Pós-Graduação em Ciência da Computação

ANTÔNIO JANAEL PINHEIRO

"APLICAÇÃO DE POLÍTICAS DE MIDDLEBOXES COM O USO DE SOFTWARE-DEFINED NETWORKING"

Dissertação de Mestrado

Universidade Federal de Pernambuco
posgraduacao@cin.ufpe.br
www.cin.ufpe.br/~posgraduacao

Recife

2016

Antônio Janael Pinheiro

"APLICAÇÃO DE POLÍTICAS DE MIDDLEBOXES COM O USO DE SOFTWARE-DEFINED NETWORKING"

Dissertação apresentada ao Programa de Pós-graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Mestre em Ciência da Computação.

Orientador: *Divanilson Rodrigo de Sousa Campelo*

Recife

2016

Catalogação na fonte
Bibliotecária Monick Raquel Silvestre da S. Portes, CRB4-1217

P654a Pinheiro, Antônio Janael
Aplicação de políticas de middleboxes com o uso de Software Defined Networking / Antônio Janael Pinheiro. – 2016.
79 f.: il., fig., tab.

Orientador: Divanilson Rodrigo de Sousa Campelo.
Dissertação (Mestrado) – Universidade Federal de Pernambuco. CIn, Ciência da Computação, Recife, 2016.
Inclui referências.

1. Engenharia de software. 2. Aplicações dinâmicas. I. Campelo, Divanilson Rodrigo de Sousa (orientador). II. Título.

005.1 CDD (23. ed.) UFPE- MEI 2016-142

Antônio Janael Pinheiro

Aplicação de políticas de middleboxes com o uso de Software Defined Networking

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Pernambuco, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação

Aprovado em: 24 /02/2016

BANCA EXAMINADORA

_____

Prof. Dr. José Augusto Suruagy Monteiro
Centro de Informática / UFPE

_____

Prof. Dr. . Fábio Luciano Verdi
Departamento de Ciência da Computação / UFSCar

_____

Prof. Dr. Divanilson Rodrigo de Sousa Campelo (Orientador)
Centro de Informática / UFPE

Dedico esta dissertação aos meus familiares, amigos e professores que me deram o apoio necessário para chegar aqui.

Agradecimentos

Agradeço à minha família o apoio e a compreensão de não me terem por perto. Aos meus pais, Ildevan e Jocilêda, e irmãs, Vitória e Roberta, que me apoiaram, aconselharam e incentivaram em todos os momentos da minha vida. Aos meus amigos, as palavras de apoio. Ao professor e orientador Divanilson, a inspiração, compreensão e incentivo constantes. À colega Ethel B. Gondim, a ajuda no desenvolvimento de trabalhos conjuntos durante o Mestrado. Aos demais professores do Centro de Informática da Universidade Federal de Pernambuco, a formação de uma base sólida para o desenvolvimento deste trabalho.

Nada, absolutamente nada resiste ao trabalho.

—EURYCLIDES DE JESUS ZERBINI

Resumo

Middleboxes são dispositivos de rede essenciais a inúmeras organizações, utilizados primordialmente na adição de serviços à rede. *Middleboxes* realizam operações complexas e variadas sobre o tráfego, introduzindo vários desafios ao funcionamento das redes atuais. Estes dispositivos são configurados manualmente pelo operador de rede, o que dificulta a aplicação correta das políticas destes *middleboxes* diante de aplicações de rede dinâmicas. Diversas soluções foram propostas para mitigar problemas gerados pela presença de *middleboxes*, porém tais soluções não tratam das dificuldades que surgem na operação de aplicações dinâmicas. Muitas destas soluções tornam a rede mais complexa, aumentam o seu custo e exigem a substituição completa dos *middleboxes* existentes. Neste trabalho, é apresentada uma arquitetura baseada em *Software-Defined Networking* (SDN) que tem como objetivo garantir a aplicação correta de políticas de *middleboxes* na presença de aplicações dinâmicas. A arquitetura emprega o controle centralizado e a programabilidade dos dispositivos de rede presentes em SDN para tornar os *middleboxes* existentes capazes de aplicar corretamente suas políticas sem introdução de complexidade à rede, sem aumento de seu custo e sem interferência no funcionamento das aplicações. Para avaliar a arquitetura proposta, foi desenvolvido um protótipo no ambiente de emulação *Mininet* com três middleboxes: um *firewall*, um *Intrusion Detection System* (IDS) e um balanceador de carga. As aplicações utilizadas foram *Voice over IP* (VoIP) e *web*, e as métricas de desempenho foram o atraso de pacotes, a perda de pacotes e o *jitter*. Testes de hipóteses baseados no *Wilcoxon Signed-Rank Test* aplicados aos resultados atestam que, apesar de adicionar um acréscimo tolerável no atraso de pacotes, a arquitetura proposta não gera perda de pacotes, tampouco impacta o *jitter*, sendo capaz de configurar corretamente políticas de *middleboxes* em um cenário de aplicações dinâmicas.

Palavras-chave: *Middleboxes*. Aplicação de políticas. *Software-Defined Networking*. Aplicações dinâmicas.

Abstract

Middleboxes are essential network devices to numerous organizations, primarily to add services to the network. Middleboxes perform complex and varied operations on the traffic, introducing several challenges to the functioning of today's networks. These devices are manually configured by the network operator, what hinders the correct application of the policies of these middleboxes dynamic network applications. Several solutions have been proposed to mitigate problems caused by the presence of middleboxes, but these solutions do not address the difficulties that arise in the operation of dynamic applications. Many of these solutions make the network more complex, increase its cost and require complete replacement of existing middleboxes. In this work, an architecture based on Software-Defined Networking (SDN) is presented that aim at ensuring the correct application of middlebox policies in the presence of dynamic applications. The architecture employs the centralized control and programmability of network devices present in SDN to make existing middleboxes able to correctly apply their policies without introducing complexity to the network, without increasing their cost and without interfering in the operation of applications. To evaluate the proposed architecture, a prototype in the Mininet emulation environment was developed with three middleboxes: a firewall, an Intrusion Detection System (IDS) and a load balancer. The applications used were Voice over IP (VoIP) calls and HTTP requests, and the performance metrics were packet delay, packet loss and jitter. Hypothesis testing based on Wilcoxon Signed-Rank Test applied to the results show that, while adding a tolerable increase in packet delay, the proposed architecture neither generates packet loss, nor impacts the jitter, being able to correctly configure middleboxes policies in a scenario of dynamic applications.

Keywords: *Middleboxes*. Policy enforcement. *Software-Defined Networking*. Dynamic applications.

Lista de Figuras

Lista de Tabelas

Lista de Acrônimos

ASA	Adaptive Security Appliance
AMD	Advanced Micro Devices
ARP	Address Resolution Protocol
ALG	Application-Level Gateway
API	Application Programming Interface
BGP	Border Gateway Protocol
CPU	Central Processing Unit
CSV	Comma-Separated Values
CLI	Command-Line Interface
DDR	Double Data Rate
XML	eXtensible Markup Language
HP	Hewlett-Packard
ICE	Interactive Connectivity Establishment
ITU-T	International Telecommunication Union - Telecommunication Standardization Sector
IP	Internet Protocol
IoT	Internet of Things
IDS	Intrusion Detection System
IPS	Intrusion Prevention System
JSON	JavaScript Object Notation
LTS	Long Term Support
MIDCOM	Middlebox Communications
MPLS	Multiprotocol Label Switching
NAT	Network Address Translation
NIDS	Network Intrusion Detection Systems
NoSQL	Not Only Structured Query Language
ODL	Open Day Light
ONOS	Open Network Operating System
ONF	Open Networking Fundation
P2P	Peer-to-peer
QoS	Quality of Service
RAM	Random Access Memory
RTP	Real-time Transport Protocol

REST	Representational State Transfer
RFC	Request for Comments
SMC	Security Management Center
SIP	Session Initiation Protocol
SIMCO	Simple Middlebox Configuration
STUN	Simple Traversal of UDP through NAT
SIPp	SIP performance
SDN	Software-Defined Networking
SDK	Software Development Kit
TCAM	Ternary Content Addressable Memory
TMG	Threat Management Gateway
URL	Uniform Resource Locator
VXOA	Virtual Acceleration Open Architecture
VM	Virtual Machine
VoIP	Voice over Internet Protocol
WSGI	Web Server Gateway Interface
WAN	Wide Area Network
WRST	Wilcoxon Rank-Sum Test
WSRT	Wilcoxon Signed-Rank Test

Sumário

1 Introdução

Em redes de computadores, *Middleboxes* são elementos intermediários no caminho entre a origem e o destino dos dados que realizam funções além do roteamento *Internet Protocol* (IP) (CARPENTER; BRIM, 2002). *Middleboxes* são essenciais à operação de inúmeras redes, pois são usados na adição de serviços (SEKAR et al., 2012). Em ambientes corporativos, chegam a ultrapassar o número de elementos de encaminhamento (SHERRY et al., 2012). *Middleboxes* realizam operações complexas e variadas sobre o tráfego da rede, como inspeção e modificação, e desempenham funções críticas de segurança (por exemplo, *firewall*, *Network Intrusion Detection Systems* (NIDS)), desempenho (por exemplo, Balanceador de carga, *proxy*, *Wide Area Network* (WAN) *optimizer*), gerenciamento de recursos (por exemplo, *Network Address Translation*, NAT), dentre outras. Esses dispositivos têm seus comportamentos definidos através de políticas, regras que definem quais ações devem ser aplicadas sobre o tráfego recebido. Essas políticas são configuradas pelo operador da rede com base nas necessidades da instituição em que os *middleboxes* são implantados.

Middleboxes podem ser implantados em dispositivos com *hardware* e *software* especializados no desempenho de uma função específica, ou em máquinas virtuais que os executam como aplicações. Geralmente, quando um novo serviço é requerido, é necessário adquirir e adicionar um novo *middlebox* à rede, o que eleva o dispêndio de recursos financeiros, de espaço e de energia.

Middleboxes são configurados manualmente pelo operador da rede, que deve antever eventos que possam ocorrer na rede e configurar políticas para lidar com esses eventos. Portanto, suas políticas são configuradas proativamente e, assim, incapazes de lidar com eventos e tráfego não previstos pelo operador. Aplicações dinâmicas estabelecem conexões entre a origem e o destino dos dados utilizando informações (como o número de porta TCP/UDP) impossíveis de prever (STIEMERLING; QUITTEK; CADAR, 2006). Esse tipo de aplicação utiliza valores aleatoriamente selecionados para identificar informações transmitidas da origem até o destino dos dados. Por exemplo, o protocolo *Real-time Transport Protocol* (RTP), usado pela aplicação VoIP para transmissão de áudio e vídeo, seleciona aleatoriamente sua porta de destino a cada nova conexão criada. Aplicações dinâmicas são extremamente prejudicadas pela presença de *middleboxes* devido à configuração estática desses dispositivos. É impossível prever o comportamento pleno dessas aplicações, o que faz com que, *middleboxes* sejam incapazes de aplicar corretamente suas políticas, impedindo, naturalmente, o funcionamento dessas aplicações. Por exemplo, a aplicação *Voice over Internet Protocol* (VoIP) enfrenta problemas ao atravessar *firewalls* (KHLIFI; GREGOIRE; PHILLIPS, 2006).

Soluções como *Application-Level Gateway* (ALG), *Interactive Connectivity Establishment* (ICE), e *Simple Traversal of UDP through NAT* (STUN) são usadas para mitigar dificul-

dades geradas por *middleboxes* em comunicação VoIP, contudo geram custos adicionais para a sua implantação e manutenção, além de tornarem o gerenciamento de rede mais complexo (KHLIFI; GREGOIRE; PHILLIPS, 2006). Além disso, essas soluções exigem que aplicações e *middleboxes* tenham suporte a elas. Os protocolos *Middlebox Communications* (MIDCOM) (STIEMERLING; QUITTEK; TAYLOR, 2008) e *Simple Middlebox Configuration* (SIMCO) (STIEMERLING; QUITTEK; CADAR, 2006) são limitados ao uso com *firewalls* e NATs e *hosts* interferem diretamente na configuração das políticas desses dispositivos. Por questões de segurança, *hosts* não devem configurar *middleboxes* (SHERRY et al., 2015). Os desafios introduzidos na rede devido a grande quantidade de protocolos motivaram o desenvolvimento de arquiteturas como *Software-Defined Networking* (SDN) que tem como objetivo simplificar o gerenciamento da rede e reduzir a complexidade instaurada por esses protocolos (SHENKER et al., 2011).

SDN é uma arquitetura de rede em que os planos de dados e controle são desacoplados e implementados em dispositivos distintos. O plano de dados é mantido em *switches* como forma de prover altas taxas de transmissão de dados, e o plano de controle é implementado em um *software* chamado de controlador. O controlador configura a infraestrutura da rede através de *Application Programming Interfaces* (APIs) *southbound*, sendo o *OpenFlow* a mais popular. Os dispositivos da infraestrutura de uma rede SDN são configurados através de políticas, que especificam um conjunto de informações do tráfego e uma ou mais ações a serem realizadas sobre esse tráfego. Essas políticas são configuradas nos *switches* como entradas em suas tabelas de fluxos.

Middleboxes são configurados através de políticas e atuam sobre o tráfego da rede. O controlador SDN utiliza informações do tráfego para configurar *switches*, e tais informações podem ser usadas também na configuração de *middleboxes*. Enquanto a infraestrutura em uma rede SDN é configurada pelo controlador através de APIs bem definidas, como o *OpenFlow*, não há atualmente uma interface para comunicação entre *middleboxes* e controlador para configuração de políticas. Há trabalhos na literatura (GEMBER-JACOBSON et al., 2014; GEMBER-JACOBSON; AKELLA, 2015; BREMLER-BARR; HARCHOL; HAY, 2015; FAYAZBAKHSH et al., 2014) que propõem APIs para interação com *middleboxes*, mas não abordam configuração de políticas. Há soluções comerciais que oferecem interfaces para configuração de políticas, mas estas APIs são limitadas pelos respectivos fabricantes. Nesta dissertação, é apresentada uma API com a qual o controlador poderá comunicar-se com *middleboxes*, e ambos poderão evoluir independentemente. *Middleboxes* estão na melhor posição para prover informações sobre suas políticas (FAYAZBAKHSH et al., 2014), pois são os responsáveis por mantê-las e aplicá-las. Dados sobre políticas aplicadas podem ser obtidos dos *middleboxes* para auxiliar na configuração de seus pares pelo controlador.

Neste trabalho, é proposto o uso da arquitetura SDN para aplicar políticas de *middleboxes* de forma dinâmica, com objetivo de mitigar problemas causados na execução de aplicações dinâmicas. Através da integração com SDN, *middleboxes* são capazes de lidar com eventos ocorridos na rede e aplicar suas políticas em resposta a requisitos de aplicações. Com o controlador realizando a configuração de políticas, *middleboxes* passam a estar cientes sobre eventos que ocorram na rede. Apresentamos uma interface *southbound* para prover comunicação entre controlador e *middleboxes*; essa interface é usada pelo controlador para configurar e receber políticas dos *middleboxes*. As aplicações em execução no topo do controlador realizam a configuração dos *middleboxes* com base nas informações de tráfego obtidas pelo controlador, em políticas recebidas de outros *middleboxes* e demais informações que o desenvolvedor julgue

úteis. O operador da rede poderá desenvolver aplicações para o gerenciamento de politicas dos *middleboxes*, estendendo as funções providas por esses dispositivos.

A escalabilidade é uma característica essencial em qualquer sistema. Portanto, um requisito da arquitetura proposta nesta dissertação é que ela seja escalável, mantendo um bom desempenho na presença de cargas de trabalho crescentes e, capaz de crescer de forma simples. Como a quantidade de tráfego gerado pela comunicação entre *middlebox* e controlador deverá ser pequena, espera-se que a arquitetura proposta nesta dissertação seja capaz de lidar com grandes quantidades destes dispositivos. Além disso, esta arquitetura pode ser facilmente ampliada com o uso de múltiplos controladores dividindo a carga de trabalho entre si, ampliando o número de *middleboxes* gerenciados sem prejuízo ao desempenho da rede. Outro requisito é gerar um impacto mínimo ao desempenho da rede, visto que não será útil caso gere um grande impacto ao desempenho da rede.

Os *middleboxes* deverão receber modificações simples para darem suporte à comunicação com o controlador. Em *middleboxes* proprietários, apenas o desenvolvedor tem permissão para modificá-los, por isso propomos o uso de agentes de *software* que implementem APIs do *middlebox* proprietário para configuração das politicas e a API proposta para comunicação com o controlador, atuando como um *proxy* entre controlador e *middlebox*. Esses agentes também podem ser usados com *middleboxes open source*, para simplificar o uso desses dispositivos, já que são *softwares* compostos por milhares de linhas de código, e modificá-los é uma tarefa muito onerosa. Com esses agentes, é possível prover comunicação com o controlador sem modificar o *middlebox*. Os agentes são componentes opcionais na arquitetura proposta e seu objetivo é simplificar a adoção da nossa interface por parte dos *middleboxes*.

A arquitetura proposta não exige qualquer dispositivo extra, portanto não gera custos para implantação de serviços. Também reduz a complexidade na configuração dos *middleboxes* ao permitir a interação com estes dispositivos de um ponto logicamente centralizado. Como o controlador é gerenciado pelo operador de rede, evitam-se problemas com *hosts* que interfiram na configuração dos *middleboxes*. Além disso, a arquitetura é agnóstica a *middlebox*, permitindo a integração de qualquer tipo desse dispositivo.

Uma característica importante da arquitetura proposta nesta dissertação é sua capacidade de obter políticas dos *middleboxes*, dado que essas informações podem ser usadas para aplicar políticas em seus pares. A arquitetura também torna *middleboxes* capazes de aplicar políticas reativamente em resposta a requisitos de aplicações dinâmicas. Esses dispositivos poderão serem adicionados à rede sem interromperem a execução das aplicações, reduzindo custos com artifícios usados para mitigar problemas causados por *middleboxes*. Com a arquitetura proposta, é muito mais simples implantar novas aplicações em uma rede. Desenvolvedores poderão criar aplicações para configuração desses dispositivos, estendendo suas funções e tornando-os capazes de lidar com situações para as quais até então não havia suporte e que atualmente demandam o uso de técnicas e ferramentas que aumentam o custo e a complexidade da rede.

Os resultados obtidos nos experimentos empregados na avaliação do protótipo desenvolvido com base na arquitetura proposta não seguem uma distribuição normal, por isso foram utilizados os testes não paramétricos *Wilcoxon Signed-Rank Test* (WSRT) (WILCOXON; KATTI; WILCOX, 1970) para amostrados pareadas (quando os dados são coletados de cenários com e sem o tratamento avaliado) e *Wilcoxon Rank-Sum Test* (WRST) (WILCOXON; KATTI; WILCOX, 1970) para amostras independentes.

O experimento controlado utilizado neste trabalho tem o objetivo de responder à seguinte questão de pesquisa: como SDN contribui no processo de aplicação de políticas de *middleboxes*

de forma dinâmica?

1.1 Objetivos

1.1.1 Objetivo geral

O presente trabalho tem como principal objetivo utilizar características da arquitetura SDN, tais como controle centralizado e programabilidade dos dispositivos de rede, para aplicar políticas de *middleboxes* de forma dinâmica.

1.1.2 Objetivos específicos

Os objetivos específicos desse trabalho de mestrado são:

- Desenvolver uma interface para comunicação entre *middleboxes* e controlador;
- Obter informações sobre políticas aplicadas pelos *middleboxes*;
- Minimizar o impacto da arquitetura proposta na performance da rede;
- Estudar o processo de aplicação das políticas em *middleboxes*.

1.2 Principais contribuições

A seguir, são apresentadas as principais contribuições desta dissertação ao estado da arte:

- Neste trabalho de mestrado, foi desenvolvida uma interface para prover comunicação entre o controlador e *middleboxes*, independente de fabricante, modelo ou classe de *middlebox*, para obter e configurar as políticas desses dispositivos. Essa interface trabalha paralelamente ao *OpenFlow*, sendo, portanto, independente deste protocolo.
- A arquitetura proposta nesta dissertação não tem como objetivo substituir completamente a maneira como *middleboxes* são configurados para aplicar suas políticas. Contudo, pode ser utilizada para substituir a forma como políticas são configuradas e aplicadas, estendendo funções ofertadas por *middleboxes*.
- Com o uso de múltiplos controladores, a arquitetura proposta pode ser facilmente ampliada para acomodar novos *middleboxes* de modo que cada controlador configure uma fração destes dispositivos.
- O protocolo *OpenFlow*, utilizado pelo controlador na configuração da infraestrutura, não sofre alteração alguma.
- Aplicações populares como VoIP, que enfrentam vários desafios introduzidos pelo uso de *middleboxes*, poderão ser adicionadas à rede sem interferências causadas por esses dispositivos.
- A arquitetura proposta simplifica o gerenciamento da rede ao eliminar a dependência de soluções complexas e dispendiosas.

Não foram encontrados esforços no uso da arquitetura SDN para aplicar políticas de forma dinâmica com objetivo de permitir o funcionamento de aplicações dinâmicas e reagir aos eventos ocorridos na rede. Considerando a lacuna na literatura com relação ao processo de aplicação das políticas de *middleboxes* de forma dinâmica com uso de SDN, conclui-se que há uma grande oportunidade para o desenvolvimento do tema proposto neste trabalho.

