will perform a quantitative analysis of control and monitoring traffic in OpenFlow networks and thus provide subsidies for building solutions in the effective use of the control channel, leading to a better understanding of the real impact of using the OpenFlow protocol. This understanding is essential for better projection of new network management systems based on the SDN paradigm. To obtain the results, an experiment will be performed considering rules installation aspects, using POX controller, as well as statistics of installed flows and different monitoring frequencies. In addition, it is the intention of this work to perform such analysis under different scenarios and different monitoring frequencies, with variations of the topologies used and specific configurations of the controller, as well as to identify how to avoid the overload of the models by the excessive monitoring, and the impact in the transfer rate and the number of messages generated in the control channel.

Keywords: OpenFlow, SDN, Software Defined Networking, POX, control channel overload, monitoring, Mininet

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE QUADROS

LISTA DE ABREVIATURAS

API	*Application Programming Interface*
ARP	*Address Resolution Protocol*
BGP	*Border Gateway Protocol*
DHCP	*Dynamic Host Configuration Protocol*
DIFANE	*DIstributed Flow Architecture for Networks Enterprises*
DoS	*Denial of Service*
HTTP	*Hypertext Transfer Protocol*
LRU	*Last Recently Used*
MPLS	*Multi-Protocol Label Switching*
NAT	*Network Address Translation*
ONF	*Open Networking Foundation*
OSPF	*Open Shortest Path First*
OVS	*Open Virtual Switch*
OXM	*OpenFlow Extensible Match*
QoS	*Quality of Service*
REST	*Representation State Transfer*
RTP	*Real-time Transport Protocol*
SDN	*Software-Defined Networking*
SNMP	*Simple Network Management Protocol*
TCAM	*Ternary Content Addressable Memory*
TCP	*Transmission Control Protocol*
UDP	*User Datagram Protocol*
VLAN	*Virtual Local Area Network*
VLC	*VideoLan Client*

Capítulo 1

Introdução

1.1 Contextualização

Software Defined Networking (SDN) é um paradigma de redes que ganhou muita atenção da indústria e do meio acadêmico nos últimos anos. Sua arquitetura separa o plano de controle do plano de dados, movendo a função de controle da rede para um elemento controlador central [1,2], que coordena as definições de encaminhamento dos demais dispositivos de rede, como roteadores e *switches*, transformando-os em simples encaminhadores de pacotes [3]. Esta separação provê uma visão centralizada e unificada do estado da rede, simplificando sua operação e gerenciamento.

Nesse paradigma, ao contrário da implementação em *hardware* das redes convencionais, a lógica de rede é implementada em uma camada de *software*, que é totalmente programável [3]. Desta forma, toda sua estrutura pode estar construída em um único equipamento físico, reduzindo custos de infraestrutura. Essas características criam um grande escopo de inovações na maneira como as redes são programadas e gerenciadas, pois permitem uma implementação de forma mais simples e ágil de diversos tipos de aplicações e serviços, como *firewalls*, algoritmos de roteamento, modelamento de tráfego, entre outros [4]. A Figura 1.1 mostra a comparação entre a rede tradicional e uma rede SDN.

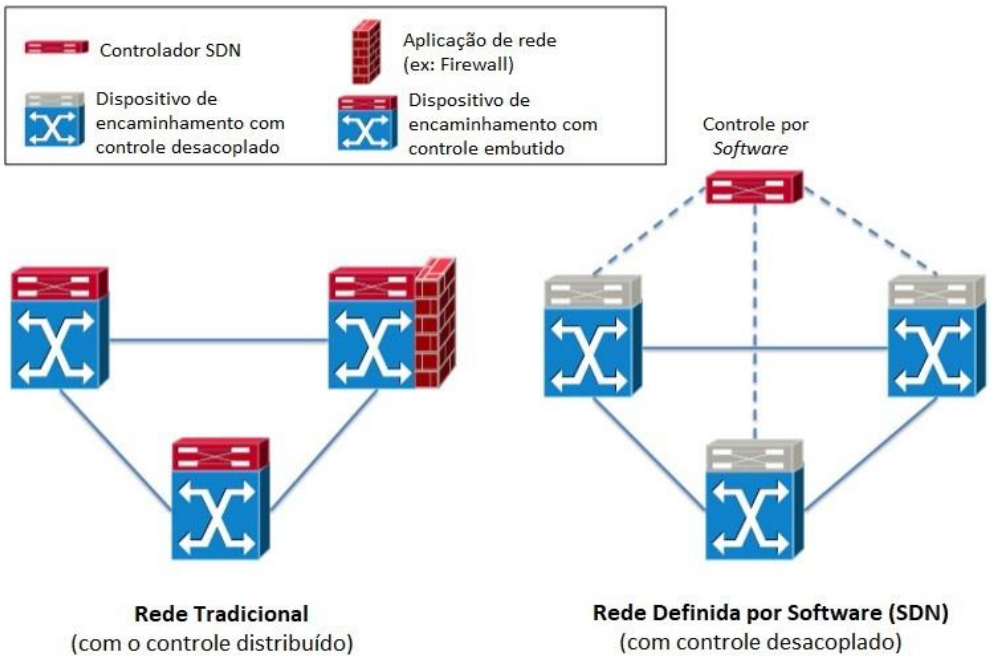


Figura 1.1 – Comparação entre as estruturas da rede tradicional e rede SDN [5]

Com a especificação do protocolo OpenFlow, que é o protocolo de comunicação que permite a separação dos planos de controle e de dados no paradigma SDN [1], as redes baseadas nesse paradigma ganharam maior destaque. Assim, a lógica da rede é concentrada em um controlador central, que é responsável por toda a inteligência e monitoramento da rede, o que reduz problemas de gerenciamento comuns em redes convencionais. Além disso, permite regular o estado da rede através da criação de políticas, que podem ser facilmente implantadas pelo elemento controlador.

O elemento controlador utiliza mensagens de controle e de monitoração, definidas pelo protocolo OpenFlow, na comunicação com os dispositivos de rede, através de um canal seguro, chamado de canal de controle, para definir as políticas e operações a serem adotadas no tratamento de pacotes, além de monitorar o estado da rede.

A abordagem centralizada da lógica da rede, utilizada pelo protocolo OpenFlow, vem sendo discutida principalmente da ocorrência de sobrecarga no canal de controle. Existe uma preocupação na definição de soluções para evitar a sobrecarga de mensagens no controlador, em relação à alternativa usual de distribuir a lógica da rede sobre os dispositivos de encaminhamento [4]. São desenvolvidas alternativas para distribuir a lógica da rede, como por exemplo, estender a inteligência da rede para múltiplos controladores, visando evitar um possível gargalo de mensagens no canal de controle [4,6].

Apesar do desenvolvimento de soluções para redução de sobrecarga [7,8], análises dos impactos reais causados pelas mensagens de controle e monitoramento geradas pelo protocolo OpenFlow ainda são necessárias. Esta análise é de suma importância para a projeção de novas formas de gerenciamento de redes de maneira eficiente e escalável, com maiores níveis de desempenho e complexidade, baseadas no paradigma SDN.

1.2 Definição do problema

Diferente das redes tradicionais, na arquitetura do paradigma SDN, onde o plano de controle está separado do plano de dados, o problema da confiabilidade se torna mais crítico no elemento controlador do que nos *switches*, por ser o controlador o responsável pela inteligência e pela decisão das políticas de toda a rede [9]. O aumento do tráfego dos diversos tipos de mensagens entre o controlador e os *switches* pode resultar em uma sobrecarga do canal de controle, comprometendo sua disponibilidade e levando à queda de desempenho da rede, ou mesmo à interrupção da comunicação.

O protocolo Openflow especifica apenas quais e como são as mensagens de controle, a exemplo das mensagens *packet-in*, porém não define melhores práticas de utilização dessas mensagens para o controle e monitoramento da rede sem comprometer seu desempenho. Neste âmbito, é necessária a análise dos reais impactos das mensagens de controle e monitoramento gerados pelo protocolo OpenFlow, e principalmente obter a classificação do nível de impacto das mensagens na sobrecarga do canal de controle em uma rede baseada em SDN e OpenFlow.

É fundamental a obtenção desses dados para possibilitar uma melhor projeção de novos sistemas de gerenciamento de redes baseadas no paradigma SDN, viabilizando melhores soluções de controle e monitoramento, sem a qual pode limitar a capacidade de construção de redes mais eficientes e flexíveis para as diversas necessidades de desempenho e confiabilidade.

1.3 Hipótese

Existem diversos tipos de mensagens definidas no protocolo OpenFlow para comunicação entre os dispositivos, onde todas impactam no tráfego do canal de controle, sejam grandes ou pequenos esses impactos. Imagina-se que existam mensagens que são utilizadas com mais frequência pelo protocolo, devido à necessidade de controle e monitoração dos *switches* pelo controlador. As possíveis mensagens que podemos inferir ser mais utilizadas são as de requisição e instalação de regras de fluxo do tipo *Packet-In, Send-Packet* e *Modify-State*, e as de coleta de estatística do tipo *Read-State*, pois são as principais para o controle básico e monitoração dos *switches.* Segue a descrição das mensagens [10]:

- *Packet-In*: enviada pelo switch para o controlador quando ocorre *table-miss* ou quando a regra for definida para isto.
- *Send-Packet*: utilizada pelo controlador para forçar o encaminhamento de um pacote que não possua regra definida, quando não é possível inserir uma nova regra na tabela (e.g. tabela está cheia);
- *Modify-State*: gerencia o estado dos switches, geralmente utilizada para a manutenção de fluxos nas tabelas e alterar as propriedades de portas;
- *Read-State*: utilizada para coletar estatísticas sobre tabelas, portas, fluxos e filas dos switches.

Para validar a questão das mensagens mais relevantes, será realizado inicialmente um cenário preliminar, mais simples, para melhor análise do tráfego das mensagens de controle e monitoramento, e assim definir as que realmente possuem maior frequência nesse tráfego.

1.4 Objetivos do projeto

Este projeto tem como objetivo realizar uma análise quantitativa do tráfego de controle e monitoramento em redes baseadas no protocolo OpenFlow, e assim, fornecer subsídios para construção de soluções para o uso efetivo do canal de controle. Isto leva a uma melhor compreensão do impacto real da utilização do protocolo OpenFlow, que pode causar sobrecarga do canal. Com isto, podemos implementar uma melhor configuração de rede e de parâmetros do controlador, além de melhor orientar na escolha das topologias e estrutura de rede a serem utilizadas.

A partir desta análise, o intuito será focar nas mensagens de maior relevância para, através de alterações simples, diminuir sua frequência e, por consequência, seu impacto na sobrecarga do canal.

No quesito de monitoramento, será analisado o impacto de se monitorar essas redes, pois as mensagens de monitoramento contribuem para o aumento da carga e da quantidade de mensagens trafegando no canal de controle. Desta forma, o objetivo é obter dados que possibilitem definir um melhor nível de granularidade e frequência de monitoramento que leve ao menor impacto possível no canal de controle.

1.5 Contribuições do projeto

As principais contribuições deste trabalho são:

- Validação da hipótese de quais são as mensagens relevantes no tráfego do canal de controle de redes SDN.
- Avaliação do impacto das mensagens de controle na configuração e manutenção dos dispositivos de encaminhamento, avaliados nas diferentes topologias e configurações específicas do controlador.
- Avaliação do impacto das mensagens de monitoramento dos dispositivos de encaminhamento, avaliados nas diferentes topologias, configurações específicas do controlador e diferentes frequências de monitoramento.

- Especificação do nível de ociosidade de regras instaladas na tabela de fluxos dos dispositivos de encaminhamento para diferentes valores de tempo de ociosidade máximo – *idle timeout*.
- Disponibilização de componente para monitoramento e coleta de estatísticas do canal de controle e dispositivos de rede no controlador POX, de autoria própria.
- Fornecer subsídios para identificar as melhores configurações de rede e topologia baseadas no paradigma SDN para as diferentes abordagens.
- Fornecer subsídios para identificar as melhores configurações do controlador de rede baseadas no paradigma SDN para as diferentes abordagens.
- Fornecer subsídios para definir modelos de alternativas de nível de granularidade de um possível monitoramento periódico da rede, visando não sobrecarregar a rede.

1.6 Metodologia

Neste projeto será utilizada uma metodologia de pesquisa experimental, com uma abordagem quantitativa, realizando simulações de redes baseadas no paradigma SDN. Serão considerados diferentes aspectos de instalação de regras nos controladores, diferentes configurações dos elementos e de topologia da rede. A pesquisa da fundamentação teórica e de trabalhos correlacionados irão fornecer o referencial de base para conduzir os experimentos, assim como o alinhamento com os resultados que devem ser esperados deles.

Dos aspectos metodológicos, experimentos serão realizados no emulador de rede Mininet, em conjunto com o virtualizador Virtual Box, com a preparação do ambiente conforme os passos a seguir:

1. Download da máquina virtual que contém o Mininet instalado no sistema operacional Ubuntu, através do site http://mininet.org;
2. Inclusão da máquina virtual do Mininet no Virtual Box e configuração de parâmetros de rede, memória, sistema operacional, entre outros;
3. Atualização de versão do *Open vSwitch* para a versão 2.7.2;
4. Instalação do reprodutor de vídeo VLC, que será utilizado como servidor de vídeo;

5. Configuração do controlador POX, com análise dos componentes a serem utilizados para os experimentos;
6. Criação de um novo componente para o controlador POX, na linguagem Python, para coleta de dados e estatísticas do controlador através de mensagens *Read-State,* pois não existe nativamente.
7. Criação de *script* na linguagem Python para realização do experimento de forma automatizada, onde realizará a inicialização do Mininet nas configurações de topologia e parâmetros de rede desejados, e executará o experimento com a geração do tráfego na rede nos tempos e intervalos definidos. Isso permite que os experimentos sejam realizados da mesma forma, sem erros humanos ou subjetividade.

1.6.1 Etapas de simulação

Serão realizadas as seguintes etapas de simulação, visando a obtenção de dados em condições similares para melhor comparação, conforme segue:

1. Criação de um cenário preliminar simplificado, com um número reduzido de estações e *switches,* para melhor verificação do tráfego de mensagens no canal de controle;
2. Inicialização do POX como controlador, com os devidos componentes escolhidos para serem utilizados nos experimentos;
3. Geração de tráfego de dados e observação do fluxo de mensagens entre o controlador e *switches*, seguindo métricas pré-definidas.
4. Obtenção das informações da quantidade de mensagens trafegadas no canal de controle e criação de uma tabela com as estatísticas de cada uma delas, de acordo com a direção da comunicação;
5. Análise do experimento preliminar para definição dos melhores parâmetros e implementações a serem utilizadas para os demais experimentos, em maior escala.
6. Criação de cenários mais complexos, utilizando as topologias estrela, linear e árvore, para os demais experimentos.
7. Geração de tráfego de dados e observação do fluxo de mensagens entre o controlador e *switches*, seguindo métricas pré-definidas, além dos parâmetros obtidos no estudo preliminar.

8. Obtenção das informações da quantidade e da carga das mensagens trafegadas no canal de controle, para geração de gráficos e análise dos impactos gerados em cada topologia.
9. Análise dos experimentos e comparação dos dados obtidos para cada topologia.
10. Alteração da frequência de monitoramento para análise do impacto das mensagens do tipo *Read-State* no canal de controle em contraste com o nível de granularidade de monitoramento.
11. Realização dos experimentos novamente para cada topologia, com os novos valores de frequência de monitoramento.
12. Análise do impacto do monitoramento e definição de melhores opções para um monitoramento de baixo impacto, comparando os dados obtidos para cada frequência.

1.6.2 Métricas

Serão avaliados quantitativamente os tráfegos específicos de controle e monitoramento da rede baseada no protocolo OpenFlow, considerando as seguintes métricas:

1. Quantidade de mensagens do tipo *Packet-In* processadas pelo controlador, por segundo;
2. Carga trafegada no canal de controle para mensagens do tipo *Packet-In* entre os dispositivos de encaminhamento e o controlador, em *bits* por segundo;
3. Quantidade de mensagens do tipo *Modify-State* processadas pelos dispositivos de encaminhamento, por segundo;
4. Carga trafegada no canal de controle para mensagens do tipo *Modify-State* entre o controlador e os dispositivos de encaminhamento, em *bits* por segundo;
5. Carga trafegada no canal de controle para mensagens do tipo *Read-State* entre o controlador e os dispositivos de encaminhamento, em *bits* por segundo;

Através dessas métricas, será avaliada a quantidade de mensagens enviadas em função do tempo, nas duas direções de comunicação entre o *switch* e o controlador. Essas métricas serão utilizadas para todos os cenários de simulação definidos, mantendo um padrão de avaliação, permitindo uma comparação entre os resultados obtidos para cada configuração de cenário.

1.7 Ferramentas utilizadas

Segue descrição das ferramentas utilizadas para a realização dos experimentos em redes SDN, propostos para o projeto em questão:

- **Mininet [17]**: um emulador de redes que cria uma rede de controladores, *switches*, *links* e *hosts* virtuais. Suporta o protocolo OpenFlow e a criação de Redes Definidas por *Software* (SDN). É possível definir topologias diversas e criar diferentes cenários para testes e pesquisas.
- **Virtual Box [19]**: é uma ferramenta de virtualização *open-source* que visa criar ambientes para instalação de sistemas distintos, chamadas máquinas virtuais. Ele permite a instalação e utilização de um sistema operacional dentro de outro, assim como seus respectivos *softwares*, como dois ou mais computadores independentes, mas compartilhando fisicamente o mesmo hardware.
- **VLC [20]**: é um reprodutor multimídia livre, de código aberto, multi-plataforma, e um arcabouço que reproduz a maioria dos arquivos de mídia, bem como DVD, CD de áudio, VCD e vários protocolos de fluxo de rede.
- **POX [9]**: é uma plataforma para criação de controladores de rede escrita na linguagem Python, de código aberto, que possui ampla interação com o OpenFlow para ajudar no desenvolvimento de Redes Definidas por *Software*.
- **Wireshark [21]**: é um analisador de pacotes e tráfego de rede, organizando por protocolos, de código aberto e multi-plataforma, que permite que você capture e navegue interativamente no tráfego de uma rede de computadores em tempo de execução.
- **PuTTY [22]**: cliente telnet e SSH desenvolvido para Windows, grátis e de código aberto, para estabelecer conexões seguras de acesso remoto a servidores. Utilizado para acessar remotamente a máquina virtual do Mininet.
- **Máquina física (desktop):** computador onde serão instalados os *softwares* descritos anteriormente, a serem utilizados nos experimentos. Suas configurações serão descritas na parte experimental do projeto.

Capítulo 2

Fundamentação teórica

2.1 Redes Definidas por Software

Apesar do grande crescimento e evolução da Internet, é possível observar que sua arquitetura não evoluiu o suficiente durante os últimos anos, estando relativamente estagnada [11]. Muitas modificações já foram realizadas, porém a alteração arquitetural da Internet já se faz mais necessária a cada dia, pois não atende aos novos conceitos e velocidade de mudanças a que estamos vivenciando no meio acadêmico e no seu uso comercial.

As redes de computadores são construídas, em geral, utilizando elementos de rede como *switches*, roteadores, placas *ethernet*, entre outros, que juntos utilizam protocolos de comunicação para formar a infraestrutura necessária que permite a troca de dados entre computadores e demais dispositivos eletrônicos. Essas redes podem ser organizadas em três planos de funcionalidades: controle, dados e gerência [3], conforme mostra a Fig. 2.1.

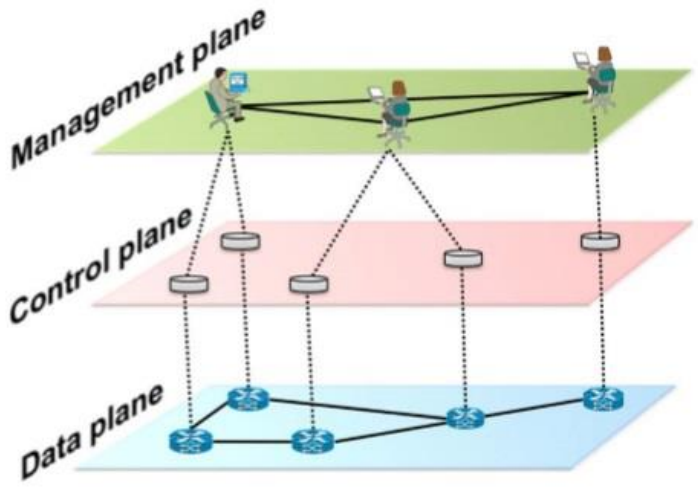


Figura 2.1 – Visão da divisão dos planos de funcionalidade [3]

O plano de controle é representado pelos protocolos que preenchem as tabelas de encaminhamento, definindo o roteamento de pacotes, ou seja, o tráfego de dados, como por exemplo, BGP (*Border Gateway Protocol*), OSPF (*Open Shortest Path First*), entre outros. O plano de gerência contém os serviços de *software*, que realizam o monitoramento e configuração da camada de controle, de maneira remota, como por exemplo o SNMP (*Simple Network Management Protocol*). O plano de dados abrange os equipamentos utilizados para

encaminhamento de pacotes, como roteadores e *switches.* Em resumo, as políticas da rede são definidas no plano de gerência, ao passo que o plano de controle as implementa e o plano de dados as executa, encaminhando o tráfego conforme definido nas políticas.

As redes IP tradicionais são complexas e difíceis de gerenciar [3]. Para se implementar políticas de rede de alto nível, os operadores de rede necessitam configurar cada um dos dispositivos de rede separadamente, usando comandos de baixo nível e muitas vezes comandos específicos definidos pelo próprio fornecedor do equipamento. Além disso, geralmente acaba ocorrendo uma mistura de equipamentos de fornecedores diferentes, onde cada um tem seu próprio sistema e suas próprias particularidades, o que dificulta em muito a manutenção e modificação das estruturas e definições de rede. Desta forma, para implementar novas políticas de rede em um ambiente como este é geralmente um grande desafio [3].

Software Defined Networking (SDN) é um paradigma de redes que ganhou muita atenção da indústria e do meio acadêmico nos últimos anos. Sua arquitetura separa o plano de controle do plano de dados, movendo a função de controle da rede para um elemento controlador central [1], que coordena as definições de encaminhamento dos demais dispositivos de rede, transformando-os em simples encaminhadores de pacotes [3]. Esta separação provê uma visão centralizada do estado da rede e simplifica sua operação e gerenciamento. A Figura 2.2 mostra uma visão simplificada da arquitetura SDN.

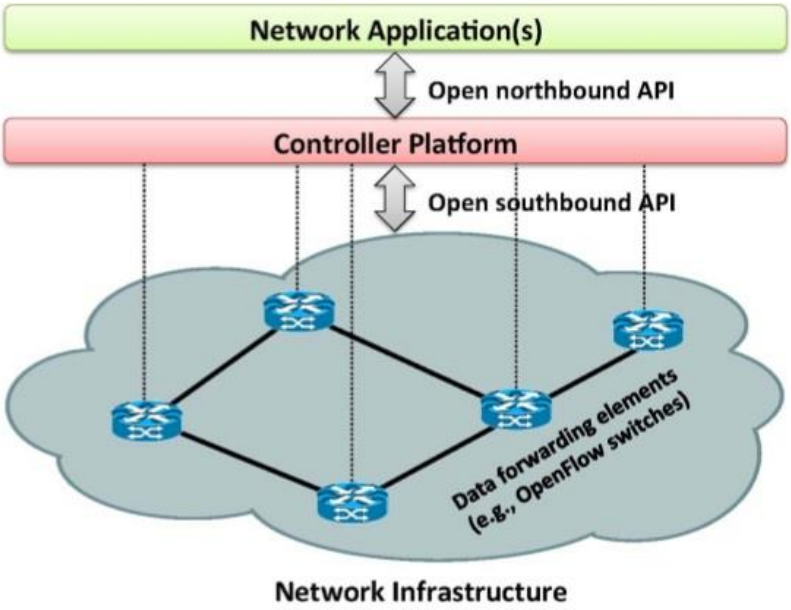


Figura 2.2 – Visão simplificada da arquitetura SDN [3]

Nesse paradigma, ao contrário da implementação em *hardware* das redes convencionais, a lógica da rede é implementada em uma camada de *software*, que é

totalmente programável [3]. A Figura 2.3 mostra uma comparação entre a arquitetura de dispositivos de encaminhamento do modelo atual e do modelo baseado nas redes SDN. Desta forma, toda sua estrutura pode estar construída em um único equipamento físico, reduzindo custos de infraestrutura. Essas características possibilitam grandes oportunidades de inovações na maneira como as redes são programadas e gerenciadas, pois permitem uma implementação de forma mais simples e ágil de diversos tipos de aplicações e serviços, como *firewalls*, algoritmos de roteamento, modelamento de tráfego, entre outros [4].

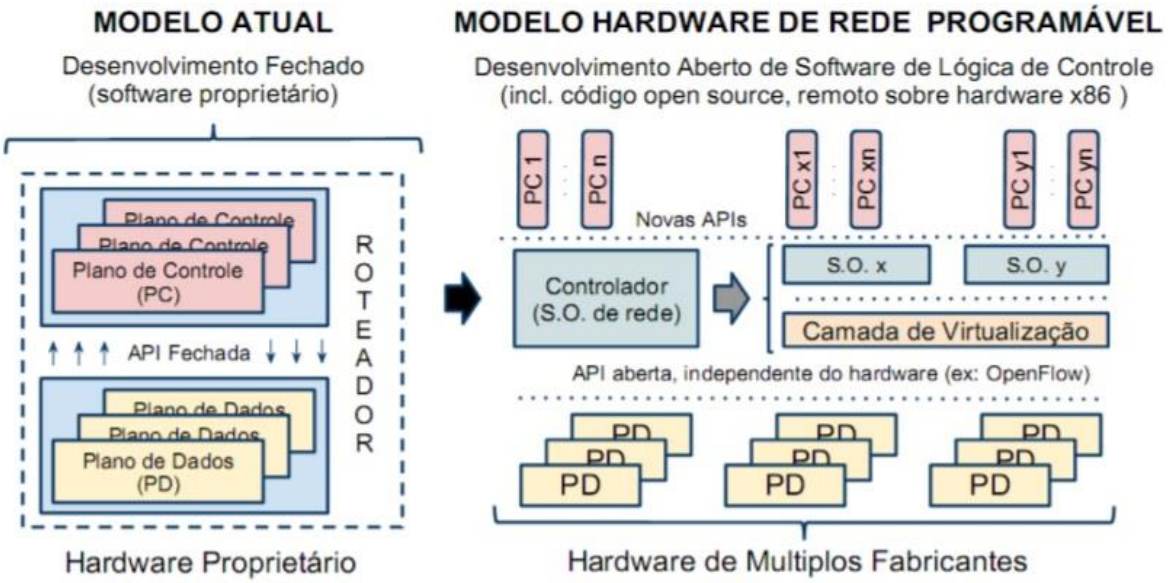


Figura 2.3 – Arquiteturas de roteadores: modelo atual e modelo programável OpenFlow [12]

A Open Networking Foundation (ONF), uma organização sem fins lucrativos a qual estuda, desenvolve e promove as Redes Definidas por Software e a padronização do protocolo OpenFlow, apresenta as redes SDN com quatro planos: gerenciamento, aplicação, controle e dados [13]. Todos esses planos comunicam entre si através de interfaces. A Figura 2.4 mostra os principais componentes e interfaces desta arquitetura.

As interfaces A-CPI e D-CPI são também conhecidas como *northbound* e *southbound* API, respectivamente, conforme mostrado anteriormente na Fig. 2.2. A interface A-CPI estabelece a comunicação bidirecional entre os planos de aplicação e de controle, enquanto a interface D-CPI estabelece a mesma comunicação, porém entre os planos de controle e de dados. O mais indicado, ou seja, o ideal, é que essas interfaces sejam padronizadas, para que possam trabalhar com os diversos tipos de equipamentos e tecnologias diferentes sem dificuldades.