1.3 Estrutura do trabalho

Esta dissertação de mestrado está dividida da seguinte forma:

- **Seção 2**: apresenta os conceitos teóricos necessários à compreensão deste trabalho, principais ferramentas utilizadas em sua implementação e avaliação, e procedimentos para análise dos resultados obtidos nos experimentos.

- **Seção 3**: apresenta a revisão da literatura realizada para a composição do presente trabalho e as contribuições para o estado da arte.

- **Seção 4**: descreve a solução proposta e a metodologia utilizada em seu desenvolvimento.

- **Seção 5**: descreve o protótipo para aplicar políticas de *middleboxes* com o uso da arquitetura SDN, implementado com base na solução proposta neste trabalho.

- **Seção 6**: apresenta os passos seguidos para avaliação do protótipo, a metodologia utilizada, e os resultados obtidos nos experimentos.

- **Seção 7**: conclui o trabalho, descrevendo suas contribuições e trabalhos futuros.

2
Fundamentação teórica

A presente seção apresenta alguns dos conceitos teóricos tratados nesta dissertação, bem como ferramentas utilizadas para o desenvolvimento da arquitetura proposta e sua avaliação. Além disso, são apresentados alguns conceitos e ferramentas estatísticas empregadas na análise dos resultados obtidos neste trabalho de mestrado.

2.1 Redes definidas por software

Abstrações são a chave para simplificar a área de redes de computadores (SHENKER et al., 2011). A única grande abstração presente na área de redes é a de camadas, que lida apenas com o plano de dados, não existindo abstrações poderosas no plano de controle. Essa abstração provê independência entre camadas, possibilitando que cada camada evolua de forma independente. Ao contrário de outros campos da Ciência da Computação, como banco de dados e sistemas operacionais, a área de redes não possui princípios básicos; seu gerenciamento é complexo e evolui muito lentamente (SHENKER et al., 2011). Essa evolução lenta se deve em parte à complexidade na implementação de novas tecnologias, pois o processo de padronização pode levar anos e depende do interesse dos fabricantes de *hardware* de adotá-las. Essa rigidez é conhecida na literatura como ossificação da rede (MCKEOWN et al., 2008).

Redes definidas por *software* (do inglês, *Software-Defined Networking, SDN*) são uma arquitetura em que controle e o encaminhamento dos dados são desacoplados e implementados em dispositivos fisicamente separados. O controle é implementado em um *software* logicamente centralizado, chamado de controlador, e o encaminhamento dos dados em *switches*, para prover altas taxas de transmissão de dados. A Figura 2.1 apresenta os componentes da arquitetura SDN. A separação entre os planos de controle e de dados permite que ambos possam evoluir independentemente. SDN possui suporte às APIs que permitem o gerenciamento da infraestrutura da rede, e permite inovação na rede através de programabilidade dos equipamentos.

SDN define três abstrações:

- **Abstração de distribuição**: gera uma visão global da rede, capaz de criar um grafo da topologia a partir de informações obtidas através de APIs. Algoritmos distribuídos não são mais necessários, apenas algoritmos de grafos.

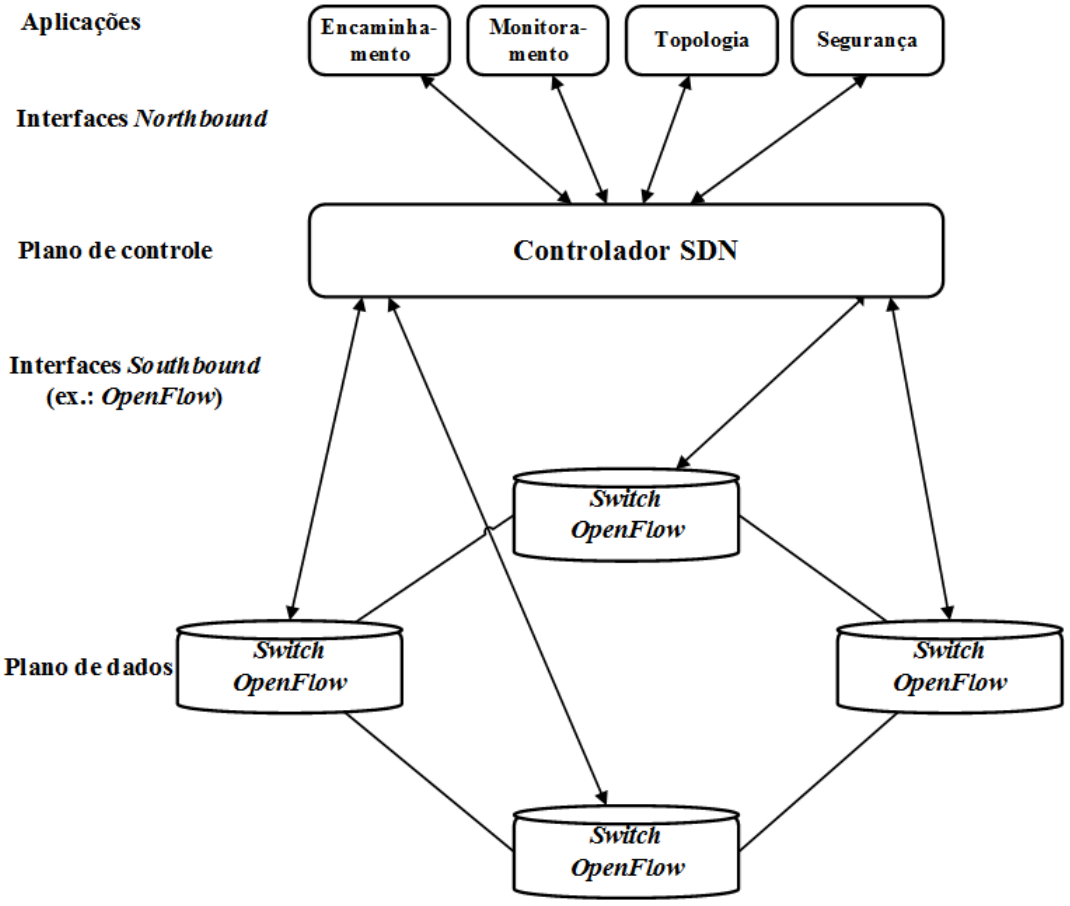
- **Abstração de configuração**: as aplicações devem expressar o comportamento desejado, mas não devem ser responsáveis por implementar esse comportamento diretamente na infraestrutura física da rede. Essa abstração mapeia a rede física para um modelo abstrato, configurado pelas aplicações, e posteriormente mapeia essa configuração para a rede física.

- **Abstração de encaminhamento**: o plano de controle precisa de um modelo de encaminhamento flexível, que dê suporte a qualquer comportamento de encaminhamento necessário e seja independente de fabricante. Essa abstração é implementada através de APIs *southbound*.

Os elementos da arquitetura SDN comunicam-se através de interfaces *southbound* para instruções de baixo nível, como comunicação entre controlador e *switch*, e *northbound* para comunicação de alto nível, como interação entre aplicações e controlador. A Figura 2.1 apresenta essas interfaces.

Como um protocolo de baixo nível, utilizado para manipular a memória dos *switches*, o *OpenFlow* não apresenta características favoráveis ao desenvolvimento de aplicações de alto nível. O controlador deve abstrair a complexidade apresentada pelo *OpenFlow*, apresentando às aplicações um modelo mais simples da rede, exportando APIs *northbound* de alto nível a serem utilizadas pelas aplicações.

Figura 2.1: Visão geral dos principais componentes da arquitetura SDN.



Fonte: (AKYILDIZ et al., 2014)

2.1.1 OpenFlow

OpenFlow é um protocolo *open source* padronizado pela *Open Networking Fundation* (ONF) utilizado no gerenciamento da infraestrutura de uma rede SDN. É uma implementação da abstração de encaminhamento definida no ecossistema SDN. O *OpenFlow* define alguns tipos de mensagens, em que as mais comuns são:

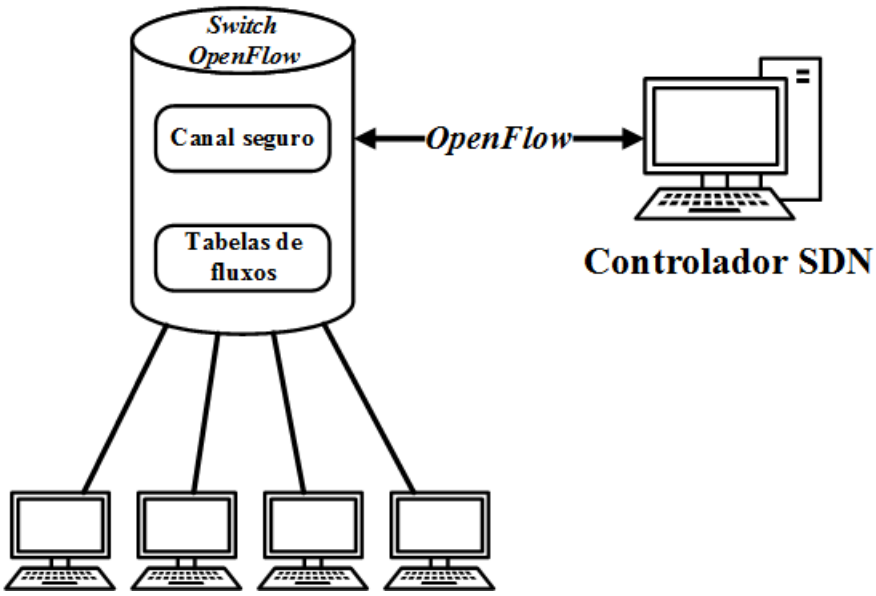
- *packet-in*: utilizado pelos *switches* para solicitar ao controlador a configuração de uma nova entrada em sua tabela de fluxos;
- *flow-mod*: usado no gerenciamento da tabela de fluxos dos *switches*, para adição, remoção ou modificação de entradas da tabela.
- *packet-out*: utilizado pelo controlador para enviar dados diretamente para a rede, sem a necessidade de adicionar entradas na tabela de fluxos.

O *Openflow* habilita *switches Ethernet* a serem remotamente configurados sem expor detalhes de sua implementação, o que o torna interessante para os fabricantes de *hardware*. Dessa forma, fabricantes podem permitir que *softwares* de terceiros gerenciem seus dispositivos.

O *OpenFlow* em sua versão 1.0 (LARA; KOLASANI; RAMAMURTHY, 2014), define um conjunto de treze campos utilizados na caracterização dos fluxos. Cada entrada na tabela de fluxos contém um conjunto desses campos, uma ação associada ao fluxo e contadores usados para obtenção de estatísticas. Ao receber um pacote, o *switch* consulta sua tabela em busca de alguma entrada que seja consistente com o pacote recebido; caso encontre, realiza a ação associada ao fluxo e atualiza os contadores. Quando não há entrada que seja consistente com o pacote recebido, o *switch* envia uma mensagem *packet-in* com os cabeçalhos do pacote ao controlador, solicitando instruções para tratar esse fluxo. Nessa versão os *switches* possuem suporte a apenas uma tabela. A partir do *OpenFlow* 1.1 (LARA; KOLASANI; RAMAMURTHY, 2014) foi adicionado suporte ao protocolo *Multiprotocol Label Switching* (MPLS) e a múltiplas tabelas. O Suporte ao protocolo *Internet Protocol version 6* (IPv6) foi adicionado na versão 1.2, com melhorias na versão 1.3 (LARA; KOLASANI; RAMAMURTHY, 2014).

Switches OpenFlow utilizam a memória *Ternary Content Addressable Memory* (TCAM) para armazenar a(s) tabela(s) de fluxos (MCKEOWN et al., 2008). Há dois tipos de *switches OpenFlow*: o *OpenFlow* puro, no qual a configuração é feita exclusivamente pelo controlador SDN, e o *OpenFlow-enabled*, em que o controlador atua na configuração do *switch*. Outros protocolos e ferramentas auxiliam na configuração do dispositivo, como o *Border Gateway Protocol* (BGP) (NAOUS et al., 2008). A comunicação com o controlador é feita através de um canal criptografado, que conecta o *switch* ao controlador, liberando-os a trocarem comandos e pacotes de forma segura. Uma visão geral do *switch OpenFlow* é apresentada na Figura 2.2.

Figura 2.2: Arquitetura do *Switch OpenFlow*



Fonte: (MCKEOWN et al., 2008)

2.1.2 Controlador

A lógica de controle em uma rede SDN é implementada em um *software* chamado controlador, também conhecido como sistema operacional de rede (GUDE et al., 2008). O controlador é logicamente centralizado, e possui uma visão global sobre a rede que gerencia, permitindo-o estar ciente sobre eventos que ocorrem na rede. A interação com a infraestrutura, como *switches* e roteadores, se dá através de interfaces *southbound*, sendo o *OpenFlow* a mais popular. Com as aplicações, a comunicação é feita através de interfaces *northbound*. Até o momento, não há uma interface *northbound* padronizada pela ONF, mas há esforços no desenvolvimento desse tipo de interface (ROBUCK, 2015). Atualmente cada controlador oferta suas próprias interfaces *northbound*. A Tabela 2.1 apresenta alguns dos principais controladores comerciais e *open source*.

A configuração dos fluxos nos *switches* pode ser realizada de duas maneiras: proativa, em que as regras são pré-definidas, e configuradas na inicialização do controlador; e reativas, em que as regras são configuradas em resposta a eventos ocorridos na rede, como a chegada de um novo fluxo.

2.1.3 Mininet

O *Mininet* (LANTZ; HELLER; MCKEOWN, 2010) é um *software* de emulação para prototipação e experimentação de redes SDN. O *Mininet* é empregado na criação de redes compostas por *hosts* virtuais, *switches*, roteadores, controladores e enlaces (KETI; ASKAR, 2015). Na criação dos *hosts*, o *Mininet* utiliza *Linux containers*, um recurso de virtualização leve em nível de sistema. O mecanismo de *Linux containers* possibilita a criação de múltiplos *hosts* em uma máquina mesmo com uma configuração modesta de *hardware* (HANDIGOL et al., 2012). Todos os *hosts* compartilham o mesmo sistema de arquivos, mas mantêm pilhas de processos, interfaces de rede e tabelas de roteamento e *Address Resolution Protocol* (ARP) individuais. *Hosts* são conectados por pares de interfaces *Ethernet* virtuais, providos por

Tabela 2.1: Relação de controladores comerciais e *open sources*.

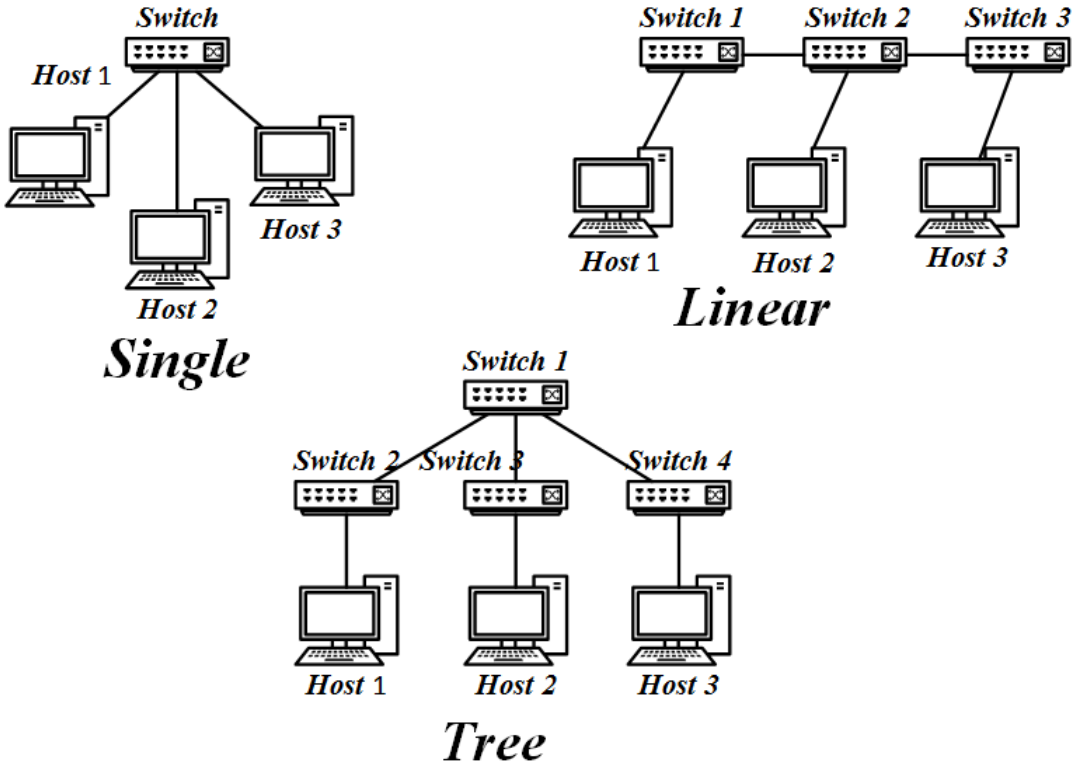
Comerciais	*Open sources*
Brocade SDN Controller	*Ryu SDN Framework*
Cisco Application Policy Infrastructure Controller	*OpenDaylight*
Ericsson SDN Controller	*OpenContrail*
Contrail	NOX
Sonus NaaS IQ	POX
Avaya SDN Fx Controller	*Open Network Operating System* (ONOS)
Big Cloud Fabric	*Trema*
ContexNet 4.0	*BEEM Controller*
Cyan Planet Operate	*Floodlight*
HP Virtual Application Networks SDN Controller	*Loom*

Linux network namespaces. *Switches* são criados com o *OpenVswitch* (CHIU; WANG, 2015) para experimentação com *OpenFlow*, e conectados ao(s) controlador(es) onde são executados *softwares* como *Ryu* (RYU SDN FRAMEWORK, 2015), *OpenDayLight* (ODL) (MEDVED et al., 2014) e *POX* (SHALIMOV et al., 2013). O *Mininet* executa código real, incluindo aplicações, *kernel* e *softwares* de rede *Linux*. Aplicações desenvolvidas e testadas no *Mininet* podem ser movidas para sistemas reais com mudanças mínimas ou sem modificações. O *Mininet* provê as seguintes características (MININET OVERVIEW, 2015):

- Possibilidade de criar um *testbed* de baixo custo para experimentação em redes SDN;
- Permite que múltiplos desenvolvedores concorrentes trabalhem de forma independente na mesma topologia;
- Dá suporte a testes de regressão em nível de sistema;
- Permite experimentos em topologias complexas;
- Inclui uma *Command-Line Interface* (CLI) para testes de depuração ou execução em toda a rede;
- Dá suporte a topologias personalizadas arbitrárias, e inclui um conjunto básico de topologias predefinidas;
- Fornece uma API *Python* simples e extensível para criação e experimentação de rede;
- Conectividade com a rede local e a *Internet*.

As topologias predefinidas que o *Mininet* dá suporte, apresentadas na Figura 2.3, são três: *single*, *linear* e *tree*. Na topologia *single*, há apenas um *switch* em que todos os *hosts* da rede são conectados; na *linear*, múltiplos *switches* são interconectados e cada um é conectado a um único *host*; por fim, na *tree*, os *switches* são agrupados em níveis (em geral, dois ou três níveis) e os *hosts* são conectados apenas aos *switches* do nível mais inferior. A topologia *tree* é muito comum em redes de *data center*.

Figura 2.3: Topologias predefinidas no *Mininet*



Fonte: elaborada pelo autor.

2.2 Middleboxes

Middlebox é uma classe de dispositivos de rede definidos como qualquer dispositivo no caminho entre origem e destino dos dados que realize funções além do roteamento IP (CARPENTER; BRIM, 2002). Podem ser implantados como dispositivos físicos distintos, em que cada função é executada em um *hardware* especializado, ou em máquinas virtuais, nas quais são executados como múltiplas aplicações independentes sobre o mesmo *hardware* virtual. O termo *middlebox* é usado para designar uma grande variedade de dispositivos, tais como dispositivos que encerram uma conexão e iniciam outra, que modificam informações dos pacotes, que multiplexam diversos pacotes sobre a mesma conexão, que alteram o destino dos pacotes, ou dispositivos que combinam várias funções. Contudo, um *middlebox* nunca é o destino de uma conexão. A Tabela 2.2 apresenta alguns dos *middleboxes* mais populares especificados na *Request for Comments* (RFC) 3234 (CARPENTER; BRIM, 2002), com exceção de NIDS e WAN *optimizer*, desenvolvidos posteriormente à escrita da referida RFC.

Middleboxes são capazes de modificar o tráfego que recebem. Por exemplo, NAT mapeia endereços IP locais em endereços válidos na *Internet*, e o balanceador de carga modifica o endereço de destino dos pacotes. Geralmente, o tráfego atravessa múltiplos *middleboxes* entre a origem e e destino, posicionados fisicamente na rede de forma que o tráfego atravesse-os em uma sequência, como por exemplo, *firewall* $\rightarrow$ *proxy* $\rightarrow$ NAT. Essas sequências são comumente chamadas de cadeia de serviços, pois são criadas para prover serviços. Quando o tráfego é modificado, *middleboxes* subsequentes não sabem sobre essas modificações e como suas políticas foram configuradas para lidar com o tráfego original – anterior à modificação – são

incapazes de aplicar políticas corretas para o tráfego recebido (FAYAZBAKHSH et al., 2014).

Tabela 2.2: Amostra de *middleboxes* e breve descrição de suas funções

Middlebox	Descrição
NAT	Ao recebe tráfego dos *hosts* de uma rede privada, sobrescreve os endereços privados de origem desses *hosts* com seu próprio endereço global único e, encaminha esse tráfego para o destino. Ao receber as respostas para suas requisições, substitui o endereço IP de destino colocando o endereço privado do *host* que originou a requisição.
NAT-PT (Protocol Translator)	Realiza a tradução de IPv4 para IPv6, e vice-versa.
IP *Tunnel Endpoints*	Tuneis usados na criação de redes privadas virtuais.
Packet classifiers	Classificam pacotes de acordo com suas políticas, podendo marcá-los para tratamento por serviços diferenciados.
TCP *performance enhancing proxies*	Modificam a temporização ou ação do protocolo TCP para melhoria de performance.
Load balancer	Realiza o balanceamento da carga entre diferentes servidores. São capazes de alterar IP e portas de origem e destino das requisições antes de redirecioná-las para os servidores.
IP *Firewall*	Aceita ou rejeita pacotes com base em campos dos cabeçalhos da camada de rede e transporte.
NIDS	Monitora o tráfego da rede em busca de atividades suspeitas, que podem ser um ataque.
WAN Optimizer	Usado para maximizar a eficiência do fluxo de dados através de uma WAN.

2.2.1 Principio fim a fim sob ameaça

Segundo o princípio fim a fim, algumas funções (como segurança e confiabilidade) só podem ser implementadas completamente e corretamente fim a fim, com o auxílio dos pontos finais (SALTZER; REED; CLARK, 1984). Nesta arquitetura, os únicos dispositivos no núcleo da rede são roteadores IP, e sua única função é determinar as rotas e encaminhar os pacotes até seus destinos. É por isso que eles não são classificados como *middleboxes* (CARPENTER; BRIM, 2002). O uso de *middleboxes* introduz vários desafios ao funcionamento da rede, tais como:

- Interferem na operação de protocolos que não consideram sua presença na rede.

Middleboxes interrompem ou prejudicam operações de inúmeros protocolos, por não serem capazes de lidar corretamente com eles;

- Introduzem novos modos de falhas. É preciso rotear o tráfego para evitar *middleboxes* com mal funcionamento;
- A Configuração não está restrita às pontas da conexão. *Middleboxes* devem ser configurados para permitir o estabelecimento de conexões com sucesso. Há situações em que é necessário configurar múltiplos e complexos *middleboxes* para permitir a conexão entre os pontos finais;
- O diagnóstico de falhas é mais complexo.

Atualmente, observa-se o distanciamento do princípio fim a fim, com a inserção de variadas funções realizadas por *middleboxes*. Ao realizarem operações complexas no núcleo da rede, *middleboxes* ferem o principio fim a fim (EDELINE; DONNET, 2015), que diz que o núcleo da rede deve ser simples e toda complexidade deve estar nas bordas.

2.2.2 Considerações sobre segurança

Cada classe de *middlebox* possui suas próprias questões de segurança, no entanto, há problemas comuns a vários *middleboxes* com relação a este tema (CARPENTER; BRIM, 2002). Alguns desses problemas são elencados a seguir:

- Criptografia fim a fim: a interferência com a transmissão de pacotes fim a fim por muitos tipos de *middleboxes* impede o uso de criptografia fim a fim na sua forma atual (SHERRY et al., 2015), e também invalida a segurança da camada de transporte em muitos cenários.
- Compartilhamento de chaves: *middleboxes* exigem novos modelos de confiança e compartilhamento de chaves (FOSSATI; GURBANI; KOLESNIKOV, 2015).
- Pontos de ataque: *middleboxes* analisam o tráfego e realizam funções essenciais em muitas redes, tornando-os um ponto de ataque potencial. Portanto, devem ser fortemente protegidos.

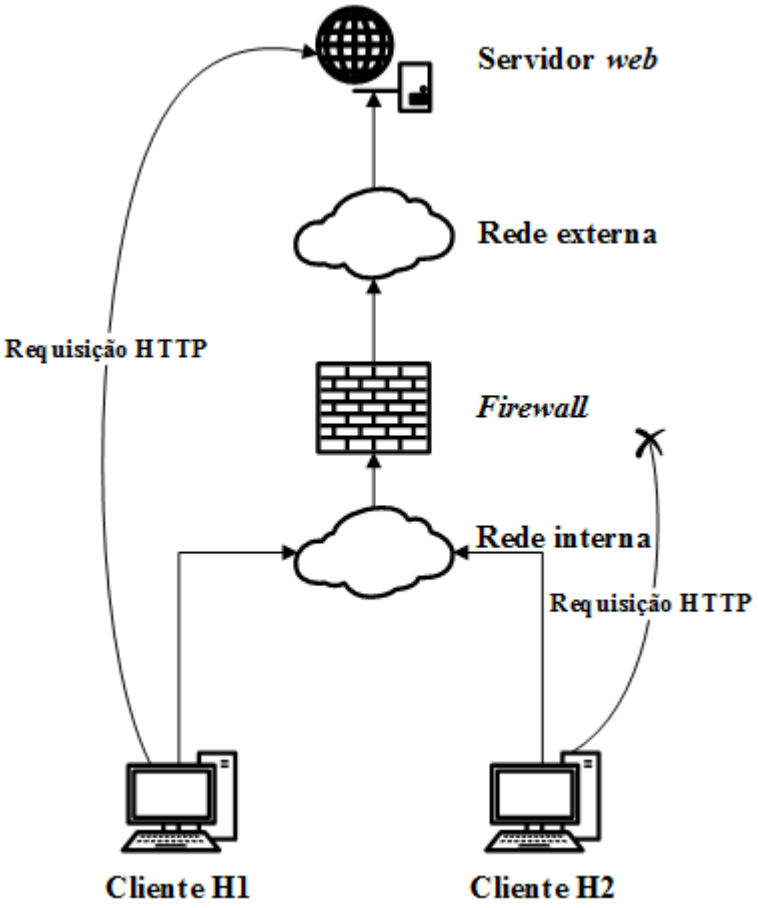
Apesar de serem constantemente utilizados para prover segurança em uma rede, *middleboxes* introduzem vários desafios relacionados a esse tema, como alguns descritos acima. Contudo, como esses dispositivos possuem um papel essencial à operação de inúmeras redes (SEKAR et al., 2012), faz-se necessário utilizá-los mesmo que representem um risco aos ativos da rede e da instituição em que são implantados.

2.2.3 Políticas

O comportamento dos *middleboxes* é definido por políticas, um conjunto de instruções que definem como esses dispositivos devem tratar o tráfego que recebem. Em geral, políticas são definidas como um conjunto de campos de cabeçalho dos protocolos, que caracterizam um determinado tráfego e uma ou mais ações que devem ser tomadas se o *middlebox* receber tráfego com tais características. Ao receber um pacote, o *middlebox* verifica se possui alguma política

que tenha consistência com as informações contidas no pacote e, em caso afirmativo, aplica ao pacote a ação definida pela política selecionada. A Figura 2.4 apresenta um *firewall* com políticas para liberar o acesso do cliente H1 e bloquear do cliente H2 a um servidor *web* na rede externa.

Figura 2.4: Modelo de operação de um *firewall*



Fonte: elaborada pelo autor.

2.3 APIs REST

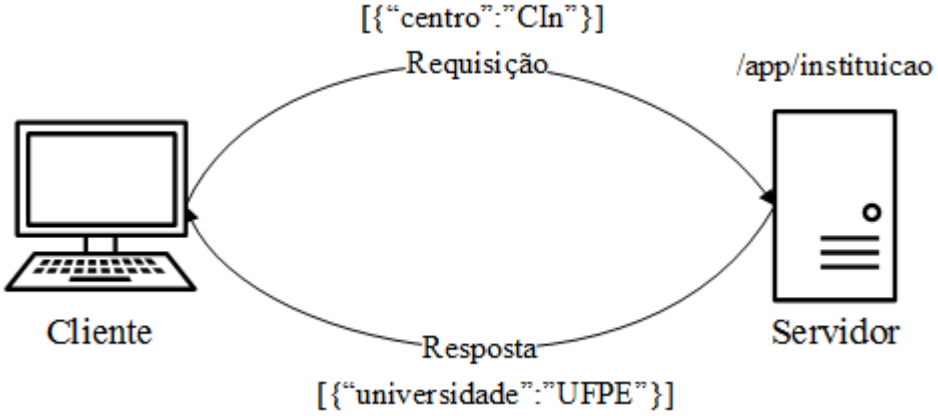
REpresentational State Transfer (REST) é um padrão *web* orientado a recursos que utiliza o protocolo *Hypertext Transfer Protocol* (HTTP) para transmissão de dados. Em REST, cada componente de um sistema é representado como um recurso, identificado por uma URL, e acessado através de métodos HTTP. Abaixo estão alguns métodos HTTP usados em serviços REST:

- *GET*: utilizado para solicitar um recurso ao servidor;
- *POST*: usado para enviar dados ao servidor;
- *PUT*: cria ou atualiza um recurso no servidor;
- *DELETE*: deleta o recurso.

Os dados transmitidos podem ser representados através de diferentes formatos, como texto puro, *eXtensible Markup Language* (XML) ou *JavaScript Object Notation* (JSON) (BRAY, 2014). A Figura 2.5 mostra um exemplo da comunicação entre cliente e servidor utilizando REST e JSON para transmissão dos dados.

O JSON é um formato para transmissão de dados independente de linguagem de programação. O modelo para representação dos dados e sua ordenação em uma lista de valores utilizado por JSON são estruturas de dados universais, aceitas por *softwares* escritos em diferentes linguagens. Os dados são representados como pares chave/valor, separados por dois pontos, em que o campo chave é do tipo *string* e o campo valor pode ser *string*, número, variável booleana, *null*, objeto ou *array* (BRAY, 2014).

Figura 2.5: Modelo de comunicação com REST e JSON



Fonte: elaborada pelo autor.

2.4 Considerações finais

Nesta seção, foi apresentada uma visão geral dos principais conceitos utilizados nesta dissertação. Para isso, foram descritas as principais características de uma rede SDN, abordando seus componentes básicos como o controlador, o *switch* e o protocolo *OpenFlow*. Além disso, foram apresentadas também as características fundamentais dos dispositivos de rede classificados como *middlebox*, bem como os desafios ao operador de rede introduzidos por sua utilização. As tecnologias para formatação e a transmissão de dados JSON e REST também tiveram seus conceitos essenciais descritos nesta seção.

3
Revisão da literatura

Esta seção contempla os trabalhos presentes na literatura que têm como objeto de estudo a aplicação de políticas de *middlebox* em redes SDN. São analisadas abordagens em diferentes contextos. Há trabalhos que tratam dos desafios introduzidos por modificações de tráfego dinâmicas, políticas redundantes, migração de políticas entre instâncias de um *middlebox* e protocolos para configuração de políticas. Além disso, são descritas as principais lacunas nessas propostas.

3.1 Análise dos trabalhos relacionados

A localização adequada dos *middleboxes* em uma rede é um processo muito oneroso. O operador da rede deve localizar manualmente estes dispositivos de forma que o tráfego os atravesse. Invariavelmente, esse posicionamento manual gera vários problemas, como *middleboxes* mal configurados ou falhas que impedem a rede de operar corretamente. Os trabalhos Qazi et al. (2013), Gushchin, Walid e Tang (2015), Cao, Kodialam e Lakshman (2014), apresentam propostas para posicionar *middleboxes* arbitrariamente na rede e encaminhar o tráfego até eles usando a arquitetura SDN. Em Gushchin, Walid e Tang (2015) propõe-se um algoritmo para roteamento do tráfego entre a origem e o destino passando por *middleboxes* consolidados (quando múltiplas funções são realizadas no mesmo dispositivo físico), considerando restrições de capacidade dos enlaces, poder de processamento dos *middleboxes* e tamanho das tabelas dos *switches*. O controlador calcula um caminho entre a origem e o destino dos dados, passando pelos *middleboxes* requeridos, e configura os *switches* que compõem esse caminho para encaminhar o tráfego até o destino. Em Cao, Kodialam e Lakshman (2014) é proposto um algoritmo que calcula todos os caminhos possíveis entre a origem e o destino dos dados passando por uma sequência ordenada de *middleboxes*. No encaminhamento dos dados, podem ser usados múltiplos caminhos desde que todos passem pelos *middleboxes* na ordem especificada. É calculado o caminho mais curto que passa por todos os *middleboxes* requeridos na ordem definida.

O controlador precisa conhecer o tráfego que será encaminhado para os *middleboxes*. Quando ocorrem modificações no tráfego, o controlador não sabe para quais *middleboxes* deve encaminhar o tráfego modificado. Em Qazi et al. (2013), propõe-se que o controlador gere um mapeamento do tráfego que entra e sai de um *middlebox*, como forma de identificar modificações geradas por esse dispositivo. O mapeamento é feito comparando o *payload* dos pacotes com origem e destino nos *middleboxes*. Portanto, há suporte apenas para *middleboxes* que modifiquem somente o cabeçalho dos pacotes, sendo incapaz de lidar com *middleboxes* que modifiquem o *payload*. Usando esse mapeamento, o controlador conhecerá a informação original do tráfego e poderá configurar corretamente a infraestrutura para encaminhá-lo até os *middleboxes* e ao seu

destino.

Os trabalhos Fayazbakhsh et al. (2013, 2014), Ngo e Kim (2014), Bremler-Barr, Harchol e Hay (2015), Ben-Itzhak et al. (2015) propõem o uso de SDN para garantir a aplicação correta das políticas de *middleboxes*. Em Fayazbakhsh et al. (2013, 2014) são apresentadas formas de mitigar problemas causados por modificações no tráfego geradas por *middleboxes*. As referências Fayazbakhsh et al. (2013) e Fayazbakhsh et al. (2014) utilizam um ID para identificar modificações geradas por *middleboxes*. Em Fayazbakhsh et al. (2014), ao receber um pacote que será modificado, o *middlebox* envia as informações desse pacote ao controlador solicitando um ID, o controlador gera este ID, cria um mapeamento entre este ID e as informações do pacote recebido e por fim envia-o para o *middlebox*, que adiciona-lo-á ao cabeçalho dos pacotes que compõem o mesmo fluxo. Ao receber um pacote que contenha um ID em seu cabeçalho, o *middlebox* consultará o controlador pelas informações originais do pacote, mantidas no mapeamento anteriormente gerado pelo controlador. Em Fayazbakhsh et al. (2013) o mapeamento é manualmente configurado pelo operador da rede, já em Fayazbakhsh et al. (2014) o mapeamento é criado e atualizado pelo controlador com informações recebidas dos *middleboxes* através de uma API.

Em Ngo e Kim (2014), também são tratados problemas causados por modificações dinâmicas geradas por *middleboxes*. O primeiro *middlebox* de uma sequência ao receber o primeiro pacote de um fluxo extrai os cabeçalhos e envia-os ao controlador, que gera um ID para o fluxo e mapeia esse ID para o conjunto de cabeçalhos recebidos. O controlador distribui o ID e os cabeçalhos para cada *middlebox* da sequência. No último *middlebox* da sequência, os cabeçalhos são usados para reconstruir o pacote. Como cada *middlebox* recebe o cabeçalho necessário à aplicação de suas políticas, eventuais modificações não afetarão a aplicação de suas políticas. Essa proposta aumenta a carga de trabalho no controlador ao receber os cabeçalhos e distribuí-los entre múltiplos *middleboxes*.

Em Ben-Itzhak et al. (2015), os processos de decisão e aplicação de políticas são desacoplados. *Middleboxes* apenas determinam quais ações devem ser aplicadas ao tráfego, enquanto os *switches* são responsáveis pela aplicação dessas ações. *Middleboxes* determinam quais ações devem ser aplicadas com base nas políticas manualmente configuradas pelo operador da rede. Uma aplicação no controlador configura a infraestrutura para encaminhar os dados até o *middlebox*, recebe a ação que deverá ser aplicada no tráfego, e configura os *switches* para aplicarem essa ação. Apenas um subconjunto de *middleboxes* pode ser utilizado nessa proposta, pois o *switch OpenFlow* dá suporte apenas a algumas das ações realizadas por *middleboxes*.

Há na literatura vários trabalhos como Wang, Butnariu e Rexford (2011), Kaur et al. (2015), Chen e Chen (2015), Suh et al. (2014), Koerner e Kao (2012) que propõem a substituição dos *middleboxes* como dispositivos especializados por aplicações executando no topo do controlador SDN. As aplicações no controlador definem quais ações devem ser aplicadas ao tráfego e configuram os *switches* para que apliquem essas ações. Logo, há a vantagem de reduzir a quantidade de dispositivos na rede. Outra vantagem desse modelo é que múltiplas funções de *middleboxes* podem ser desempenhadas pelo mesmo dispositivo físico, no caso um *switch OpenFlow*, que contribui para a redução dos custos.

Os trabalhos na literatura que tem como proposta substituir *middleboxes* por *switches OpenFlow* gerenciados por aplicações SDN abordam uma variada gama de funções desempenhadas por *middleboxes*. Contudo, nem todas as funções realizadas por *middleboxes* podem ser desempenhadas pelos *switches OpenFlow* atuais. Por exemplo, *middleboxes* do tipo *stateful* requerem o armazenamento de dados sobre as conexões que manipulam, uma característica

inexistente nos *switches OpenFlow* atuais. Logo, são necessários novos *switches* capazes de armazenar dados sobre as conexões que manipulam. Portanto, o uso de *middleboxes* como dispositivos especializados ainda se faz necessário, visto que muitas de suas funções não podem ser realizadas pelos *switches OpenFlow* atuais. Como *middleboxes* realizam as mais diversas funções, para que eles sejam completamente substituídos por *switches OpenFlow* será preciso que esses dispositivos sejam capazes de implementar todas as funções realizados pelos *middleboxes*.

Visto que uma grande variedade de funções desempenhadas por *middleboxes* pode ser realiza por *switches OpenFlow*, uma solução visando a redução do número de dispositivos, quando possível, é implementar funções de *middleboxes* como aplicações no controlador SDN. As demais funções que requerem dispositivos especializados podem ser implementadas utilizando os devidos dispositivos. Portanto, ao usar uma abordagem híbrida constituída por *switches OpenFlow* e dispositivos especializados para implementar funções de *middleboxes*, obtém-se uma redução no número de dispositivos na rede e mantêm-se todos os serviços requeridos.

Em Anwer et al. (2013, 2015) o controle dos *middleboxes* é desacoplado do *hardware*. Funções desempenhadas por *middleboxes* são realizadas por *softwares* e, executados no *hardware* desses dispositivos. O controlador SDN realiza a migração desses *softwares* entre o *hardware* dos *middleboxes*, e encaminha o tráfego até o dispositivo responsável por tratá-lo.

Em Bremler-Barr, Harchol e Hay (2015), os planos de dados e controle dos *middleboxes* são desacoplados e o plano de dados de múltiplos *middleboxes* é unificado em entidades chamadas de *service instances*. Aplicações que definem o comportamento do plano de dados dos *middleboxes* são executadas no plano de controle logicamente centralizado. Há situações em que múltiplos *middleboxes* executam funções similares, como *Intrusion Prevention Systems* (IPSs) e *firewalls* filtrando a mesma porta, ou *middleboxes* com políticas sobrepostas. Em Bremler-Barr, Harchol e Hay (2015), as políticas são estruturadas de forma que múltiplos *middleboxes* que tratam o mesmo tráfego não apliquem políticas redundantes.

O *framework* projetado em Gember et al. (2012), posteriormente implementando e avaliado em Gember-Jacobson et al. (2014), tem como objetivo prover capacidades para o controlador SDN migrar políticas entre instâncias de um *middlebox*. Após o processo de migração ser concluído, os *switches* são configurados para encaminhar para a nova instância os dados tratados pelas políticas transferidas. O controlador recebe as políticas da instância de origem e as envia para a instância de destino, utilizando APIs. As referências Gember-Jacobson e Akella (2015) e Kothandaraman, Du e Sköldström (2015) apresentam melhorias ao *framework* implementado em Gember- Jacobson et al. (2014). Em Gember-Jacobson e Akella (2015), são apresentadas duas melhorias, o reprocessamento de pacotes, permitindo que uma instância de *middlebox* continue processando os pacotes durante a migração das políticas e, transferência *peer-to-peer* (P2P) que libera instâncias a transmitirem políticas diretamente entre si, reduzindo a carga de trabalho do controlador. Em Kothandaraman, Du e Sköldström (2015), propõe-se também a transferência P2P das políticas. O controlador atua apenas trocando mensagens com *middleboxes* para coordenar o processo de transferência, mas não participa diretamente da transmissão das políticas. Para que o controlador insira políticas em uma instância, essas políticas devem ter sido previamente configuradas em outra instância.

Os protocolos MIDCOM e SIMCO são utilizados para configurar *firewalls* e NATs para lidarem com aplicações dinâmicas (KHLIFI; GREGOIRE; PHILLIPS, 2006). Esses protocolos exigem que pontos finais estejam cientes sobre a existência de *middleboxes* na rede e quais tratam seu tráfego. Esse comportamento fere diretamente o princípio dos *middleboxes*: *hosts* não devem saber sobre sua presença na rede (CRAVEN; BEVERLY; ALLMAN, 2014). Outro

problema é o uso de *hosts* na configuração desses dispositivos, já que por motivos de segurança os pontos finais da comunicação não devem conhecer as políticas dos *middleboxes* Sherry et al. (2015). Portanto, além de estarem limitados a apenas dois tipos de *middleboxes*, esses protocolos apresentam sérios riscos a segurança desses dispositivos e consequentemente aos recursos por eles protegidos.