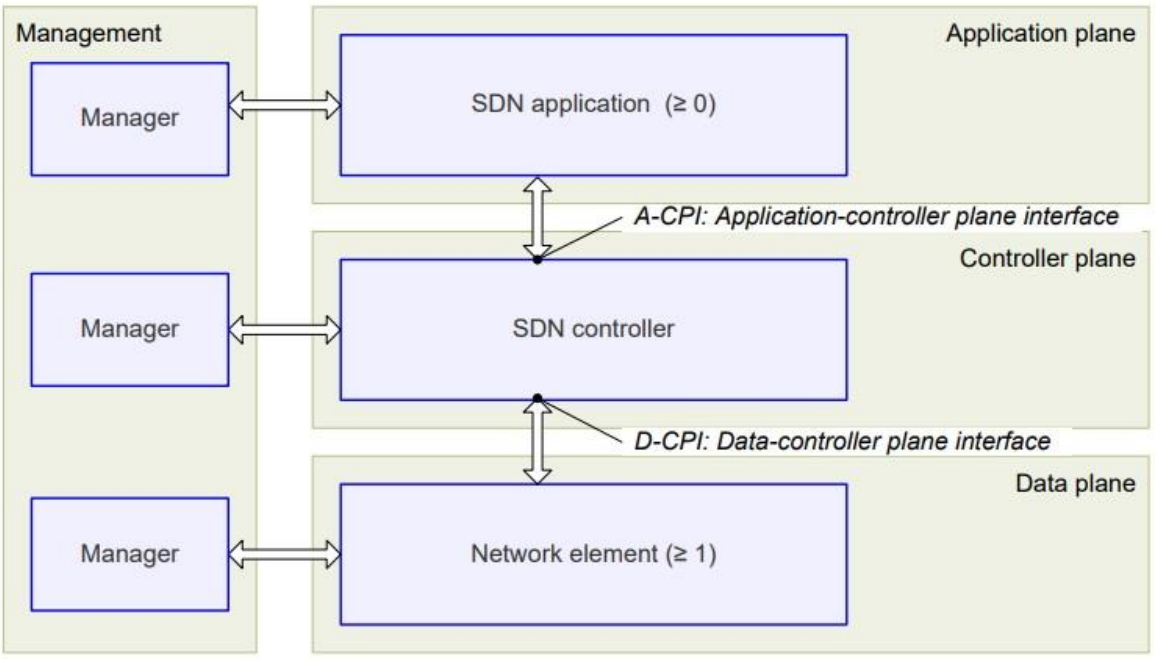


Figura 2.4 – Principais componentes e interfaces da arquitetura SDN [13]

Cada um dos quatro planos possui um conjunto de funções específicas, conforme segue descrito abaixo [13].

- **Plano de gerenciamento:** situado ao lado de cada um dos demais planos, é capaz de coordenar cada um deles através de interfaces de gerenciamento individuais. Contém uma ou mais soluções de gerenciamento SDN responsável por gerenciar os elementos dos outros planos SDN, como por exemplo, monitorar o estado de um dispositivo, configurar e alocar recursos, entre outros.
- **Plano de aplicação:** localizado no topo da arquitetura, contém uma ou mais aplicações que podem servir para diferentes tipos de propósitos, como por exemplo, *firewalls*, balanceador de carga, entre outros. Cada aplicação possui acesso exclusivo a um conjunto de recursos fornecidos por um ou mais controladores SDN.
- **Plano de controle**: localizado no meio da arquitetura, entre os planos de aplicação e controle. Contém um ou mais controladores que coordenam os dispositivos de rede, como por exemplo, NOX, POX, Ryu, entre outros. Ao menos um controlador precisa executar as requisições vindas do plano de aplicação. Estes controladores também possuem uma lógica interna própria para lidar com eventos de rede e decisões de encaminhamento de tráfego.

- **Plano de dados**: localizado na base da arquitetura, contém um conjunto de um ou mais elementos de rede, os quais contém recursos de encaminhamento de tráfegos ou processamento de tráfegos.

Basicamente, a arquitetura SDN foi desenhada para permitir inovações de rede baseadas em quatro pilares [3, 13]:

i. Os planos de controle e dados estão desacoplados. A função de controle é removida dos dispositivos de rede, que se tornam simples dispositivos de encaminhamento de pacotes.

ii. Decisões de encaminhamento são baseadas em fluxos ao invés de baseadas em destino. Um fluxo é definido como um conjunto de valores do cabeçalho de pacotes que atuam com um filtro e um conjunto de ações a serem realizadas para todos os pacotes que possuam esses valores definidos. A programação de fluxos permite uma flexibilidade sem precedentes, limitado apenas à capacidade das tabelas de fluxos. Além disso, permite unificar o comportamento de diversos tipos de dispositivos de rede, como roteadores, *switches*, *firewalls*, entre outros.

iii. Lógica de controle é levada para um agente externo, chamado controlador SDN. Este controlador é uma plataforma de *software* que provê os recursos essenciais e abstrações para facilitar a programação de dispositivos de encaminhamento baseados em redes SDN.

iv. A lógica de encaminhamento de pacotes é abstraída do *hardware* para uma camada de *software* programável. A rede é programável através de aplicações que rodam no nível do controlador, que interage com os dispositivos do plano de dados abaixo dele.

A utilização destes quatro pilares de forma integrada, permite projetar, desenvolver e implantar inovações de rede mais facilmente, pois possibilita maior flexibilidade e agilidade na incorporação de novos serviços e recursos, sem precisar expor o funcionamento interno de equipamentos por parte de seus fornecedores [1].

2.2 OpenFlow

Com a especificação do protocolo OpenFlow, que é o protocolo de comunicação que permite a separação dos planos de controle e de dados no paradigma SDN [1], as redes baseadas nesse paradigma ganharam maior destaque. A lógica da rede concentrada em um

controlador central, que é responsável por toda a inteligência e monitoramento da rede, reduz problemas de gerenciamento comuns em redes convencionais. Além disso, permite regular o estado da rede através da criação de políticas, que podem ser facilmente implantadas pelo elemento controlador.

O OpenFlow provê um protocolo aberto que permite a programação de tabelas de fluxo em diferentes roteadores e *switches.* Este protocolo explora o fato de que a maioria dos roteadores e *switches* modernos possuem tabelas de fluxo, geralmente TCAM (*Ternary Content Addressable Memory),* para a implementação de *firewalls,* NAT (*Network Address Translation*), QoS (*Quality of Service),* e coleta de estatísticas [1]. Apesar de serem diferentes de fornecedor para fornecedor, essas tabelas possuem um conjunto comum de funções que são utilizadas na maioria dos *switches* e roteadores. Desta forma, é possível o controle e gerenciamento de fluxos da rede, além de testes de novos protocolos de roteamento, segurança, e demais inovações [1]. As memórias TCAM, utilizadas como a tabela de fluxos dos dispositivos de encaminhamento, podem ser muito caras e consumidoras de recursos, sendo uma característica a ser levada em consideração nos cenários que exijam uma utilização mais intensa da tabela de fluxos [5].

A proposta inicial do protocolo OpenFlow era mais acadêmica, para possibilitar a realização de experimentos em uma rede de campus universitário de maneira simplificada [1], estabelecendo padrões de comunicação entre um controlador e seus dispositivos de rede, utilizando um canal seguro de comunicação, em uma arquitetura SDN, onde seu protocolo atua. Com este protocolo é possível criar um ambiente de rede de teste que seja programável, unindo as vantagens da virtualização de redes com o conceito do paradigma SDN.

As vantagens e o potencial da tecnologia OpenFlow têm ganhado atenção da indústria, com a criação de novas empresas voltadas ao estudo desta tecnologia, assim como parceria entre empresas existentes, onde são desenvolvidos protótipos com suporte a OpenFlow [12]. A indústria está enxergando as possibilidades que este protocolo traz, dando seu passo no desenvolvimento e adaptação de seus produtos e *hardware* para sua utilização. Existem diversos cenários de rede onde o uso desta tecnologia é promissor [12]:

a. Redes corporativas: possibilita a implementação de novos mecanismos de controle de acesso, gerência de redes cabeada e sem fio de modo unificado, suporte à mobilidade, entre outros [14].

b. Redes celulares: possibilidade de uso transparente de diferentes redes de acesso, como Wifi e 3G, separação dos provedores de infraestrutura e de serviços, entre outros [15].

c. *Backbone*: balanceamento de tráfego web, mobilidade de máquinas virtuais, possibilidade de convergência de redes de pacotes com redes de circuitos, o que abre caminho para novas lógicas de roteamento e engenharia de tráfego, entre outros [16].

d. Data center: possibilidade de novas técnicas de conservação de energia e engenharia de tráfego, suporte à virtualização de estações e *switches* [17].

e. Redes domésticas: possibilita a terceirização da gerência da rede, assim como seu compartilhamento com diversos provedores de serviços, gerência de energia (*smart grid),* entre outros.

f. Redes de ensino e pesquisa: possibilidade de montar cenários e infraestrutura para estudos e experimentações de novas arquiteturas, protocolos e mecanismos de rede, além de validação de *hardware* já comercial, entre outros.

Nas redes utilizando o protocolo OpenFlow, o controlador pode gerenciar as regras da tabela de fluxos do *switch.* Um fluxo de entrada é encaminhado de acordo com a ação especificada na entrada da tabela de fluxos que corresponda à um fluxo em questão, situação chamada de *match*. Caso não exista uma entrada definida para o fluxo, situação chamada de *table-miss*, o *switch* envia uma mensagem do tipo *packet-in* ao controlador, para que este defina a ação a ser tomada – o que pode resultar na criação de uma nova entrada [18]. Contudo, o envio de mensagens *packet-in* para todo *table-miss* pode causar sobrecarga da comunicação entre os *switches* e o controlador. Desta forma, pode haver um aumento do tempo de processamento para o encaminhamento de pacotes, e isto pode ser crítico para a rede, pois o *buffer* do *switch* pode ultrapassar seu limite e causar a perda de pacotes, o que pode limitar a viabilidade do uso desse paradigma em muitos cenários.

2.2.1 Mensagens do protocolo OpenFlow

O protocolo OpenFlow suporta três categorias de mensagens: controlador-para-switch, assíncronas e simétricas, cada uma com vários tipos de mensagens diferentes [10]. As mensagens utilizadas pelo OpenFlow estão descritas abaixo, segregadas por suas categorias.

2.2.1.1 Controlador-para-switch

São mensagens enviadas do controlador para os dispositivos de encaminhamento, que podem ou não requerer uma resposta destes.

- ***Features***: após o estabelecimento da sessão TLS (*Transport Layer Security*), o controlador envia um *features request* para o dispositivo de encaminhamento. Este deve enviar um *features reply* que especifica as capacidades suportadas pelo dispositivo.
- ***Configuration*:** o controlador é capaz de solicitar e aplicar parâmetros configurações nos dispositivos de encaminhamento. O dispositivo responde solicitações apenas do controlador.
- ***Modify-State*:** são mensagens enviadas pelo controlador para gerenciar o estado dos dispositivos de encaminhamento. Seu propósito principal é adicionar, remover e modificar fluxos na tabela de fluxos e definir parâmetros de porta dos dispositivos de encaminhamento.
- ***Read-State*:** são utilizadas pelo controlador para coletar estatísticas das tabelas de fluxos, portas e registros individuais de fluxos, de cada dispositivo de encaminhamento.
- ***Send-Packet*:** utilizadas pelo controlador para encaminhar pacotes para uma porta específica do dispositivo de encaminhamento. Geralmente utilizada para tratar pacotes que não possuem nenhum fluxo definidos para eles.
- ***Barrier*:** são utilizadas pelo controlador para garantir se as dependências da mensagem foram alcançadas ou para receber notificação da conclusão de operações.

2.2.1.2 Assíncronas

São mensagens enviadas sem que o controlador as solicite de um dispositivo de encaminhamento. Os dispositivos enviam mensagens assíncronas ao controlador para informar uma chegada de pacote, mudança de estado, ou erro. As quatro principais mensagens assíncronas são descritas abaixo.

- ***Packet-in***: para todos os pacotes que não possuem um registro de fluxo correspondente, uma mensagem do tipo *packet-in* é enviada ao controlador. Se o dispositivo possuir memória suficiente para armazenar os pacotes que são enviados ao controlador, a mensagem *packet-in* irá conter uma fração do cabeçalho do pacote (por padrão 128 *bytes*) e um *buffer ID* a ser utilizado pelo controlador quando estiver tudo preparado para que o dispositivo possa encaminhar o pacote. Esse *buffer ID* faz referência a área da memória onde o

pacote em questão está armazenado. Dispositivos que não suportam armazenamento interno (ou que teve seu *buffer* esgotado) devem enviar todo o pacote como parte da mensagem a ser enviada ao controlador.

- ***Flow-removed*:** quando uma entrada é adicionada na tabela de fluxos de um dispositivo de encaminhamento através de uma mensagem *Modify-state,* um valor do parâmetro *idle timeout* indica quando esta entrada deve ser removida devido à falta de atividade, assim como um parâmetro de *hard timeout,* que indica quando a entrada deve ser removida independentemente de estar ativa ou não. A mensagem *Modify-state* também especifica se o dispositivo deve enviar uma mensagem *flow-removed* para o controlador quando a entrada expira.

- ***Port-status*:** é esperado que o dispositivo de encaminhamento envie mensagens *port-status* para o controlador quando as configurações de estado da porta mudarem. Esse evento inclui também a mudança do estado da porta (por exemplo, se for desativada por usuário).

- ***Error*:** o dispositivo de encaminhamento pode alertar o controlador sobre problemas utilizando mensagens de erro.

2.2.1.3 Simétricas

São mensagens enviadas mesmo sem solicitação, tanto do controlador para os dispositivos de encaminhamento quanto ao contrário. Abaixo estão descritas as mensagens desta categoria.

- ***Hello***: são trocadas entre os dispositivos de encaminhamento e o controlador quando a conexão é iniciada.

- ***Echo***: podem ser enviadas tanto do controlador quanto dos dispositivos de encaminhamento, e devem retornar uma resposta. Podem ser utilizadas para indicar a latência, largura de banda, e para verificar se a conexão ainda está ativa entre o controlador e os dispositivos.

- ***Vendor***: provém um meio padrão para dispositivos de encaminhamento baseados no OpenFlow possam oferecer funções adicionais dentro do espaço de mensagens OpenFlow.

2.2.2 Switch Openflow

Um *switch* OpenFlow consiste em uma tabela de fluxos, que realiza a checagem e encaminhamento de pacotes, e de um canal seguro de conexão com um controlador externo [3,10]. O controlador gerencia o *switch* através do canal seguro utilizando o protocolo OpenFlow. Desta forma, não é necessário que o *switch* seja programado diretamente, sendo possível realizar a alteração de vários deles a partir do controlador. A Figura 2.5 mostra como é idealizado o *switch* OpenFlow, conforme explicado anteriormente.

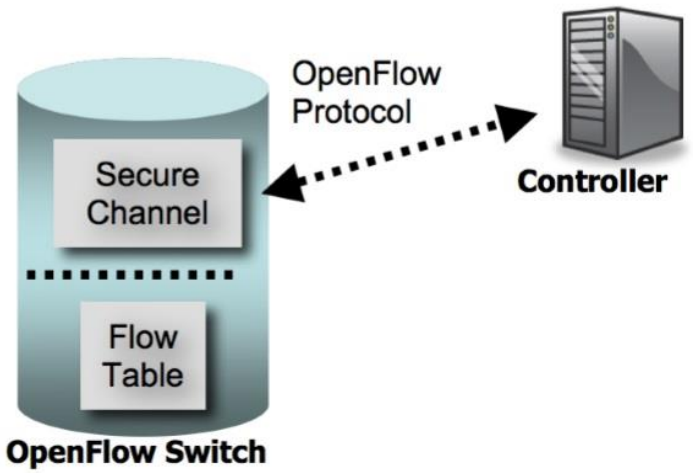


Figura 2.5 – Definição de um switch OpenFlow [10]

A tabela para armazenamento de regras de encaminhamento, chamada de tabela de fluxos, possui um conjunto de registros de fluxos que definem as ações a serem tomadas pelos *switches* no encaminhamento de pacotes. Os principais componentes da tabela de fluxos são exibidos na Tabela 2.1 [10], com suas definições listadas em seguida.

Tabela 2.1 – Principais componentes de um registro da tabela de fluxos [10]

Header Fields	*Counters*	*Actions*

Os campos são definidos abaixo:

- ***header fields***: campos que são observados para realizar a comparação com os pacotes processados. Consistem basicamente na porta de entrada e no cabeçalho dos pacotes. Podem ser chamadas de regras, que definem a combinação dos pacotes com o respectivo registro de fluxo. Esses campos são mostrados na Tab. 2.2.

- ***counters***: contador que é atualizado quando ocorre *match* de pacotes com a respectiva entrada, também chamado de estatísticas (*stats*).
- ***actions***: a ação a ser tomada para os pacotes com obtiveram *match* com a entrada em questão.

Uma das principais abstrações definidas na especificação do protocolo OpenFlow é o conceito de fluxo [12]. Como já mencionado, um fluxo é composto pela combinação dos valores dos campos do cabeçalho de um pacote que está sendo processado pelo dispositivo de encaminhamento. Esses campos são mostrados na Tab. 2.2, os quais formam o componente *header fields* dos registros da tabela de fluxos [10]. Os dispositivos de encaminhamento utilizam esses campos para comparar os valores do pacote em processamento com os registros da tabela de fluxos, onde ao obter uma combinação, situação chamada de *match,* encaminha o pacote utilizando as ações definidas no campo *actions* do respectivo registro. Em seguida, incrementa o contador (*counters* da tabela de fluxos) para indicar mais um *match* para o registro em questão. Caso não seja encontrado nenhum registro na tabela de fluxos que corresponda aos valores do pacote, situação chamada de *table-miss*, o pacote é encaminhado ao controlador, através do canal seguro, para definição da ação a ser tomada [18], utilizando mensagens do tipo *Packet-In*.

O controlador é quem determina como os pacotes com *table-miss* devem ser tratados, podendo realizar a instalação de novos registros na tabela de fluxos, utilizando mensagens do tipo *Modify-State*, ou mesmo tratando o pacote diretamente, através de mensagens do tipo *Send-Packet*. O envio de mensagens *Packet-In* para todo *table-miss* pode causar sobrecarga da comunicação entre os *switches* e o controlador. Desta forma, pode haver um aumento do tempo de processamento para o encaminhamento de pacotes, e isto pode ser crítico para a rede, pois o *buffer* do *switch* pode ultrapassar seu limite e causar a perda de pacotes, o que pode limitar a viabilidade do uso desse paradigma em muitos cenários.

Tabela 2.2 – Campos dos pacotes utilizados para comparar com os registros de fluxos da tabela [10]

Ingress Port	Ethernet			VLAN		IP				TCP/UDP	
	src addr	dst addr	type	id	priority	src addr	dst addr	proto	ToS bits	src port	dst port

Os registros de fluxos podem ser instalados especificando todos os campos possíveis para se realizar um *match*, ou utilizar *wildcards*, que permitem aceitar qualquer valor para os campos definidos, possibilitando a combinação de campos diversos, com valores específicos

em alguns e genéricos em outros [3, 10]. Utilizando *wildcards* temos registros de fluxos mais genéricos, aumentando a abrangência de pacotes aos quais ele pode obter um *match*, diminuindo também a quantidade de registros instalados na tabela de fluxos. Com isso, pode ocorrer de um pacote obter *match* em mais de um registro. Neste caso, o registro com um maior número de campos especificados, ou seja, que possui uma correspondência mais específica, é o escolhido para determinar a ação a ser tomada para o pacote. Caso haja apenas registros genéricos, eles possuem uma prioridade associada a cada um deles. As prioridades mais altas devem ser escolhidas antes das mais baixas. Se existir mais de um registro com a mesma prioridade, o *switch* é livre para escolher a ordem de utilização dos registros [10].

A Tabela 2.3 mostra a especificação dos campos de um registro na tabela de fluxos, onde são definidos valores específicos apenas para a porta de origem e endereço *ethernet* de origem e destino. Os demais campos são configurados com *wildcard* '*', que permite *match* para qualquer valor encontrado. Desta forma, todos os pacotes que sejam originados a partir da porta 1, com endereço *ethernet* de origem 1:1 e com destino ao endereço 1:2, realizarão *match* com este registro, independentemente do valor dos demais campos. É importante ressaltar que a granularidade com que são definidos os registros da tabela de fluxos também afeta o uso de recursos e as métricas da carga no canal de controle.

Tabela 2.3 – Exemplo de registro mais genérico, utilizando *wildcards*.

Ingress Port	Ethernet			VLAN		IP				TCP/UDP	
	src addr	dst addr	type	id	priority	src addr	dst addr	proto	ToS bits	src port	dst port
1	1:1	1:2	*	*	*	*	*	*	*	*	*

Cada registro na tabela de fluxos possui ações simples associadas à eles, onde listamos três básicas, as quais todo *switch* OpenFlow deve conter [1]:

- Encaminhe os pacotes deste fluxo para uma determinada porta ou portas. Isto permite que os pacotes sejam roteados através da rede.
- Encapsule e encaminhe os pacotes deste fluxo para um controlador. O pacote é encapsulado e enviado ao controlador através do canal seguro. Geralmente é utilizado ao se processar o primeiro pacote de um novo fluxo, onde o controlador pode decidir se adiciona este fluxo na tabela. Ou em alguns casos todos os pacotes daquele fluxo podem ser processados pelo próprio controlador.

- Descarte os pacotes deste fluxo. Pode ser utilizado por segurança, para evitar, por exemplo, ataques do tipo DoS (*Denial of Service)*, ou reduzir a quantidade de *broadcast* advindos das estações da rede.

A Figura 2.6 resume o funcionamento do *switch* OpenFlow, em conjunto com a tabela de fluxos e o controlador.

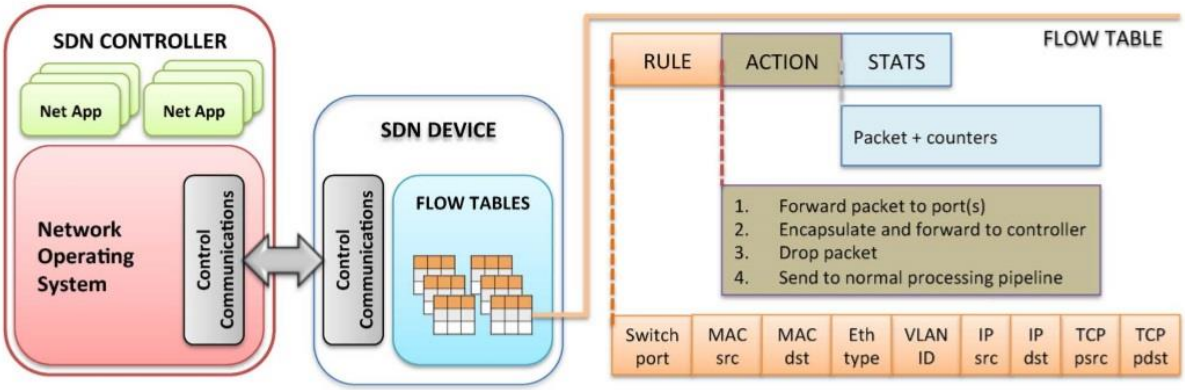


Figura 2.6 – Estrutura dos dispositivos OpenFlow baseados no paradigma SDN [3]

2.2.3 Evolução do Openflow

Este protocolo continua em constante desenvolvimento, evoluindo para melhor se adaptar as necessidades da indústria e aprimorar os recursos e funções nas quais se baseia. Sua evolução aconteceu em grande parte com o processo de padronização realizado pela *Open Networking Foundation* (ONF), partindo da versão 1.0, onde existem apenas 12 campos fixos de *match* e uma única tabela de fluxos, até a última versão que já implementa múltiplas tabelas de fluxos, novas funções e novos campos de *match* [19].

A Tabela 2.4 mostra as principais mudanças entre cada versão, enquanto a Fig. 2.7 mostra a linha do tempo em que foram produzidas as novas versões do protocolo.

Tabela 2.4 – Principais mudanças entre cada versão do protocolo OpenFlow [19]

Versão	**Principais características**	**Motivo**	**Casos de uso**
1.0 para 1.1	Múltiplas tabelas	Evita ultrapassar o limite de registros	
	Tabelas de grupo	Permite aplicar ações para grupos de fluxos	Balanceamento de carga, Link Agreggation
	Suporte total a VLAN e MPLS		

1.1 para 1.2	Formato OXM de *match*	Estende a flexibilidade de *matches*	
	Múltiplos controladores	Escalabilidade, balanceamento de carga	Balanceamento de carga dos controladores
1.2 para 1.3	Tabela de medida	Adiciona capacidade de QoS e DiffServ	
	Registro de *table-miss*	Provê flexibilidade	
1.3 para 1.4	Tabelas sincronizadas	Aprimora a escalabilidade das tabelas	Mac *learning* / Encaminhamento
	Bundle	Aprimora sincronização de *switches*	Configuração múltipla de *switches*
1.4 para 1.5	Tabela de saída	Permite que o processamento seja feito na porta de saída	
	Bundle agendado	Maior aprimoramento da sincronização de *switches*	

Figura 2.7 – Linha do tempo das mudanças do protocolo OpenFlow [19]

2.3 Controladores

É um programa de *software* responsável por manipular as tabelas de fluxos dos dispositivos de encaminhamento, usando o protocolo OpenFlow, através de um canal seguro de comunicação que conecta o controlador a todos os dispositivos [20]. É um dos principais elementos de uma Rede Definida por *Software*. Ele exerce a função de uma camada de abstração da infraestrutura física, que facilita a criação de serviços e aplicações de gerenciamento dos registros das tabelas de fluxos [12].

Os controladores estão no coração das redes baseadas no paradigma SDN. Eles estabelecem fluxos, monitoram o estado dos dispositivos de encaminhamento, calculam os caminhos dos fluxos através dos dispositivos, e manipulam as tabelas de fluxos para orientar na forma como os dispositivos devem lidar com os pacotes da rede [21]. Podem ser tanto um servidor físico quanto uma máquina virtual, como nuvem. Os controladores centralizam a lógica do plano de controle e possuem uma visão global do estado de toda a rede, o que simplifica o desenvolvimento de funções, serviços e aplicações de rede mais sofisticadas [3]. A partir desta premissa é possível um melhor cálculo e controle das rotas na rede, pois a tomada de decisões é simplificada devido ao conhecimento amplo e unificado da rede, o que diminui a complexidade dos demais dispositivos, trazendo a inteligência da rede para a camada de controle.

Os dispositivos de encaminhamento são totalmente dependentes dos controladores para montar sua tabela de fluxos, não possuindo nenhuma lógica de controle local [22]. Alguns estudos são realizados para o uso de mais de um controlador na rede, um recurso já suportado em versões mais recente do protocolo OpenFlow. Em [22] é analisado qual a quantidade de controladores que são necessários em uma rede, e como deve ser desenhado o plano de controle e a distribuição dos controladores para que haja redução da latência da rede, mantendo uma melhor performance. Com a utilização de mais de um controlador, pode-se também balancear melhor a carga do canal de controle e reduzir as chances de sobrecarga, além não comprometer toda a rede no caso de falha de um controlador.

Os parâmetros chaves que definem a performance de um controlador são o número de fluxos que podem ser configurados por segundo e o tempo de configuração desses fluxos [21]. A performance de um controlador varia de acordo com o número de *switches* que ele suporta e a quantidade de fluxos que precisam ser configurados. Conforme já citado, pode haver um único ou múltiplos controladores em uma rede, dependendo do cenário e das necessidades. Quando são utilizados múltiplos controladores, estes formam um plano de controle, onde realizam a comunicação entre si para gerenciar a rede em conjunto [21].

Existem ao menos três camadas bem definidas na maior parte das plataformas de controle, sendo elas [3]:

- Camada de aplicação e serviços, que fica no topo e gerencia a lógica da rede. A conexão com essa camada é realizada pelo controlador através de interfaces *northbound*, onde comunica com sistemas externos de gerenciamento e serviços de rede. Podem ser utilizadas APIs REST (*Representation State Transfer*), no caso por aplicações que utilizem linguagens diferentes da do

controlador, ou diretamente através da linguagem de programação na qual o controlador foi desenvolvido;

- Camada das funções centrais de controle. É o núcleo do controlador, sendo caracterizada como a combinação dos serviços básicos de rede (gerenciamento de topologia, estado da rede, ARP – *Address Resolution Protocol*) com suas várias interfaces de comunicação. Realiza as funções de controle da rede, implementando as políticas nos dispositivos de encaminhamento, monitorando, coletando dados e repassando para as aplicações e serviços da camada superior;
- Camada de elementos que compõe a comunicação *southbound.* O controlador se comunica com essa camada através das interfaces *southbound*, para controle dos dispositivos de encaminhamento. É a camada do plano de dados, que realiza o encaminhamento do tráfego através dos dispositivos de rede, implementando as políticas determinadas pelo controlador.

A Figura 2.8 mostra a estrutura básica de um controlador, ilustrando as camadas e interfaces descritas, com suas funções e elementos.

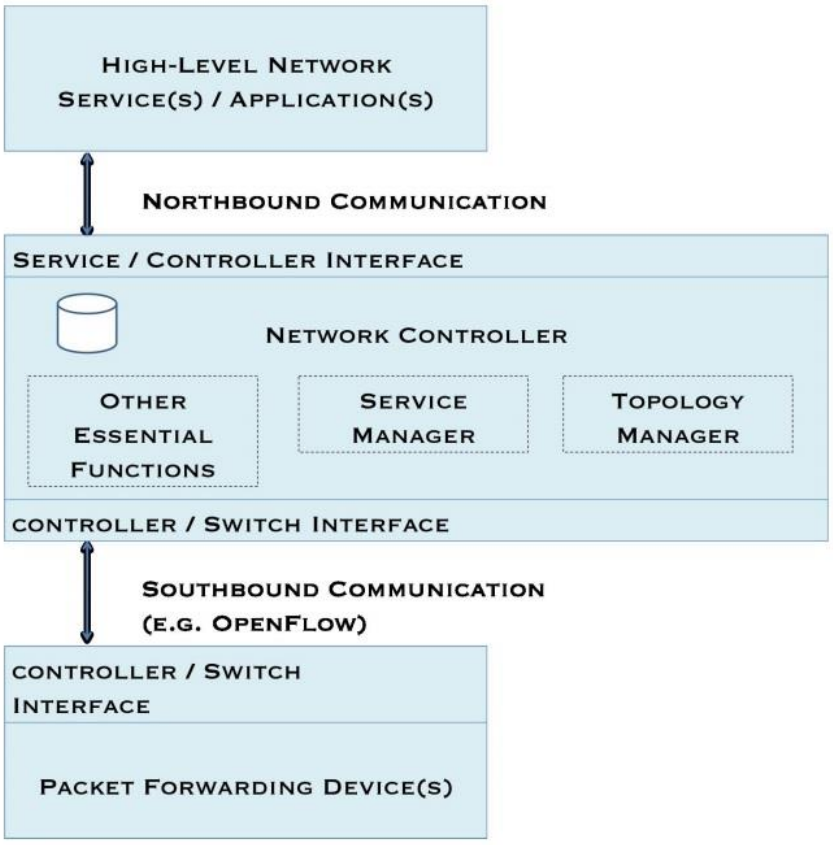


Figura 2.8 – Estrutura de um controlador com suas interfaces *northbound* e *southbound* [5]

Em resumo, um controlador SDN deve prover [23]:

- Gerenciamento do estado da rede: o gerenciamento deste estado pode envolver uma base de dados, a qual serve como um repositório para a informações obtidos dos dispositivos de rede;
- Uma API moderna, de preferência com suporte a REST, para abrir os serviços do controlador para aplicações remotas e facilitar as operações realizadas entre eles.
- Modelamento de dados de alto nível, que captura as relações entre os recursos gerenciados, serviços e políticas.
- Uma sessão de controle TCP segura entre o controlador e os elementos da rede aos quais se conecta.
- Um mecanismo de descoberta de serviços, dispositivos e topologias

Existem diversos controladores criados para o paradigma SDN, cada um com sua particularidade e conjunto de funções próprios, desenvolvidos em diversas linguagens de programação. A Tabela 2.5 lista alguns dos controladores criados para redes SDN.

Tabela 2.5 – Lista de alguns dos controladores SDN

Controlador	**Linguagem**	**Plataforma**	**Desenvolvedor**	**Característica**
NOX [24]	C++, Python	Linux	Nicira	Primeiro controlador SDN
POX [25]	Python	Windows, Mac, Linux	Nicira	Evolução do NOX
Floodlight [26]	Java	Windows, Mac, Linux	Big Switch	Baseado no Beacon. Pode ser integrado com redes não OpenFlow
Ryu [27]	Python	Linux	NTT	Suporta diversos protocolos e versões do OpenFlow, com APIs bem definidas
Beacon [28]	Java	Windows, Mac, Linux, Android	Stanford	*Multithread* e multiplataforma
Maestro [29]	Java	Windows, Mac, Linux	Rice University	Explora paralelismo e controle de redes modulares
Onix [17]	C++, Python, Java	Windows, Mac, Linux	Nicira, Google, NEC	Gerencia redes de grande porte
Trema [30]	C, Ruby	Linux	NEC	*Script*/Emulador
SNAC [31]	C++	Linux	Nicira, Big Switch	Extensão do NOX. Monitora redes OpenFlow

Dentre todas as opções de controladores para serem utilizados neste projeto, foi escolhido o controlador POX, por ser um controlador de código aberto bastante utilizado e com uma grande comunidade de suporte, mais simples para realizar modificações, e pela maior afinidade com a linguagem Python. Este controlador é explicado com mais detalhes na seção 2.3.1 a seguir.

2.3.1 Controlador POX

Este controlador é na verdade uma plataforma de desenvolvimento de código aberto, baseada na linguagem Python, para criação de aplicações de controle de redes definidas por *software* [25]. Foi desenvolvido a partir do NOX [24,32], com uma arquitetura mais estável e interface mais elegante, o que resulta em um controlador mais simples e moderno, sendo muito aceito pela comunidade. Pode ser utilizado em várias plataformas, como Windows, Linux e Mac, e requer a versão 2.7 do Python. Suporta apenas a versão 1.0 do protocolo Openflow, ainda sem previsão para suporte de novas versões.

Ele vem sendo cada vez mais utilizado, estando em constante desenvolvimento, sendo o sucessor do controlador NOX. Embora o POX seja mais estável, o NOX ainda permanece como um ambiente adequado para implementações que possuam demandas com maiores exigências de desempenho. A Figura 2.9 mostra o comparativo do desempenho no controlador POX em relação ao NOX, onde perde para o NOX executado em C++, porém ganha em relação ao NOX executado em Python.

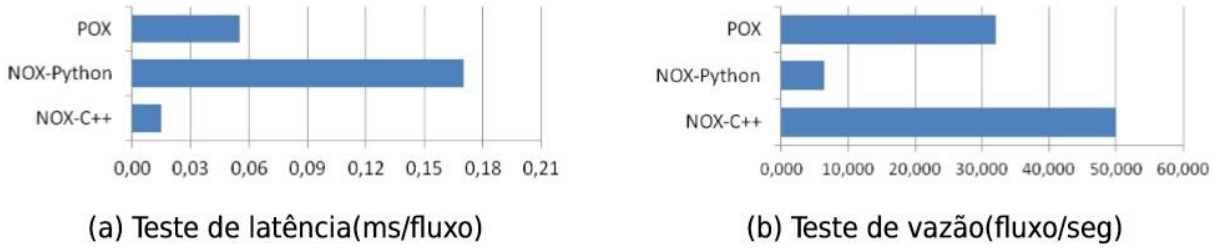


Figura 2.9 – Gráficos de comparação de desempenho entre os controladores NOX e POX [24]

O *script* pox.py é o que inicializa o controlador, podendo fazer a chamada também de módulos e componentes adicionais, nativos ou personalizados, conforme o comando exibido no Quadro 2.1:

Quadro 2.1 – Comando para inicialização do controlador POX

```
./pox.py forwarding.l2_learning openflow_of01 --address 10.1.1.1 –port=6634
```

Este comando faz a inicialização do controlador POX, chamando também o componente *forwarding.l2_learning*, que simula um dispositivo de encaminhamento de nível 2, e passa parâmetros para o protocolo OpenFlow informando o endereço e a porta onde se encontra o controlador. O componente *openflow.of_01* é iniciado por padrão pelo controlador, sendo especificado no comando apenas quando são realizadas alterações de endereço IP ou porta onde o controlador atual. Caso contrário, o controlador estará atendendo no endereço 127.0.0.1 e porta 6633. Também é possível especificar o parâmetro *--no-openflow*, o qual não inicializa o módulo OpenFlow automaticamente. Outro parâmetro interessante é o *--verbose*, que exibe informações extras ao se iniciar o do controlador, sendo útil para a depuração de problemas de inicialização.

O POX possui uma diversidade de componentes e códigos de exemplo em sua Wiki [25], que podem ser utilizados para implementar função nativas ou criar novas conforme a necessidade de cada projeto.

2.4 Open vSwitch

O *Open Virtual Switch* (OVS) é um *switch* virtual multi-camada, de código aberto, que suporta um grande número de protocolos de gerenciamento e interfaces em conjunto com o OpenFlow [4]. Ele reside dentro de um *hypervisor*, que é um sistema de virtualização, ou gerenciador de domínio, e provê conectividade entre máquinas virtuais e as interfaces físicas [33].

Sua arquitetura consiste em um *Control Cluster*, um *ovsdb-server*, um *ovs-vswitchd*, e um módulo de *Kernel* que fica mais abaixo da estrutura, conforme ilustra a Fig. 2.10, e são descritas abaixo [23, 34]:

- ***ovsdb-server*:** é um banco de dados utilizado para armazenar configurações de *switch* que contenham as definições das pontes, interfaces e túneis de rede, assim como os endereços do OVSDB e do controlador OpenFlow. A tabela raíz é chamada "*Open-vSwitch"* e contém as configurações para um *daemon* (*ovs-vswitchd*) exato;
- ***ovs-vswitchd*:** é o *daemon*, ou programa, que implementa o *switch* de fato, em conjunto com o módulo de *kernel* para encaminhamentos baseados em fluxos.

É configurado pelo utilitário *ovs-vsctl* e fornece comunicação com os controladores através de interfaces que utilizam o protocolo OpenFlow, além de se conectar ao OVSDB através do *ovsdb-server* e ao módulo do *Kernel* por meio de uma conexão de rede;

- ***ovs-vsctl*:** utilitário utilizado para consultar e alterar as configurações do *ovs-vswitchd*, sendo uma interface de alto nível para o banco de dados de configurações (*ovsdb-server*), de onde obtém as informações de configuração;
- ***ovs-appctl*:** utilitário que envia comandos para *daemons* do *Open vSwitch* que estão em execução no ambiente;
- ***ovs-ofctl*:** utilitário utilizado para consultar e controlar os *switches* e controladores OpenFlow;
- ***ovs-dpctl*:** ferramenta específica para configurar o módulo de *kernel* do *switch OpenFlow;*
- ***OVS Kernel Module*:** módulo de *kernel* do OVS capaz de estabelecer uma comunicação eficaz entre o *hardware* e os recursos de sistema em *software.*

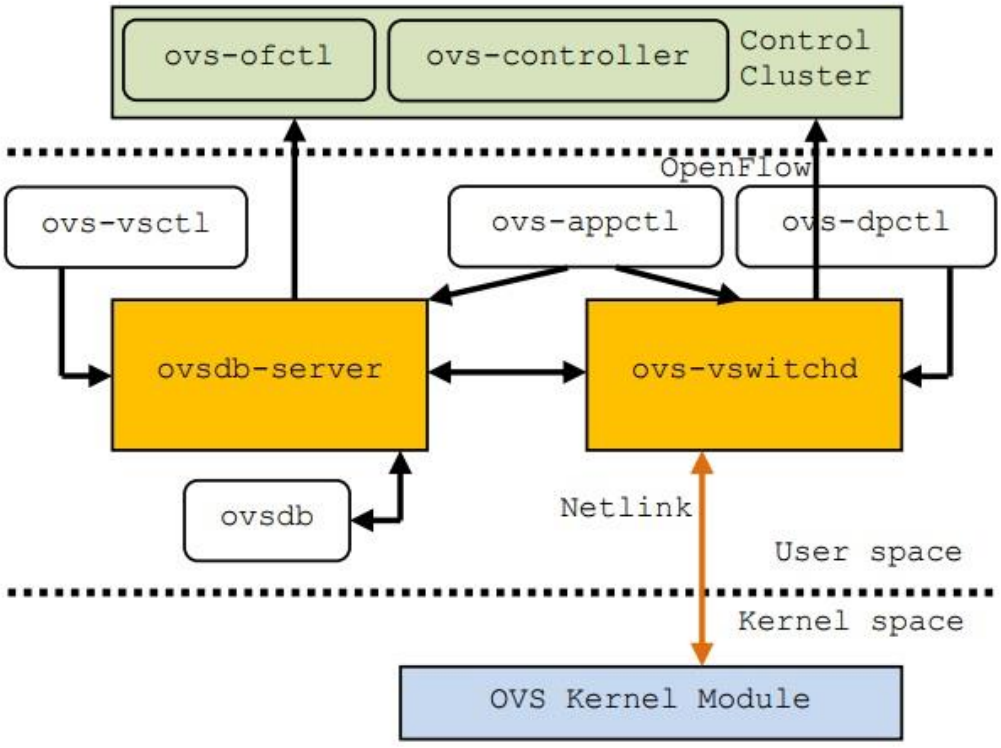


Figura 2.10 – Arquitetura do *Open vSwitch* com seus principais componentes [23]

2.5 Mininet

É uma ferramenta de emulação de redes, capaz de emular *switches*, roteadores, estações e conexões utilizando um único *kernel* do linux. É considerado um sistema que possibilita uma rápida prototipação de grandes redes utilizando os recursos limitados de um simples computador [35]. Esta ferramenta é muito útil para estudos, desenvolvimento e pesquisa devido à capacidade de interagir com toda a rede através da API e interface de linha de comandos (CLI) próprias do Mininet, onde é possível customizar, compartilhar com outros ou mesmo desenvolver em um *hardware* real [36].

O Mininet está em constante desenvolvimento e possui suporte de uma grande comunidade, onde é lançado sob uma licença BSD *Open Source*, encorajando sempre a contribuição de usuários com códigos, relatórios de problemas, resoluções, documentação e tudo o que for possível para melhorar o sistema. Possui uma grande biblioteca de códigos de exemplo, com tutoriais, dicas, FAQ's e documentação para melhor uso de sua API e comandos, possibilitando a criação de códigos que utilizem sua estrutura em conjunto com programas externos.

Em [36] é possível obter todas as informações citadas sobre o Mininet, e fazer o download da máquina virtual completa, contendo o sistema operacional Linux Ubuntu, com o Mininet instalado, onde já possui nativamente também o *Open vSwitch* e o controlador POX, sendo necessário apenas um sistema de virtualização, como VirtualBox, para executar a máquina virtual.

O primeiro passo para criação de uma rede é utilizar a ferramenta de linha de comando "mn" [35]. Através dela é criada toda a estrutura de rede, definindo controladores, *switches, hosts,* entre outros parâmetros. Um exemplo de comando é mostrado no Quadro 2.2:

Quadro 2.2 – Comando para inicialização do Mininet

```
mn --switch ovsk --controller nox --topo tree,depth=2,fanout=8 \
--test pingAll
```

Este comando inicia uma rede Openflow, utilizando *switches* do tipo *Open vSwitch kernel*, com uma topologia em árvore de profundidade 2 e largura 8, apontando para um controlador NOX externo, e realizando um teste de ping entre todas as estações logo após iniciar a rede. Para a criação desta rede, o Mininet emula *switches, hosts, links* e controladores, os quais são descritos abaixo [35].

- ***Switches**: software* baseado em *switches OpenFlow* que provê o mesmo comportamento de encaminhamento de pacotes encontrado em um *switch* físico. Estão disponíveis tanto *switches* baseados em *kernel* quanto em *user-space*.
- ***Hosts**:* é simplesmente um processo *shell*, movido para seu próprio *namespace* de rede com chamada de sistema não compartilhada, que simula uma estação de trabalho. Cada *host* possui sua própria interface *Ethernet* virtual e possui conexão com o processo do Mininet (*mn*) que o criou, do qual recebe comandos.
- ***Links**:* um par virtual *Ethernet* que simula um cabo que conecta duas interfaces virtuais. Pacotes enviados por uma interface chegam até a outra, onde estas atuam como portas *Ethernet* totalmente funcionais para todos os sistemas e aplicações de *software*.
- **Controladores**: os controladores podem estar em qualquer lugar da rede simulada, ou mesmo da rede real, assim como em um computador ou máquina virtual externa, contanto que os *switches* possuam conectividade com ele.

Também é possível customização de experimentos, topologias e elementos de rede, através de um API em Python fornecida pelo Mininet. Algumas poucas linhas de código são suficientes para criar uma rede customizada, executar comandos em vários elementos e exibir resultados e estatísticas da rede [35]. O Quadro 2.3 mostra o exemplo de um código de customização do Mininet, que utiliza sua API:

Quadro 2.3 – Exemplo de código de customização do Mininet

```
from mininet.net import Mininet
from mininet.topolib import TreeTopo
tree4 = TreeTopo(depth=2,fanout=2)
net = Mininet(topo=tree4)
net.start()
h1, h4 = net.hosts[0], net.hosts[3]
print h1.cmd('ping -c1 %s' % h4.IP())
net.stop()
```

Este código cria uma pequena rede com 4 estações, utilizando uma topologia árvore com profundidade 2 e largura 2. A rede é iniciada, e em seguida são emitidos comandos para a estação *h1* enviar uma requisição de *ping* para a estação *h4*. Após o término do comando, a

rede é encerrada. Partindo deste exemplo, pode-se utilizar muitos outros parâmetros e funções para criar novos elementos, topologias, aplicações, entre outros, sendo uma ferramenta bem completa e poderosa.

2.6 Wireshark

É uma ferramenta utilizada para análise de tráfego de rede, multiplataforma e de código aberto, que permite capturar e analisar pacotes trafegando nos diversos sentidos e interfaces da rede, sendo capaz de exibir as informações dos pacotes e filtrá-los por critérios específicos [37].

A Figura 2.11 mostra a interface do Wireshark em atividade, onde lista os pacotes capturados em tempo real. Neste caso, os pacotes estão filtrados para o tipo OpenFlow através do filtro "*openflow_v1*", onde em alguns casos se resume apenas à "*of*", gerando uma melhor visualização dos pacotes desejados. Para filtrar por tipo de pacote OpenFlow, pode-se clicar no pacote desejado, e no cabeçalho OpenFlow clicar com o botão direito em *Type* e selecionar a opção "*Apply as Filter*" > "*Selected*", para que sejam visualizados todos os pacotes com mensagens daquele tipo selecionado.

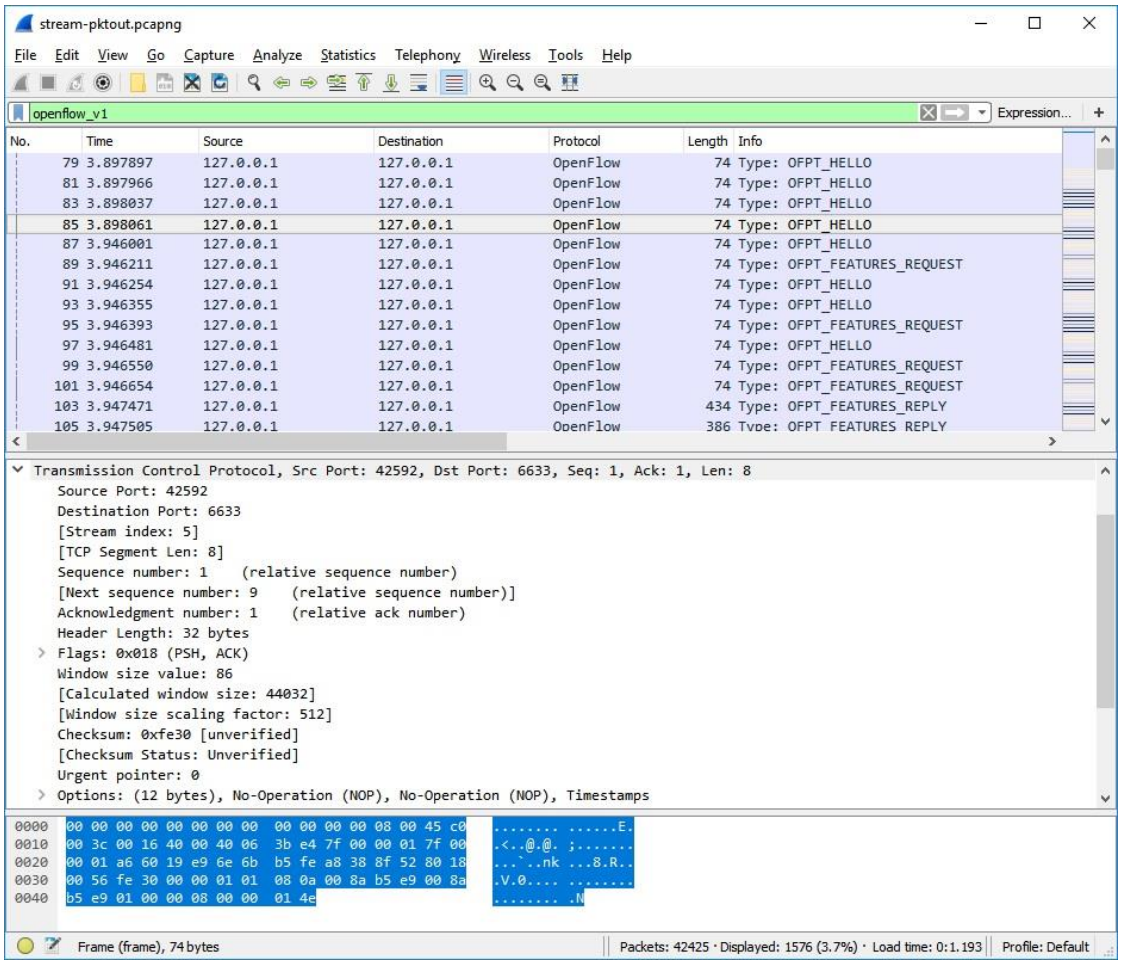


Figura 2.11 – Interface da ferramenta Wireshark

As informações exibidas no painel de listagem de pacotes podem ter seus campos customizados, adicionando ou removendo informações conforme o necessário. Em geral, são apresentadas da seguinte forma, com sua descrição listada em seguida:

Tabela 2.6 – Campos das informações apresentadas pelo Wireshark na listagem de pacotes

No.	Time	Source	Destination	Protocol	Length	Info

- **No**: contém a numeração dos pacotes que foram capturados.
- **Time:** o tempo, geralmente em segundos, em que foram transmitidos os respectivos pacotes, a partir do início da captura.
- **Source:** exibe o endereço IP ou MAC de onde o pacote foi originado
- **Destination:** exibe o endereço de IP ou MAC para onde o pacote foi destinado.
- **Protocol:** exibe o nome do protocolo ao qual o pacote pertence.
- **Length:** exibe o tamanho do pacote, em *bytes*.
- **Info:** exibe informações e detalhes adicionais sobre o pacote que está sendo apresentado. No caso do OpenFlow demonstrado na Fig. 2.11, exibe o tipo de mensagem OpenFlow contida no pacote em questão.

Além da captura e exibição das informações de pacotes, o Wireshark possui muitas funcionalidades e recursos interessantes para ajudar na análise dos dados, como por exemplo, criação de gráficos baseado na entrada e saída de pacotes (I/O), visualização de janela de transmissão em função do tempo, gráficos de fluxos, estatísticas de conversação dos protocolos, rastreio de *streams* TCP e UDP, entre outros. Suporta também a leitura de diversos padrões de protocolos, e possibilita exportar e importar capturas salvas para análises posteriores, além da capacidade de instalar *plug-ins* para estender suas funcionalidades de acordo com a necessidade do usuário.

2.7 Revisão bibliográfica

A abordagem centralizada da lógica da rede, utilizada pelo protocolo OpenFlow, vem sendo discutida principalmente sobre a ocorrência de sobrecarga no canal de controle. Alguns estudos e pesquisas investigam possíveis gargalos no canal de controle entre os dispositivos de encaminhamento e o controlador.

O caminho mais seguido na solução de problemas de sobrecarga deste canal é a descentralização do plano de controle, inserido múltiplos controladores e distribuindo a lógica da rede [4, 6, 21, 22]. Em [22] é avaliado, por exemplo, qual a melhor disposição a ser inserido o controlador, a depender da quantidade deles na rede, para que haja uma menor latência no plano de controle.

Em [18] é proposta uma solução que visa realizar um melhor gerenciamento das regras instaladas na tabela de fluxos, e assim diminuir a quantidade de *table-miss* nos dispositivos de encaminhamento, o que impacta diretamente no número de requisições feita ao controlador e mensagens trafegando no canal de controle. A solução se baseia em duas premissas: um dispositivo de encaminhamento deve manter temporariamente o as regras inativas ao invés de deletá-las; a deleção das regras inativas é feita de acordo com um algoritmo de simples, chamado LRU (*Last-recently-used*). Desta forma, as regras são deletadas de forma mais otimizada, com base no uso mais recente, diminuindo a taxa de regras que ficam inativas na tabela de fluxos, e aumentando a taxa de *match* nos dispositivos de encaminhamento.

Em [7] é apresentada a solução DevoFlow que em sua essência permite o uso mais acentuado de *wildcards* nas tabelas de fluxos dos dispositivos de encaminhamento, através de mecanismos que detectam os fluxos mais significantes, diminuindo a quantidade de interações entre o controlador e os dispositivos, assim como o número de registros na tabela de fluxos. Além disso, delega parte do trabalho do controlador para os dispositivos de encaminhamento, permitindo que estes façam decisões de roteamento locais, sem necessidade de consultar o controlador. Os principais mecanismos criados por ele são a clonagem de regras, que através de uma *flag* específica copia uma regra mais genérica criada pelo controlador para atender um determinado fluxo, e um pequeno conjunto de ações locais que permitem tomadas de decisões simples. O trabalho apresenta algumas estimativas de sobrecargas com a utilização do OpenFlow, porém são uma aproximação média, que podem variar de acordo com o tipo de rede, e sem um estudo específico das mensagens de controle.

Outra solução é apresentada em [8], chamada DIFANE (*DIstributed Flow Architecture for Networked Enterprises*), que implementa os chamados *switches* de autoridade, que são dispositivos que fazem a intermediação entre os controladores e os dispositivos de encaminhamento, mantendo temporariamente os registros de fluxos. Neste caso, o controlador entrega parte do controle para os dispositivos de encaminhamento mais próximos dele, onde quando os demais dispositivos não encontram uma regra na tabela de fluxos, invocam o *switch* de autoridade mais próximo ao invés de invocar diretamente o

controlador. A Figura 2.12 mostra a arquitetura de gerenciamento de fluxos da solução DIFANE, para melhor compreensão.