Há na literatura vários esforços no uso de SDN para resolução de problemas gerados por *middleboxes*. Há esforços na tentativa de solucionar problemas causadas por modificações no tráfego, e como elas impactam a aplicação de políticas. Há esforços para evitar políticas redundantes e para usar *switches OpenFlow* na aplicação das políticas. Contudo, políticas continuam sendo configuradas manualmente pelo operador, que, como citado anteriormente, leva a inúmeros problemas. Geralmente são propostas soluções que exigem a substituição completa dos *middleboxes* existentes, uma alternativa demasiadamente dispendiosa, considerando a grande quantidade desses dispositivos normalmente encontrada nas rede atuais (SHERRY et al., 2012).

3.2 Coautoria em publicação sobre *middleboxes* em SDN

Em Gondim, Pinheiro e Campelo (2015) é apresentada uma arquitetura para o monitoramento de desempenho de aplicações na presença de *middleboxes* em uma rede SDN. Seu objetivo é mitigar problemas no monitoramento de aplicações geradas por esses dispositivos. São obtidas informações sobre fluxos da rede a partir do controlador e sobre políticas aplicadas pelos middleboxes ao processar dados das aplicações monitoradas. Os dados sobre os fluxos são capturados e analisados pelo *software Wireshark*. As informações sobre as políticas dos *middleboxes* são recebidas utilizando o formato JSON.

Um *host* obtém dados do controlador e dos *middleboxes* e os usa para obter duas métricas fundamentais para o monitoramento de desempenho: o tempo de resposta e a disponibilidade. As informações recebidas são filtradas a partir dos endereços IP e das portas de destino das requisições, e em seguida são processados pelo *software* de monitoramento para que ele compute o tempo de resposta e disponibilidade da aplicação. Também é computada a quantidade de *bytes* transmitidos em cada conexão e são disponibilizadas métricas específicas de cada *middlebox*.

Esse trabalho é resultado da colaboração com a engenheira Ethel B. Gondim mantida durante o mestrado. Apesar de o principal objetivo do trabalho não estar diretamente ligado à aplicação de políticas, há elementos relacionados com a pesquisa apresentada nesta dissertação, como desafios gerados por *middleboxes* em uma rede SDN e a obtenção de informações sobre a atuação desses dispositivos. A API utilizada na obtenção de informações dos *middleboxes* foi inicialmente desenvolvida no presente trabalho de mestrado e aprimorada durante a colaboração. Desafios enfrentados no monitoramento de aplicações são causados pela aplicação das políticas dos *middleboxes*, objeto de estudo deste trabalho de mestrado.

Foram empregadas APIs específicas para cada *middlebox*, com objetivo de obter informações sobre as políticas aplicadas por eles (GONDIM; PINHEIRO; CAMPELO, 2015). Esses dados são utilizados no monitoramento das aplicações que atravessam esses dispositivos. De forma semelhante, a presente pesquisa obtém dados sobre políticas aplicadas por *middleboxes*, usando-os na configuração de seus pares. Portanto, pontos importantes de ambos os trabalhos estão diretamente relacionados.

3.3 Considerações finais

Nesta seção, foram apresentados, de forma mais detalhada, os trabalhos que tratam de desafios relacionados à aplicação de políticas de *middleboxes* e que possuem características mais próximas ao trabalho desenvolvido nesta dissertação. Ademais, foi realizada uma análise que identificou as principais brechas nas pesquisas destes trabalhos que são preenchidas por esta dissertação.

4
Arquitetura para aplicação de políticas

Nesta seção é apresentada a arquitetura desenvolvida neste trabalho para aplicação das políticas de *middleboxes* em uma rede SDN. Seu objetivo é realizar a configuração dinâmica de políticas em *middleboxes*, com o emprego de características da arquitetura SDN, oferecendo uma maneira simples para obter e configurar políticas com impacto mínimo no desempenho da rede. Políticas podem ser configuradas usando informações sobre o tráfego da rede, políticas obtidas dos *middleboxes*, ou outras fontes de informações que o desenvolvedor desejar.

4.1 Metodologia

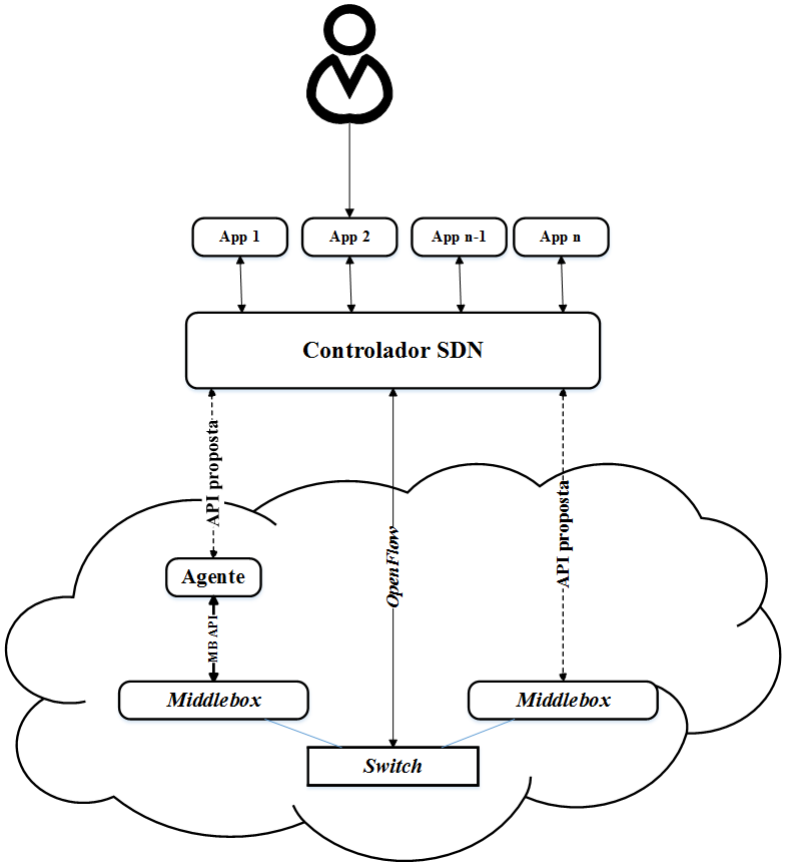
Para a elaboração da arquitetura para aplicação de políticas e o desenvolvimento e avaliação do protótipo, foram realizadas as seguintes etapas:

- Determinação do modelo de funcionamento da arquitetura: elaboração de um modelo de interação entre o controlador SDN e *middleboxes*, incluindo uma descrição de como seriam utilizadas características da arquitetura SDN na configuração das políticas nos *middleboxes*.
- Escolha do mecanismo para comunicação com *middleboxes*: verificação de qual mecanismo de comunicação seria mais adequado para interação entre *middleboxes* e o controlador SDN. Definição de qual formato e método para transferência de dados cumpriria melhor os requisitos para comunicação com *middleboxes*.
- Escolha do modelo para representação de políticas de *middleboxes*: revisão dos modelos de representação de políticas usados por *middleboxes* mais comuns, e proposta de um modelo unificado para representação de políticas de diferentes *middleboxes*.
- Implementação do protótipo: desenvolvimento de um protótipo para aplicação de políticas que utilize a arquitetura proposta como referência.
- Validação: planejamento e execução de uma série de experimentos para avaliar o protótipo, verificando a viabilidade da arquitetura proposta com base nos resultados obtidos.

4.2 Funcionamento da arquitetura

A arquitetura proposta neste trabalho tem como finalidade estender SDN para aplicação de políticas de *middleboxes*. A arquitetura é apresentada na Figura 4.1. Características como controle logicamente centralizado e programabilidade do comportamento dos dispositivos, presentes em SDN, são utilizadas na configuração das políticas. Atualmente, *middleboxes* são configurados manualmente, tornando-os incapazes de aplicar corretamente suas políticas na presença de aplicações dinâmicas e eventos não previstos pelo operador da rede. O controlador SDN é logicamente centralizado e possui uma visão global sobre a rede que gerencia, permitindo-o conhecer características do tráfego da rede. Dispositivos gerenciados pelo controlador SDN são dinamicamente configurados em resposta a eventos ocorridos na rede. O controlador poderá instruir *middleboxes* para que apliquem suas políticas de forma dinâmica com base em sua visão sobre o tráfego da rede. A programabilidade presente em SDN é muito valiosa no desenvolvimento de aplicações para configuração e aplicação de políticas de *middleboxes*. Mais detalhes sobre o desenvolvimento de aplicações são apresentados na Seção 4.3.

Figura 4.1: Modelo conceitual da arquitetura para aplicação de políticas de *middleboxes*



Fonte: elaborada pelo autor.

Middleboxes mantêm seu modelo de configuração manual, mas têm suas capacidades estendidas com o uso da arquitetura proposta, que os torna capazes de aplicar suas políticas para atender requisitos de aplicações dinâmicas e lidar com eventos ocorridos na rede. O operador de rede poderá configurar políticas de uma posição logicamente centralizada, ao invés de configurar dispositivos individuais fisicamente dispersos na rede, e poderá desenvolver aplicações para automatizar a configuração de políticas, simplificando o processo de aplicação de políticas.

Para tornar a arquitetura proposta nesta dissertação simples e compatível com implementações SDN e de *middleboxes* existentes, procurou-se utilizar tecnologias universalmente usadas, como JSON e REST. O controlador requer apenas modificações simples para dar suporte à comunicação com *middleboxes*, que, por sua vez, exigem modificações igualmente simples para receber e aplicar políticas enviadas pelo controlador. Nos *middleboxes*, essas modificações podem ser evitadas com o uso de *softwares* atuando como *proxies*. Aplicações SDN também não sofrem alterações, apenas precisam utilizar a interface para comunicação com *middleboxes* na configuração das políticas. No projeto da interface, foi definido o uso das tecnologias REST e JSON, populares em APIs de *middleboxes* proprietários e implementação de interfaces *northbound* em controladores SDN. Mais detalhes são apresentados na Seção 4.4.

Aplicações no topo do controlador utilizam a interface proposta nesta dissertação para configurar e receber políticas dos *middleboxes*. Informações sobre políticas são armazenadas no próprio controlador. Na seção 4.7, é descrito com mais detalhes o armazenamento dessas informações. Os *middleboxes*, além de receberem, utilizam a interface para enviar ao controlador políticas que tenham aplicado, que poderão ser utilizadas na configuração de outros *middleboxes*. Para a transmissão dos dados das políticas, foram avaliadas formas de minimizar a quantidade de informação transmitidas, reduzindo, assim, o impacto no desempenho da rede.

Em redes de grande escala, o uso de múltiplos controladores é essencial, visto que o controlador possui recursos limitados e lidar com grandes quantidades de *switches* demanda mais recursos do que os disponíveis (DIXIT et al., 2014). Em ambientes com múltiplos controladores, cada um gerencia uma parcela dos *switches*, dividindo a carga de trabalho entre si e permitindo que novos dispositivos possam ser adicionados sem prejuízo ao funcionamento da rede.

É fundamental que o controlador seja escalável, mantendo um bom desempenho. Para obter escalabilidade em um controlador, é crucial reduzir sua carga de trabalho, através, por exemplo, da redução da quantidade de dados para ele transmitidos (WANG et al., 2014). Para evitar problemas de desempenho e escalabilidade do controlador, a arquitetura reduz a quantidade de dados transmitidos ao utilizar apenas as informações estritamente necessárias para caracterização de políticas e, utiliza o formato JSON para transmissão das políticas.

4.3 Aplicações e controlador

No controlador foram executadas aplicações SDN com suporte à interface para a comunicação com *middleboxes*. Essas aplicações foram passadas como parâmetro de linha de comando para o *script python* do controlador *Ryu* responsável por executá-las. Não há mudanças significativas no desenvolvimento das aplicações. Aplicações obtêm informações tanto da infraestrutura da rede, através do controlador, quanto dos *middleboxes*, via interface proposta.

Ao receber uma mensagem *packet-in* do *switch*, o controlador repassa essa mensagem para as aplicações em execução. Neste trabalho, como essas aplicações foram passadas como parâmetro do *script python* do controlador *Ryu*, as mensagens *packet-in* são repassadas para as aplicações como objetos da linguagem *python*. O *packet-in* recebido pelas aplicações contém informações que caracterizam um fluxo da rede, e essas informações poderão ser extraídas e usadas para configurar políticas nos *middleboxes*. Por exemplo, ao receber um pacote contendo uma requisição HTTP, a aplicação poderá verificar se o cliente tem permissão para acessar o servidor *web*, e então configurar um *firewall* para liberar ou bloquear essa requisição, de acordo com os direitos de acesso do cliente. Quando o *firewall* receber os pacotes HTTP, aplicará a

Tabela 4.1: APIs e SDKs de middleboxes proprietários e respectivos fabricantes

Fabricante	API
A10	aXAPI
Cisco	ASA REST API
Silver-Peak	VXOA REST API
BlueCoat	Director XML API
Microsoft	Microsoft Forefront TMG SDK
Hewlett-Packard	HP TippingPoint Advanced Threat API
Intel/McAfee	SMC REST API
Brocade	ADX OpenScript API
Procera	*Network Application Visibility Library* SDK

política configurada pelo controlador. Ao receber uma política de um *Intrusion Detection System* (IDS) alertando sobre a presença de tráfego malicioso na rede, poderá usar as informações dessa política para configurar um *firewall* para bloquear esse tráfego.

4.4 Comunicação com middleboxes

Middleboxes proprietários geralmente possuem APIs e *Software Development Kits* (SDKs), que permitem sua interação com agentes externos e desenvolvimento de aplicações que estendam suas funções. Contudo, o controle sobre ações desempenhadas pelo dispositivo é limitado pelo fabricante. Devido à sua natureza proprietária, não é possível modificar tais dispositivos, ficando-se sujeito ao escasso conjunto de opções permitidas pelo fabricante. Por outro lado, *middleboxes open source* podem ser modificados para atender às necessidades do desenvolvedor. Entretanto, tais *middleboxes* são complexos, compostos por milhares de linhas de código, que torna a tarefa de modificá-los muito onerosa.

Não há uma API padrão para comunicação com *middleboxes* proprietários, e muitos *middleboxes open source* nem sequer possuem API. Na Tabela 4.1 são apresentados exemplos de APIs para *middleboxes* comerciais. Mesmo quando o *middlebox* possui API para gerenciamento, considerando a grande quantidade desses dispositivos presentes em uma rede (SHERRY et al., 2012), parece-nos muito dispendioso trabalhar com tantas APIs distintas. Os desafios acima citados motivaram o desenvolvimento da interface proposta nesta dissertação.

Os requisitos para o desenvolvimento da interface são baixo impacto no desempenho da rede, simplicidade, e utilização de tecnologias universalmente adotadas. O impacto no desempenho da rede gerado pela interface é determinante para seu sucesso ou fracasso, visto que sua utilização torna-se inviável caso gere um grande impacto. A quantidade de dados transmitidos entre *middleboxes* e controlador deve ser minimizada, usando uma quantidade mínima de informações para caracterizar políticas. As políticas são transmitidas como uma tupla formada por:

- **contexto**: informações que caracterizam pacotes a serem tratados pela política, como campos de cabeçalhos.
- **ação**: a ação que será aplicada aos pacotes que condizem com o contexto especificado.

Atendendo aos requisitos para o desenvolvimento da interface, foram escolhidos JSON para formatação e REST para transferência dos dados. Provavelmente o uso de REST e JSON

simplificará a adoção da API proposta nesta dissertação, visto que ambas as tecnologias são muito simples e universalmente aceitas. Alguns dos controladores SDN mais populares possuem mecanismos para desenvolvimento de APIs REST, utilizando esse tipo de API para desenvolvimento de interfaces *northbound* (ZHOU et al., 2014). Grandes empresas do mercado de *middleboxes*, como *Cisco*, *Silver-Peak* e *Intel/McAfee*, possuem APIs que utilizam JSON e REST para comunicação com os *middleboxes*. Atualmente, a interface possui duas operações e ambas utilizam o método HTTP *POST* para o envio das políticas. As operações disponíveis na API são:

- InsertPolicy: utilizada pelo controlador para configurar políticas nos *middleboxes*;
- InformCtr: usada pelos *middleboxes* para enviar políticas ao controlador.

A estrutura de uma requisição é apresentada no código abaixo:
Operação **InsertPolicy** para um *Firewall*:

```
{"context":{"src_ip": "192.168.1.10",
"dst_ip": "168.168.1.11",
"src_port": "54532",
"dst_port": "4345"},
"action":{"action": "block"}
}
```

Operação **InformCrt** par um IPS:

```
{context:{"src_ip": "172.16.0.11",
"dst_ip": "172.16.0.12",
"src_port": "5432",
"dst_port": "44345"},
"action":{"status": "liberado"}}
```

Abaixo estão as respostas para requisições, com códigos 200 e 500, requisição bem sucedida e erro no servidor, respectivamente:

```
{"code": "200"}
```

```
{"code": "500",
"message": "Server overloaded"}
```

Em uma requisição bem sucedida, o código HTTP 200 é suficiente para identificar o *status* da requisição, e garante que apenas as informações necessárias são transmitidas, minimizando o tráfego gerado na comunicação entre *middleboxes* e controlador. Por outro lado, quando ocorre algum erro, o código que identifica o erro é transmitido junto com uma mensagem contendo o problema reportado pelo *middlebox*. Essa mensagem contém informações úteis para identificar a(as) causa(s) do problema.

4.5 Agentes de *software*

Para não depender do fabricante de *middlebox* que é o único a possuir permissão para modificar este dispositivo para que passe a dar suporte à interface proposta nesta dissertação, propomos o uso de *softwares* que atuam como um *proxy* entre o *middlebox* e o controlador SDN.

Como mencionado anteriormente, esses agentes são usados para simplificar a integração da arquitetura proposta nesta dissertação com *middleboxes* proprietários. Estes *softwares* podem ser utilizados também para simplificar a integração com *middleboxes open source* que, apesar de estarem aptos à modificações são *softwares* muito complexos e por isso modificá-los torna-se uma tarefa muito dispendiosa.

Os agentes implementam a API do *middlebox* disponibilizada por seu fabricante para configurar políticas neste dispositivo sem precisar modificá-lo. Por outro lado, estes agentes implementam também a API proposta nesta dissertação para que sejam capazes de receber e enviar as políticas ao controlador.

Esses agentes são componentes opcionais na arquitetura proposta e seu objetivo é simplificar a adoção da interface proposta nesta dissertação por parte dos *middleboxes*. Estes *softwares* são transparentes ao controlador que, crê estar interagindo diretamente com um *middlebox*. Portanto, estes *softwares* além de evitarem modificações nos *middleboxes*, não exigem adaptação alguma no controlador.

A princípio, a responsabilidade pelo desenvolvimento dos agentes fica a cargo do operador da rede, visto que esses agentes são usados quando o fabricante do *middlebox* não adiciona suporte à API proposta neste trabalho de mestrado. Contudo, os fabricantes de *middleboxes* podem adicionar suporte em seus dispositivos à API proposta neste trabalho por meio de agentes ou *plugins*. Dessa forma, os fabricantes poderão adicionar suporte à API proposta sem precisar modificar seus *softwares*. Como motivação para que os fabricantes adicionem suporte à API proposta, há a possibilidade de desenvolvimento de novos serviços a partir da integração entre SDN e *middleboxes* apresentada neste trabalho mestrado.

4.6 *Middleboxes* e estado

Há uma grande variedade de tipos, implementações e fabricantes de *middleboxes*, cada um com suas próprias formas de representar o estado interno desses dispositivos. Representar o estado de múltiplos *middleboxes* de forma unificada é um grande desafio. Contudo, geralmente todos os *middleboxes* têm uma característica comum: inspecionam e manipulam o tráfego da rede. Portanto, compreendem informações comuns, tais como endereços IP e portas de transporte. Com base nessa premissa, acreditamos que uma forma de representar o estado dos *middleboxes* seja através das informações sobre o tráfego que manipulam e ações que aplicam, uma proposta semelhante à apresentada em (GEMBER et al., 2012). A Tabela 4.2 apresenta políticas representadas com esse modelo.

Os *middleboxes* foram escolhidos de forma a abranger implementações comuns em ambientes de produção, tendo como referência (SHERRY et al., 2012). Os *middleboxes* escolhidos foram o *firewall*, o balanceador de carga e o sistema de detecção e prevenção de intrusão (*Intrusion Detection/Prevention System*, IDS/IPS). São obtidas informações sobre as políticas aplicadas pelo balanceador de carga e pelo IDS/IPS. Essas informações são usadas para caracterizar uma conexão, como os IPs e portas de origem e destino, e a ação aplicada nesta conexão.

4.7 Armazenamento dos dados

Para o armazenamento dos dados sobre políticas, uma opção seria o uso de um banco de dados relacional, mas essa opção se mostra muito onerosa, pois, como mostrado em (KRISH-

Tabela 4.2: Modelo de representação das políticas de *middleboxes*

Middlebox	informações	ação
Firewall	origem 10.0.0.2	bloquear
NAT	origem 10.0.0.5	mapear para 200.10.3.5
IPS	Pacote contém TCP SYN	alertar
Proxy	www.cin.ufpe.br	armazenar no cache

NAMURTHY; CHANDRABOSE; GEMBER-JACOBSON, 2014), o uso de um sistema externo para armazenamento de informações presentes no controlador impacta consideravelmente o desempenho da rede. Portanto, para evitar tal impacto, o armazenamento dados sobre políticas é realizado no próprio controlador. Na subseção 7.2.2, são discutidos elementos que justificam o uso de um banco de dados para o armazenamento das políticas, objeto de trabalhos futuros.