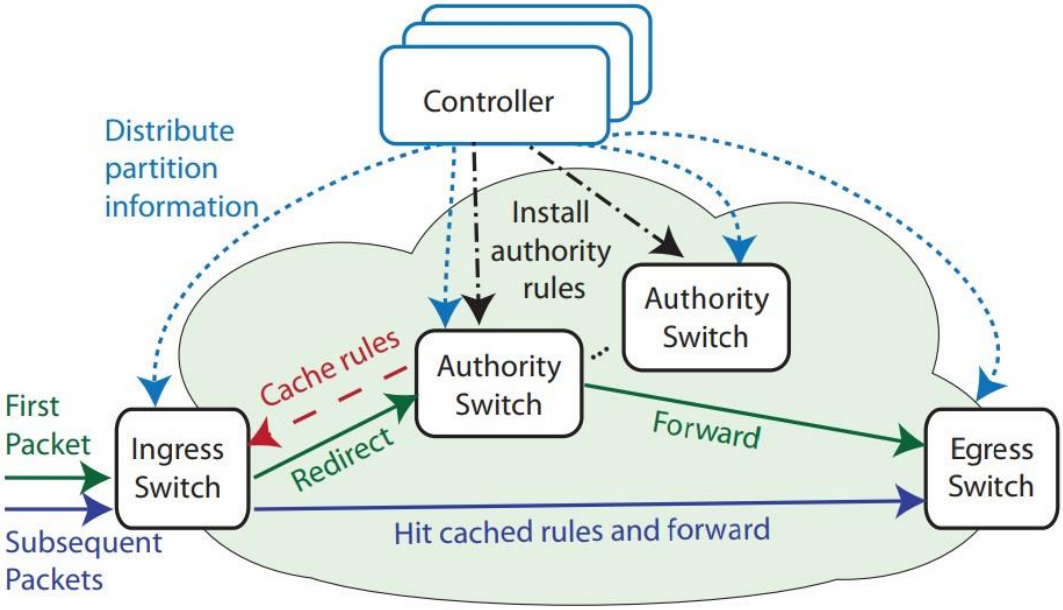


Figura 2.12 – Arquitetura de gerenciamento de fluxos da solução DIFANE [8]

Em todos os trabalhos citados são realizadas análises e propostas para a redução do uso do canal de controle e assim, evitar possíveis gargalos. Entretanto, não é apresentado um estudo mais profundo das mensagens que trafegam no canal, não sendo avaliado o real impacto das mensagens de controle e de monitoramento geradas pelo protocolo OpenFlow no canal de controle. Esta análise é de suma importância para a projeção de novas formas de gerenciamento de redes de maneira eficiente e escalável, com maiores níveis de desempenho e complexidade, baseadas no paradigma SDN.

Este trabalho visa fornecer os subsídios necessários mensurar o impacto gerado pelo tráfego de controle e monitoramento, em uma análise quantitativa, de forma a evidenciar as mensagens utilizadas com maior frequência e a carga gerada por essas mensagens no canal de controle.

Capítulo 3

Desenvolvimento do trabalho

Com base nos conhecimentos exibidos anteriormente sobre a arquitetura do paradigma SDN e do protocolo OpenFlow, a seção 3.1 apresenta um estudo prévio do comportamento deste protocolo em um cenário mais simplificado, para definir os critérios mais relevantes para a análise do uso do canal de controle. Na seção 3.2 são apresentados os cenários e topologias criados e suas características para as diversas simulações realizadas, assim como a metodologia e métricas utilizadas para análise e obtenção dos dados experimentais. Na seção 3.3 são exibidos os resultados experimentais, com análises de cada um, enquanto na seção 3.4 é realizada a síntese dos resultados obtidos para cada parâmetro de configuração estudado, com as análises finais.

3.1 Estudo Preliminar

Para uma análise inicial da relevância das mensagens encaminhadas pelo protocolo OpenFlow foi desenhada uma topologia simplificada contendo um único *switch*, conectado à 14 estações de trabalho, um servidor de vídeo e um servidor web, com intuito de verificar o impacto de cada mensagem de controle e de monitoramento que trafega no canal entre o controlador e os dispositivos de encaminhamento. O experimento foi realizado utilizando o controlador POX com a implementação de alguns componentes adicionais, listados e descritos abaixo.

- ***forwarding.l2_learning*:** componente nativo do controlador, que simula um dispositivo de encaminhamento nível dois, onde através de uma tabela interna ele mapeia todos os endereços MAC com base na sua origem e porta, no momento em que é iniciada a comunicação entre o controlador e os dispositivos, sempre atualizando no decorrer das trocas de mensagens.
- ***ic_monitor*:** componente de própria autoria, criado para realizar o monitoramento do tráfego do protocolo OpenFlow, através do envio de requisições de estatísticas (*Read-State*), e geração de relatórios com informações de fluxos e quantidade de pacotes para cada tipo de mensagem. É explicado com mais detalhes na seção 3.1.4.

- ***info.packet_dump*:** componente nativo do controlador que envia as informações das mensagens do tipo *packet-in* para a log. Semelhante a executar um *tcpdump* em um *switch*.
- ***samples.pretty_log*:** componente nativo do controlador que formata o log de modo a ficar mais bonito e funcional.
- **openflow.of_01:** componente nativo de controlador que faz a comunicação com os *switches* utilizando o protocolo OpenFlow 1.0. Ele é iniciado automaticamente pelo controlador, mesmo sem realizar a chamada via comando. Através dele podem ser definidos o endereço de IP e porta do controlador, caso não queira utilizar os valores padrões.

Este cenário foi escolhido para simplificar a identificação das mensagens mais comuns, e que possuem mais peso no uso do canal de controle, gerando quantidade suficiente de tráfego para uma análise mais assertiva durante o período do experimento.

Os servidores web e de vídeo são os responsáveis, em conjunto com as estações, pela geração do tráfego através da resposta às requisições de arquivos e de *streaming* de vídeo, respectivamente, que são realizadas em média a cada 20 segundos pelas estações. Este tipo de tráfego foi escolhido baseado em estudos prévios do perfil de tráfego de usuários na internet [38, 39], onde prevalece o tráfego HTTP e de vídeo. A Tabela 3.1 mostra os parâmetros definidos para gerar o tráfego nos experimentos realizados. As requisições dos usuários foram divididas nas proporções de 25% para vídeo e 75% para web, ou seja, 1 usuário faz requisição de vídeo para cada 3 que fazem requisições web, devido ao maior tráfego gerado pelo *streaming*, pois desta forma a quantidade de tráfego gerado pelos dois servidores é semelhante, fazendo com que o tráfego web também tenha significância.

Tabela 3.1 – Parâmetros de configuração de perfil de tráfego de usuário

Parâmetros	Valores
Protocolo de transmissão	Web: HTTP / Vídeo: RTP
Tamanho do arquivo web	618 KB
Tempo de duração vídeo	1 minuto
Taxa de bits / Codec do vídeo	358 kbps / H.264
Perfil de usuário	1 usuário de vídeo para cada 3 usuários Web
Frequência monitoramento	5 segundos
Tempo entre requisições	20 segundos
Duração do experimento	5 minutos

Foi utilizada apenas uma máquina física para as todas as simulações realizadas neste projeto, em conjunto com uma máquina virtual rodando o sistema Ubuntu, uma rede emulada no emulador Mininet com *switches* do tipo *Open vSwitch* em modo *kernel* e um controlador POX, como citado. A especificações são descritas na Tab. 3.2.

Tabela 3.2 – Especificações de *hardware* e *software* dos experimentos

Especificações de *Hardware* e *Software* utilizados	
Processador	Intel Core i5-4460 3.2 GHz
Memória RAM	16 GB
Sistema Operacional	Windows 10 Pro
Simulador	Mininet 2.2.1 – Ubuntu 14.04 LTS 64 bit
Switch Virtual	Open vSwitch 2.7.2
Largura de banda	1 Gb
Controlador	POX 0.2.0
Máquina Virtual	Virtual Box 5.2
Servidor de vídeo	VLC 2.1.6 Rincewind
Servidor web	SimpleHTTPServer (Python)
Analisador de pacotes	Wireshark 1.10.6 (nativo do Mininet)
Acesso remoto (SSH)	PuTTY 0.69

3.1.1 Configuração do Virtual Box e Mininet

A máquina virtual do Mininet foi baixada do seu site oficial [36], e importada para o Virtual Box. Algumas configurações foram realizadas para um melhor desempenho da máquina virtual, e disponibilização de conectividade para acesso remoto. Foram compartilhados os 4 núcleos do processador e 8 Gb de memória RAM com a máquina virtual, para permitir uma execução dos experimentos com o máximo de recursos disponíveis. Para a rede, foi criada uma interface NAT para comunicação da máquina virtual com a internet, permitindo o acesso a repositórios para instalação e atualização de programas. Para acesso remoto, foi configurada uma segunda interface do tipo *HostOnly*, que possibilita conectividade entre a máquina virtual e a máquina física. Esta configuração cria uma interface virtual na máquina física para comunicação com a máquina virtual. É necessário habilitar a interface como servidor DHCP, nas ferramentas globais do Virtual Box [40], para que forneça endereços de IP para as demais interfaces virtuais. As Figuras 3.1 e 3.2 mostram as configurações utilizadas para a máquina virtual.

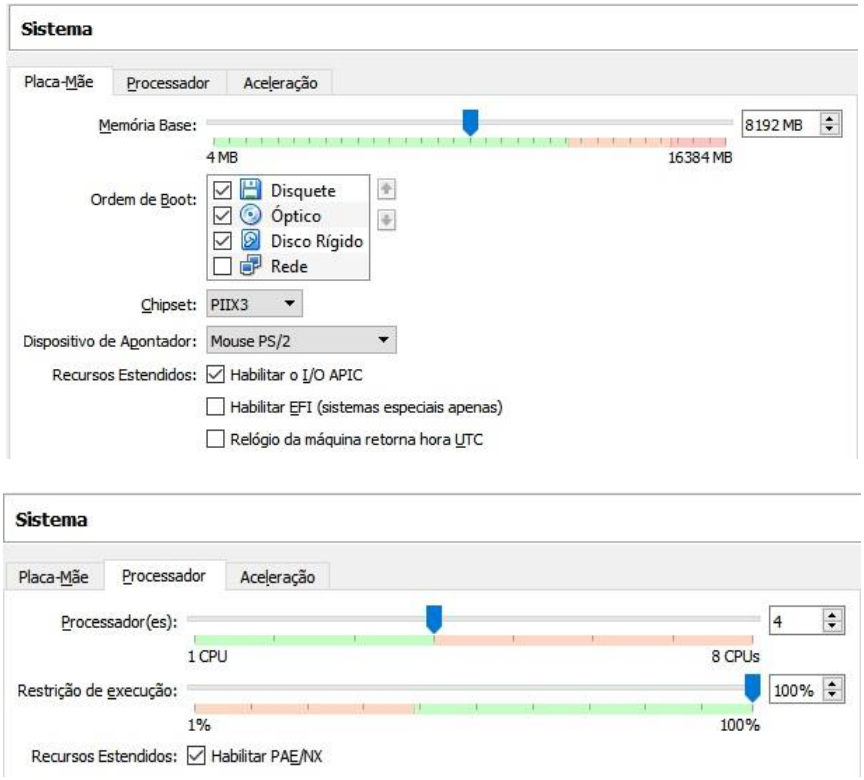
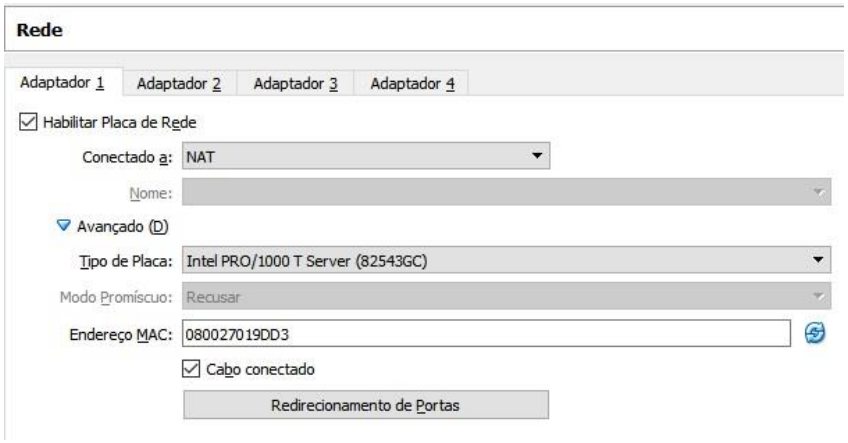


Figura 3.1 – Configurações de processador e memória da máquina virtual



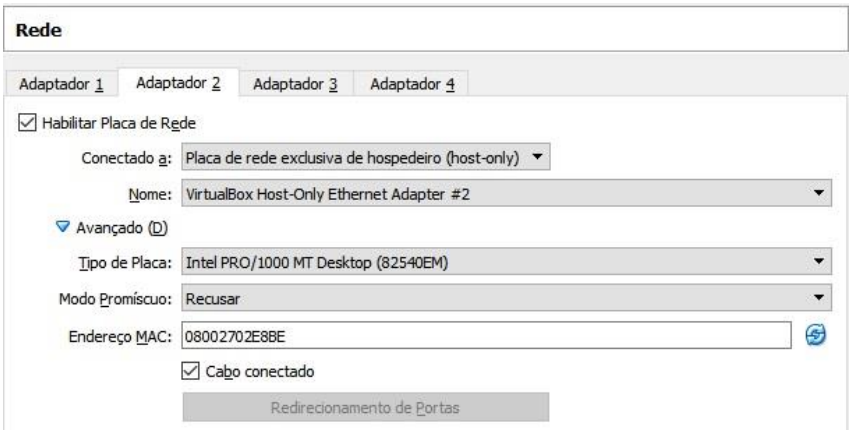


Figura 3.2 – Configurações de rede da máquina virtual

Após as configurações, a máquina virtual foi inicializada, com a autenticação padrão de usuário *mininet* e senha *mininet.* Ao realizar o comando *ifconfig -a*, podemos observar que a interface *eth0* já possui um IP configurado, sendo ela a interface HostOnly para acesso remoto. A interface *eth1*, que é a interface NAT, não possui nenhum IP, sendo necessário executar o comando "*dhclient eth1"* para obter um IP do servidor DHCP. Após isto, todas as interfaces se encontram configuradas e com conectividade, conforme mostra a Fig. 3.3 abaixo.

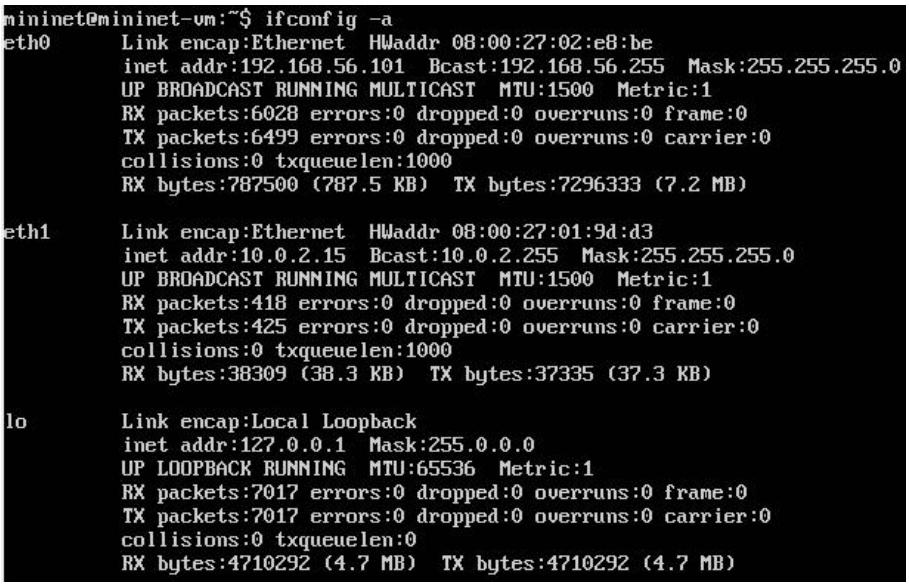


Figura 3.3 – Interfaces de rede da máquina virtual através do comando *ifconfig*

Para que a alocação do IP na interface *eth1* pelo servidor DHCP seja automática, deve ser inserido o comando abaixo no arquivo *"/etc/network/interfaces*".

Quadro 3.1 – Comando para configuração de DHCP nas interfaces da máquina virtual

```
auto eth1
iface eth1 inet dhcp
```

3.1.2 Configuração do PuTTY

Após as configurações de rede, o acesso remoto é feito através da ferramenta PuTTY [41]. Para isto, basta inserir o IP da interface *hostonly* do Mininet, no caso 192.168.56.101, na porta 22 do SSH e conectar ao destino, inserindo o usuário e senha. Podem ser abertas diversas sessões SSH do PuTTY com a mesma máquina virtual, o que é útil para inserir comandos e inicializar aplicações em telas diferentes, melhorando a organização do experimento. Entretanto, para que seja possível executar aplicações gráficas, como por

exemplo o Wireshark, é necessário configurar o ambiente gráfico do Linux, o X11, para que execute as aplicações gráficas remotamente. Para isso é preciso ter um X Server instalado no Windows, no caso o Xming, para que possa abrir as telas do Linux em seu ambiente. Com isso, foi instalado o servidor Xming e realizada as configurações conforme exibidas na Fig. 3.4 para habilitar o recurso, onde o ponto principal é a seleção da caixa "*Enable X11 forwarding*".

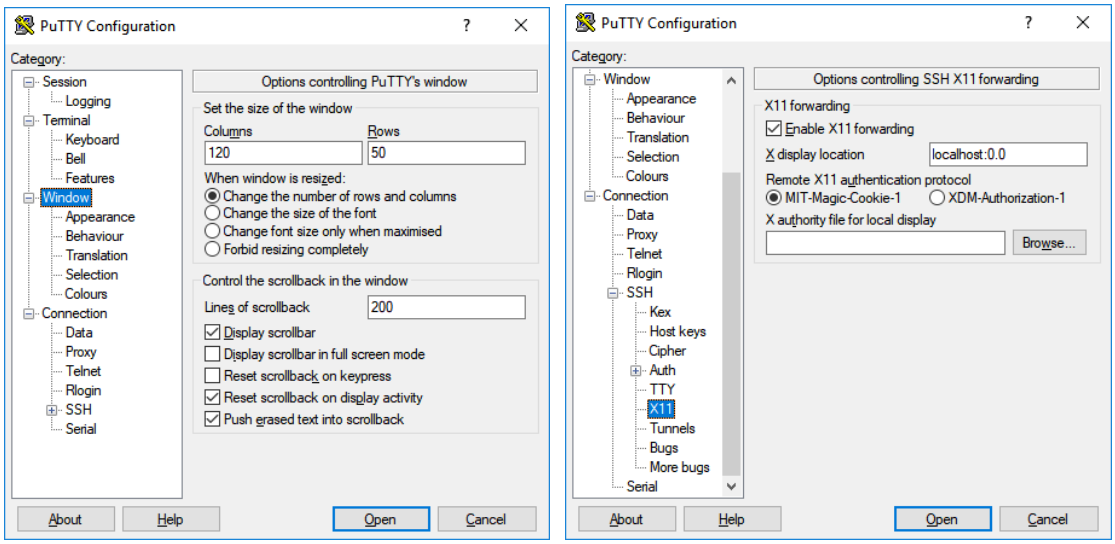


Figura 3.4 – Configurações do PuTTY para uso de aplicações gráficas remotamente

3.1.3 Código *stream.py*

Para execução do experimento é utilizado o arquivo *stream.py*, um *script* feito na linguagem Python que faz a inicialização da rede virtual no Mininet e dos servidores de vídeo e web, assim como as requisições das estações para os servidores. Nele são definidos os tempos de duração e de intervalo do experimento, qual a topologia utilizada e o número de estações que farão requisições de vídeo e as que farão requisições web, entre outros parâmetros da rede.

A chamada da classe Mininet() define os parâmetros de inicialização da rede. No código do Quadro 3.2, que faz parte do *script stream.py*, é iniciada uma instância do Mininet onde são passados os parâmetros desejados para a rede, explicados logo em seguida. Ao final, o comando *net.start()* inicializa a rede com os parâmetros informados.

Quadro 3.2 – Código de inicialização do Mininet no *script stream.py*

```
if sys.argv[1] == "1":
topo = SingleSwitchTopo( 64 )

elif sys.argv[1] == "2":
```

```
topo = LinearTopo(8,8)

elif sys.argv[1] == "3":
topo = TreeTopo( depth=3, fanout=4 )

net = Mininet(topo=topo, switch=partial(OVSKernelSwitch,protocols=None),
controller=RemoteController,link=partial(TCLink,bw=1000), autoSetMacs=True,
cleanup=True)

net.start()
```

- **topo**: tipo de topologia a ser utilizada. É definida pela variável topo, que depende do parâmetro passado na inicialização do *script.*
- **switch**: o tipo de *switch* a ser utilizado. No caso, é configurado o *Open vSwitch* no modo *kernel* (OVSK).
- **controller**: controlador a ser utilizado na rede. No caso é definido um controlador remoto, que será buscado no IP/porta padrão 127.0.0.1:6633.
- **link**: configuração dos *links* entre as estações e *switches.* É configurada um link com largura de banda de 1 Gbps.
- **autoSetMacs**: define se a numeração dos endereços MAC deverá ser sequencial, que facilita a visualização.
- **cleanup**: define se ao finalizar a rede deverá ser realizada limpeza dos elementos e configurações.

A chamada do *script* é feita através do comando do Quadro 3.3, onde o parâmetro passado define qual a topologia a ser utilizada – 1 para estrela, 2 para linear e 3 para árvore:

Quadro 3.3 – Comando de inicialização do *script stream.py*

```
> sudo python2.7 stream.py 1
```

Neste *script* também é feita a inicialização dos servidores de vídeo e servidor web, a partir dos comandos dos Quadros 3.4 e 3.5, que são enviados aos *hosts* selecionados como os respectivos servidores. Para o caso do servidor de vídeo, foi criada uma função específica que define o *streaming* para todos os IP's de destino das estações definidas.

Quadro 3.4 – Códigos de inicialização do servidor web no *script stream.py*

```
print 'Inicializando servidor web...'
            h[servidor_web].sendCmd("python -m SimpleHTTPServer 8000")
print "Inicia streaming do servidor de vídeo..."
                  self.iniciaServidor(h[servidor_video],n_videos,intervalo)
```

Quadro 3.5 – Códigos de inicialização do servidor de vídeo no *script stream.py*

```
def iniciaServidor(self,h,n_videos,tempo):
              dst = ""
              for v in range(n_videos):
                     dst += "dst=rtp{dst=10.0.0.%i,mux=ts}," % (v+1)
              h.sendCmd('cvlc -vvv video.mp4 --sout \
"#transcode{vcodec=h264,acodec=mpga,ab=128,channels=2,samplerate=44100}:\
duplicate{%s}" --run-time %i vlc://quit' % (dst,tempo))
```

O comando *cvlc* é referente ao reprodutor VLC [42], por linha de comando, que faz o *streaming* do arquivo *vídeo.mp4* via protocolo RTP para os IP's definidos na variável *dst*, de acordo com a quantidade de estações de vídeo definidos pela variável *n_videos*, durante o tempo obtido na variável *tempo*, que é o intervalo de 20 segundos escolhido para o experimento.

3.1.4 Componente ic_monitor

Para a obtenção dos dados e estatísticas do tráfego no canal de controle, foi criado o componente *ic_monitor*. Ele é escrito na linguagem Python e para instalá-lo no controlador POX basta copiá-lo para dentro da pasta *"~/pox/pox/ext*". A chamada dele é feita na inicialização do controlador POX, pelo próprio nome, conforme Quadro 3.10 da seção 3.1.5. Este componente possui uma função de monitoramento "_monitoracao()", descrita no código do Quadro 3.6, onde faz o envio de mensagens do tipo *Read-State* para os *switches,* e contabiliza cada mensagem enviada separando por tipo de estatística (*flowstat, aggstat e portstat*). As respostas destas mensagens trazem as informações de estatísticas necessários para se levantar os dados propostos no experimento.

Quadro 3.6 – Código da função de monitoramento do componente *ic_monitor*

```
def _monitoracao ():
  for connection in core.openflow._connections.values():
       flowstat = of.ofp_flow_stats_request()
       aggstat = of.ofp_aggregate_stats_request()
       portstat = of.ofp_port_stats_request()
       # Envia as mensagens de requisicao de estatisticas e contabiliza
       connection.send(of.ofp_stats_request(body=flowstat))
       ic_monitor.flowstats += 1
       connection.send(of.ofp_stats_request(body=portstat))
```

```
        ic_monitor.portstats += 1
        connection.send(of.ofp_stats_request(body=aggstat))
        ic_monitor.aggstats += 1
```

O *framework* do controlador POX possui funções que lidam com eventos de recebimento de mensagens, para que estas possam ser tratadas, que possuem o prefixo *_handle_*. Estas funções são utilizadas para se obter as respostas das requisições e contabilizar a quantidade de mensagens recebidas pelo controlador. O código do Quadro 3.7 demonstra a função para lidar com as mensagens do tipo *Packet-In*, onde recebe a variável *event* que contém a mensagem recebida, com suas informações e cabeçalho. O atributo *event.ofp.header_type* indica o valor do tipo da mensagem, onde no caso do *Packet-In* é 10. Cada tipo de mensagem possui um número correspondente, e este número é utilizado para compor o índice da matriz *pkt_contador*, que armazena a quantidade e o tamanho total de cada mensagem recebida. A variável *tcp_size* contém o tamanho do pacote TCP onde a mensagem OpenFlow está encapsulada, que é de 66 bytes, e soma com o valor da mensagem OpenFlow para chegar ao tamanho final do pacote que trafega no canal. Este mesmo padrão de função é feito para cada tipo de mensagem recebida, e contabilizadas individualmente.

Quadro 3.7 – Código da função que lida com mensagens *packet-in* no controlador POX

```
def _handle_PacketIn (self,event):
        self.pkt_contador[event.ofp.header_type][0] += 1
        self.pkt_contador[event.ofp.header_type][1] += len(event.ofp)+self.tcp_size
```

Para a contabilização das mensagens enviadas do controlador para os *switches*, foi necessário a alteração do componente *openflow.of_01*. Este componente é o responsável por lidar com as mensagens OpenFlow, tanto para receber, ativando os eventos mencionados anteriormente, quanto para enviar mensagens. Ele possui uma função *send()*, utilizada pelo controlador toda vez que precisa enviar alguma mensagens aos *switches.* Com base nisso, foi inserido dentro desta função o código exibido no Quadro 3.8. O componente *core* faz a chamada do componente *ic_monitor* para acessar a variável *pkt_contador*, onde incrementa nesta, da mesma forma feita nas mensagens *Packet-In*, a quantidade e o tamanho de cada pacote que envia aos *switches* (*Send-Packet*, *Modify-State*, entre outros), separados por tipo. Neste caso, o pacote gerado está contido na variável *data*, ao contrário dos pacotes recebidos, que estão na variável *event* das funções *_handle_*.

Quadro 3.8 – Código para obtenção de dados de mensagens enviadas pelo controlador

```
def send (self, data):
    core.ic_monitor.pkt_contador[data.header_type][0] += 1
    core.ic_monitor.pkt_contador[data.header_type][1] += len(data)+66
```

A Figura 3.5 mostra um exemplo da tela de estatísticas gerada pelo componente *ic_monitor* ao se encerrar a conexão com o controlador POX.

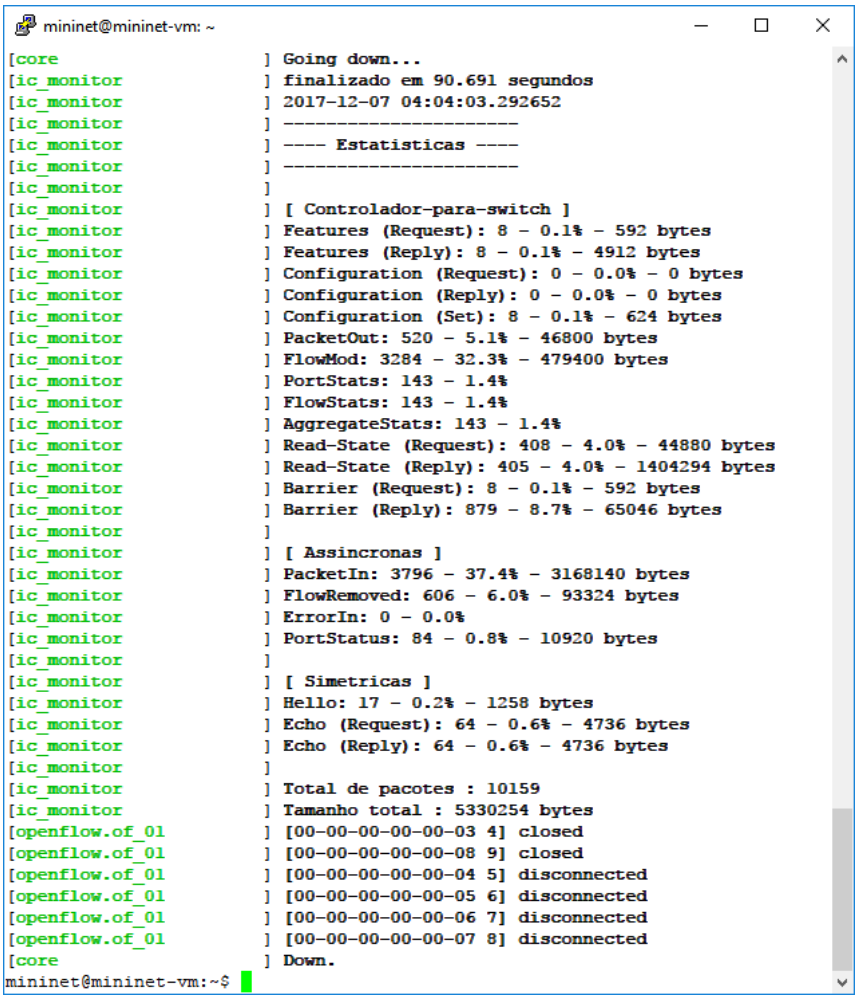


Figura 3.5 – Tela de estatísticas geradas pelo componente de monitoramento *ic_monitor*

Este componente também gera uma lista de fluxos totais e inativos instalados a cada intervalo de monitoramento, analisando as estatísticas obtidas das mensagens *Read-State* do tipo *FlowStats,* conforme código exibido no Quadro 3.9. Ao final dele faz a gravação dos dados de estatística dos pacotes e dos fluxos nos arquivos "fluxos_%topo%.txt" e "estat_%topo%.txt", respectivamente, onde %topo% é o nome da topologia utilizada em cada caso, definido a partir do número de *switches* contabilizados. No caso da topologia estrela, por exemplo, seriam os arquivos "fluxo_estrela.txt" e "estat_estrela.txt".

Quadro 3.9 – Código que trata as mensagens de estatísticas de fluxos para obtenção de dados

```
def _handle_FlowStatsReceived (self,event):

        # contabiliza o reply de FlowStats

        self.flowstats += 1
       stats = flow_stats_to_list(event.stats)

        f_inativos = 0
       self.soma_fluxos += 1

        # verifica se houve incremento nos fluxos

        if self.flowstats > 1 :

             for f in event.stats:

                    for k in self.fluxos[event.connection.dpid]:

                           if f.match == k.match:

                                  if f.packet_count == k.packet_count:

                                         f_inativos += 1

        # soma a quantidade de fluxos de todos os switches

        self.soma_f_inativos += f_inativos

        self.soma_f_total += len(event.stats)

        # verifica se todos os switches responderam e contabiliza o total

        if self.soma_fluxos == self.n_switches:

             self.fluxos_total.append(self.soma_f_total)

             self.fluxos_inativos.append(self.soma_f_inativos)

             self.soma_fluxos = 0

             self.soma_f_inativos = 0

             self.soma_f_total = 0

        # armazena a ultima estatistica de fluxos de cada switch

        self.fluxos[event.connection.dpid] = event.stats
```

3.1.5 Experimentos

O controlador POX é inicializado utilizando uma tela de sessão do PuTTY, com os componentes especificados anteriormente, através do comando do Quadro 3.10, com a adição de um nível de log do tipo DEBUG, que permite um maior nível de detalhamento no log:

Quadro 3.10 – Comando de inicialização do controlador POX com os devidos componentes

```
> sudo ~/pox/pox.py forwarding.l2_learning ic_monitor info.packet_dump \
samples.pretty_log log.level --DEBUG
```

Utilizando outra sessão do PuTTY, é feita a inicialização do código *stream.py*, conforme apresentado no Quadro 3.3 da seção 3.1.3.

Durante todo o experimento foram instaladas diversas regras de encaminhamento no *switch* para permitir a comunicação entre os servidores de vídeo e web, e as demais estações que fazem as respectivas requisições. Todos os dados sobre a quantidade de mensagens de cada tipo e os fluxos instalados são gravados em arquivo ao fim de cada experimento, através do componente *ic_monitor*. No total foram realizados cinco experimentos com duração de 5 minutos, com um total 1.500 requisições de estatísticas de monitoramento do tipo *Read-State* realizadas em cada um deles, para uma melhor precisão dos dados e eliminação de erros. Ao final de todos os experimentos, foi gerada a estatística geral de cada tipo de mensagem do protocolo OpenFlow, para analisar quais são as mais frequentes e suas respectivas contribuições no total de carga do canal de controle.

A Tabela 3.3 apresenta todos os tipos de mensagens do protocolo OpenFlow organizados por categoria (Assíncronas, Simétricas e Controlador-switch), com a respectiva estatística obtida durante os experimentos. Existem duas direções pelas quais essas mensagens podem trafegar na rede: dos *switches* para o controlador (S→C) e do controlador para os *switches* (C→S). Para cada tipo de mensagem foi informado a respectiva porcentagem de pacotes processados e sua carga no canal de controle, segregados pela direção de envio das mensagens, baseando sua porcentagem em cada uma dessas direções.

Tabela 3.3 – Estatísticas de pacotes processados e a carga no canal de controle para cada mensagem OpenFlow transmitida no experimento preliminar

Categoria de mensagem	**Tipo de mensagem**	**Mensagens processadas**	**Carga no canal de controle**	**Direção**
Simétrica	*Hello (request)*	0,11%	0,06%	C → S
	Hello (reply)	0,06%	0,003%	S → C
	Echo (request)	1,28%	0,08%	S → C
	Echo (reply)	1,28%	0,67%	C → S
	Vendor (request)	0,00%	0,00%	C → S
	Vendor (reply)	0,00%	0,00%	S → C
Assimétrica	*Packet-In*	88,47%	77,77%	S → C
	Flow-Removed	0,00%	0,00%	S → C
	Port-Status	0,06%	0,005%	S → C
	Error	0,00%	0,00%	S → C
Controlador para switch	*Features (request)*	0,06%	0,03%	C → S
	Features (reply)	0,06%	0,04%	S → C
	Configuration (request)	0,00%	0,00%	C → S
	Configuration (reply)	0,00%	0,00%	S → C
	Configuration (set)	0,06%	0,03%	C → S
	Modify-State	87,65%	90,86%	C → S
	Send-Packet	0,78%	0,50%	C → S
	Read-State (request)	10,02%	7,82%	C → S
	Read-State (reply)	10,03%	22,09%	S → C
	Barrier (request)	0,06%	0,03%	C → S
	Barrier (reply)	0,06%	0,003%	S → C

Os resultados dos experimentos demonstram que as mensagens de maior frequência são as do tipo *Packet-In*, *Modify-State* e *Read-State*, sendo as mais utilizadas na comunicação entre o controlador e os *switches*. As mensagens do tipo *Echo* também se sobressaem um pouco mais devido aos intervalos de requisições e o número reduzido de hosts, o que faz com que a rede possua momentos de inatividade. Quando a rede está inativa, o controlador e os *switches* utilizam mensagens do tipo *Echo* para garantir a conectividade entre eles.

As mensagens do tipo *Send-Packet* geralmente são utilizadas pelo controlador com frequência, pois quando este trata uma mensagem do tipo *Packet-In*, envia uma mensagem do tipo *Modify-State* para inclusão de uma nova regra na tabela de fluxos do dispositivo de encaminhamento, seguida de uma mensagem *Send-Packet* para lidar com o pacote que gerou o *Packet-In*. A baixa estatística deste pacote na Tab. 3.3 se deve a um recurso nativamente explorado pelo componente *l2_learning* do controlador POX, que faz a referência do pacote através do parâmetro *buffer_id* diretamente na mensagem *Modify-State*, ao invés de o fazer na mensagem *Send-Packet.* Desta forma, com apenas uma mensagem o novo fluxo é instalado no dispositivo de encaminhamento, assim como o pacote que gerou a requisição deste novo fluxo é tratado, sem a necessidade de uma mensagem adicional de *Send-Packet.*

As estatísticas da Tabela 3.3 demonstram que apenas 0,78% das mensagens processadas são do tipo *Send*-Packet, pois estes são utilizados apenas para *broadcast*, instruindo o dispositivo de encaminhamento a enviar o respectivo pacote para todas as portas. A Tabela 3.4 mostra o resultado do mesmo experimento realizado após implementar o envio de mensagens do tipo *Send-Packet*, ao invés do uso apenas da mensagem *Modify-State.*

Tabela 3.4 – Estatísticas de pacotes processados e carga no canal de controle para cada mensagem OpenFlow transmitida no experimento preliminar com implementação de *Send-Packet*

Categoria de mensagem	**Tipo de mensagem**	**Mensagens processadas**	**Carga no canal de controle**	**Direção**
Simétrica	*Hello (request)*	0,06%	0,04%	C → S
	Hello (reply)	0,06%	0,003%	S → C
	Echo (request)	1,29%	0,08%	S → C
	Echo (reply)	0,69%	0,44%	C → S
	Vendor (request)	0,00%	0,00%	C → S
	Vendor (reply)	0,00%	0,00%	S → C
Assimétrica	*Packet-In*	88,24%	77,86%	S → C
	Flow-Removed	0,00%	0,00%	S → C
	Port-Status	0,22%	0,03%	S → C
	Error	0,00%	0,00%	S → C
Controlador para switch	*Features (request)*	0,03%	0,02%	C → S
	Features (reply)	0,06%	0,04%	S → C
	Configuration (request)	0,00%	0,00%	C → S

	Configuration (reply)	0,00%	0,00%	S → C
	Configuration (set)	0,03%	0,02%	C → S
	Modify-State	46,68%	58,21%	C → S
	Send-Packet	47,07%	36,18%	C → S
	Read-State (request)	5,41%	5,07%	C → S
	Read-State (reply)	10,08%	21,99%	S → C
	Barrier (request)	0,03%	0,02%	C → S
	Barrier (reply)	0,06%	0,003%	S → C

O crescimento foi expressivo, de 0,78% para 47,07% do total de mensagens processadas no controlador, representando um aumento de cerca de 30% na quantidade total de mensagens processadas e 7% de carga adicional no canal de controle, em relação à estatística anterior.

As regras de encaminhamento instaladas nos *switches* possuem campos específicos a serem comparados com o cabeçalho dos pacotes recebidos, para que seja definido um *match* com a determinada regra. Desta forma, o *switch* utiliza a regra de encaminhamento mais específica, ou seja, a que obteve um maior número de campos iguais aos do pacote recebido, para definir qual ação a ser tomada. Estas regras também podem ser mais genéricas, sem especificar todos os campos possíveis, abrangendo um maior número de pacotes sem a necessidade de instalação de regras adicionais. O componente *l2_learning* do controlador POX nativamente instala regras com o maior nível de especificação, restringindo em grande parte o número de *matches* possíveis e, assim, aumentando a quantidade de regras necessárias para atender todos os pacotes que trafegam no *switch.* O Quadro 3.11 mostra o código que faz a instalação de regras nos *switches*, onde para se estabelecer regras mais genéricas foram inibidas as especificações de porta TCP de origem e destino.

Quadro 3.11 – Código para definição de regras mais genéricas no componente *l2_learning*

```
msg = of.ofp_flow_mod() # cria mensagem do tipo Modify-State
msg.match = of.ofp_match.from_packet(packet, event.port) # informacoes do pacote
msg.match.tp_src = None # inibe a especificacao de portas nos fluxos
msg.match.tp_dst = None # inibe a especificacao de portas nos fluxos
msg.idle_timeout = 30 # define o valor de idle timeout para as regras
msg.hard_timeout = 0
msg.actions.append(of.ofp_action_output(port = port))
msg.data = event.ofp # trata o pacote que gerou a mensagem packet in
self.connection.send(msg)
```

A Tabela 3.5 mostra as estatísticas obtidas ao configurar o componente *l2_learning* com instalação de regras mais genéricas, no mesmo cenário preliminar. Neste experimento foram removidas as especificações de portas TCP de origem e destino, por se tratar de uma especificação não muito usual ou necessária, pois muitas vezes teremos regras basicamente iguais, sendo diferenciadas apenas por suas portas TCP. Esta especificação poderia ser indicada para controle de acesso ou segregação de tráfego em determinadas portas, com intuito de prover um caminho ou tratamento diferenciado para esses casos. Entretanto, especificá-las sem nenhum propósito acaba por praticamente gerar regras duplicadas na tabela de fluxos, causando assim um aumento no número de mensagens trafegadas no canal de controle sem nenhum ganho real.

Tabela 3.5 – Estatísticas de pacotes processados e carga no canal de controle para cada mensagem OpenFlow transmitidas no experimento preliminar com regras genéricas

Categoria de mensagem	**Tipo de mensagem**	**Mensagens processadas**	**Carga no canal de controle**	**Direção**
Simétrica	*Hello (request)*	0,47%	0,28%	C → S
	Hello (reply)	0,24%	0,02%	S → C
	Echo (request)	8,02%	0,54%	S → C
	Echo (reply)	7,98%	4,79%	C → S
	Vendor (request)	0,00%	0,00%	C → S
	Vendor (reply)	0,00%	0,00%	S → C
Assimétrica	*Packet-In*	48,35%	19,12%	S → C
	Flow-Removed	0,00%	0,00%	S → C
	Port-Status	0,24%	0,03%	S → C
	Error	0,00%	0,00%	S → C
Controlador para switch	*Features (request)*	0,23%	0,14%	C → S
	Features (reply)	0,24%	0,20%	S → C
	Configuration (request)	0,00%	0,00%	C → S
	Configuration (reply)	0,00%	0,00%	S → C
	Configuration (set)	0,23%	0,15%	C → S
	Modify-State	45,07%	53,83%	C → S
	Send-Packet	3,29%	2,43%	C → S
	Read-State (request)	42,49%	38,24%	C → S
	Read-State (reply)	42,69%	80,08%	S → C
	Barrier (request)	0,23%	0,14%	C → S
	Barrier (reply)	0,24%	0,02%	S → C

Com a configuração de regras genéricas houve uma diminuição significativa no número de mensagens do tipo *Packet-In*, e, por consequência, uma diminuição nas mensagens do tipo *Modify-State*. No caso, ocorre um maior equilíbrio na proporção de mensagens em relação às do tipo *Read-State*, pois a quantidade destas mensagens é a mesma nos dois experimentos, variando apenas a quantidade total de pacotes no canal. Em comparação com o experimento inicial, demonstrado na Tab. 3.3, houve uma queda de cerca de 87% do número

de mensagens do tipo *Packet-In* e *Modify*-State. Isto acontece devido a uma única regra atender um número maior de pacotes, independente das portas TCP de origem e destino que utilizem, fazendo com que o *switch* não precise consultar o controlador para a instalação de novas regras. Desta forma, o uso do canal de controle é reduzido, contribuindo para a prevenção de sua sobrecarga.

3.2 Análise Experimental

Nesta seção serão realizados os experimentos voltados para análise do uso do canal de controle entre os dispositivos de encaminhamento e o controlador da rede, com base nos parâmetros e análises obtidos no estudo preliminar. O foco do estudo será nas mensagens de maior frequência que trafegam no canal, com três variações de topologias para simular diferentes cenários.

3.2.1 Cenários

Para uma análise maior e diversificada, os experimentos foram realizados em três diferentes topologias, com variação no número de *switches*, porém mantendo fixo o número de estações, para criar uma padronização na quantidade de tráfego e requisições entre as diversas estações. Desta maneira, é possível avaliar de forma mais equivalente o tráfego de mensagens do protocolo OpenFlow entre as topologias citadas.

Os cenários emulados consistem em 62 estações, 1 servidor de vídeo e 1 servidor web para as seguintes topologias:

(i) estrela com apenas um *switch*;

(ii) árvore com profundidade 3 e largura 4, totalizando 21 *switches*;

(iii) linear com 8 *switches* conectados cada um a 8 estações.

As Figuras 3.6 a 3.8 mostram as topologias mencionadas, com as devidas conexões e a posição dos servidores de vídeo e web, para melhor visualização. As estações estão agrupadas, sendo indicada sua quantidade ao lado de cada uma delas.

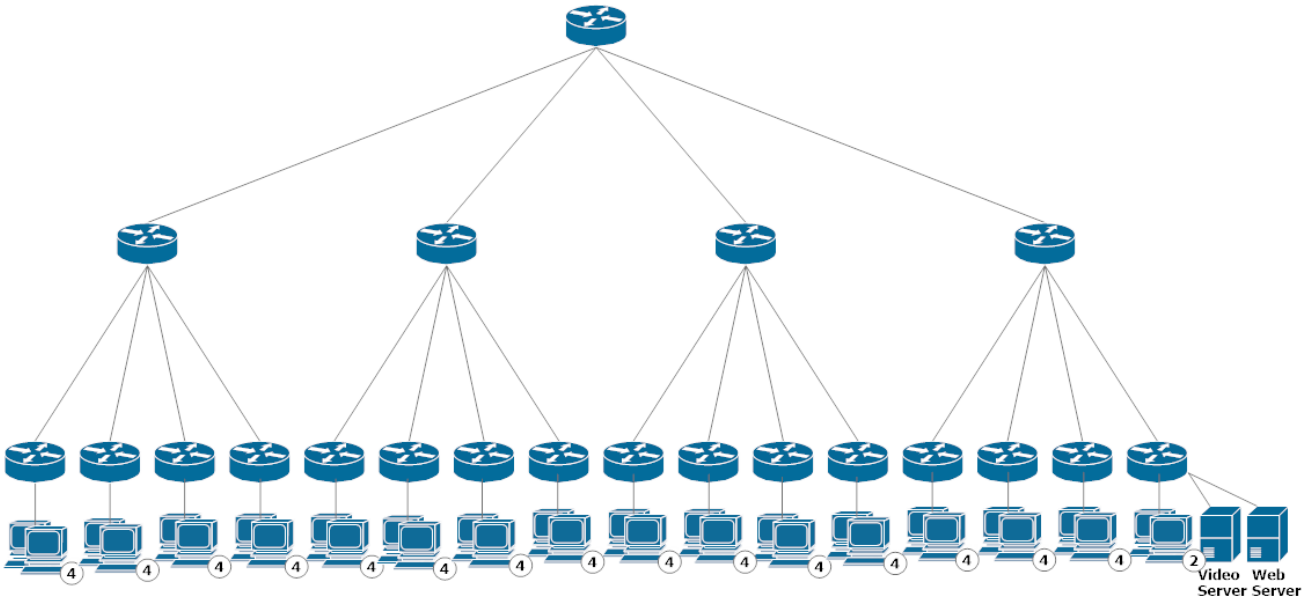


Figura 3.6 – Topologia árvore

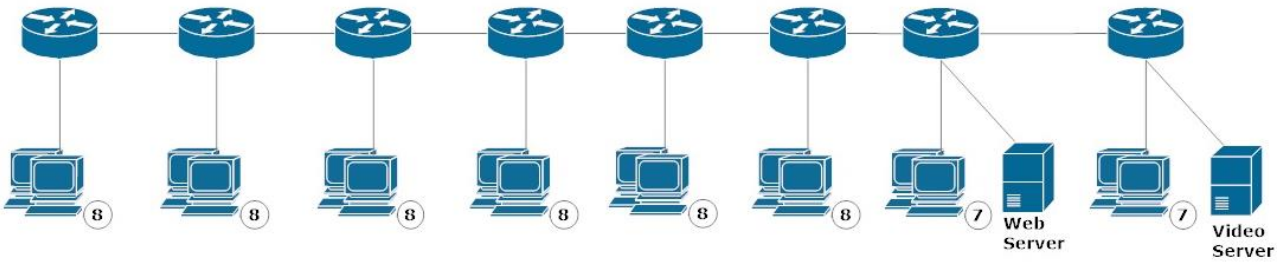


Figura 3.7 – Topologia linear

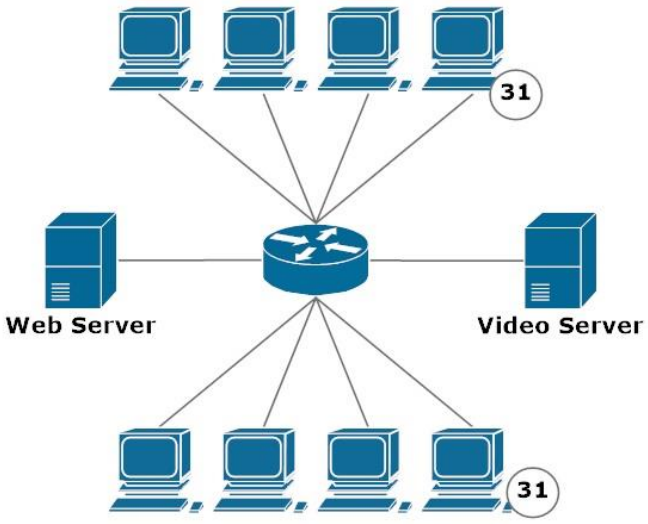


Figura 3.8 – Topologia estrela

O tráfego gerado nestes experimentos segue o mesmo padrão de perfil utilizado no estudo preliminar, com os parâmetros descritos na Tab. 3.1. Desta forma, 1 usuário faz requisição de vídeo para cada 3 que fazem requisições web, devido ao maior tráfego gerado pelo *streaming*, pois desta forma a quantidade de tráfego gerado pelos dois servidores é semelhante, fazendo com que o tráfego web também tenha significância, em todos os experimentos. As requisições de cada estação, tanto para web quanto para vídeo, são realizadas com o intervalo de 20 segundos, com todos eles ativos durante todo o experimento. Foi utilizada apenas uma máquina física para as todas as simulações realizada, em conjunto com uma máquina virtual rodando o sistema Ubuntu, uma rede emulada no emulador Mininet com *switches* do tipo *Open vSwitch* em modo *kernel* e um controlador POX, conforme descrito na Tab. 3.2 do experimento preliminar.

Com base no estudo preliminar, foi mantida a implementação nativa do componente *l2_learning* que evita o envio de mensagens do tipo *Send-Packet*, habilitando o tratamento dos pacotes através das mensagens do tipo *Modify-State* apenas. Da mesma forma, foi implementada a instalação de regras mais genéricas na tabela de fluxos, com remoção dos campos de porta TCP de origem e destino, pois não é de interesse deste trabalho a segregação por diferentes portas. Desta maneira, é possível focar nas mensagens de maior frequência e peso no canal de controle, definidas anteriormente como sendo as mensagens do tipo *Packet-In, Modify-State* e *Read-State.*

O monitoramento e obtenção dos dados estatísticos referentes ao protocolo OpenFlow e suas mensagens trafegadas no canal de controle foram realizados através do componente *ic_monitor*, de própria autoria, para a requisição de estatísticas através de mensagens do tipo *Read-State* enviadas aos dispositivos de encaminhamento. Como essas mensagens também contribuem para o aumento do tráfego no canal de controle, onde de acordo com o nosso estudo preliminar essa contribuição é, a princípio, significativa, elas também precisam ser analisadas em conjunto com as mensagens de controle.

Todos os experimentos foram repetidos cinco vezes, com duração de 3 minutos cada, totalizando 15 minutos para cada um deles, com intuito de obter dados mais confiáveis e menor margem de erro.

3.2.2 Metodologia de Avaliação

Para a análise quantitativa do uso de canal de controle com base no protocolo OpenFlow, tanto por tráfego de mensagens de controle quanto mensagens de monitoramento, foram utilizadas métricas para melhor orientar os experimentos e avaliações, conforme segue:

- quantidade de mensagens do tipo *Packet-In* processadas pelo controlador, por segundo;
- carga trafegada no canal de controle para mensagens do tipo *Packet-In* entre os dispositivos de encaminhamento e o controlador, em *bits* por segundo;
- quantidade de mensagens do tipo *Modify-State* processadas pelos dispositivos de encaminhamento, por segundo;
- carga trafegada no canal de controle para mensagens do tipo *Modify-State* entre o controlador e os dispositivos de encaminhamento, em *bits* por segundo.
- carga trafegada no canal de controle para mensagens do tipo *Read-State* entre o controlador e os dispositivos de encaminhamento, em *bits* por segundo.

Essas métricas auxiliam na avaliação quantitativa das mensagens que trafegam no canal de controle em função do tempo, tanto na direção dos dispositivos de encaminhamento quanto na direção do controlador, com base nas principais mensagens utilizadas.

Em conjunto com as métricas definidas, um parâmetro de configuração das regras é variado, estando este presente em qualquer rede baseada no protocolo OpenFlow, que é o tempo de ociosidade da regra, chamado *idle timeout.* Este parâmetro indica o tempo necessário para que um registro na tabela de fluxos de um *switch* com base no protocolo OpenFlow seja removido, indicando que aquela regra de encaminhamento expirou. A variação deste parâmetro é importante para a compreensão dos diferentes comportamentos do protocolo OpenFlow, influenciando no aumento ou diminuição na quantidade de mensagens trafegadas no canal de controle. O *idle timeout* é configurado no componente *l2_learning* do controlador POX, que por padrão possui o valor de 30 segundos, o que significa que cada regra instalada pelo controlador nos *switches* possui um tempo máximo de 30 segundos em que pode permanecer inativa antes de ser removida. A inatividade é definida pela falta de *match* de pacotes para aquela regra específica em um determinado intervalo de tempo, tendo seu valor máximo aceitável definido pelo parâmetro *idle timeout*, conforme citado.

Em relação às mensagens de monitoramento do tipo *Read-State*, a frequência em que são solicitadas é fixada em 5 segundos, com intuito inicial de verificar o impacto dessas mensagens no canal de controle em relação ao seu tamanho, ou seja, a carga gerada por elas, ao invés de sua quantidade. As requisições deste tipo de mensagem possuem um tamanho fixo, porém as respostas variam conforme a quantidade de *switches*, regras instaladas, tipo de estatística, entre outros fatores. Em um segundo momento, a frequência de monitoramento será variada entre 1, 5 e 10 segundos, e o parâmetro *idle timeout* fixado em 10 segundos, para

análise do aumento das mensagens de monitoramento. A Tabela 3.6 mostra os parâmetros de configuração adotados nas duas etapas dos experimentos.

Tabela 3.6 – Parâmetros definidos para cada etapa dos experimentos

Parâmetros	**Etapa 1**	**Etapa 2**
Tempo de *idle timeout*	1,5,10,30,60,120 segundos	Fixo em 10 segundos
Frequência de monitoramento	Fixo em 5 segundos	1, 5, 10 segundos

3.3 Resultados experimentais

Com a realização dos experimentos nos parâmetros definidos anteriormente, foram obtidos dados relevantes para o levantamento de informações e análise das mensagens mais frequentes do protocolo OpenFlow, possibilitando a avaliação do uso do canal de controle. A partir destes dados foram gerados gráficos para melhor visualização e comparação, separados por topologia e pelas métricas de quantidade de pacotes processados e da carga gerada no canal de controle.

3.3.1 Mensagens processadas no canal de controle

As Figuras 3.9 a 3.11 mostram o número de pacotes processados por segundo, tanto pelo controlador quanto pelos dispositivos de encaminhamento, em relação à variação do parâmetro *idle timeout*, para as topologias estrela, linear e árvore, respectivamente. As mensagens do tipo *Modify-State* e *Read-State (request)* são enviadas do controlador para os dispositivos de encaminhamento, enquanto as mensagens do tipo *Packet-In* e *Read-State (reply)* são enviadas dos dispositivos para o controlador, totalizando as mensagens de maior peso e frequência nos dois sentidos de comunicação no canal de controle. Os gráficos demonstram que quanto maior o tempo definido para expiração das regras – *idle timeout* – menor é a quantidade de pacotes processados, tanto no controlador quanto nos dispositivos de encaminhamento, em todas as topologias experimentadas.

Para o valor de *idle timeout* de 1 segundo, temos uma grande elevação no número de mensagens processadas, pois a exclusão das regras de encaminhamento é quase imediata, fazendo com que os dispositivos de encaminhamento requisitem ao controlador a instalação de novas regras com uma frequência muito maior. Em contrapartida, para valores de *idle timeout* mais altos, como 60 ou 120 segundos, temos uma grande redução do número de mensagens processadas. Contudo, a tendência é que muitas das regras instaladas estejam

ociosas e sejam desnecessárias, podendo esgotar o espaço da tabela de fluxos do dispositivo de encaminhamento.