Informações armazenadas no controlador são acessadas mais rapidamente pelas aplicações do que por um sistema de armazenamento externo (KRISHNAMURTHY; CHANDRABOSE; GEMBER-JACOBSON, 2014). Assim, com informações mantidas no controlador, dados sobre políticas serão acessados mais rapidamente e, consequentemente, a configuração dos *middleboxes* será mais ágil.

Políticas são armazenadas na estrutura "dicionário" da linguagem *Python* e agrupadas em uma lista. Em *Python*, "dicionário" é uma coleção de pares chave:valor separados por vírgulas. JSON emprega estruturas similares na formatação dos dados transmitidos, tornando o processo de transição entre o recebimento e o armazenamento dos dados mais simples. Ao receber uma política, a aplicação extrai os dados dessa política e os posiciona em suas respectivas chaves. Por exemplo, ao receber o endereço IP de origem coloca seu valor na chave "src", como "src":"192.168.2.101". Depois que todos os dados de uma política são armazenados no "dicionário", ela é adicionada a uma lista contendo todos as políticas recebidas.

4.8 Considerações finais

Nesta seção, foi apresentada uma descrição detalhada da arquitetura proposta nesta dissertação, assim como os seus componentes e como interagem entre si. Foi descrita como as características da arquitetura SDN são empregadas na aplicações das políticas, como é realizada a comunicação entre controlador SDN e middleboxes e como se dá a representação e armazenamento das políticas. Além disso, foi descrito como é possível integrar *middleboxes* à arquitetura proposta nesta dissertação sem realizar modificação alguma nestes dispositivos.

5 Protótipo para aplicação de políticas

Nesta seção, é descrito o protótipo implementado com base na arquitetura proposta nesta dissertação e apresentada na seção anterior. Além disso, são descritas as ferramentas e tecnologias empregadas na implementação deste protótipo, assim como as configurações realizadas nos *middleboxes* avaliados e no ambiente de emulação do *Mininet*.

5.1 Descrição do protótipo

O protótipo para aplicação das políticas de *middleboxes* foi implementado em uma máquina virtual (do inglês, *Virtual Machine*, VM) com o *software* de emulação *Mininet* 2.2.1, disponível na página *web* do projeto *Mininet*. Essa VM possui o sistema operacional Ubuntu 14.04 *Long Term Support* (LTS) de 64 *bit*, binários OpenFlow, *Wireshark*, *OpenVswitch*, dentre outras ferramentas. Essa VM foi configurada com 4 *Gigabytes* de memória *Random Access Memory* (RAM) *Double Data Rate* (DDR) 3 de 1333 Mhz, processador *Advanced Micro Devices* (AMD) *Phenom*^{TM} II b97 de 3.20 Ghz de 64 *bit* e virtualização por *hardware* habilitada. O sistema operacional da máquina hospedeira é o *Windows* 7 *professional* de 64 bit. Essa VM contém o *OpenVswitch* 2.0.2, que não possui suporte completo ao *OpenFlow* 1.3. Por isso, foi compilado o código fonte do *OpenVswitch* 2.3 que possui suporte completo a essa versão do *OpenFlow*. Nessa VM foi criada a topologia em que foram realizados os experimentos para avaliação deste protótipo. Esta topologia possui múltiplos *switches OpenVswitch* gerenciados pelo controlador *Ryu* usando o *OpenFlow* 1.3 como interface *southbound*. Foram utilizados apenas *middleboxes open source*, devido aos custos com licenças e restrições impostas por *softwares* proprietários. Foram utilizados o *firewall Iptables* (XUAN; WU, 2015), o IDS/IPS *Snort* (ALHOMOUD et al., 2011) e o balanceador de carga *HAproxy* (GARG; BAGGA, 2015). Para cada *middlebox* foi desenvolvido um agente de *software* na linguagem *Python*, que o configura ou obtém suas políticas. A linguagem *Python* foi utilizada devido à sua simplicidade de desenvolvimento e ser multiplataforma. O controlador *Ryu* usa a API proposta para comunicar-se com os agentes, que por sua vez, comunicam-se diretamente com os *middleboxes* atuando como *proxies*.

As seções a seguir detalham a implementação das aplicações SDN que gerenciam *middleboxes*, modificações realizadas no controlador, o desenvolvimento da API para comunicação com *middleboxes*, a implementação dos agentes que configuram e obtêm políticas, a configuração dos *middleboxes* e o ambiente de emulação *Mininet*.

5.2 Aplicações

As aplicações em execução no controlador praticamente não sofreram modificações. A única mudança é que essas aplicações passam a utilizar a interface de comunicação com *middleboxes* para configuração e obtenção das políticas desses dispositivos. As aplicações buscam nas mensagens *packet-in* recebidas por informações pré-definidas, como tipo de protocolo da camada de aplicação e endereços IP. Ao encontrarem as informações definidas pelo desenvolvedor, as utiliza para configurar políticas nos *middleboxes* a serem aplicadas por esses dispositivos. Por exemplo, a aplicação responsável por configurar *firewalls* para permitir o funcionamento da comunicação VoIP busca nas mensagens *packet-in* pelos protocolos *Session Initiation Protocol* (SIP) e RTP. Esses protocolos identificam a comunicação VoIP.

O *Ryu* possui suporte a alguns protocolos populares como IP, TCP, UDP e *Ethernet*. Esses protocolos são representados por estruturas pré-definidas e seus campos são armazenados em variáveis, enquanto os demais protocolos são representados como *strings*. No caso do SIP e do RTP, as *strings* que representam seus cabeçalhos contêm o nome do respectivo protocolo. Dessa forma, para identificar se o *packet-in* contém o cabeçalho do SIP, basta buscar o termo SIP no *packet-in* recebido. Se confirmada a presença de cabeçalhos SIP ou RTP, os endereços IP de origem e destino e porta de destino são utilizados na configuração das políticas no *firewall*. Essa aplicação é apresentada no algoritmo 5.1.

É preciso garantir que políticas sejam configuradas antes que o tráfego tratado por elas seja encaminhado até o *middlebox*. Caso contrário, o *middlebox* não será capaz de aplicar as políticas corretas ao receber o tráfego. Para isso, primeiro as políticas são configuradas nos *middleboxes* e só depois são configurados os *switches* para encaminhar o tráfego até esses dispositivos.

Aplicações que recebem políticas dos *middleboxes* permanecem em execução à espera das políticas e, ao recebê-las as armazena no controlador. As políticas recebidas por essas aplicações podem ser usadas para configurar outros *middleboxes*. Por exemplo, a aplicação que obtém políticas do IPS é capaz de utilizar as informações recebidas para configurar políticas no *firewall*, como bloquear tráfego identificado como malicioso.

5.3 Controlador Ryu

O controlador *Ryu* possui uma função de servidor *web* correspondente ao *Web Server Gateway Interface* (WSGI), que permite o desenvolvimento de APIs REST. No controlador, a API proposta nesta dissertação foi implementada utilizando recursos nativos do *Ryu*, como a função WSGI. A função WSGI mantém um mapeamento entre *Uniform Resource Locators* (URL) e funções *Python* para saber qual função deve ser chamada ao receber uma requisição para determinada URL. Esse mapeamento é chamado de *rota*. O *Ryu* usa *@route* para indicar que uma função *python* deve ser mapeada para uma URL. Para gerar as requisições usadas na transmissão das políticas para os *middleboxes*, foi utilizada a biblioteca *pycurl*[1].

[1] Usada para obter objetos identificados por uma URL a partir de um programa *Python*

5.4 Interface de comunicação com *middleboxes*

Na implementação da interface nos agentes, foi utilizado o *Flask* (GRINBERG, 2014), um micro *framework web* escrito em *python* para desenvolvimento de APIs REST. Nesse contexto, micro significa que o *Flask* possui apenas as caraterísticas mínimas necessárias ao desenvolvimento de APIs REST. Por exemplo, o *Flask* não possui conexão com banco de dados. Uma grande vantagem de usar esse *framework* é o baixo consumo de recursos computacionais que contribui para melhorar o desempenho dos agentes. O algoritmo 5.2 apresenta um exemplo de uma aplicação que utiliza o *framework Flask* para implementar uma API *REST*.

A operação *InsertPolicy* foi implementada no agente do *firewall iptables* como *InsertPolicyFW*. Para enviar políticas ao *firewall*, o controlador gera requisições com o método HTTP *POST* ao endereço *http://<ip_firewall>/InsertPolicyFW*, passando as informações da política no formato JSON. Nas aplicações responsáveis por receber políticas dos *middleboxes*, a operação *InformCtr* foi implementada como *InformCtrIPS* e *InformCtrLB* para o IPS e o balanceador de carga, respectivamente. Ao informar sua aplicação sobre políticas aplicadas, o IPS gera requisições HTTP *POST* para o endereço *http://<ip_controlador>/InformCtrIPS*. O balanceador de carga gera requisições HTTP *POST* ao endereço http://<ip_controlador>/InformCtrLB, informando suas políticas aplicadas. Ao receberem requisições HTTP *POST*, tanto o agente do *firewall* quanto o controlador executam a função *python* mapeada para a URL solicitada: *insertFW* para o *firewall*, e *InformIPS* e *InformLB* para as aplicações do IPS e balanceador de carga, respectivamente.

Neste trabalho de mestrado, as operações da API proposta *InsertPolicy* e *InformCtr* foram implementadas em um *firewall*, um *IPS* e um balanceador de carga, mas essas operações poderão ser implementadas outros *middleboxes*.

5.5 Agentes de *software*

Os agentes foram desenvolvidos na linguagem *python*, por possuir uma sintaxe simples que facilita o desenvolvimento das aplicações. O código total de cada agente não ultrapassou 50 linhas, sendo mais simples de gerenciar do que, por exemplo, modificar o *software Snort*, composto por mais de 30.000 linhas. O algoritmo 5.3 apresenta o agente para configuração de um *firewall*, em que é implementada a API proposta utilizando o *flask*. Para a configuração de políticas no *iptables*, foi usado um módulo que implementa a API do *iptables* na linguagem *python* chamado de *python-iptables* (PYTHON-IPTABLES'S, 2015). O agente implementa a interface proposta neste trabalho para receber as políticas do controlador e depois as configura usando a API do *iptables*. Esse agente permanece em execução a espera de políticas enviadas pelo controlador SDN.

Para obter informações sobre políticas aplicadas pelos *middleboxes* sem precisar modificá-los, foram utilizados agentes para extrair informações relevantes sobre políticas dos *logs* dos *middleboxes*. Os *logs* contêm todas as informações necessárias e, como são gerados pelos próprios *middleboxes*, representam fielmente o que ocorre nesses dispositivos. É uma maneira mais simples do que modificar os *middleboxes* e evita o aumento na carga de trabalho desses dispositivos, pois eles não desempenharão mais as funções de coletar e enviar ao controlador dados sobre sua execução. Sempre que uma politica é adicionada ao *log*, o agente de *software* extrai as informações necessárias e as envia ao controlador através da interface proposta.

Algoritmo 5.1: Aplicação para configuração de políticas em um *firewall*

```
   Entrada: Packet-in
   Saída: Política e regra de fluxo
 1 início
 2     repita
 3         Recebe o Packet-in;
 4         if Não existe política com essas informações then
 5             Ler o primeiro cabeçalho;
 6             repita
 7                 if Pacote da camada de enlace then
 8                     Cria uma regra de fluxo;
 9
10                 if Pacote da camada de rede then
11                     Extrai endereços de origem e destino;
12                     Atualiza a regra de fluxo criada;
13
14                 if Pacote da camada de transporte then
15                     Extrai porta de destino;
16                     Atualiza a regra de fluxo;
17                     if Pacote UDP then
18                         Atualiza a regra de fluxo;
19
20
21                 if Pacote RTP ou SIP then
22                     Cria uma política com informações extraídas do packet-in;
23                     Envia a política para o firewall;
24                     if política configurada com sucesso then
25                         Atualiza a regra de fluxo para encaminhar os dados até seu
                           destino;
26
27                     else
28                         Atualiza a regra de fluxo para descartar o tráfego
29                     end
30
31                 Ler o próximo cabeçalho;
32             até Chegar ao último cabeçalho;
33             Envia a regra de fluxo para os switches;
34
35         else
36             Ignorar
37         end
38     até A aplicação ser parada;
39 fim
```

Algoritmo 5.2: Implementação miníma de uma API no flask

```
1          from flask import Flask
2          from flask import request, redirect
3          import requests
4
5          app = Flask(__name__)
6
7          @app.route('/', methods = ['POST'])
8          def index():
9                  ...
10                 return ""
11         if __name__ == '__main__':
12                 app.run(host='0.0.0.0', port=5000)
```

Do *log* do *snort*, são extraídos endereços IP e de porta de origem e de destino, bem como a ação aplicada (liberar/bloquear). Do *HAproxy*, são extraídas informações sobre endereços IP e porta de origem e destino das requisições e IP e porta do *HAproxy* usado na requisição criada para o servidor *web* ao balancear as requisições. A Tabela 5.1 apresenta as informações obtidas de cada *middlebox*.

Na obtenção de informações dos *logs* do *Snort* foi utilizado o *watchdog* 0.8.3 (WATCHDOG DOCUMENTATION, 2015), uma ferramenta para monitoramento de sistemas de arquivos. Com essa ferramenta, é possível capturar eventos gerados pelo sistema operacional ao criar/modificar/apagar um arquivo. O *watchdog* permite o desenvolvimento de aplicações que respondam a esses eventos gerados pelo sistema operacional. Dessa forma, é possível implementar uma aplicação que execute uma dada função apenas quando um determinado arquivo é modificado. Uma vantagem dessa ferramenta é que ela permite a uma aplicação monitorar um arquivo sem precisar checá-lo periodicamente, contribuindo para a melhora de desempenho dessa aplicação.

No agente que monitora os *logs* do *Snort*, foi implementada uma função usando o *watchdog* que só é executada quanto ocorre alguma modificação no arquivo de *logs*. Foi utilizada uma ferramenta específica para a leitura de arquivos de sistemas IPS, chamada de *idstools* 0.5.1 (IDSTOOLS, 2015), e leitura do arquivo de *logs* do *Snort*. Essa ferramenta faz a conversão direta das informações presentes no arquivo de *logs* para o formato JSON, que facilita na transferência dessas informações para o controlador. Portanto, o agente usa as ferramentas *watchdog* para monitorar o arquivo de *logs* e *idstools* para ler esse arquivo.

Para tratar os *logs* do *HAproxy*, também foi utilizado o *watchdog*. Sempre que ocorre modificação no arquivo de *logs* monitorado pelo *watchdog*, uma função do agente que gerencia o *HAproxy* é chamada e são extraídas as informações sobre a política aplicada. Como o *watchdog* permite que essa função seja executada apenas quando o sistema operacional gerar um evento específico para modificação de arquivo, a aplicação que monitora os *logs* do *HAproxy* é executada somente quando o arquivo por ela monitorado é modificado. Nesse agente, a leitura do arquivo de *logs* é feita utilizando funções padrão da linguagem *python*.

Algoritmo 5.3: Agente para configuração de políticas em um firewall

```
1   from flask import Flask
2   from flask import request, redirect
3   import iptc
4   import commands
5   import os
6   import requests
7   import pycurl,json
8   def fw(rule,chain):
9   chain.insert_rule(rule)
10
11  app = Flask(__name__)
12
13  @app.route('/InsertPolicyFW', methods = ['POST'])
14  def insertPolicy():
15  rule = iptc.Rule()
16  rule.src = str(request.json['src_ip'])
17  rule.dst = str(request.json['dst_ip'])
18  rule.protocol = str(request.json['protocol'])
19  match = rule.create_match(str(request.json['protocol']))
20  match.dport = str(request.json['dst_port'])
21  rule.add_match(match)
22  rule.target = iptc.Target(rule, str(request.json['action']
23  chain = iptc.Chain(iptc.Table(iptc.Table.FILTER), "FORWARD
24  fw(rule,chain)
25  return ""
26  if __name__ == '__main__':
27  app.run(host='0.0.0.0', port=5000)
```

Tabela 5.1: Informações extraídas dos *middleboxes*

Middlebox	informações
Firewall	IP de origem e destino; porta de destino; protocolo da camada de transporte.
IPS	IP e porta de origem e destino; status.
Balanceador de carga	IP de origem e destino; portas de origem e destino; IP e porta do balanceador de carga usados na nova requisição[2].

5.6 *Middleboxes*

Como foram implementados agentes, *middleboxes* não sofreram qualquer modificação. Entretanto, como esses dispositivos são capazes de registrar uma grande quantidade de informações em seus *logs*, foi necessário configurá-los para restringir a quantidade de informações registradas para simplificar a extração dessas informações por parte dos agentes. Tanto no *Snort* quanto no *HAproxy*, apenas as informações que serão enviadas ao controlador são registradas nos *logs*.

O *Snort* foi configurado para trabalhar como IPS, bloqueando tráfego considerado malicioso. Para executar o *Snort* como IPS, foi necessário compilar e instalar a biblioteca *Data Acquisition* (DAQ), e compilar o código fonte do *Snort*.

5.7 Mininet

Por padrão, o *Mininet* não possui conexão com a rede local ou *Internet*. Conexões originadas no controlador com destino a *hosts* no ambiente *Mininet* são identificadas como tendo origem em *hosts* externos à rede *Mininet* e são bloqueadas. Foi necessário configurar uma interface virtual entre a rede *Mininet* e a rede local para que o controlador fosse capaz de enviar políticas para os *middleboxes*. Essa interface virtual é configurada através de uma interface anexada a um *switch* da rede *Mininet*, possibilitando que *hosts Mininet* acessem *hosts* que estejam fora do ambiente de emulação e vice-versa. A máquina local passa a ser percebida pelo *Mininet* como uma máquina pertencente à rede emulada, tornando possível a comunicação com *hosts* na rede emulada. Essa interface foi anexada ao *switch* S2 da topologia apresentada na Figura 6.1, com o endereço IP 192.168.3.254.

5.8 Considerações finais

Nesta seção, foram descritas as ferramentas e tecnologias utilizadas na construção do protótipo com base na arquitetura proposta nesta dissertação. Foi descrito também como os elementos da arquitetura SDN como aplicações e controlador foram implementados para configurar dinamicamente políticas em *middleboxes*. Além disso, foi mostrado como JSON e REST atuam na transmissão das políticas entre controlador e *middleboxes*. Por fim, foram descritas as configurações realizadas nos *middleboxes* para que adicionem a seus *logs* apenas as

[2]Essas informações são usadas para identificar as modificações geradas por esse *middlebox*

informações enviadas ao controlador e o ambiente de emulação do *Mininet* foi configurado para permitir a comunicação do controlador com os *middleboxes* por ele gerenciados.

6
Avaliação do protótipo

Nesta seção, são apresentados a metodologia e os experimentos utilizados na avaliação do protótipo. São descritas as ferramentas utilizadas na construção da rede para a avaliação, as métricas de desempenho usadas e a análise dos resultados. Por fim, são apresentadas as ameaças à validade dos resultados e medidas tomadas neste trabalho de mestrado para reduzir essas ameaças.

6.1 Metodologia

Quanto ao método de pesquisa, foi escolhido o experimento controlado. Em um experimento controlado são manipuladas variáveis independentes, enquanto observa-se o impacto que as mudanças nestas variáveis causam na(s) variável(eis) dependente(s). O impacto sobre variáveis dependentes é observado a fim de testar as hipóteses levantadas durante a pesquisa (EASTERBROOK et al., 2008). Abaixo algumas características desta pesquisa que justificam a utilização do método de experimento controlado:

- Controle sobre variáveis;
- Possibilidade de aleatorização;
- Replicação possível e de baixo custo.

A validação desta pesquisa está diretamente relacionada ao uso de experimentos em laboratório, estabelecidos no método de experimento controlado, pois há controle sobre as variáveis, é possível aleatorizar e é fácil de replicar os experimentos com baixo custo.

6.1.1 Projeto dos experimentos

De acordo com (PFLEEGER, 1995) as seguintes etapas são necessárias para o planejamento de um experimento:

- Concepção;
- Projeto;
- Preparação;
- Execução;

- Análise;
- Disseminação e tomada de decisão.

Nas seções seguintes serão descritos cada um desses passos para este trabalho.

6.1.1.1 Concepção

Na fase de concepção, foram definidos o objeto de estudo e os objetivos do experimento. Características como alto controle sobre as variáveis, replicação de alto nível e de baixo custo indicam que o experimento controlado é o método mais adequado nesta pesquisa. O objetivo desta pesquisa é avaliar a utilização da arquitetura SDN na aplicação dinâmica das políticas de *middleboxes*. As métricas de desempenho utilizadas para avaliar o protótipo são:

- Atraso: Especifica quanto tempo os dados levam para irem da origem até o destino. Em chamadas VoIP, um atraso muito alto pode gerar problemas na conversação, como integrantes interrompendo um ao outro.
- Perda de pacotes: Ocorre quando um ou mais pacotes são perdidos. É de suma importância que a perda de pacotes seja a menor possível. Em aplicações de transmissão de voz e vídeo em redes comutadas por pacotes, essa é uma métrica de qualidade de serviço (do inglês, *Quality of service* (QoS)) fundamental, uma vez que altas taxas de perda de pacotes inviabilizam o funcionamento da aplicação.
- *Jitter*: Variação do atraso na entrega dos pacotes em uma rede. O ouvido humano é altamente intolerante a lacunas de áudio de curto prazo, portanto, o *jitter* deve ser minimizado (KARAPANTAZIS; PAVLIDOU, 2009).