A topologia estrela é a mais simples em comparação as demais por ter apenas um único *switch*. Por conter um número elevado de estações conectadas, o *switch* produz muitas mensagens de controle e necessita instalar diversas regras de encaminhamento, pois toda a comunicação da rede irá obrigatoriamente passar por ele. Entretanto, para mensagens de monitoramento do tipo *Read-State*, temos um menor número trafegando no canal de controle, pois apenas um *switch* é monitorado e responde ao controlador com suas estatísticas. Esse tipo de topologia é geralmente utilizado em ambientes domésticos e de pequenas empresas, tendo um número menor de estações e tende a ter um baixo custo de implementação, em relação às topologias que utilizam uma maior quantidade de *switches*.

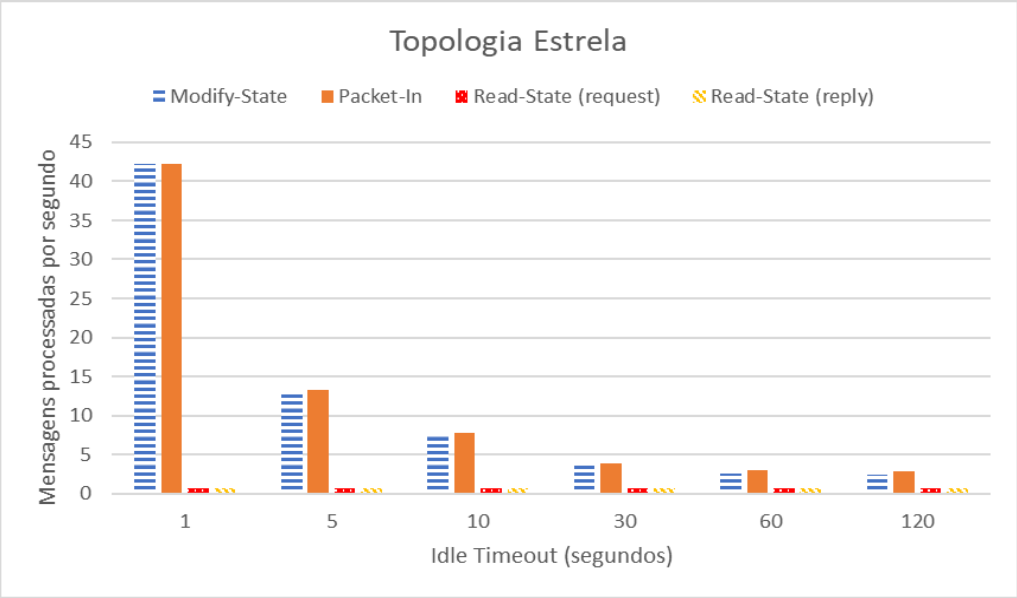


Figura 3.9 – Gráfico dos pacotes processados por segundo vs configuração de idle timeout, para a topologia estrela

A topologia linear possui um número maior de *switches*, no total de 8, o que implica em um maior número de mensagens de controle e de monitoramento, como é possível observar no gráfico da Fig. 3.10. Apesar da quantidade de estações e de tráfego de rede ser o mesmo, a quantidade de mensagens processadas é cerca de duas a três vezes maior que o observado na topologia estrela. Devido à grande parte das estações estarem conectadas em *switches* diferentes, se faz necessária a instalação de regras em vários deles até que uma determinada estação alcance outra. Por consequência, ocorre uma maior troca de mensagens entre o controlador e os diversos dispositivos de encaminhamento.

Em relação às mensagens de monitoramento do tipo *Read-State*, temos um número oito vezes maior em relação à topologia estrela, sendo diretamente proporcional ao número de

switches. Esse tipo de topologia é utilizado para diversos tipos de rede, desde domésticas até de grandes empresas, podendo ser integrada com outros tipos de topologias.

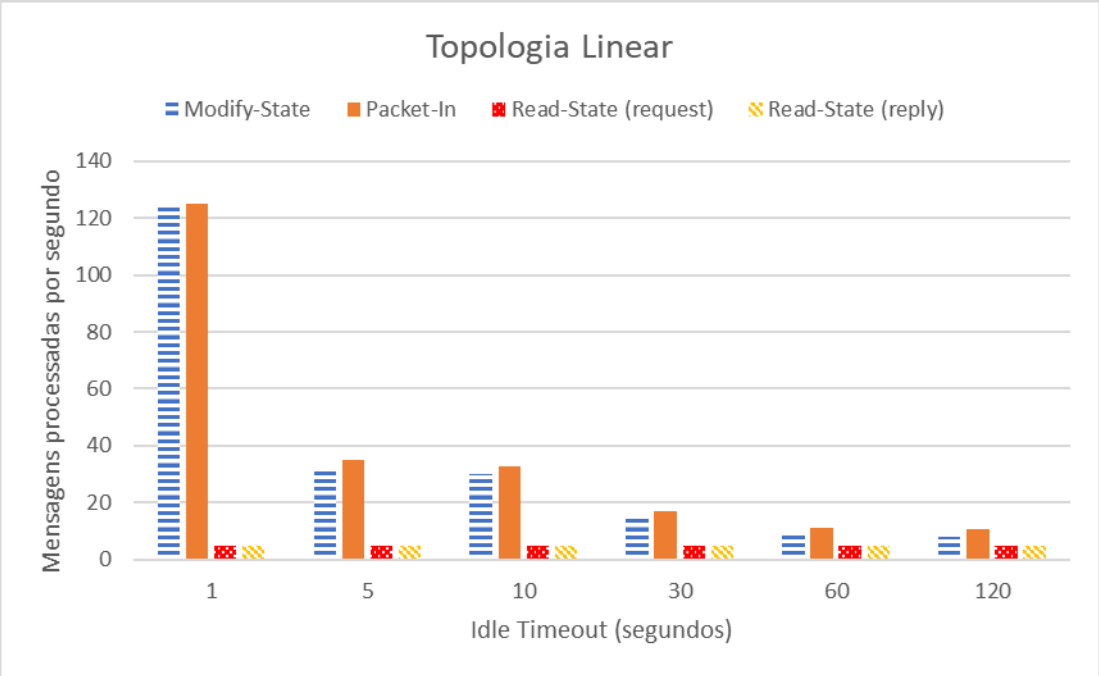


Figura 3.10 – Gráfico dos pacotes processados por segundo vs configuração de idle timeout, para a topologia linear

A topologia árvore possui um número ainda maior de *switches*, no total de 21, o que implica em um número ainda maior de mensagens de controle e de monitoramento, como é possível observar no gráfico da Fig. 3.11. A quantidade de mensagens processadas é cerca de quatro a cinco vezes maior que o observado na topologia estrela. Neste caso, além de grande parte das estações estarem em *switches* diferentes, existem *switches* de interconexão, os quais definem os níveis da árvore, que aumentam a quantidade de *switches* existentes entre cada estação. Desta forma, o tráfego entre 75% das estações necessita percorrer cinco *switches* para que uma estação alcance a outra, o que influencia diretamente na quantidade de mensagens de controle que são processadas.

Em relação às mensagens de monitoramento do tipo *Read-State*, temos um número 21 vezes maior em relação à topologia estrela, mostrando novamente a influência causada pelo número de *switches.* Logo, deve sempre ser levado em conta a necessidade real de um número maior de *switches* em detrimento do impacto causado no canal de controle, pois cada um deles contribui para o aumento não só de mensagens de monitoramento como também de controle. Esse tipo de topologia é mais utilizado em estruturas de datacenter, onde são construídos diferentes níveis de rede baseados nos níveis de árvore da topologia.

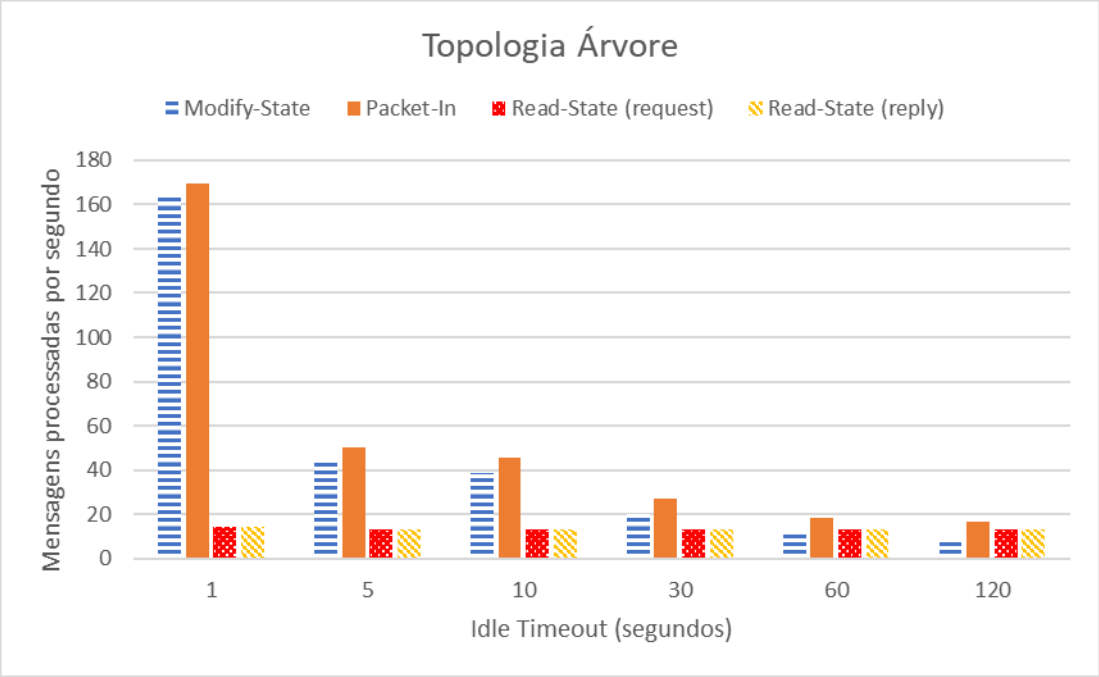


Figura 3.11 – Gráfico das mensagens processadas por segundo vs configuração de idle timeout, para a topologia árvore

As diferenças observadas entre quantidade de mensagens do tipo *Modify-State* e *Packet-In* são devido às mensagens do tipo *Send-Packet*, que somadas às do tipo *Modify-State* constituem o mesmo valor que as do tipo *Packet-In*. Afinal, para cada mensagem *Packet-In* deve existir uma resposta, seja do tipo *Modify-State* ou *Send-Packet*, para que os pacotes sejam tratados. É possível observar também que as mensagens do tipo *Read-State* se mantêm constante para os diferentes valores de *idle timeout*, tanto para *request* quanto *reply*, pois a quantidade de mensagens independe do valor de *idle timeout* configurado, mas sim do número de *switches* e da frequência de monitoramento utilizada. Logo, para topologias com um maior número de *switches*, teremos por consequência uma maior quantidade de mensagens de monitoramento trafegando no canal de controle, conforme já mencionado.

3.3.2 Carga no canal de controle

A quantidade de pacotes que trafegam no canal de controle certamente é um critério crucial na avaliação do uso do canal, porém o peso de cada mensagem pode ser diferente mesmo com quantidades semelhantes, devido à diferença no tamanho das mensagens. Com isso, precisamos avaliar tanto a quantidade de mensagens trafegadas quanto a carga que cada uma delas possui, para que assim possamos ter uma noção real do verdadeiro impacto provocado no canal de controle. As Figuras 3.12 a 3.14 mostram a carga que trafega no canal de controle, em *bits* por segundo, tanto na direção do controlador quanto na dos dispositivos

de encaminhamento, em relação à variação do parâmetro *idle timeout*, para as topologias estrela, linear e árvore, respectivamente.

Assim como nos gráficos de quantidade de mensagens processadas, para o valor de *idle timeout* de 1 segundo, temos uma elevação na carga total trafegada no canal de controle em relação aos demais valores, porém neste caso com mais ênfase nas mensagens do tipo *Packet-In*. Isto ocorre, pois, os *switches* possuem um limite de *buffer* para armazenar os pacotes, onde ao atingi-lo faz com que as mensagens do tipo *Packet-In* incluam dentro delas todo o pacote a ser tratado, ao invés de apenas referenciar o *buffer id* no qual está armazenado. Desta forma, o tamanho da mensagem *Packet-In* é somado ao tamanho do pacote, tornando-o muitas vezes bem maior que o original, o que impacta diretamente na carga do canal de controle. Em contrapartida, para valores de *idle timeout* mais altos como 60 ou 120 segundos, temos uma grande redução da carga das mensagens do tipo *Packet-In* e *Modify-State*, porém é possível observar um grande aumento nas mensagens de monitoramento.

Nesta análise em questão, temos um comportamento diferente para as mensagens de monitoramento do tipo *Read-State.* Com valores de *idle timeout* pequenos temos um número menor de regras instaladas na tabela de fluxos, pois expiram mais rapidamente, enquanto que para valores maiores, estas tabelas tendem a conter mais regras instaladas. Por esse motivo, o tamanho das mensagens do tipo *Read-State (reply)* são muito maiores para um maior valor de *idle timeout*, pois haverá muito mais regras a serem reportadas na resposta de estatísticas de monitoramento solicitada pelo controlador – *Read-State* –, aumentando o tamanho da mensagem. Para mensagens do tipo *Read-State (request)* o tamanho é fixo, variando apenas sua quantidade de acordo com o número de *switches* na rede.

O gráfico da topologia estrela na Fig. 3.12 demonstra um crescimento superior de mensagens *Packet-In* em relação às demais, principalmente ao se comparar com a quantidade de mensagens processadas, demonstradas na Fig. 3.9, onde mensagens do tipo *Modify-State* possuem quase o mesmo valor das mensagens *Packet-In*. Neste caso o crescimento também é proporcionalmente maior que nas demais topologias para um valor de *idle timeout* de 5 segundos, pois como possui apenas um *switch*, todas as mensagens do tipo *Packet-In* trafegam por ele, o que tende a esgotar sua capacidade de *buffer* com mais rapidez.

Para mensagens do tipo *Read-State (request)* foi obtido uma carga constante de cerca de 66 *bits* por segundo, o que é basicamente inexpressivo em comparação a carga das demais mensagens avaliadas. Desta forma, nos gráficos da Fig. 3.12 quase não é possível visualizar a barra desse tipo de mensagem devido à escala comparativa com as outras mensagens.

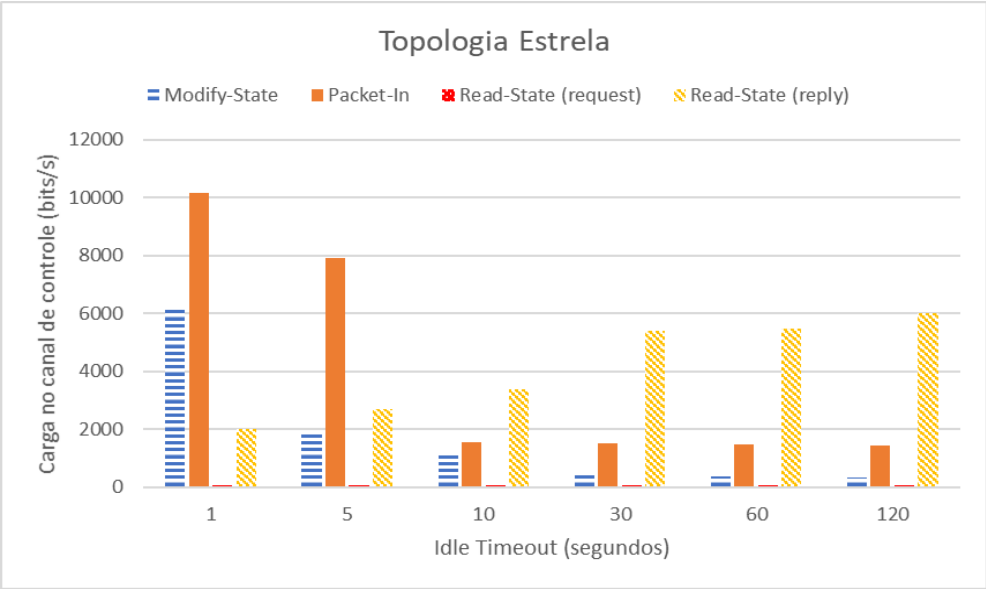


Figura 3.12 – Gráfico da carga no canal de controle vs configuração de idle timeout, para a topologia estrela

Para o gráfico da topologia linear, demonstrado na Fig. 3.13, temos um crescimento notável de mensagens do tipo *Packet-In* para o valor de *idle timeout* de 1 segundo, sendo cerca de 3 vezes maior que para o valor de 5 segundos. Entretanto, para valores de *idle timeout* maiores essas mensagens diminuem drasticamente, enquanto as mensagens do tipo *Read-State (reply)* crescem em grandes proporções. Como esta topologia possui um número maior de *switches*, temos por consequência uma carga maior de mensagens de monitoramento, com cerca de três vezes mais carga em relação à topologia estrela. Para mensagens do tipo *Read-State (request)* foi obtido uma carga constante de cerca de 542 *bits* por segundo, sendo possível visualizar no gráfico, apesar do baixo impacto que possui.

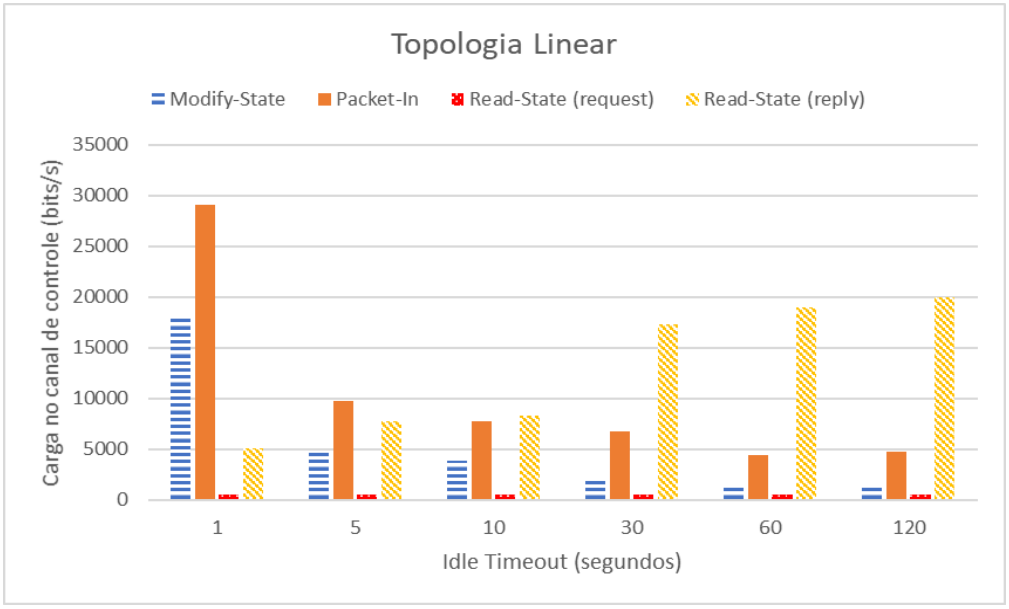


Figura 3.13 – Gráfico da carga no canal de controle vs configuração de idle timeout, para a topologia linear

Por fim, o gráfico da topologia árvore, demonstrado na Fig. 3.14, segue o mesmo crescimento expressivo de mensagens do tipo *Packet-In* para o valor de *idle timeout* de 1 segundo, diminuindo também para valores de *idle timeout* maiores, enquanto as mensagens do tipo *Read-State (reply)* crescem em grandes proporções. Por possuir alguns *switches* centralizadores, que fazem a interconexão dos níveis da árvore, a diminuição de mensagens *Packet-In* com o valor de *idle timeout* de 5 segundos não chega a ser tão expressiva quanto na topologia linear, pois estes *switches* tendem a esgotar seu *buffer* com maior facilidade, devido à grande quantidade de mensagens recebidas dos demais *switches*. Para mensagens do tipo *Read-State (request)* foi obtido uma carga constante de cerca de 1424 *bits* por segundo, sendo possível visualizar no gráfico, porém com baixo impacto na carga do canal de controle.

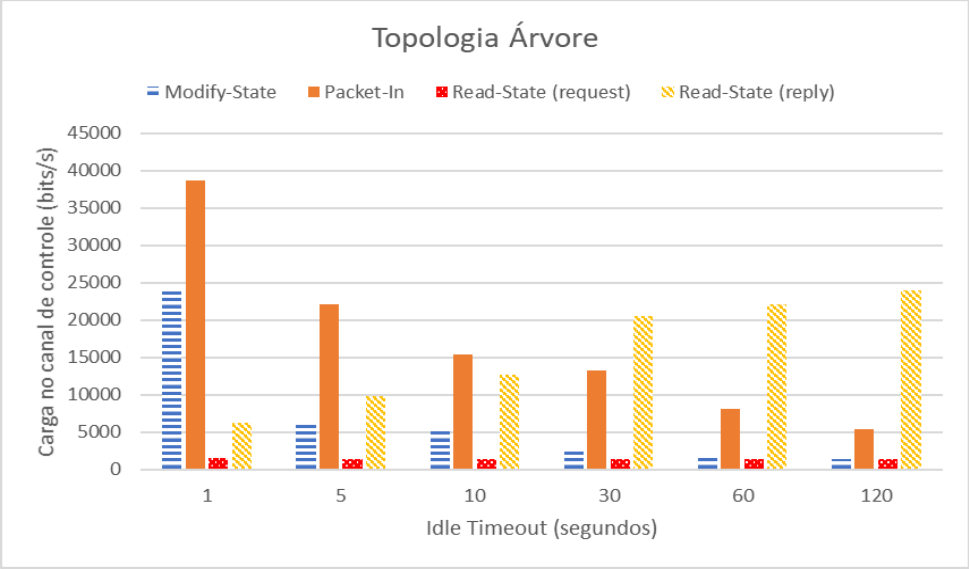


Figura 3.14 – Gráfico da carga no canal de controle vs configuração de idle timeout, para a topologia árvore

Ao se analisar as estatísticas da quantidade de mensagens trafegadas no canal de controle em conjunto com a carga que cada uma possui, para os diversos valores de *idle timeout*, obtemos uma melhor visão do uso efetivo do canal. Em uma primeira percepção, aumentar o valor de *idle timeout* parece ser o mais vantajoso para reduzir o uso do canal de controle, pois à medida que esse valor aumenta a quantidade de pacotes processados é reduzida. Porém ao analisar a carga que cada mensagem possui em diferentes configurações, percebemos a desvantagem em aumentar o valor de *idle timeout*, devido à maior carga trafegada no canal de controle. Enquanto um *idle timeout* de 1 segundo eleva em muito o número de pacotes processados na rede e no controlador, também diminui bastante a carga gerada pelas mensagens de monitoramento. Para valores maiores de *idle timeout*, temos o comportamento inverso, com diminuição da quantidade de pacotes processados e aumento da

carga de monitoramento. Além das questões citadas, o parâmetro *idle timeout* também influencia na quantidade de regras instaladas na tabela de fluxos dos *switches* que ficam ociosas, sendo um ponto a ser levado em consideração na escolha de um valor de *idle timeout* ideal.

3.3.3 Regras ociosas na tabela de fluxos

Os gráficos das Figuras 3.15 a 3.17 mostram a quantidade total de regras de encaminhamento instaladas na tabela de fluxos dos *switches* e quantas delas ficam ociosas, em média, para diferentes valores de *idle timeout.* Uma regra de encaminhamento é considerada ociosa quando não há acréscimo no número de pacotes que realizaram *match* com ela, entre dois instantes seguidos de monitoramento. É possível observar, no geral, que a medida em que o valor de *idle timeout* aumenta, maior é a quantidade de regras de encaminhamento ociosas na tabela de fluxos dos *switches,* para as três topologias em questão. Isto ocorre pelo fato de que a maior parte do tráfego da rede tem um curto período de duração, com requisições que levam menos de segundos para serem concluídas, criando assim espaços de tempo ociosos, onde as regras de encaminhamento não são utilizadas. Com valores mais altos de *idle timeout,* as requisições realizadas pelas estações, que possuem um tempo médio de 20 segundos, acabam por utilizar as regras antes que estas expirem. Por consequência estas regras não são excluídas mesmo que em boa parte do tempo estejam ociosas.

A capacidade máxima de fluxos que um *switch* pode instalar é limitada, pois a maioria dos *switches* baseados no protocolo *OpenFlow* utilizam memória TCAM, que possui uma capacidade pequena de armazenamento e um alto custo [3]. Por esse motivo, é fundamental estar atento à quantidade de regras instaladas na tabela de fluxos dos *switches,* para que não haja problemas por falta de recurso, o que pode levar ao ponto crítico onde a rede pare de funcionar.

Em todas as topologias a variação da quantidade de regras que estão ociosas em relação ao total instalado é, em suas devidas proporções, semelhante. Na topologia estrela, para valores de *idle timeout* de 1 segundo temos cerca 31 regras instaladas, com 3,6% delas ociosas, enquanto para um *idle timeout* de 120 segundos, temos um total de 234 regras instaladas, com 71% delas ociosas. O valor de *idle timeout* com melhor eficiência neste caso é o de 5 segundos, onde possui apenas 1,73% de regras ociosas do total instalado. Isto ocorre devido ao perfil de tráfego da rede, onde o valor de *idle timeout* em conjunto com o tempo de requisições permite fazer melhor uso das regras instaladas, mantendo ativa as mais utilizadas e descartando as que já cumpriram o seu papel de encaminhamento.

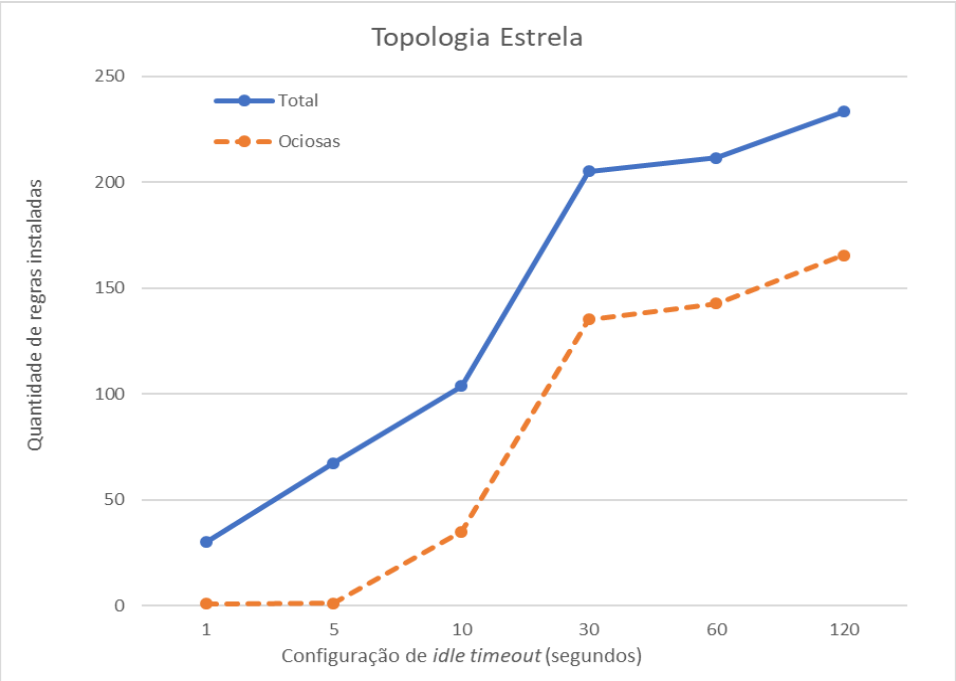


Figura 3.15 – Comparação entre a quantidade de regras total instaladas vs ociosas em função do idle timeout, para a topologia estrela.

Na topologia linear, para um *idle timeout* de 1 segundo temos cerca 95 regras instaladas, com 3,75% ociosas, enquanto para um *idle timeout* de 120 segundos, temos um total de 898 regras instaladas, com 67% ociosas. Novamente o valor de *idle timeout* com melhor eficiência é o de 5 segundos, com apenas 0,9% de regras ociosas em relação ao total.