6.1.1.2 Projeto

Na fase de projeto são definidas as hipóteses nula e alternativa:

H_0 : A arquitetura habilita aplicação dinâmica de políticas

H_1 : A arquitetura não habilita aplicação dinâmica de políticas

Nesta dissertação, é proposta uma arquitetura para aplicação dinâmica de políticas de *middlebox*. Portanto, a hipótese levantada durante esta pesquisa e avaliada nesta seção é que a arquitetura proposta é capaz de aplicar dinamicamente políticas de *middlebox*. Logo, a hipótese citada acima levantada durante esta pesquisa é representada por H_0 e seu complemento é representado por H_1 que afirma que a arquitetura não habilita aplicação dinâmica de políticas.

6.1.1.3 Preparação

Na preparação dos experimentos foram definidas as ferramentas utilizadas na construção do ambiente de testes e características da rede usada na avaliação. Na avaliação do protótipo foram utilizadas as ferramentas a seguir:

- *Mininet* 2.2.1: foi escolhido para uso na avaliação por ser amplamente utilizado em experimentos com SDN. O *Mininet* é um *software* de emulação que permite a construção de redes SDN para experimentação. Não foi possível realizar os experimentos em uma rede física devido ao custo para construção de tal rede.

- *OpenVswitch* 2.3: *openVswitch* é um *switch OpenFlow* implementado em *software*. O *OpenVswitch* implementa as mesmas funções que *switches OpenFlow* físicos, com a vantagem de receber suporte a novas versões do *OpenFlow* com maior rapidez. Ao tempo de escrita deste trabalho, a versão 2.3 era a mais recente a possuir suporte completo ao *OpenFlow* 1.3.

- OpenFlow 1.3: essa versão possui suporte aos protocolos MPLS e IPv6, multitabelas e melhorias nas operações com múltiplos controladores. Apesar de existirem versões mais recentes como a 1.4 e 1.5, a versão 1.3 foi escolhida por ser mais popular.

- Ryu Framework 3.2: o *Ryu* foi usado por ser um dos únicos controladores com suporte ao *OpenFlow 1.3* no início desta pesquisa e por possuir uma comunidade muito ativa que o mantém sempre atualizado com novas versões *OpenFlow*. O *Ryu* é implementado em *python*, uma linguagem poderosa e que possui uma sintaxe muito simples, que facilita a compreensão e o desenvolvimento de aplicações no *Ryu*.

- SIPp: o SIP *performance* (SIPp) (SIP PERFORMANCE, 2015) é um *software* para avaliação de sistemas VoIP. Com SIPp pode-se gerar chamadas para uma central VoIP, medir a taxa de chamadas bem sucedidas e o atraso das chamadas.

- *HTTPing*: *HTTPing* (HTTPING, 2015) é uma ferramenta para criação de requisições HTTP, atuando semelhante ao comando *ping*. Essa ferramenta foi utilizada para medir o atraso e a perda de pacotes de requisições HTTP.

- R: é uma linguagem de programação e ambiente *open source* para desenvolvimento de análise estatística sofisticada. R suporta a execução de uma grande variedade de testes estatísticos e, construção de gráficos. O R pode ser estendido por intermédio de pacotes que adicionam suporte a novos testes.

- *Wireshark* e *tshark*: o *wireshark* (OREBAUGH; RAMIREZ; BEALE, 2006) é um *software* popular para captura e análise de tráfego. O *wireshark* foi utilizado nos experimentos para capturar o tráfego da rede e armazená-lo em um arquivo. Os dados capturados pelo *wireshark* são armazenados em um arquivo. O *Tshark* (TSHARK, 2015) foi utilizado para extrair informações sobre *jitter* e perda de pacotes de chamadas VoIP, a partir do arquivo gerado pelo *wireshark*. Com o *Tshark*, essas informações foram extraídas e enviadas para um arquivo *Comma-Separated Values* (CSV), posteriormente lido pelo R para obter os dados usados nos testes estatísticos.

- *Middleboxes* avaliados: os *middleboxes Iptables*, *Snort* e *HAproxy* foram escolhidos para avaliação neste trabalho por serem largamente utilizados em ambientes de produção, que nos permite generalizar os resultados para outros contextos.

- *Asterisk* (VAN MEGGELEN; MADSEN; SMITH, 2007): uma solução *open source* para implantação de sistemas VoIP. Com *asterisk*, uma máquina pode ser transformada em um servidor de comunicação VoIP. Nos experimentos o *asterisk* foi utilizado para implementar uma central de telefonia VoIP em uma máquina do ambiente de emulação *Mininet*, possibilitando o estabelecimento de chamadas para essa máquina.

- *Apache web server*: é o processo servidor que processa solicitações HTTP dos clientes e gera as respostas. O *Apache* é o servidor web mais popular no mundo.

- RTP e SIP: o protocolo SIP é empregado na criação, modificação e finalização de chamadas VoIP. RTP é um protocolo para aplicações multimídia de tempo real, utilizado no transporte de áudio e vídeo. No RTP as portas são selecionadas aleatoriamente.

6.1.1.4 Execução

Na fase de execução, os experimentos foram executados de acordo com o planejamento feito na Seção 6.2. Nesta fase, foram obtidos os resultados posteriormente analisados com os procedimentos descritos na próxima subseção.

6.1.1.5 Análise

Foram realizados testes de hipóteses para auxiliarem as conclusões sobre os resultados. Testes de hipóteses além de serem amplamente utilizados na literatura, empregam intervalos de confiança na análise dos dados (MONTGOMERY, 2014). Portanto, intervalos de confiança também são usados ao se realizarem testes de hipóteses.

Primeiro é preciso identificar se os dados seguem uma distribuição Normal. Essa identificação é essencial, visto que em dados normalmente distribuídos, são aplicados testes paramétricos; e são aplicados testes não paramétricos para dados que não seguem uma distribuição Normal. Portanto, o tipo de teste estatístico depende da distribuição dos dados.

Neste trabalho, foi usado o teste de aderência de *Kolmogorov-Smirnov*, amplamente utilizado na literatura, para identificar se os dados seguem uma distribuição Normal. Toda a análise dos dados foi realizada no *software* R 3.1.1 para *Ubuntu Linux*.

Os testes WSRT e WRST são os mais comuns para análise de dados não paramétricos (MONTGOMERY, 2014). As amostras possuem o tamanho de 20.000[1]. Foram realizados testes de hipóteses para auxiliarem na tomada de decisões sobre os resultados obtidos. Abaixo estão as hipóteses adotadas neste trabalho:

$$H_0 : \text{O protótipo não impacta no desempenho da rede}$$
$$H_1 : \text{O protótipo impacta no desempenho da rede}$$

Testes não paramétricos empregam a mediana como medida de tendência central ao invés da média que é usada em testes paramétricos. Portanto, as hipóteses acima foram expressas no teste WSRT pelas hipóteses:

$$H_0 : \tilde{\mu_0} = \tilde{\mu_1}$$
$$H_1 : \tilde{\mu_0} \neq \tilde{\mu_1}$$

em que $\tilde{\mu_0}$ é a mediana da amostra sem o protótipo, e $\tilde{\mu_1}$ a mediana da amostra com o protótipo. Portanto, se a mediana das duas amostras não forem estatisticamente diferentes, o protótipo não causa impacto o desempenho da rede. Em todos os testes estatísticos foram utilizados nível de confiança de 95% e $\alpha = 0,05$.

[1] Esse valor é suficientemente alto para avaliar o protótipo com base em (YANG; LEE; KO, 2008) e (ZHOU; HUANG; MO, 2005) que utilizam 100 e 1080 chamadas, respectivamente, para avaliar suas propostas.

6.1.1.6 Disseminação e tomada de decisão

Ao final da análise dos resultados, chega-se a uma conclusão sobre como as variáveis avaliadas afetaram os resultados. É importante documentar os procedimentos adotados na avaliação, de forma que outros pesquisadores sejam capazes de replicar os experimentos e confirmar as conclusões em um cenário semelhante. Aspectos chave empregados na avaliação desta pesquisa como objetos, variáveis, configuração do ambiente experimental e resultados são descritos nas próximas seções. As hipóteses são apresentadas na Seção anterior e as ferramentas empregadas na construção do ambiente experimental foram descritas na Subseção 6.1.1.3.

6.1.1.7 Formulação do problema

Ao comparar o uso com o não uso de um tratamento, deve-se estabelecer os objetos de controle e experimentais. O objeto de controle recebe o tratamento, já o objeto experimental não recebe o tratamento. Nesta pesquisa, o objeto de controle foi nomeado como 'políticas dinâmicas', cenário em que o protótipo está sendo avaliado, com a aplicação do tratamento proposto neste trabalho. O objeto experimental foi nomeado como 'políticas estáticas', visto que a configuração das políticas é completamente manual, pois o protótipo não está em execução.

Nos experimentos desta pesquisa, as variáveis dependentes são: as quantidades de chamadas VoIP, requisições HTTP e *middleboxes* na rede. Variáveis independentes são o atraso, *jitter* e perda de pacotes, além das políticas aplicadas pelos *middleboxes*.

6.2 Ambiente experimental

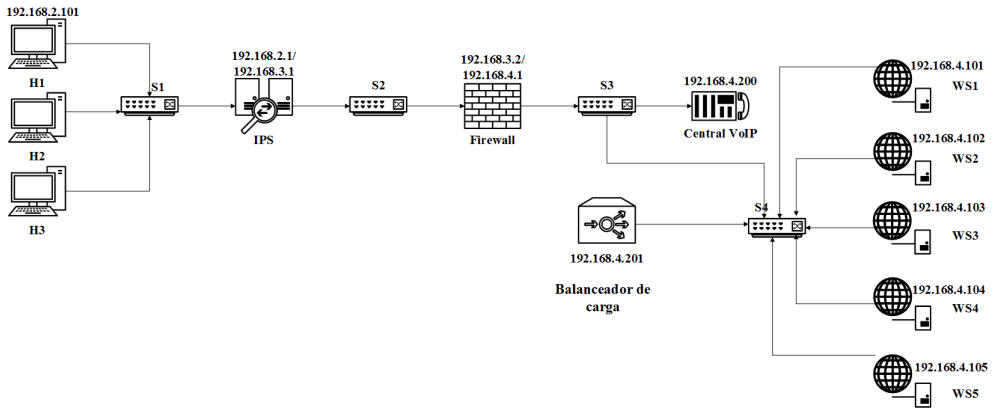
Nesta seção, é descrito o ambiente experimental utilizado na avaliação do protótipo. O protótipo foi submetido a quatro avaliações parciais, todas na mesma topologia, com variações nos serviços e *middleboxes* utilizados. A topologia usada nos experimentos descritos resumidamente na Tabela 6.1 é apresentada na Figura 6.1. Além disso, esses testes para avaliar o protótipo são descritos em detalhes nas próximas subseções. Na Subseção 6.2.3 é apresentado um teste conduzido para avaliar o protótipo em uma rede relativamente maior que a apresentada na Figura 6.1. O objetivo desse teste é avaliar o desempenho do protótipo em redes maiores considerando a quantidade de *switches*.

Os *middleboxes* foram posicionados na rede seguindo a distribuição que normalmente ocorre em ambientes de produção, em que são localizados de forma que o tráfego obrigatoriamente os atravesse. Foi considerado o uso de múltiplas instâncias do mesmo *middlebox*. Contudo cada dispositivo foi usado para oferecer um serviço: *firewall* para controle, IPS para análise do tráfego a fim de prover segurança e balanceador de carga para melhoria de desempenho. Portanto, múltiplas instâncias seriam ineficientes ao ofertar serviços redundantes. Em cada avaliação parcial, o(s) *middlebox*(*es*) que não estão sendo avaliados foram configurados para atuar como roteadores encaminhando o tráfego. Nas seções seguintes, são descritos os testes com *firewall*, IPS, balanceador de carga e múltiplos controladores.

6.2.1 Caso de uso: comunicação VoIP atravessando um *firewall*

VoIP utiliza o RTP no transporte dos dados de voz e vídeo. O problema com *firewalls* ocorre porque o RTP não utiliza portas predefinidas; elas são escolhidas aleatoriamente de um conjunto de portas. Geralmente, *firewalls* são configurados para liberar apenas portas de

Figura 6.1: Topologia usada na avaliação do protótipo



Fonte: elaborada pelo autor.

Tabela 6.1: Descrição dos testes realizados na avaliação do protótipo.

Teste	*Middleboxes*	Descrição
T1	*Firewall*	O *firewall* é configurado para bloquear chamadas VoIP e o protótipo configura políticas nesse *firewall* para que permita a comunicação VoIP.
T2	Firewall e IPS	O *firewall* possui políticas para bloquear as chamadas VoIP e o IPS para analisar o tráfego VoIP que o atravessa. O protótipo configura políticas no *firewall* para que libere as chamadas VoIP e obtém as políticas aplicadas pelo IPS.
T3	Balanceador de carga e IPS	O balanceador de carga possui políticas para balancear as requisições HTTP por ele recebidas e o IPS possui políticas para analisar essas requisições HTTP. O protótipo obtém as políticas aplicadas pelo balanceador de carga e pelo IPS.
T4	*Firewall* e IPS	O *firewall* é configurado para bloquear chamadas VoIP e o IPS para analisar o tráfego dessas chamadas. É avaliada a capacidade de a arquitetura escalar usando múltiplos controladores. São utilizados três controladores que dividem entre si a carga de trabalho da arquitetura proposta nesta dissertação.

serviços conhecidos e portas específicas, enquanto as demais portas são bloqueadas por questão de segurança. Portas liberadas no *firewall* representam a superfície de ataque – o conjunto de possíveis brechas de um sistema passíveis de serem exploradas por atacantes – à rede. Quanto menor essa superfície, isto é quanto menor o número de portas liberadas no *firewall*, melhor. O conjunto de portas disponíveis para escolha do RTP geralmente é bloqueado nos *firewalls*.

Assim, dados RTP são descartados pelo *firewall*, impossibilitando a comunicação VoIP.

Várias soluções foram propostas para mitigar problemas causados por *middleboxes* em comunicação VoIP. Foram propostos *Application-Level Gateways* (ALGs), *Middlebox Communications (MIDCOM) Protocol* (STIEMERLING; QUITTEK; TAYLOR, 2008), dentre outros. De forma geral, essas soluções apesar de resolverem um problema, geram vários outros ao tornarem a rede mais complexa, difícil de gerenciar e suscetível a erros (KHLIFI; GREGOIRE; PHILLIPS, 2006).

No teste desta seção, foi avaliado o protótipo na presença de um *firewall* bloqueando a comunicação VoIP. O *firewall iptables* utilizado na avaliação exige a especificação do protocolo de transporte ao definir uma política contendo informações sobre portas. No *firewall*, foi configurada uma política na *chain forward*[2] para bloquear tráfego UDP com porta de destino superior a 1023. Com essa política no *firewall*, requisições VoIP são descartadas, impedindo a realização de chamadas, semelhante ao que ocorre em ambientes de produção. Ainda no *firewall*, foi configurada uma política na *chain INPUT*[3] para liberar conexões destinadas à porta 5.000, utilizada pelo agente para receber políticas do controlador. Esse agente é executado paralelamente ao *iptables* na máquina 'Firewall' da Figura 6.1. No *iptables*, as políticas são armazenadas em uma lista. A primeira política que combine com os dados recebidos é aplicada e as demais políticas da lista são ignoradas. Políticas configuradas pelo controlador são adicionadas ao início da lista, de forma que, quando o *iptables* encontra a política para liberar a chamada, ela é aplicada e as demais são ignoradas, incluindo a que instrui o *firewall* a descartar tráfego com porta destino superior a 1023.

No *asterisk*, foi configurado um usuário para receber as chamadas VoIP. O *SIPp* foi executado no *host* H1, gerando 50 chamadas simultâneas para o usuário configurado no *asterisk*. Acima de 50 chamadas simultâneas o SIPp apresentou instabilidade, descartando chamadas mesmo quando o protótipo não estava sendo avaliado. Foi verificado o uso de memória e *Central Processing Unit* (CPU), que poderiam ser a causa desse descarte de chamadas, mas ambos os recursos não apresentaram sobrecarga e, a VM não apresentou qualquer problema com lentidão. Aparentemente, esse problema foi causado pelo próprio SIPp. No total foram criadas 20.000 chamadas.

O *stream* de áudio RTP é gerado a partir do arquivo de captura *g711a.pcap* (G711A, 2015) padrão do SIPp, utilizando o *codec* de áudio G.711 padrão do *International Telecommunication Union - Telecommunication Standardization Sector* (ITU-T), com cada chamada tendo uma duração de 1 segundo. O *codec* é definido pelo arquivo de captura. Cada chamada tem duração de 1 segundo porque o SIPp foi incapaz de criar chamadas com duração maior, mesmo quando o protótipo não estava sendo avaliado. A duração das chamadas não influencia no desempenho do protótipo, já que ele atua apenas na liberação das chamadas. O *Wireshark* realiza a captura dos pacotes do *host* H1. Ao final do experimento, o *Tshark* é usado para extrair do arquivo de captura criado pelo *Wireshark* informações sobre *jitter* e perda de pacote das chamadas e exportá-las para um arquivo CSV. O SIPp mede o atraso de cada chamada e o armazena em um arquivo CSV.

As Figuras 6.2 a 6.5 apresentam as telas do protótipo em execução. Na Figura 6.2, são apresentadas informações sobre as chamadas a partir do tráfego capturado pelo *Wireshark*. A última coluna dessa Figura contém o *jitter* médio de cada chamada com duração de 1 segundo, usado na análise dos resultados. A Figura 6.3 mostra dados obtidos pelo controlador *Ryu* sobre as chamadas. Na Figura 6.4, são mostradas as requisições usadas pelo controlador para enviar

[2]Local onde são armazenadas as políticas que tratam o tráfego que atravessa o *firewall*

[3]Local onde políticas que tratam o tráfego destinado ao *firewall* são armazenadas

políticas ao agente que gerencia o *firewall iptables*. Na Figura 6.5, são apresentadas as políticas recebidas do controlador e aplicadas pelo *iptables*.

Figura 6.2: Chamadas capturadas pelo *wireshark*

Src addr	Src port	Dst addr	Dst port	SSRC	Payload	Packets	Lost	Max Delta (ms)	Max Jitter (ms)	Mean Jitter (ms)
192.168.2.101	6000	192.168.4.200	19808	0xDEE0EE8F	g711A	67	0 (0.0%)	36.14	1.16	0.67
192.168.2.101	6004	192.168.4.200	18406	0xDEE0EE8F	g711A	67	0 (0.0%)	33.17	1.02	0.78
192.168.2.101	6008	192.168.4.200	13882	0xDEE0EE8F	g711A	67	0 (0.0%)	32.56	0.85	0.67
192.168.2.101	6012	192.168.4.200	18962	0xDEE0EE8F	g711A	67	0 (0.0%)	31.77	0.92	0.74
192.168.2.101	6016	192.168.4.200	19612	0xDEE0EE8F	g711A	67	0 (0.0%)	33.91	1.36	0.87
192.168.2.101	6020	192.168.4.200	12730	0xDEE0EE8F	g711A	67	0 (0.0%)	33.96	1.01	0.72
192.168.2.101	6024	192.168.4.200	12496	0xDEE0EE8F	g711A	67	0 (0.0%)	33.29	1.19	0.80
192.168.2.101	6028	192.168.4.200	11480	0xDEE0EE8F	g711A	67	0 (0.0%)	35.10	1.19	0.83
192.168.2.101	6032	192.168.4.200	10350	0xDEE0EE8F	g711A	67	0 (0.0%)	35.30	1.26	0.90
192.168.2.101	6036	192.168.4.200	18120	0xDEE0EE8F	g711A	67	0 (0.0%)	33.35	1.26	0.83
192.168.2.101	6040	192.168.4.200	18340	0xDEE0EE8F	g711A	67	0 (0.0%)	34.80	1.16	0.87
192.168.2.101	6044	192.168.4.200	18696	0xDEE0EE8F	g711A	68	0 (0.0%)	33.33	1.25	0.78
192.168.2.101	6048	192.168.4.200	13752	0xDEE0EE8F	g711A	67	0 (0.0%)	35.93	1.07	0.69
192.168.2.101	6052	192.168.4.200	19406	0xDEE0EE8F	g711A	67	0 (0.0%)	32.55	0.68	0.59
192.168.2.101	6056	192.168.4.200	11556	0xDEE0EE8F	g711A	67	0 (0.0%)	33.54	1.32	0.81
192.168.2.101	6060	192.168.4.200	11952	0xDEE0EE8F	g711A	68	0 (0.0%)	34.35	1.05	0.77
192.168.2.101	6064	192.168.4.200	14674	0xDEE0EE8F	g711A	68	0 (0.0%)	31.65	0.64	0.60
192.168.2.101	6068	192.168.4.200	12012	0xDEE0EE8F	g711A	67	0 (0.0%)	32.85	0.80	0.76
192.168.2.101	6072	192.168.4.200	11176	0xDEE0EE8F	g711A	67	0 (0.0%)	33.74	1.31	0.88
192.168.2.101	6076	192.168.4.200	14168	0xDEE0EE8F	g711A	67	0 (0.0%)	41.36	3.03	1.44
192.168.2.101	6080	192.168.4.200	11280	0xDEE0EE8F	g711A	67	0 (0.0%)	33.53	0.80	0.77

Fonte: elaborada pelo autor.

Figura 6.3: Informações das chamadas obtidas pelo controlador

```
root@mininet-vm:/home/mininet/ryu# ./bin/ryu-manager ryu/app/iptables.py
loading app ryu/app/iptables.py
loading app ryu.controller.ofp_handler
instantiating app ryu/app/iptables.py of Fw
instantiating app ryu.controller.ofp_handler of OFPHandler
src: 192.168.2.101 --> dst: 192.168.4.200 on port: 19808
src: 192.168.2.101 --> dst: 192.168.4.200 on port: 18406
src: 192.168.2.101 --> dst: 192.168.4.200 on port: 13882
src: 192.168.2.101 --> dst: 192.168.4.200 on port: 18962
src: 192.168.2.101 --> dst: 192.168.4.200 on port: 19612
src: 192.168.2.101 --> dst: 192.168.4.200 on port: 12730
src: 192.168.2.101 --> dst: 192.168.4.200 on port: 12496
src: 192.168.2.101 --> dst: 192.168.4.200 on port: 11480
src: 192.168.2.101 --> dst: 192.168.4.200 on port: 10350
src: 192.168.2.101 --> dst: 192.168.4.200 on port: 18120
src: 192.168.2.101 --> dst: 192.168.4.200 on port: 18340
src: 192.168.2.101 --> dst: 192.168.4.200 on port: 18696
src: 192.168.2.101 --> dst: 192.168.4.200 on port: 13752
src: 192.168.2.101 --> dst: 192.168.4.200 on port: 19406
src: 192.168.2.101 --> dst: 192.168.4.200 on port: 11556
src: 192.168.2.101 --> dst: 192.168.4.200 on port: 11952
src: 192.168.2.101 --> dst: 192.168.4.200 on port: 14674
src: 192.168.2.101 --> dst: 192.168.4.200 on port: 12012
src: 192.168.2.101 --> dst: 192.168.4.200 on port: 11176
src: 192.168.2.101 --> dst: 192.168.4.200 on port: 14168
src: 192.168.2.101 --> dst: 192.168.4.200 on port: 11280
```

Fonte: elaborada pelo autor.

Figura 6.4: Requisições do controlador recebidas pelo agente do *iptables*

```
root@mininet-vm:/home/mininet/firewall# python ipt_agent.py
 * Running on http://192.168.3.2:5000/
192.168.3.254 - - [09/Jan/2016 21:22:02] "POST /InsertPolicyFW HTTP/1.1" 200 -
192.168.3.254 - - [09/Jan/2016 21:22:02] "POST /InsertPolicyFW HTTP/1.1" 200 -
192.168.3.254 - - [09/Jan/2016 21:22:02] "POST /InsertPolicyFW HTTP/1.1" 200 -
192.168.3.254 - - [09/Jan/2016 21:22:02] "POST /InsertPolicyFW HTTP/1.1" 200 -
192.168.3.254 - - [09/Jan/2016 21:22:02] "POST /InsertPolicyFW HTTP/1.1" 200 -
192.168.3.254 - - [09/Jan/2016 21:22:02] "POST /InsertPolicyFW HTTP/1.1" 200 -
192.168.3.254 - - [09/Jan/2016 21:22:02] "POST /InsertPolicyFW HTTP/1.1" 200 -
192.168.3.254 - - [09/Jan/2016 21:22:02] "POST /InsertPolicyFW HTTP/1.1" 200 -
192.168.3.254 - - [09/Jan/2016 21:22:03] "POST /InsertPolicyFW HTTP/1.1" 200 -
192.168.3.254 - - [09/Jan/2016 21:22:03] "POST /InsertPolicyFW HTTP/1.1" 200 -
192.168.3.254 - - [09/Jan/2016 21:22:03] "POST /InsertPolicyFW HTTP/1.1" 200 -
192.168.3.254 - - [09/Jan/2016 21:22:03] "POST /InsertPolicyFW HTTP/1.1" 200 -
192.168.3.254 - - [09/Jan/2016 21:22:03] "POST /InsertPolicyFW HTTP/1.1" 200 -
192.168.3.254 - - [09/Jan/2016 21:22:03] "POST /InsertPolicyFW HTTP/1.1" 200 -
192.168.3.254 - - [09/Jan/2016 21:22:03] "POST /InsertPolicyFW HTTP/1.1" 200 -
192.168.3.254 - - [09/Jan/2016 21:22:03] "POST /InsertPolicyFW HTTP/1.1" 200 -
192.168.3.254 - - [09/Jan/2016 21:22:03] "POST /InsertPolicyFW HTTP/1.1" 200 -
192.168.3.254 - - [09/Jan/2016 21:22:03] "POST /InsertPolicyFW HTTP/1.1" 200 -
192.168.3.254 - - [09/Jan/2016 21:22:04] "POST /InsertPolicyFW HTTP/1.1" 200 -
192.168.3.254 - - [09/Jan/2016 21:22:04] "POST /InsertPolicyFW HTTP/1.1" 200 -
192.168.3.254 - - [09/Jan/2016 21:22:04] "POST /InsertPolicyFW HTTP/1.1" 200 -
192.168.3.254 - - [09/Jan/2016 21:22:04] "POST /InsertPolicyFW HTTP/1.1" 200 -
192.168.3.254 - - [09/Jan/2016 21:22:04] "POST /InsertPolicyFW HTTP/1.1" 200 -
```

Fonte: elaborada pelo autor.

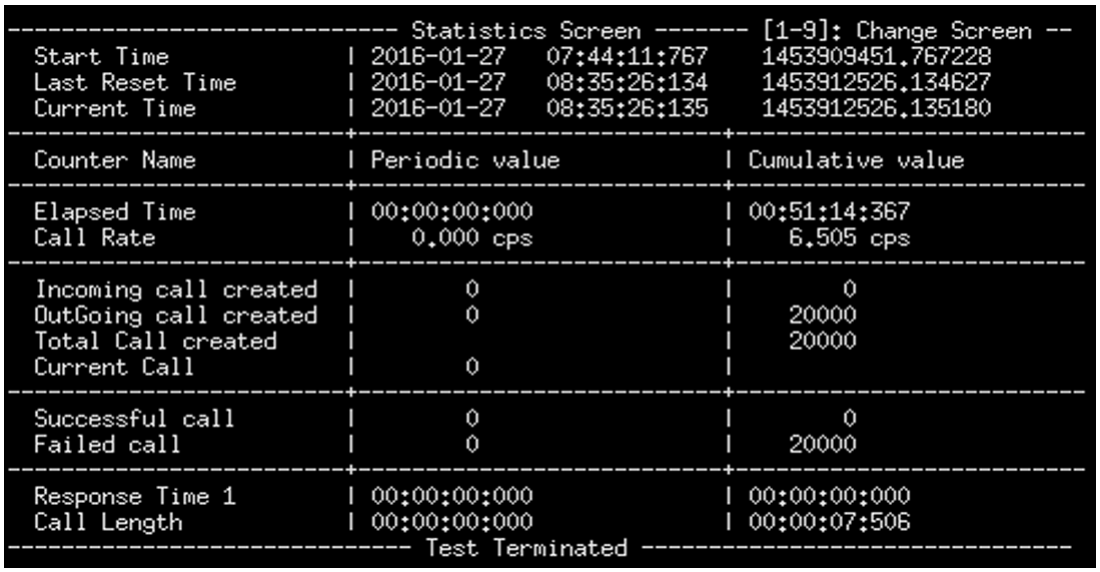
Figura 6.5: Políticas configuradas no *iptables*

```
root@mininet-vm:~# iptables -L
Chain INPUT (policy ACCEPT)
target     prot opt source               destination
ACCEPT     tcp  --  192.168.3.254        192.168.3.2          tcp dpt:5000

Chain FORWARD (policy ACCEPT)
target     prot opt source               destination
ACCEPT     udp  --  192.168.2.101        192.168.4.200        udp dpt:11280 udp dpt:11280
ACCEPT     udp  --  192.168.2.101        192.168.4.200        udp dpt:14168 udp dpt:14168
ACCEPT     udp  --  192.168.2.101        192.168.4.200        udp dpt:11176 udp dpt:11176
ACCEPT     udp  --  192.168.2.101        192.168.4.200        udp dpt:12012 udp dpt:12012
ACCEPT     udp  --  192.168.2.101        192.168.4.200        udp dpt:14674 udp dpt:14674
ACCEPT     udp  --  192.168.2.101        192.168.4.200        udp dpt:11952 udp dpt:11952
ACCEPT     udp  --  192.168.2.101        192.168.4.200        udp dpt:11556 udp dpt:11556
ACCEPT     udp  --  192.168.2.101        192.168.4.200        udp dpt:19406 udp dpt:19406
ACCEPT     udp  --  192.168.2.101        192.168.4.200        udp dpt:13752 udp dpt:13752
ACCEPT     udp  --  192.168.2.101        192.168.4.200        udp dpt:18696 udp dpt:18696
ACCEPT     udp  --  192.168.2.101        192.168.4.200        udp dpt:18340 udp dpt:18340
ACCEPT     udp  --  192.168.2.101        192.168.4.200        udp dpt:18120 udp dpt:18120
ACCEPT     udp  --  192.168.2.101        192.168.4.200        udp dpt:10350 udp dpt:10350
ACCEPT     udp  --  192.168.2.101        192.168.4.200        udp dpt:11480 udp dpt:11480
ACCEPT     udp  --  192.168.2.101        192.168.4.200        udp dpt:12496 udp dpt:12496
ACCEPT     udp  --  192.168.2.101        192.168.4.200        udp dpt:12730 udp dpt:12730
ACCEPT     udp  --  192.168.2.101        192.168.4.200        udp dpt:19612 udp dpt:19612
ACCEPT     udp  --  192.168.2.101        192.168.4.200        udp dpt:18962 udp dpt:18962
ACCEPT     udp  --  192.168.2.101        192.168.4.200        udp dpt:13882 udp dpt:13882
ACCEPT     udp  --  192.168.2.101        192.168.4.200        udp dpt:18406 udp dpt:18406
ACCEPT     udp  --  192.168.2.101        192.168.4.200        udp dpt:19808 udp dpt:19808
ACCEPT     udp  --  192.168.2.101        192.168.4.200        udp dpt:sip
ACCEPT     udp  --  192.168.4.200        192.168.2.101        udp dpt:sip
DROP       udp  --  anywhere             anywhere             udp dpts:1024:65535

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
root@mininet-vm:~#
```

Fonte: elaborada pelo autor.

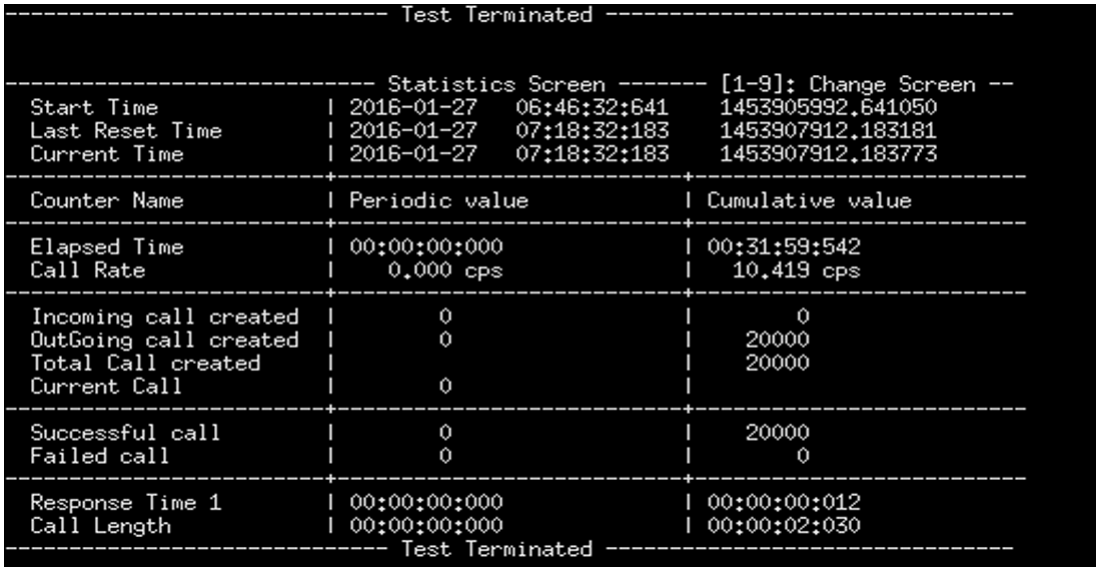
Foi realizado um teste que comprovou que, na configuração acima, o *firewall* realmente bloqueia todas as chamadas. Esse teste foi realizado para garantir que o responsável por permitir a realização das chamadas são as políticas aplicadas pelo *firewall* com auxílio do protótipo. A Figura 6.6 apresenta a tela do *software* SIPp (usado para gerar as chamadas VoIP) com o resultado do teste acima citado em que o *firewall* possui políticas para bloquear a comunicação VoIP e o protótipo não é utilizado. Na avaliação do protótipo, considerando as métricas *Jitter*, atraso e perda de pacotes, foram realizados testes considerando os dois cenários descritos na Seção 6.1.1.7, políticas dinâmicas e políticas estáticas. As Tabelas 6.2 e 6.3 apresentam os resultados do experimento tendo como métricas o atraso e o *jitter*, respectivamente. Como apresentado na Figura 6.7, todas as chamadas foram realizadas com sucesso quando o protótipo é utilizado na aplicação das políticas do *firewall*. A perda de pacotes foi nula em ambos os cenários.

Figura 6.6: Chamadas VoIP bloqueadas por um *firewall*



Fonte: elaborada pelo autor.

Figura 6.7: Chamadas VoIP realizadas com sucesso em cenário em que o protótipo é utilizado



Fonte: elaborada pelo autor.

Tabela 6.2: *Delay* de chamadas VoIP atravessando um *firewall*

	Média	Mediana	Min	Max
Políticas dinâmicas (ms)	5,968	4,391	0,756	542,0
Políticas estáticas (ms)	5,154	4,274	0,834	72,090
Diferença (ms)	0,814	0,117	–	–

Na análise dos dados sobre o atraso de chamadas VoIP, o teste WSRT para dados pareados e n=20.000 retornou um *p-value* de $2,2^{-16}$. Como $2,2^{-16} > \alpha = 0,05$, a hipótese

Tabela 6.3: *Jitter* de chamadas VoIP atravessando um *firewall*

	Média	Mediana	Min	Max
Políticas dinâmicas (ms)	0,5262	0,5	0,18	1,34
Políticas estáticas (ms)	0,5596	0,51	0,3	7,5
Diferença (ms)	-0,0334	-0,01	–	–

nula apresentada na Seção 6.1.1.5 de que as medianas das amostras estatisticamente não são diferentes deve ser rejeitada. Portanto, estatisticamente há impacto no desempenho da rede tendo como métrica o atraso.

Apesar de o protótipo incrementar o atraso, trata-se de um acréscimo pequeno e esperado, devido à comunicação entre controlador e *middleboxes*. Esse resultado mostra que o protótipo é capaz de gerenciar *middleboxes* interferindo minimamente no desempenho da rede.

Na análise dos resultados sobre o *jitter*, com o teste WSRT para dados pareados e n=20.000, foi obtido um *p-value* de $3,81^{-8}$. Portanto, como $3,81^{-8} < \alpha = 0,05$, a hipótese $H_0 : \tilde{\mu_0} = \tilde{\mu_1}$ deve ser rejeitada. Contudo, a hipótese $H_0 : \tilde{\mu_0} = \tilde{\mu_1}$ é usada para testar se o protótipo tem um desempenho estatisticamente igual ao cenário sem o protótipo. Logo, é possível que os dados tenham levado à rejeição desta hipótese nula porque o protótipo apresentou um desempenho suficientemente superior para ser estatisticamente diferente do desempenho de quando o protótipo não está em execução. Assim, foi realizado um segundo teste WSRT para avaliar se o protótipo apresentou um desempenho superior em relação ao cenário em que o protótipo não é executado. Foram utilizadas as seguintes hipóteses:

$$H_0 : \tilde{\mu_0} > \tilde{\mu_1}$$
$$H_1 : \tilde{\mu_0} < \tilde{\mu_1}$$

em que $\tilde{\mu_0}$ é a mediana da amostra sem o protótipo e $\tilde{\mu_1}$ a mediana da amostra com o protótipo. Nesse teste, o *p-value* obtido foi 1, como 1 é maior que 0,05, H_0 não deve ser rejeitada, logo a mediana do *jitter* com o protótipo em execução é estatisticamente menor do que a mediana sem o protótipo. Portanto, estatisticamente não há impacto no desempenho da rede, considerando o *jitter* das chamadas VoIP. Os resultados mostram que o protótipo torna um *firewall* capaz de aplicar corretamente suas políticas, permitindo a comunicação VoIP.

6.2.2 Caso de uso: comunicação VoIP com Firewall e IPS

Neste teste, além da configuração apresentada na Seção 6.2.1 para o *firewall*, foi utilizado um IPS na análise do tráfego VoIP. No IPS, foi configurada uma política que gera um alerta ao identificar dados com destino ao IP da central VoIP. Essa política é adicionada ao arquivo de *logs* sempre que é aplicada. Quando uma política é adicionada no arquivo de *logs*, o agente que monitora esse arquivo, extrai as informações apresentadas na Tabela 5.1, e as envia para uma aplicação responsável por gerenciar o IPS, em execução no controlador. Esse agente é executado paralelamente ao IPS *snort* na máquina 'IPS' apresentada na Figura 6.1. A Figura 6.8 apresenta a tela do protótipo em execução recebendo políticas aplicadas pelo IPS *Snort*. As Tabelas 6.4 e 6.5 apresentam os resultados para atraso e *jitter*, respectivamente.

Figura 6.8: Requisições do agente do IPS recebidas pelo controlador *Ryu*

```
root@mininet-vm:/home/mininet/ryu# ./bin/ryu-manager ryu/app/iptables.py ryu/app/snort.py
loading app ryu/app/iptables.py
loading app ryu/app/snort.py
loading app ryu.controller.ofp_handler
creating context wsgi
instantiating app ryu/app/iptables.py of Fw
instantiating app ryu/app/snort.py of Snort
instantiating app ryu.controller.ofp_handler of OFPHandler
(26510) wsgi starting up on http://0.0.0.0:8080/
(26510) accepted ('192.168.3.1', 38890)
192.168.3.1 - - [10/Jan/2016 00:26:33] "POST /InformCtrIPS HTTP/1.1" 200 115 0.027356
192.168.3.1 - - [10/Jan/2016 00:26:33] "POST /InformCtrIPS HTTP/1.1" 200 115 0.000842
192.168.3.1 - - [10/Jan/2016 00:26:33] "POST /InformCtrIPS HTTP/1.1" 200 115 0.004670
192.168.3.1 - - [10/Jan/2016 00:26:33] "POST /InformCtrIPS HTTP/1.1" 200 115 0.002695
192.168.3.1 - - [10/Jan/2016 00:26:33] "POST /InformCtrIPS HTTP/1.1" 200 115 0.002449
192.168.3.1 - - [10/Jan/2016 00:26:33] "POST /InformCtrIPS HTTP/1.1" 200 115 0.001625
192.168.3.1 - - [10/Jan/2016 00:26:33] "POST /InformCtrIPS HTTP/1.1" 200 115 0.002477
192.168.3.1 - - [10/Jan/2016 00:26:33] "POST /InformCtrIPS HTTP/1.1" 200 115 0.004872
192.168.3.1 - - [10/Jan/2016 00:26:33] "POST /InformCtrIPS HTTP/1.1" 200 115 0.000988
192.168.3.1 - - [10/Jan/2016 00:26:33] "POST /InformCtrIPS HTTP/1.1" 200 115 0.008297
192.168.3.1 - - [10/Jan/2016 00:26:33] "POST /InformCtrIPS HTTP/1.1" 200 115 0.000993
192.168.3.1 - - [10/Jan/2016 00:26:33] "POST /InformCtrIPS HTTP/1.1" 200 115 0.003387
192.168.3.1 - - [10/Jan/2016 00:26:33] "POST /InformCtrIPS HTTP/1.1" 200 115 0.005566
192.168.3.1 - - [10/Jan/2016 00:26:33] "POST /InformCtrIPS HTTP/1.1" 200 115 0.001108
192.168.3.1 - - [10/Jan/2016 00:26:33] "POST /InformCtrIPS HTTP/1.1" 200 115 0.003125
192.168.3.1 - - [10/Jan/2016 00:26:33] "POST /InformCtrIPS HTTP/1.1" 200 115 0.002471
192.168.3.1 - - [10/Jan/2016 00:26:33] "POST /InformCtrIPS HTTP/1.1" 200 115 0.007160
192.168.3.1 - - [10/Jan/2016 00:26:33] "POST /InformCtrIPS HTTP/1.1" 200 115 0.000978
192.168.3.1 - - [10/Jan/2016 00:26:33] "POST /InformCtrIPS HTTP/1.1" 200 115 0.001159
192.168.3.1 - - [10/Jan/2016 00:26:33] "POST /InformCtrIPS HTTP/1.1" 200 115 0.005706
192.168.3.1 - - [10/Jan/2016 00:26:33] "POST /InformCtrIPS HTTP/1.1" 200 115 0.001222
192.168.3.1 - - [10/Jan/2016 00:26:33] "POST /InformCtrIPS HTTP/1.1" 200 115 0.000690
192.168.3.1 - - [10/Jan/2016 00:26:33] "POST /InformCtrIPS HTTP/1.1" 200 115 0.001877
192.168.3.1 - - [10/Jan/2016 00:26:33] "POST /InformCtrIPS HTTP/1.1" 200 115 0.004244
```

Fonte: elaborada pelo autor.