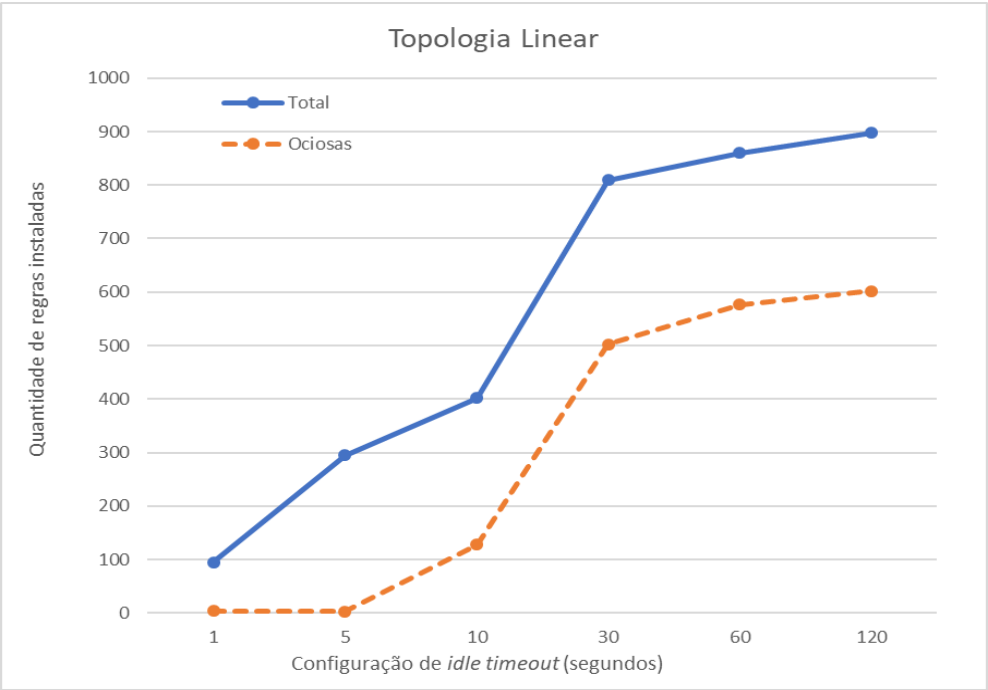


Figura 3.16 – Comparação entre a quantidade de regras total instaladas vs ociosas em função do idle timeout, para a topologias linear.

Na topologia árvore, para valores de *idle timeout* de 1 segundo temos cerca 102 regras instaladas, com 5,16% delas ociosas, enquanto para um *idle timeout* de 120 segundos, temos um total de 968 regras instaladas, com 69% delas ociosas. Mais uma vez o valor de *idle timeout* com melhor eficiência neste caso é o de 5 segundos, onde possui apenas 1,2% de regras ociosas do total instalado.

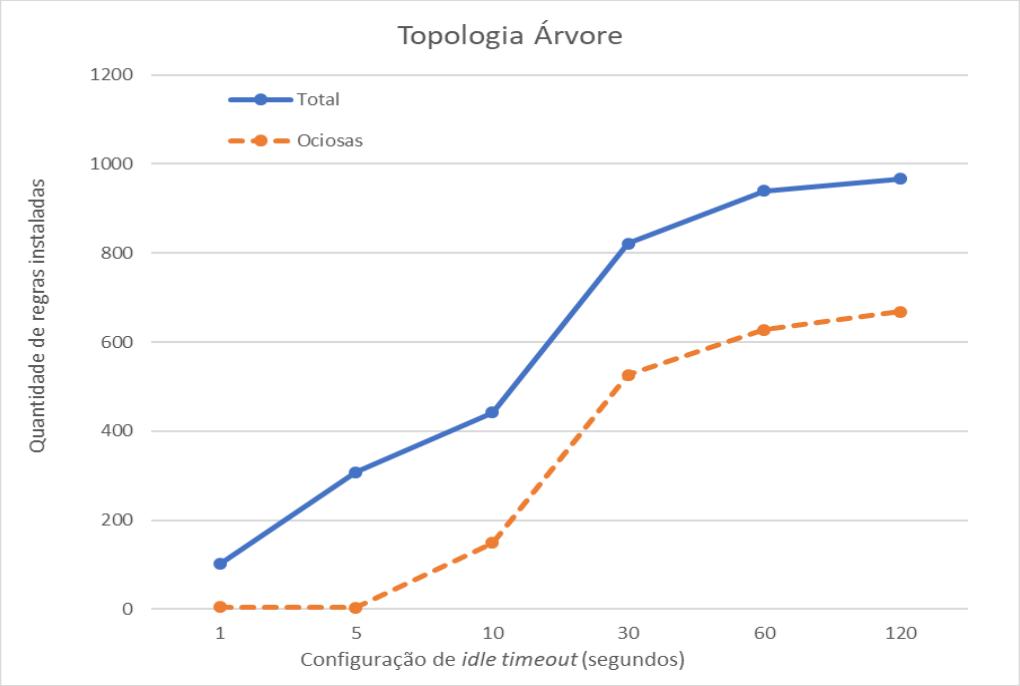


Figura 3.17 – Comparação entre a quantidade de regras total instaladas vs ociosas em função do idle timeout, para a topologia árvore

O conhecimento da quantidade de regras ociosas para determinados valores de *idle timeout* é uma ferramenta muito útil para definir o melhor valor de *idle timeout* a ser utilizado. Em geral, as análises realizadas nos mostram como apenas um simples parâmetro, no caso o *idle timeout*, pode afetar significativamente o tráfego do canal de controle e o consumo de recursos em uma rede baseada no protocolo *OpenFlow*. Unindo as informações obtidas nos experimentos, podemos melhor avaliar qual o valor de *idle timeout* mais próximo do ideal para as diferentes propostas de rede, baseado no custo-benefício, de acordo com o que for mais importante para a rede avaliada.

3.3.4 Variação da frequência de monitoramento

Na segunda etapa deste projeto foram analisadas as quantidades de pacotes processados e a carga no canal de controle ao se variar a frequência de monitoramento. Para se obter uma melhor comparação entre os cenários, o valor do parâmetro *idle timeout* foi fixado em 10 segundos. As simulações foram realizadas para as frequências de

monitoramento de 1, 5 e 10 segundos, e os resultados são mostrados nos gráficos das Figuras 3.18 a 3.23.

Nos gráficos apresentados é possível observar uma variação expressiva da carga no canal de controle gerado pelas mensagens do tipo *Read-State (reply)* para todas as topologias, quando comparada com a carga das demais mensagens analisadas. A variação da carga e quantidade de mensagens do tipo *Read-State (reply)* tende a ser linear, pois o número de requisições de estatísticas é fixo, independente da frequência. Desta maneira, para um monitoramento que realiza requisições de estatísticas a cada 1 segundo, isto é, com frequência de monitoramento de 1 segundo, temos um número de requisições 5 vezes maior do que para um monitoramento com frequência de 5 segundos. Entretanto, ao se comparar essa variação com a das demais mensagens, essa diferença tende a ser exponencial, pois para um mesmo valor de *idle timeout*, a variação das demais mensagens é pequena em relação às mensagens de monitoramento.

Com base na frequência de monitoramento de 5 segundos, temos uma carga no canal de controle relativa às mensagens de monitoramento *Read-State* cinco vezes maior quando utilizamos uma frequência de 1 segundo, e 2 vezes menor quando alteramos para 10 segundos, demonstrando a linearidade das variações causadas pela mudança da frequência de monitoramento. O mesmo ocorre para a quantidade de mensagens processadas, demonstrada na Fig 3.19. Em comparação às demais mensagens, isto é, a soma das mensagens *Packet-In* e *Modify-State*, a carga das mensagens *Read-State* no canal de controle é 6,5 vezes maior para a frequência de 1 segundo, 30% maior para 5 segundos e 35% menor para 10 segundos.

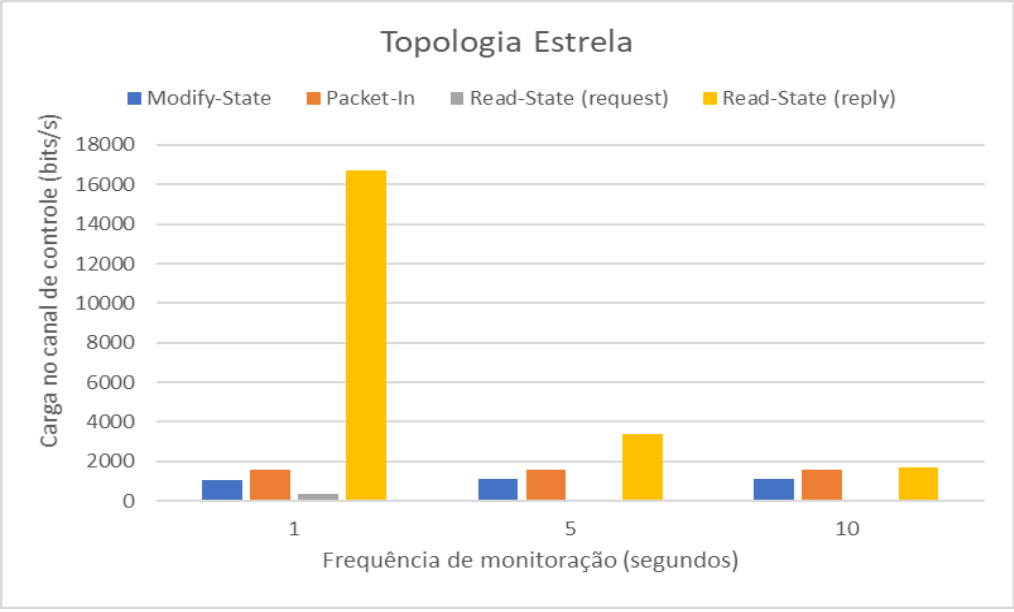


Figura 3.18 – Gráfico da carga no canal de controle em diferentes frequências de monitoramento, para a topologia estrela

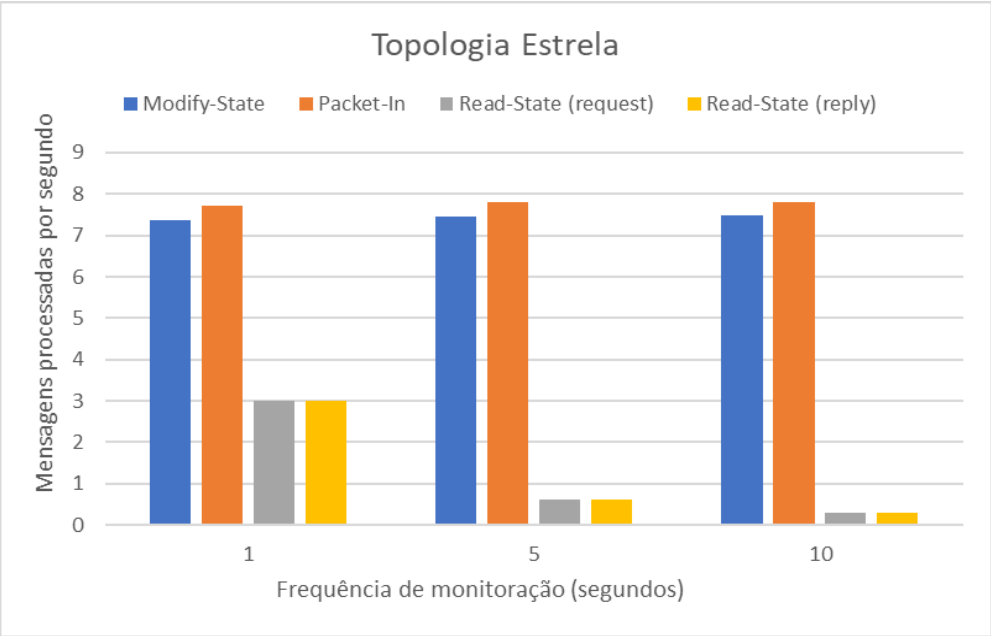


Figura 3.19 – Gráfico da quantidade de mensagens processadas por segundo em diferentes frequências de monitoramento, para a topologia estrela

Para a topologia linear, demonstrada no gráfico das Fig. 3.20 e 3.21, a variação das mensagens de monitoramento para os diferentes valores de frequência de monitoramento também é linear, tanto para a quantidade de mensagens processadas quanto para a carga. Realizando a mesma comparação com as demais mensagens, a carga das mensagens *Read-State* no canal de controle é 4 vezes maior para a frequência de 1 segundo, 24% menor para 5 segundos e 57% menor para 10 segundos.

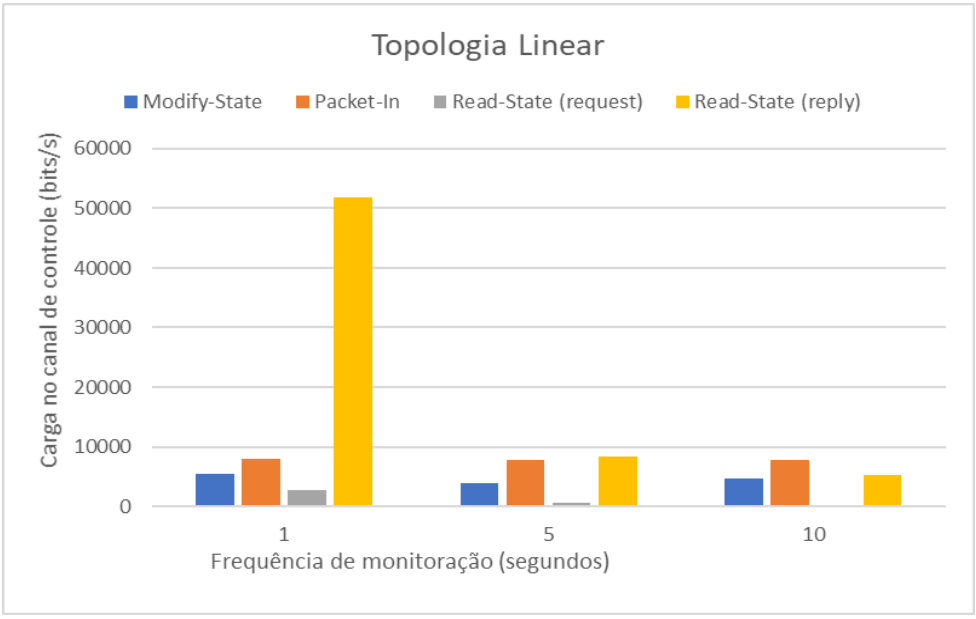


Figura 3.20 – Gráfico da carga no canal de controle em diferentes frequências de monitoramento, para a topologia linear

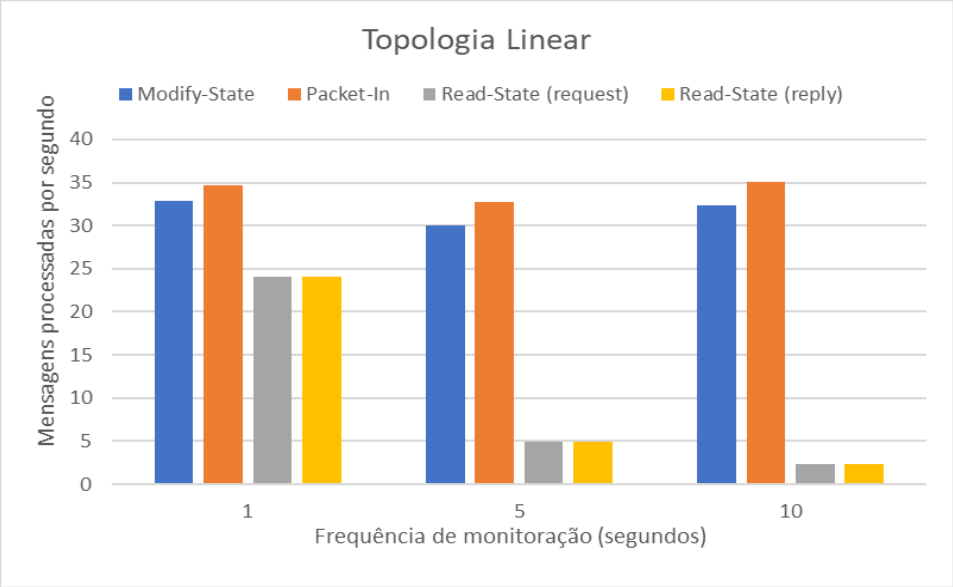


Figura 3.21 – Gráfico da quantidade de mensagens processadas por segundo em diferentes frequências de monitoramento, para a topologia linear

Para a topologia árvore, demonstrada no gráfico da Fig. 3.22 e 3.23, a variação das mensagens de monitoramento para os diferentes valores de frequência de monitoramento também é linear, tanto para a quantidade de mensagens processadas quanto para a carga. Entretanto, a quantidade de mensagens de monitoramento processadas quando monitorado a cada 1 segundo ultrapassa a quantidade das mensagens do tipo *Packet-In* e *Modify-State.* Isto deve ser levado em consideração, pois além da grande carga no canal de controle, o controlador tem um limite máximo de pacotes que pode processar por segundo, a depender dos recursos que dispõe. Realizando a mesma comparação com as demais mensagens, a carga das mensagens *Read-State* no canal de controle é 3 vezes maior para a frequência de 1 segundo, 36% menor para 5 segundos e 56% menor para 10 segundos.

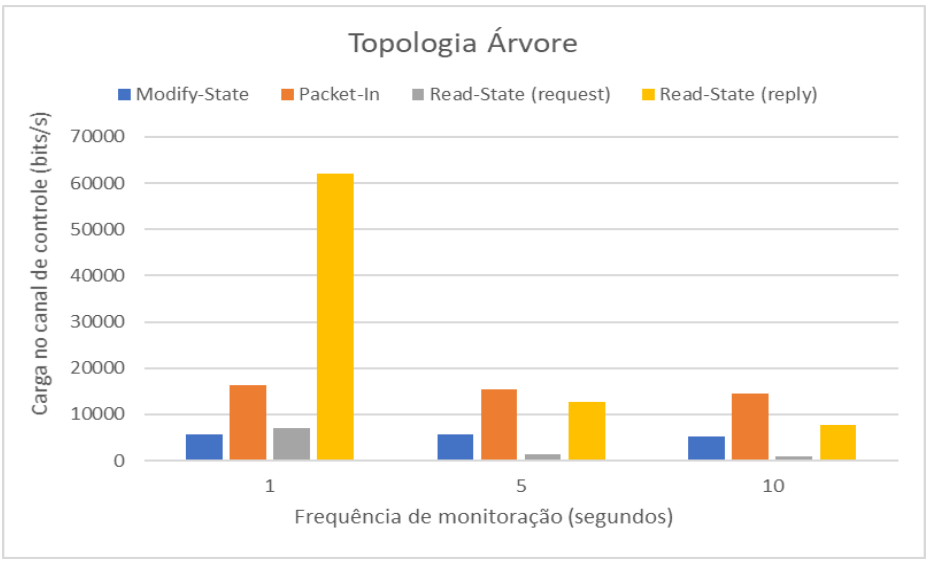


Figura 3.22 – Gráfico da carga no canal de controle em diferentes frequências de monitoramento, para a topologia árvore

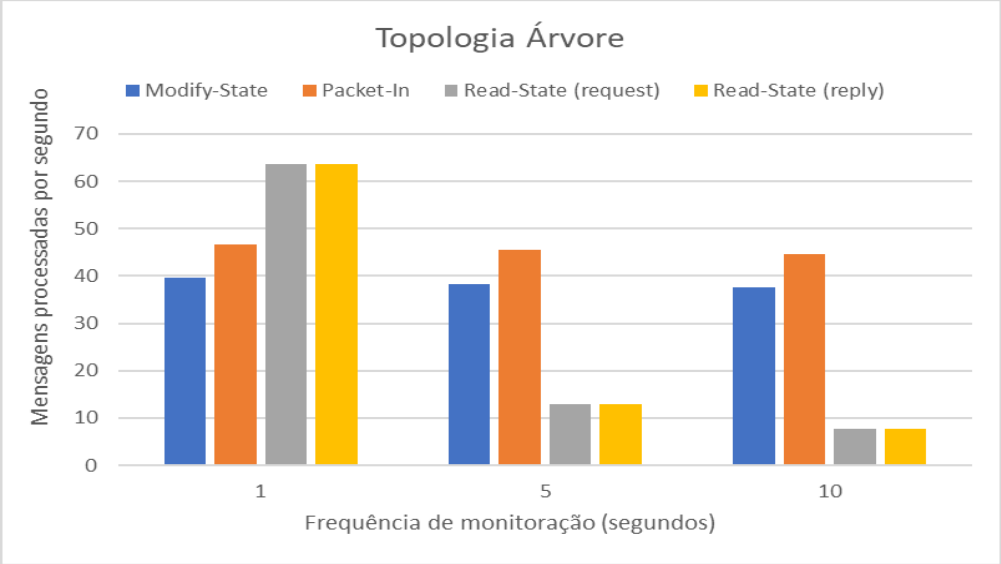


Figura 3.23 – Gráfico da quantidade de mensagens processadas por segundo em diferentes frequências de monitoramento, para a topologia árvore

A frequência de monitoramento é importante para melhor precisão dos dados estatísticos, onde para redes com um *idle timeout* mais baixo uma frequência de monitoramento baixa implica em uma perda de parte dos dados, gerando uma estatística não tão assertiva. É importante equilibrar o valor de monitoramento com o de expiração de regras ociosas, de modo que não aumente expressivamente a quantidade de mensagens e carga no canal de controle, e nem comprometa a confiabilidade dos dados de estatística.

Por se tratar de uma variação linear na quantidade de pacotes e carga das mensagens de monitoramento, é mais fácil avaliar o aumento e impacto que será causado no canal de controle ao alterar a frequência de monitoramento. Desta forma podemos estipular um intervalo de tempo em que a frequência de monitoramento pode ser aumentada sem causar sobrecarga no canal de controle, dependendo do perfil de tráfego da rede e seus dispositivos.

3.4 Análise dos resultados

Com os dados de todos os experimentos já relatados, é possível obter subsídios para definir um uso mais efetivo e consciente do canal de controle, com objetivo de diminuir as chances de sobrecarga no canal. Em resumo, os parâmetros de tempo de ociosidade – *idle timeout* – e de frequência de monitoramento são os mais cruciais para o melhor uso do canal de controle. Também existe a influência da quantidade de *switches* que a rede possui, que em conjunto com os demais parâmetros pode trabalhar contra ou a favor do canal de controle.

Desta forma, foi realizada uma síntese das vantagens e desvantagens desses parâmetros em relação a sua variação, conforme os dados obtidos nos experimentos, dividas

nas subseções que seguem. Além da definição desses parâmetros, também foram mostradas duas implementações simples para a diminuição dos pacotes no canal de controle, que nem sempre estarão definidas em todos os diversos controladores existentes para as redes definidas por *software*. Essas implementações foram explicadas e demonstradas no estudo preliminar, e estão listadas abaixo, para ressaltá-las novamente.

- Tratamento dos pacotes através das mensagens *Modify-State*, ao invés das mensagens *Send-Packet*, explicado em conjunto com a Tabela 3.3 da seção 3.1.5.
- Instalação de regras mais genéricas na tabela de fluxos, através da não especificação das portas TCP de origem e destino, explicado em conjunto com a Tabela 3.5 da seção 3.1.5.

3.4.1 Tempo de ociosidade das regras

Este é o parâmetro que define quanto tempo leva para que uma regra instalada na tabela de fluxos seja removida, caso esteja ociosa durante todo o intervalo. A definição deste parâmetro não afeta na quantidade de mensagens de monitoramento que trafegam no canal de controle, não importa o valor utilizado.

Parâmetro *idle timeout* com valores baixos:

- Maior quantidade de mensagens de controle no canal, devido à remoção mais rápida de regras da tabela de fluxos;
- Maior carga de mensagens de controle no canal, devido à limitação de *buffer* do *switch*, que depende da topologia e tráfego da rede;
    - Caso o perfil de tráfego da rede não contenha rajadas de pacotes, esta carga tende a ser bem menor, devendo ser avaliado caso a caso;
- Menor carga de mensagens de monitoramento, devido à diminuição das estatísticas relativas às regras instaladas na tabela de fluxos;
- Menor número de regras instaladas, ocupando menos espaço da tabela;
- Menor taxa de regras ociosas na tabela de fluxos.

Parâmetro *idle timeout* com valores altos:

- Menor quantidade de mensagens de controle no canal, pois demoram a expirar;
- Menor carga de mensagens de controle no canal, pois as regras ficam ativas por mais tempo, evitando também a sobrecarga do *buffer* do *switch;*

- Maior carga de mensagens de monitoramento, devido ao aumento das estatísticas;
- Maior número de regras instaladas na tabela de fluxos;
- Maior taxa de regras ociosas na tabela de fluxos;

3.4.2 Frequência de monitoramento

Este é o parâmetro que define o intervalo em que devem ser feitas as requisições de estatísticas aos dispositivos da rede, através das mensagens *Read-State.* É recomendável que o valor definido não seja menor que o valor de *idle timeout*, para o caso de um monitoramento mais precisa da instalação e remoção das regras de encaminhamento.

A definição deste parâmetro não afeta a quantidade de mensagens de controle que trafegam no canal, independentemente do valor utilizado.

Alta frequência de monitoramento:

- Maior nível de granularidade de monitoramento, com dados mais precisos;
- Maior quantidade de mensagens de monitoramento;
    - Para um número elevado de *switches*, pode representar uma quantidade muito maior que a quantidade de mensagens de controle;
- Maior carga de mensagens de monitoramento, na mesma proporção do aumento de sua quantidade;
    - Para o mesmo valor de *idle timeout*, pode representar uma carga muito maior que a carga das mensagens de controle;

Baixa frequência de monitoramento:

- Menor nível de granularidade de monitoramento;
- Menor quantidade de mensagens de monitoramento no canal de controle;
    - Para um número elevado de *switches*, pode representar uma quantidade muito maior que a quantidade de mensagens de controle;
- Menor carga de mensagens de monitoramento, na mesma proporção da diminuição de sua quantidade;
    - Para o mesmo valor de *idle timeout*, pode representar uma carga muito maior que a carga das mensagens de controle;

3.4.3 Quantidade de *switches* na rede

A quantidade de *switches* varia de acordo com a topologia e a necessidade de cada rede, nem sempre sendo possível a alteração deste fator. Entretanto, caso este não possa ser alterado, deve ser levado em conta como base para definição dos parâmetros descritos anteriormente.

Quantidade baixa de *switches*:

- Menor quantidade de mensagens de controle no canal;
- Menor quantidade de mensagens de monitoramento, pois são proporcionais ao número de *switches;*
- Menor carga de mensagens de monitoramento, devido à diminuição das estatísticas e da quantidade de mensagens;
- Possibilita uma frequência de monitoramento mais alto, devido à menor carga e quantidade de mensagens de monitoramento;
- Maior sobrecarga da tabela de fluxos, devido à menor distribuição entre os *switches* das conexões com as estações
- Tende a uma maior sobrecarga do *buffer*, dependendo do comportamento do tráfego da rede.

Quantidade elevada de *switches*:

- Maior quantidade de mensagens de controle no canal;
- Maior quantidade de mensagens de monitoramento;
- Maior carga de mensagens de monitoramento, devido ao aumento das estatísticas e da quantidade de mensagens;
- Necessita de maior atenção ao se definir uma frequência de monitoramento, devido à maior carga e quantidade de mensagens de monitoramento que possui;
- Menor sobrecarga da tabela de fluxos, devido à maior distribuição entre os *switches* das conexões com as estações*;*
- Tende a uma menor sobrecarga do *buffer*, dependendo do comportamento do tráfego da rede.

Capítulo 4

Considerações finais

Neste trabalho foi discutido a abordagem centralizada da lógica da rede e a segregação do plano de controle no contexto do paradigma SDN e do protocolo OpenFlow. Neste sentido, foi discutida e avaliada a preocupação acerca da sobrecarga do canal de controle, onde por haver apenas um controlador, as chances de o canal de controle ser sobrecarregado eram grandes em algum momento, sendo consideradas muitas vezes uma questão de tempo até que ocorra. Foram apresentadas algumas soluções para este problema, que diminuem o gargalo gerado pela problemática da centralização do controle, porém nenhuma delas define o custo e peso das mensagens utilizadas pelo protocolo OpenFlow. As especificações do protocolo definem apenas quais e como são as mensagens de controle e monitoramento, sem definir melhores práticas para sua utilização, ou quais parâmetros devem ser levados em maior consideração ao se definir uma rede baseada neste protocolo, sem comprometer seu desempenho. Desta maneira, para se obter essas informações, foi proposto um estudo quantitativo das mensagens que trafegam no canal de controle, com um maior foco nas que possuem uma maior frequência de utilização, pois tendem a gerar mais carga no canal de controle. O objetivo é obter uma maior compreensão do impacto dessas mensagens e fornecer subsídios para mensurar o uso efetivo do canal de controle em cenários diversos.

Os resultados obtidos após a realização de todos os experimentos propostos, baseados nas métricas e cenários definidos, mostram que as mensagens do tipo *Packet-In, Modify-State* e *Read-State* são as mais frequentes, e tem comportamentos variados de acordo com a definição do tempo limite de ociosidade das regras – parâmetro *idle timeout.* As mensagens de monitoramento *Read-State* são mais afetadas pela frequência de monitoramento, quantidade de regras instaladas na tabela de fluxos e pelo número de dispositivos que a rede possui. Já as mensagens de controle são mais influenciadas pelo valor definido para o *idle timeout,* onde a quantidade de mensagens crescer ou diminuir expressivamente de acordo com o valor utilizado. Além disso, este parâmetro também influencia na quantidade de regras que ficam ociosas em relação ao número total de regras instaladas na tabela de fluxos, levando a mais um ponto a ser avaliado, a nível de custo-benefício, quando da definição do valor de *idle timeout.*

Ao fim, foi realizado uma sintetização da influência do parâmetro *idle timeout,* da frequência de monitoramento e da quantidade de *switches* na rede, para melhor comparação e

análise dos dados. Com isto, temos os subsídios necessários para melhor avaliar o uso do canal de controle, fazendo as definições de rede, controle e monitoramento mais indicadas para manter o bom desempenho da rede e evitar possíveis sobrecargas do canal de controle.

4.1 Trabalhos futuros

Para futuros trabalhos pretende-se expandir esta análise para outros parâmetros, como *hard timeout,* e diferentes implementações no controlador, realizando mais experimentos para comparar até mesmo com o trabalho atual. Também é pretendido realizar experimentos com outros controladores, como Ryu, Floodlight, Beacon, Maestro, entre outros, para avaliar o desempenho de cada um deles, com variações na quantidade de estações definidas na rede e diferentes perfis e geradores de tráfego, a exemplo da ferramenta *iperf.* É possível também analisar o funcionamento do uso de mais de um controlador para a rede, de forma a distribuir o controle da lógica da rede e dispor de mais canais de controle.

Os experimentos realizados também podem ser estendidos para outras versões do protocolo OpenFlow, analisando as diferenças de comportamento e das mensagens geradas, podendo implementar novos métodos para redução de carga no canal de controle através dos novos recursos implementados nas demais versões. Existem também metodologias que implementam mais de uma tabela de fluxos por *switch,* o que pode ser objeto de análise para novos trabalhos.

Uma questão a ser estudada é a correlação dos parâmetros levantados neste trabalho, *idle timeout,* frequência de monitoramento e quantidade de *switches* instalados, onde pode ser elaborado uma fórmula matemática que faça essa correlação, para assim definir os melhores valores a serem utilizados por cada parâmetro, em cada cenário. Com isto, é possível buscar a criação de um algoritmo que utilize esta correlação para, de forma dinâmica e automatizada, definir os melhores valores dos parâmetros em questão à medida em que a rede tem seu comportamento modificado, seja por uma diferença na quantidade de tráfego ou mudança em sua estrutura, mantendo seu funcionamento no padrão desejado.

Por fim, é de intenção futura realizar todos esses experimentos e análises em redes reais, utilizando *switches* de diversos fornecedores, para avaliar quantas regras são suportadas em suas respectivas tabelas de fluxos, além de analisar qual o limite de processamento de regras pelo controlador, encontrando assim o ponto de sobrecarga do canal de controle para diversos cenários e equipamentos específicos.

REFERÊNCIAS BIBLIOGRÁFICAS

1. McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S., e Turner, J. (2008). **Openflow: enabling innovation in campus networks.** ACM SIGCOMM Computer Communication Review, 38(2):69–74.

2. Kim, H. e Feamster, N. (2013). **Improving network management with software defined networking**. IEEE Communications Magazine, 51(2):114–119.

3. Kreutz, D.; Ramos, F. M. V.; Veríssimo, P.; Rothenberg, C. E.; Azodolmolky, S.; Uhlig, S. **Software-Defined Networking: a comprehensive survey**. Computing Research Repository, v.abs/1406.0440, 2014.

4. Roy, A. R., Bari, M. F., Zhani, M. F., Ahmed, R., e Boutaba, R. (2014). **Design and management of dot: A distributed openflow testbed**. In Preceedings of the 14th IEEE/IFIP Network Operations and Management Symposium (NOMS 2014).

5. Bruno Astuto A. Nunes, Marc Mendonca, Xuan-Nam Nguyen, Katia Obraczka, e Thierry Turletti, "**A Survey of Software-Defined Networking: Past, Present, and Future of Programmable Networks**", IEEE Communication Surveys & Tutorials, VOL 16, NO. 3, Third Quarter, 2014

6. Bari, M. F., Roy, A. R., Chowdhury, S. R., Zhang, Q., Zhani, M. F., Ahmed, R., e Boutaba, R. (2013). **Dynamic controller provisioning in software defined networks**. In Proceedings of the 9th IEEE/ACM/IFIP International Conference on Network and Service.

7. Andrew R. Curtis, Jeffrey C. Mogul, Jean Tourrilhes, and Praveen Yalagandula, "**DevoFlow: Scaling Flow Management for HighPerformance Networks**," in Proc. SIGCOMM'11, 2011

8. Minlan Yu, Jennifer Rexford, Michael J. Freedman, and Jia Wang, "**Scalable Flow-Based Networking with DIFANE**," in Proc. SIGCOMM'10, 2010.

9. Park, Hyungbae, Song, Sejun, Choi, Baek-Young, Zhu, R. (2015). **Software-Defined Networking (SDN) Control Message Classification, Verification, and Optimization System**. 2015 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW). Gaithersburg, MD, USA.

10. **OpenFlow Switch Specification Version 1.0.0**. Open Networking Foundation (ONF). December 31, 2009.

11. Chowdhury, M.; Boutaba, R. **Network Virtualization: State of the Art and Research Challenges**. IEEE Communications Magazine, v. 47, n. 7, p. 20-26, 2009.

12. C.E.Rothenberg; M.R.Nascimento; M.R.Salvador; e M.F.Magalhães. **OpenFlow e redes definidas por software: um novo paradigma de controle e inovação em redes de pacotes**. Cad. CPqD Tecnologia,7 (1): 1–6, July 2010

13. Open Networking Foundation, Project: Architecture & Framework, "**SDN Architecture 1.0**", 11/2014. Acessado em 11/2017. Disponível em: <https://www.opennetworking.org/software-defined-standards/archives>.

14. CASADO, M. et al. Ethane: Taking Control of the Enterprise. In: SIGCOMM '07. Proceedings... New York, USA: ACM, 2007.

15. YAP, K. K. et al. Blueprint for Introducing Innovation into Wireless Mobile Networks. In: VISA'10, New Delhi, India, 2010. Proceedings... New York: ACM, 2010

16. GUDLA, V. et al. Experimental Demonstration of OpenFlow Control of Packet and Circuit Switches. In: OPTICAL FIBER CONFERENCE (OFC/NFOEC'10). Proceedings... San Diego, USA, 2010.

17. T. Koponen et al., ‘‘**Onix: A distributed control platform for large-scale production networks**,’’ in Proc. 9th USENIX Conf. Oper. Syst. Design Implement., 2010, pp. 1–6.

18. Eun-Do Kim; Seung-Ik Lee; Yunchul Choi; Myung-Ki Shin; Hyoung-Jun Kim (2014). **A Flow Entry Management Scheme for Reducing Controller Overhead**. Broadband Network Technol., UST (Univ. of Sci. & Technol.), Daejeon, South Korea.

19. Ching-Hao, Chang and Dr. Ying-Dar Lin, “**OpenFlow Version Roadmap**”, Study, Dept. of Computer Science, National Chiao Tung University, Taiwan. September 11, 2015.

20. Adrian Lara, Anisha Kolasani, and Byrav Ramamurthy, “**Network Innovation using OpenFlow: A Survey**”, Dept. of Comput. Sci. & Eng., Univ. of Nebraska-Lincoln, Lincoln, NE, USA, 2013. IEEE Communications Surveys & Tutorials.

21. Burak Görkemli, A. Murat Parlakışık, Seyhan Civanlar and Aydın Ulaş, “**Dynamic Management of Control Plane Performance in Software-Defined Networks**”, NetSoft Conference and Workshops (NetSoft), 2016 IEEE. Seoul, South Korea. June 2016.

22. Heller, B., Sherwood, R., e McKeown, N. (2012). “**The controller placement problem”**. In Proceedings of the first workshop on Hot topics in software defined networks, pp.7–12. ACM

23. Muhammad Anan, Ala Al-Fuqaha, Nidal Nasser, Ting-Yu Mu, Husnain Bustam. “**Empowering Networking Research and Experimentation through Software Defined Networking”**. May, 2016.

24. NOX/POX. [Online – Acessado em 11/2017]. Disponível em: http://www.noxrepo.org.

25. **OpenFlow Pox Wiki.** [Online – Acessado em 11/2017]. Disponível em: https://openflow.stanford.edu/display/ONL/POX+Wiki

26. Big Switch Networks, ‘‘**Project floodlight**,’’ 2013. [Online – Acessado em 11/2017]. Disponível em: http://www. projectfloodlight.org

27. Nippon Telegraph and Telephone Corporation, ‘‘**RYU network operating system**,’’ 2012. [Online – Acessado em 11/2017]. Disponível em: http://osrg.github.com/ryu/.

28. **Site oficial sobre o Controlador Beacon** [Online – Acessado em 11/2017]. Disponível em: https://openflow.stanford.edu/display/Beacon/Home

29. Z. Cai, A. L. Cox, and T. S. E. Ng, ‘‘**Maestro: A system for scalable OpenFlow control,**’’ Rice Univ., Houston, TX, USA, Tech. Rep., 2011.

30. Y. Takamiya and N. Karanatsios, ‘‘**Trema OpenFlow controller framework**,’’ 2012. [Online – Acessado em 11/2017]. Disponível em: https://github.com/trema.

31. G. Appenzeller, ‘‘**SNAC**,’’ 2011. [Online – Acessado em 11/2017]. Disponível em: http://www. openflowhub.org/ display/Snac.

32. N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker, “**NOX: Towards an Operating System for Networks**,” ACM SIGCOMM Computer Communication Review, 38(3), pp. 105-110, July 2008.

33. B. Pfaff, J. Pettit, T. Koponen, K. Amidon, M. Casado, S. Shenker, "**Extending Networking into the Virtualization Layer**", HotNets-VIII, Oct. 22-23, 2009.

34. **Open vSwitch: An Open Virtual Switch**. [Online – Acessado em 11/2017]. Disponível em: http://openvswitch.org/

35. Lantz, B., Heller, B., and McKeown, N. (2010). **A network in a laptop: rapid prototyping for software-defined networks**. In Proceedings of the Ninth ACM SIGCOMM Workshop on Hot Topics in Networks, Hotnets '10, pages 19:1–19:6, New York, NY, USA. ACM

36. **Mininet: An Instant Virtual Network on your Laptop (or other PC).** [Online – Acessado em 11/2017]. Disponível em: http://mininet.org/

37. **Site Oficial do Wireshark** [Online – Acessado em 11/2017]. Disponível em: https://www.wireshark.org/

38. K. Katsaros, G. Xylomenos, and G. Polyzos, "**GlobeTraff: a traffic workload generator for the performance evaluation of future Internet architectures,**" in Proceedings of the 5th International Conference on New Technologies, Mobility and Security (NTMS), May 2012, pp. 1–5.

39. G. Maier, A. Feldmann, V. Paxson, and M. Allman, "**On dominant characteristics of residential broadband internet traffic**" in ACM IMC, 2009, pp. 90–102

40. **Site Oficial do VirtualBox** [Online – Acessado em 11/2017]. Disponível em: https://www.virtualbox.org/

41. **Site Oficial do PuTTY** [Online – Acessado em 11/2017]. Disponível em: https://www.putty.org/

42. **Site Oficial do VLC** [Online – Acessado em 11/2017]. Disponível em: https://www.videolan.org/

ANEXOS

I. Código do script stream.py

```
#!/usr/bin/python

import sys
import time
from threading import Thread

from mininet.topo import *
from mininet.topolib import *
from mininet.net import Mininet
from mininet.node import CPULimitedHost
from mininet.link import TCLink
from mininet.util import dumpNodeConnections
from mininet.log import setLogLevel
from mininet.cli import CLI
from mininet.clean import cleanup

from functools import partial
from mininet.node import *
import subprocess
from os.path import isfile, join
import os


################################################################################
# codigo que faz a inicializacao do Mininet com os parametros escolhidos e
# faz o stream de video e requisicoes web entre as estacoes e os servidores
################################################################################

class stream(object):

        def __init__( self ):

                if sys.argv[1] == "1":
                        topo = SingleSwitchTopo( 64 )
                elif sys.argv[1] == "2":
                        topo = LinearTopo(8,8)
                elif sys.argv[1] == "3":
                        topo = TreeTopo( depth=3, fanout=4 )

                net = Mininet( topo=topo, switch=partial(OVSKernelSwitch,protocols=None),
controller=RemoteController,link=partial(TCLink,bw=1000) ,autoSetMacs=True, cleanup=True)

                net.start()

                print "Conexoes da rede"
                dumpNodeConnections(net.hosts)

                duracao = 180 # segundos
                intervalo = 20 # segundos
                h = [len(net.hosts)]
                n_videos = (h[0]/4)-1 # 25% para trafego de video
                n_web = (h[0]*3/4)-1  # 75% para trafego web

                servidor_video = h[0]
                servidor_web = h[0]-1
                for k in net.hosts:
                  h.append(k)

                print 'Inicializando servidor web...'
                h[servidor_web].sendCmd("python -m SimpleHTTPServer 8000")

                print "Iniciando requisicoes..."

                for i in range(duracao/intervalo):
                        s = time.time()
                        for v in range(n_videos):

                                print "Aguarda termino h%i"% (v+1)
                                h[v+1].waitOutput()
                                print "Inicia streaming h%i"%(v+1)
```

```
                                h[v+1].sendCmd('cvlc rtp://@:5004 --run-time %i vlc://quit &'%
(intervalo))

                        print "Inicia streaming do servidor de video..."
                        self.iniciaServidor(h[servidor_video],n_videos,intervalo)

                        print "Requisicao #%i"%i
                        for w in range(n_videos,n_web+n_videos):
                                h[w+1].cmdPrint("wget -P web
10.0.0.%i:8000/of1.pdf"%servidor_web)

                        print "Aguarda termino de streaming do servidor de video..."
                        h[servidor_video].waitOutput()

                        t_concluido = float( time.time() - s )
                        info( 'finalizado em %0.3f segundos\n' % t_concluido )

                for i in range(h[0]):
                        h[i+1].stop()


                net.stop()

                t_total = float( time.time() - inicio )
                info( 'completed in %0.3f seconds\n' % t_total )

                os.system("sudo killall -s SIGINT python2.7")
                os.system("sudo rm web/of1.pdf.*")

        def iniciaServidor(self,h,n_videos,tempo):
                s = time.time()
                dst = ""

                for v in range(n_videos):
                        dst += "dst=rtp{dst=10.0.0.%i,mux=ts}," % (v+1)

                h.sendCmd('cvlc -vvv video.mp4 --sout \
"#transcode{vcodec=h264,acodec=mpga,ab=128,channels=2,samplerate=44100}:\
duplicate{%s}" --run-time %i vlc://quit' % (dst,tempo))

                t_concluido = float( time.time() - s )
                info( 'finalizado em %0.3f segundos\n' % t_concluido )

if __name__ == '__main__':

        setLogLevel( 'info' )
        inicio = time.time()

        try:
                stream()
        except KeyboardInterrupt:
                info( "\n\nKeyboard Interrupt. Shutting down and cleaning up...\n\n")
                elapsed = float( time.time() - inicio )
                info( 'finalizado em %0.3f segundos\n' % elapsed )
                os.system("sudo killall -s SIGINT python2.7")
                cleanup()
```

II. Código do componente ic_monitor

```
#!/usr/bin/python
# Copyright 2017 Igor Cezar
# igorcezar@gmail.com
#

"""
Codigo criado para obter estatisticas de fluxos e portas
dos switches utilizando OpenFlow 1.0
Ao final eh gerado um relatorio com a quantidade de pacotes de cada tipo
que foram transmitidos pelo protocolo OpenFlow
"""

# importacoes
from pox.core import core
from pox.lib.util import dpidToStr
from pox.lib.recoco import Timer
import pox.openflow.libopenflow_01 as of
```

```
from pox.lib.revent.revent import *
from pox.openflow import *
from datetime import datetime
from pox.openflow.of_json import *
import time

log = core.getLogger()


#############################################################################
# funcao para realizar as requisicoes de estatisticas para
# os switches da rede
#############################################################################
def _monitoracao ():
  for connection in core.openflow._connections.values():
        flowstat = of.ofp_flow_stats_request()
        aggstat = of.ofp_aggregate_stats_request()
        portstat = of.ofp_port_stats_request()

        # Envia as mensagens de requisicao de estatisticas e contabiliza
        connection.send(of.ofp_stats_request(body=flowstat))
        ic_monitor.flowstats += 1
        connection.send(of.ofp_stats_request(body=portstat))
        ic_monitor.portstats += 1
        connection.send(of.ofp_stats_request(body=aggstat))
        ic_monitor.aggstats += 1

  log.debug("Enviada(s) %i requisicao(oes) de estatistica(s)",
len(core.openflow._connections))


class ic_monitor(object):

        # Contadores de pacotes
        flowstats = 0.0
        portstats = 0.0
        aggstats = 0.0
        errorin = 0.0
        fluxos = [{},{}]*32
        qtdfluxos = []
        fluxos_inativos = []
        fluxos_total = []
        tcp_size = 66 # tamanho do encapsulamento tcp que contem o pacote openflow
        n_switches = 0
        soma_fluxos = 0
        soma_f_inativos = 0
        soma_f_total = 0
        intervalo = 5 # intervalo de monitoracao
        pkt_contador = [[0 for x in range(64)] for y in range(64)]
        idle_timeout = 0

        start = time.time()

        def __init__(self):

          Timer(self.intervalo, _monitoracao, recurring=True)
          core.openflow.addListeners(self)
          core.addListeners(self)
          log.debug("Iniciando monitoracao...")

        def _handle_FlowStatsReceived (self,event):
          # contabiliza o reply de FlowStats
          self.flowstats += 1
          stats = flow_stats_to_list(event.stats)

          f inativos = 0
          self.soma_fluxos += 1

          # verifica se houve incremento nos fluxos
          if self.flowstats > 1 :
                for f in event.stats:
                        for k in self.fluxos[event.connection.dpid]:
                                if f.match == k.match:
                                        if f.packet_count == k.packet_count:
                                                f_inativos += 1

          # soma a quantidade de fluxos de todos os switches
          self.soma_f_inativos += f_inativos
          self.soma_f_total += len(event.stats)
```

```
  # verifica se todos os switches responderam e contabiliza o total
  if self.soma_fluxos == self.n_switches:
        self.fluxos_total.append(self.soma_f_total)
        self.fluxos_inativos.append(self.soma_f_inativos)
        self.soma_fluxos = 0
        self.soma_f_inativos = 0
        self.soma_f_total = 0

  # armazena a ultima estatistica de fluxos de cada switch
  self.fluxos[event.connection.dpid] = event.stats

def _handle_PortStatsReceived (self,event):
  # contabiliza o reply de PortStats
  self.portstats += 1
  stats = flow_stats_to_list(event.stats)

def _handle_AggregateFlowStatsReceived (self,event):
  # contabiliza o reply de AggStats
  self.aggstats += 1
  self.qtdfluxos.append(event.stats.flow_count)

def _handle_RawStatsReply (self,event):
  self.pkt_contador[event.ofp.header_type][0] += 1
  self.pkt_contador[event.ofp.header_type][1] += len(event.ofp)+self.tcp_size

def _handle_PortStatus (self,event):
  self.pkt_contador[event.ofp.header_type][0] += 1
  self.pkt_contador[event.ofp.header_type][1] += len(event.ofp)+self.tcp_size

def _handle_FlowRemoved (self,event):
 # nao contabiliza os fluxos deletados pelo controlador
 if not event.deleted:
  self.pkt_contador[event.ofp.header_type][0] += 1
  self.pkt_contador[event.ofp.header_type][1] += len(event.ofp)+self.tcp_size

  # exibe o motivo da remocao da regra
  if event.idleTimeout:
    motivo = "Inatividade"
  elif event.hardTimeout:
    motivo = "Expirado"
  elif event.deleted:
    motivo = "requisicao do controlador"
  log.debug("Regra removida por: %s", motivo)

def _handle_PacketIn (self,event):
  self.pkt_contador[event.ofp.header_type][0] += 1
  self.pkt_contador[event.ofp.header_type][1] += len(event.ofp)+self.tcp_size

def _handle_BarrierIn (self,event):
  self.pkt_contador[event.ofp.header_type][0] += 1
  self.pkt_contador[event.ofp.header_type][1] += len(event.ofp)+self.tcp_size

def _handle_ErrorIn (self,event):
  self.pkt_contador[event.ofp.header_type][0] += 1
  self.pkt_contador[event.ofp.header_type][1] += len(event.ofp)+self.tcp_size

def _handle_FeaturesReceived (self,event):
  self.pkt_contador[event.ofp.header_type][0] += 1
  self.pkt_contador[event.ofp.header_type][1] += len(event.ofp)+self.tcp_size

def _handle_EchoReply (self,event):
  self.pkt_contador[event.ofp.header_type][0] += 1
  self.pkt_contador[event.ofp.header_type][1] += len(event.ofp)+self.tcp_size

def _handle_ConnectionUp(self,event):
  self.n_switches += 1

def _handle_GoingDownEvent (self, event):

  tempo = float( time.time() - self.start )
  log.info( 'finalizado em %0.3f segundos' % tempo )

  total_pacotes = 0.0
  tam_total = 0.0
  pacotes = self.pkt_contador

  if self.n switches == 1:
        estat = "estat_estrela.txt"
        fluxos = "fluxos_estrela.txt"
  elif self.n_switches == 8:
```

```
            estat = "estat_linear.txt"
            fluxos = "fluxos_linear.txt"
        else:
            estat = "estat_arvore.txt"
            fluxos = "fluxos_arvore.txt"


        # escreve as estatisticas de fluxos e mensagens em arquivo
        data = datetime.now()
        arq_stats = open(estat,"a")
        arq_fluxos = open(fluxos,"a")

        arq_stats.write("\n{}".format(data))
        arq_stats.write("\nSwitches: {}".format(self.n_switches))
        arq_stats.write("\nIdle_Timeout: {}".format(self.idle_timeout))
        arq_stats.write("\nDuracao: {:.3f}\n".format(tempo))
        arq_fluxos.write("\n{}".format(data))
        arq_fluxos.write("\nSwitches: {}".format(self.n_switches))
        arq_fluxos.write("\nIdle_Timeout: {}".format(self.idle_timeout))
        arq_fluxos.write("\nDuracao: {:.3f}\n".format(tempo))

        log.info(data)

        for i in range(22):
          total_pacotes += pacotes[i][0]
          tam_total += pacotes[i][1]

        for i in range(22):
          e = "Tipo {};{};{:.2f};{};{:.2f}
\n".format(i,pacotes[i][0],(pacotes[i][0]/total_pacotes)*100,pacotes[i][1],(pacotes[i][1]/tam_
total)*100)
          arq_stats.write(e)

        arq_stats.write("Total pacotes: {}\n".format(total_pacotes))
        arq_stats.write("Total bytes: {}\n".format(tam_total))

        # exibe as estatisticas de mensagens no log

        log.info("----------------------")
        log.info("---- Estatisticas ----")
        log.info("----------------------")

        log.info("")
        log.info("[ Controlador-para-switch ]")
        log.info("Features (Request): %i - %.1f%% - %i bytes",
pacotes[5][0],(pacotes[5][0]/total_pacotes)*100,pacotes[5][1])
        log.info("Features (Reply): %i - %.1f%% - %i bytes",
pacotes[6][0],(pacotes[6][0]/total_pacotes)*100,pacotes[6][1])
        log.info("Configuration (Request): %i - %.1f%% - %i bytes",
pacotes[7][0],(pacotes[7][0]/total_pacotes)*100,pacotes[7][1])
        log.info("Configuration (Reply): %i - %.1f%% - %i bytes",
pacotes[8][0],(pacotes[8][0]/total_pacotes)*100,pacotes[8][1])
        log.info("Configuration (Set): %i - %.1f%% - %i bytes",
pacotes[9][0],(pacotes[9][0]/total_pacotes)*100,pacotes[9][1])
        log.info("PacketOut: %i - %.1f%% - %i bytes",
pacotes[13][0],(pacotes[13][0]/total_pacotes)*100,pacotes[13][1])
        log.info("FlowMod: %i - %.1f%% - %i bytes",
pacotes[14][0],(pacotes[14][0]/total_pacotes)*100,pacotes[14][1])
        log.info("PortStats: %i - %.1f%%", self.portstats,(self.portstats/total_pacotes)*100)
        log.info("FlowStats: %i - %.1f%%", self.flowstats,(self.flowstats/total_pacotes)*100)
        log.info("AggregateStats: %i - %.1f%%",
self.aggstats,(self.aggstats/total_pacotes)*100)
        log.info("Read-State (Request): %i - %.1f%% - %i
bytes",pacotes[16][0],(pacotes[16][0]/total_pacotes)*100,pacotes[16][1])
        log.info("Read-State (Reply): %i - %.1f%% - %i
bytes",pacotes[17][0],(pacotes[17][0]/total pacotes)*100,pacotes[17][1])
        log.info("Barrier (Request): %i - %.1f%% - %i bytes",
pacotes[18][0],(pacotes[18][0]/total_pacotes)*100,pacotes[18][1])
        log.info("Barrier (Reply): %i - %.1f%% - %i bytes",
pacotes[19][0],(pacotes[19][0]/total_pacotes)*100,pacotes[19][1])

        log.info("")
        log.info("[ Assincronas ]")
        log.info("PacketIn: %i - %.1f%% - %i bytes",
pacotes[10][0],(pacotes[10][0]/total_pacotes)*100,pacotes[10][1])
        log.info("FlowRemoved: %i - %.1f%% - %i bytes",
pacotes[11][0],(pacotes[11][0]/total pacotes)*100,pacotes[11][1])
        log.info("ErrorIn: %i - %.1f%%", self.errorin,(self.errorin/total_pacotes)*100)
        log.info("PortStatus: %i - %.1f%% - %i bytes",
pacotes[12][0],(pacotes[12][0]/total_pacotes)*100,pacotes[12][1])
```

```
          log.info("")
          log.info("[ Simetricas ]")
          log.info("Hello: %i - %.1f%% - %i bytes",
pacotes[0][0],(pacotes[0][0]/total_pacotes)*100,pacotes[0][1])
          log.info("Echo (Request): %i - %.1f%% - %i bytes",
pacotes[2][0],(pacotes[2][0]/total_pacotes)*100,pacotes[2][1])
          log.info("Echo (Reply): %i - %.1f%% - %i bytes",
pacotes[3][0],(pacotes[3][0]/total_pacotes)*100,pacotes[3][1])
          log.info("")
          log.info("Total de pacotes : %i",total_pacotes)
          log.info("Tamanho total : %i bytes",tam_total)


          # grava em arquivo a quantidade de fluxos instalados e os inativos
          for i,inativos in enumerate(self.fluxos_inativos):
            f = "{};{};{} \n".format((i+1)*self.intervalo,inativos,self.fluxos_total[i])
            arq_fluxos.write(f)

          arq_fluxos.close()
          arq_stats.close()

# funcao principal para chamar a classe ic_monitor
def launch ():

  core.registerNew(ic_monitor)
```