Tabela 6.4: Atraso para chamadas VoIP atravessando *firewall* e IPS

	Média	Mediana	Min	Max
Políticas dinâmicas (ms)	5,577	4,460	1,076	78,58
Políticas estáticas (ms)	6,465	5,236	1,128	130,0
Diferença (ms)	-0,888	-0,776	–	–

Para o atraso, novamente foi aplicado o teste WSRT em amostras com n=20.000. Como o *p-value* foi de $2,2^{-16}$ é menor que 0,05, rejeita-se H_0, levando à conclusão de que as medianas são estatisticamente diferentes. Contundo, o protótipo apresentou um atraso inferior, levando à realização de um segundo teste WSRT para avaliar se o protótipo possui um desempenho superior em relação a quando ele não é executado. Foram usadas as seguintes hipóteses:

$$H_0 : \tilde{\mu_0} > \tilde{\mu_1}$$
$$H_1 : \tilde{\mu_0} < \tilde{\mu_1}$$

Nesse novo teste, o *p-value* obtido foi 1. Como 1 é maior que 0,05, H_0 não deve rejeitada. Assim, a mediana do atraso com o protótipo em execução é estatisticamente menor do que a mediana sem o protótipo. Apesar do tráfego introduzido pela comunicação entre controlador e *middleboxes*, o protótipo interfere minimamente no atraso de chamadas VoIP. Assim, tendo como métrica o atraso, o protótipo não gerou qualquer impacto no desempenho da rede, apresentando um desempenho superior em relação ao cenário em que não é executado.

Na análise do *jitter*, novamente o teste WSRT retornou um *p-value* de $2,2^{-16}$, levando à rejeição de H_0. Assim como na análise do atraso, foi realizado um novo teste WSRT com as

Tabela 6.5: *Jitter* para chamadas VoIP atravessando *firewall* e IPS

	Média	Mediana	Min	Max
Políticas dinâmicas (ms)	0,541	0,49	0,3	5,17
Políticas estáticas (ms)	0,5876	0,54	0,03	4,33
Diferença (ms)	-0,0466	-0,05	–	–

hipóteses:

$$H_0 : \tilde{\mu_0} > \tilde{\mu_1}$$
$$H_1 : \tilde{\mu_0} < \tilde{\mu_1}$$

o *p-value* obtido desse teste foi 1, que, por ser maior que 0,05, conduz à não rejeição da hipótese nula. Portanto, o protótipo não introduz impacto no desempenho da rede, tendo como métrica o *jitter* de chamadas VoIP.

6.2.3 Caso de uso: comunicação VoIP com Firewall e IPS em uma topologia com 51 *switches*

O objetivo deste teste é avaliar o desempenho do protótipo em uma rede relativamente maior em relação a topologia utilizada nos testes anteriores. É fundamental avaliar a capacidade do protótipo gerenciar redes de diferentes escalas.

Para esse teste, a configuração dos *middleboxes* apresentada na subseção anterior foi mantida. A topologia utilizada neste teste é apresentada na Figura 6.9. Essa topologia foi derivada da mostrada na Figura 6.1 e a diferença está na quantidade de *switches*, visto que a topologia anterior possui apenas quatro *switches* enquanto a nova contém 51. Foi utilizada uma rede com 51 *switches* por questões de limitação do *hardware* da máquina disponível para hospedar a VM do *Mininet* usada nos testes.

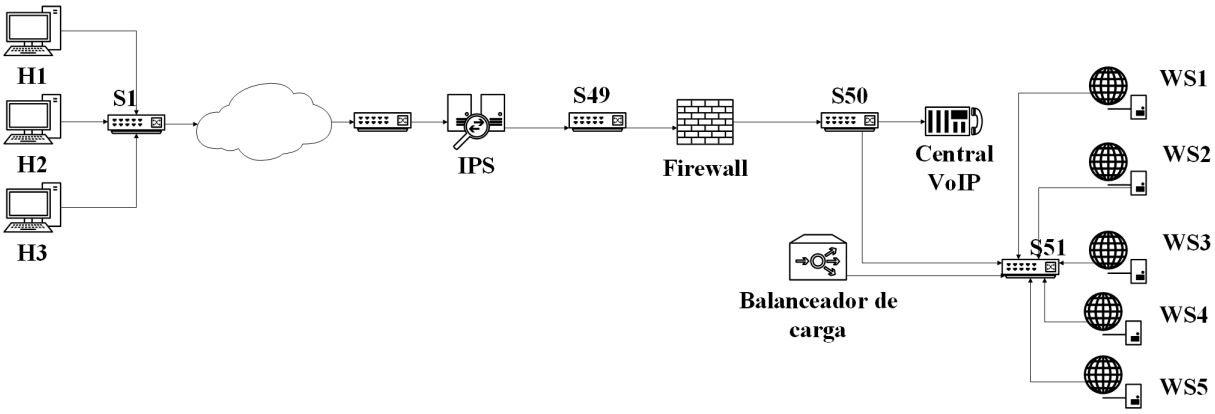
Podem ser observados na Figura 6.9 apenas quatro *switches*, pois a interligação entre os demais *switches* é representada por uma nuvem. Esses *switches* são interligados entre si de forma que entre os clientes (H1,H2 e H3) e a central VoIP os dados passem obrigatoriamente por 50 *switches*. O *switch* 'S51' que interliga os servidores *web* não está no caminho entre os clientes e a central VoIP.

Os resultados obtidos nessa nova topologia sobre o atraso e o *jitter* de chamadas VoIP são apresentados nas Tabelas 6.6 e 6.7, respectivamente. A seguir esses resultados são analisados estatisticamente através de testes de hipóteses.

Tabela 6.6: Atraso para chamadas VoIP atravessando *firewall* e IPS em uma rede com 51 *switches*

	Média	Mediana	Min	Max
Políticas dinâmicas (ms)	20,880	18,050	1,213	858,300
Políticas estáticas (ms)	23,620	17,840	1,299	535,300
Diferença (ms)	-2,74	0,21	–	–

Figura 6.9: Topologia com 51 switches



Fonte: elaborada pelo autor.

Na análise do atraso, foi aplicado o teste WSRT em amostras com n=20.000. Como foi obtido $p\text{-}value = 0,1029 > \alpha = 0,05$, não é possível rejeitar H_0, levando à conclusão de que com 95% de nível de confiança as medianas estatisticamente não são diferentes. Assim, tendo como métrica o atraso, o protótipo não gerou qualquer impacto ao desempenho da rede.

Tabela 6.7: *Jitter* para chamadas VoIP atravessando *firewall* e IPS em uma rede com 51 *switches*

	Média	Mediana	Min	Max
Políticas dinâmicas (ms)	1,174	1,09	0,4	7,76
Políticas estáticas (ms)	1,244	1,1	0,42	10,28
Diferença (ms)	-0,07	-0,01	–	–

Na análise do *jitter*, o teste WSRT retornou $p\text{-}value = 0,03097 < \alpha = 0,05$, levando à rejeição de H_0. Assim como na análise realizada nas seções anteriores, foi realizado um novo teste WSRT com as hipóteses:

$$H_0 : \tilde{\mu_0} > \tilde{\mu_1}$$
$$H_1 : \tilde{\mu_0} < \tilde{\mu_1}$$

sendo obtido $p\text{-}value = 0,9845 > \alpha = 0,05$, conduz a não rejeição da hipótese nula. Portanto, o protótipo não introduz impacto ao desempenho da rede tendo como métrica o *jitter* de chamadas VoIP.

A perda de pacotes para ambos os cenários, políticas dinâmicas e politicas estáticas, foi nula. Esses resultados indicam que a arquitetura proposta neste trabalho de mestrado é capaz de gerenciar topologias em diferentes escalas e manter um bom desempenho. A arquitetura foi capaz de manter as hipóteses sobre seu desempenho mesmo com o crescimento da quantidade de dispositivos de encaminhamento gerenciados.

6.2.4 Comunicação *web* com IPS e Balanceador de carga

Neste teste, é avaliada a capacidade da arquitetura proposta nesta dissertação obter informações sobre as políticas aplicadas pelos *middleboxes*. Para avaliar esse aspecto da arquitetura, foram utilizados um IPS e um balanceador de carga enviando ao controlador suas políticas aplicadas em requisições HTTP. No IPS, foi configurada uma política para alertar sobre tráfego HTTP com destino ao IP do balanceador de carga. O agente extrai informações do arquivo de *logs* do *Snort*, e as envia a uma aplicação no controlador. Novamente esse agente foi executado na máquina 'IPS' da Figura 6.1. O servidor *web Apache* foi executado nos *hosts* WS [1-5]. No *HAproxy*, foram configuradas políticas para balancear as requisições dos clientes *web* para os servidores WS [1-5], utilizando o algoritmo *round-robin*[4]. Outro agente é responsável por extrair informações do arquivo de *logs* do *HAproxy* e as enviar a uma aplicação no controlador. Esse agente é executado paralelamente ao *HAproxy* na máquina 'Balanceador de carga' apresentada na Figura 6.1. O *HAproxy* realiza o balanceamento de carga apenas para HTTP e TCP, logo não é capaz de balancear a comunicação VoIP, que utiliza o protocolo UDP. Os *middleboxes* foram avaliados tendo como métricas o atraso e a perda de pacotes. A Figura 6.10 apresenta as requisições com políticas do balanceador de carga recebidas pelo controlador *Ryu*.

Na avaliação do *Snort* e do *HAproxy*, foi utilizada a ferramenta *HTTPing*, para medir o atraso e a perda de pacotes em requisições HTTP. O *HTTPing* foi executado no *host* H1, gerando requisições *HEAD*[5] para o IP do balanceador de carga, que as redireciona para os servidores. Foram geradas 20.000 requisições. No *Ryu*, foram executadas as aplicações que recebem políticas do IPS e do balanceador de carga, e a aplicação *simple_switch_13.py* padrão do *Ryu* para configuração de *switches* utilizando o protocolo *OpenFlow* 1.3. Os resultados sobre o atraso são apresentados na Tabela 6.8.

A perda de pacotes foi nula. O teste WSRT retornou um $p\text{-}value$ de $2,2^{-16}$, logo H_0 deve ser rejeitada, indicando que há impacto no desempenho da rede, considerando o atraso de requisições HTTP.

[4]Os servidores são dispostos em uma fila circular. O balanceador de carga redireciona a requisição para o servidor na primeira posição, e envia-o para o final da fila.

[5]O servidor retorna apenas os cabeçalhos da resposta.

Figura 6.10: Requisições do agente do balanceador de carga recebidas pelo *Ryu*

```
root@mininet-vm:/home/mininet/ryu# ./bin/ryu-manager ryu/app/simple_switch_13.py ryu/app/haproxy.py ryu/app/snort.py
loading app ryu/app/simple_switch_13.py
loading app ryu/app/haproxy.py
loading app ryu/app/snort.py
loading app ryu.controller.ofp_handler
creating context wsgi
instantiating app ryu/app/simple_switch_13.py of SimpleSwitch13
instantiating app ryu/app/snort.py of Snort
instantiating app ryu.controller.ofp_handler of OFPHandler
instantiating app ryu/app/haproxy.py of HAproxy
(21567) wsgi starting up on http://0.0.0.0:8080/
(21567) accepted ('192.168.4.201', 32909)
192.168.4.201 - - [09/Jan/2016 23:58:15] "POST /InformCtrLB HTTP/1.1" 200 115 0.044032
192.168.4.201 - - [09/Jan/2016 23:58:19] "POST /InformCtrLB HTTP/1.1" 200 115 0.031600
192.168.4.201 - - [09/Jan/2016 23:58:23] "POST /InformCtrLB HTTP/1.1" 200 115 0.037048
192.168.4.201 - - [09/Jan/2016 23:58:27] "POST /InformCtrLB HTTP/1.1" 200 115 0.042105
192.168.4.201 - - [09/Jan/2016 23:58:31] "POST /InformCtrLB HTTP/1.1" 200 115 0.035094
192.168.4.201 - - [09/Jan/2016 23:58:35] "POST /InformCtrLB HTTP/1.1" 200 115 0.036019
192.168.4.201 - - [09/Jan/2016 23:58:39] "POST /InformCtrLB HTTP/1.1" 200 115 0.040051
192.168.4.201 - - [09/Jan/2016 23:58:43] "POST /InformCtrLB HTTP/1.1" 200 115 0.043864
192.168.4.201 - - [09/Jan/2016 23:58:47] "POST /InformCtrLB HTTP/1.1" 200 115 0.031966
192.168.4.201 - - [09/Jan/2016 23:58:51] "POST /InformCtrLB HTTP/1.1" 200 115 0.030117
192.168.4.201 - - [09/Jan/2016 23:58:55] "POST /InformCtrLB HTTP/1.1" 200 115 0.035537
192.168.4.201 - - [09/Jan/2016 23:58:59] "POST /InformCtrLB HTTP/1.1" 200 115 0.035832
192.168.4.201 - - [09/Jan/2016 23:59:03] "POST /InformCtrLB HTTP/1.1" 200 115 0.081284
192.168.4.201 - - [09/Jan/2016 23:59:07] "POST /InformCtrLB HTTP/1.1" 200 115 0.027918
192.168.4.201 - - [09/Jan/2016 23:59:11] "POST /InformCtrLB HTTP/1.1" 200 115 0.030734
192.168.4.201 - - [09/Jan/2016 23:59:15] "POST /InformCtrLB HTTP/1.1" 200 115 0.027030
192.168.4.201 - - [09/Jan/2016 23:59:19] "POST /InformCtrLB HTTP/1.1" 200 115 0.034387
192.168.4.201 - - [09/Jan/2016 23:59:23] "POST /InformCtrLB HTTP/1.1" 200 115 0.030889
192.168.4.201 - - [09/Jan/2016 23:59:27] "POST /InformCtrLB HTTP/1.1" 200 115 0.025326
192.168.4.201 - - [09/Jan/2016 23:59:31] "POST /InformCtrLB HTTP/1.1" 200 115 0.027572
192.168.4.201 - - [09/Jan/2016 23:59:35] "POST /InformCtrLB HTTP/1.1" 200 115 0.035594
192.168.4.201 - - [09/Jan/2016 23:59:39] "POST /InformCtrLB HTTP/1.1" 200 115 0.019825
```

Fonte: elaborada pelo autor.

Tabela 6.8: *Delay* para requisições *web* atravessando IPS e Balanceador de carga.

	Média	Mediana	Min	Max
Políticas dinâmicas (ms)	4,994	3,9	1,310	36,130
Políticas estáticas (ms)	3,501	2,38	1,240	121,200
Diferença (ms)	1,493	1,52	–	–

6.2.5 Caso de uso: comunicação VoIP com Firewall, IPS e múltiplos controladores

Objetivo deste teste é avaliar a escalabilidade da arquitetura quanto a sua capacidade de crescer de forma simples e com baixo custo. A configuração dos *middleboxes* e da rede apresentada na Subseção 6.2.2 foi mantida e foram utilizados três controladores *Ryu*: c0, c1 e c2. A carga de trabalho gerada pelos *switches* e *middleboxes* é balanceada entre esses três controladores da seguinte forma: c0 é responsável por receber políticas do IPS, c1 gerencia os *switches* s2 e s3 e atua na configuração do *firewall* e c2 gerencia s1 e s4. O c2 foi encarregado de gerenciar todos os *switches*. Contudo, c1 precisa coordenar s2 e s3 porque eles estão diretamente conectados ao *firewall* e é preciso garantir que as políticas sejam enviadas a ele antes dos dados.

Novamente, o *SIPp* foi executado no *host* H1, gerando 50 chamadas simultâneas para o usuário configurado no *asterisk*, até chegar ao valor de 20.000 chamadas. Todas as chamadas foram realizadas com sucesso e a perda de pacotes foi nula. Um comparativo entre os resultados obtidos e os dados sobre o protótipo apresentados na Subseção 6.2.2 são mostrados nas Tabelas 6.9 e 6.10. As Figuras 6.11 a 6.13 apresentam as telas dos três controladores em execução, em que cada controlador executa uma aplicação da arquitetura proposta nesta dissertação.

Figura 6.11: Controlador c0 executando a aplicação responsável por receber políticas do IPS

```
root@mininet-vm:/home/mininet/ryu# ./bin/ryu-manager --ofp-tcp-listen-port 6633 ryu/app/snort.py
loading app ryu/app/snort.py
creating context wsgi
instantiating app ryu/app/snort.py of Snort
(9571) wsgi starting up on http://0.0.0.0:8080/
(9571) accepted ('192.168.3.1', 38888)
192.168.3.1 - - [27/Jan/2016 09:54:06] "POST /InformCtrIPS HTTP/1.1" 200 115 0.004667
192.168.3.1 - - [27/Jan/2016 09:54:06] "POST /InformCtrIPS HTTP/1.1" 200 115 0.004973
192.168.3.1 - - [27/Jan/2016 09:54:06] "POST /InformCtrIPS HTTP/1.1" 200 115 0.002850
192.168.3.1 - - [27/Jan/2016 09:54:06] "POST /InformCtrIPS HTTP/1.1" 200 115 0.002025
192.168.3.1 - - [27/Jan/2016 09:54:06] "POST /InformCtrIPS HTTP/1.1" 200 115 0.006099
192.168.3.1 - - [27/Jan/2016 09:54:06] "POST /InformCtrIPS HTTP/1.1" 200 115 0.000941
192.168.3.1 - - [27/Jan/2016 09:54:06] "POST /InformCtrIPS HTTP/1.1" 200 115 0.003479
192.168.3.1 - - [27/Jan/2016 09:54:06] "POST /InformCtrIPS HTTP/1.1" 200 115 0.001851
192.168.3.1 - - [27/Jan/2016 09:54:06] "POST /InformCtrIPS HTTP/1.1" 200 115 0.001913
192.168.3.1 - - [27/Jan/2016 09:54:06] "POST /InformCtrIPS HTTP/1.1" 200 115 0.002070
192.168.3.1 - - [27/Jan/2016 09:54:06] "POST /InformCtrIPS HTTP/1.1" 200 115 0.001682
192.168.3.1 - - [27/Jan/2016 09:54:06] "POST /InformCtrIPS HTTP/1.1" 200 115 0.000510
192.168.3.1 - - [27/Jan/2016 09:54:06] "POST /InformCtrIPS HTTP/1.1" 200 115 0.002302
192.168.3.1 - - [27/Jan/2016 09:54:06] "POST /InformCtrIPS HTTP/1.1" 200 115 0.001686
192.168.3.1 - - [27/Jan/2016 09:54:06] "POST /InformCtrIPS HTTP/1.1" 200 115 0.001785
192.168.3.1 - - [27/Jan/2016 09:54:06] "POST /InformCtrIPS HTTP/1.1" 200 115 0.002020
192.168.3.1 - - [27/Jan/2016 09:54:06] "POST /InformCtrIPS HTTP/1.1" 200 115 0.004340
192.168.3.1 - - [27/Jan/2016 09:54:06] "POST /InformCtrIPS HTTP/1.1" 200 115 0.002217
192.168.3.1 - - [27/Jan/2016 09:54:06] "POST /InformCtrIPS HTTP/1.1" 200 115 0.006043
192.168.3.1 - - [27/Jan/2016 09:54:06] "POST /InformCtrIPS HTTP/1.1" 200 115 0.000609
192.168.3.1 - - [27/Jan/2016 09:54:06] "POST /InformCtrIPS HTTP/1.1" 200 115 0.002204
192.168.3.1 - - [27/Jan/2016 09:54:06] "POST /InformCtrIPS HTTP/1.1" 200 115 0.004150
192.168.3.1 - - [27/Jan/2016 09:54:06] "POST /InformCtrIPS HTTP/1.1" 200 115 0.003477
192.168.3.1 - - [27/Jan/2016 09:54:06] "POST /InformCtrIPS HTTP/1.1" 200 115 0.001052
192.168.3.1 - - [27/Jan/2016 09:54:06] "POST /InformCtrIPS HTTP/1.1" 200 115 0.000734
192.168.3.1 - - [27/Jan/2016 09:54:06] "POST /InformCtrIPS HTTP/1.1" 200 115 0.001930
192.168.3.1 - - [27/Jan/2016 09:54:06] "POST /InformCtrIPS HTTP/1.1" 200 115 0.017389
192.168.3.1 - - [27/Jan/2016 09:54:06] "POST /InformCtrIPS HTTP/1.1" 200 115 0.005960
192.168.3.1 - - [27/Jan/2016 09:54:06] "POST /InformCtrIPS HTTP/1.1" 200 115 0.000970
192.168.3.1 - - [27/Jan/2016 09:54:06] "POST /InformCtrIPS HTTP/1.1" 200 115 0.003333
```

Fonte: elaborada pelo autor.

Figura 6.12: Controlador c1 executando a aplicação responsável por configurar políticas no *firewall*

```
root@mininet-vm:/home/mininet/ryu# ./bin/ryu-manager --ofp-tcp-listen-port 6635 ryu/app/iptables.py
loading app ryu/app/iptables.py
loading app ryu.controller.ofp_handler
instantiating app ryu/app/iptables.py of Fw
instantiating app ryu.controller.ofp_handler of OFPHandler
src: 192.168.2.101 --> dst: 192.168.4.200 on port 5060 -- from switch S2
src: 192.168.4.200 --> dst: 192.168.2.101 on port 5060 -- from switch S3
src: 192.168.2.101 --> dst: 192.168.4.200 on port 16426 -- from switch S2
src: 192.168.2.101 --> dst: 192.168.4.200 on port 15794 -- from switch S2
src: 192.168.2.101 --> dst: 192.168.4.200 on port 15104 -- from switch S2
src: 192.168.2.101 --> dst: 192.168.4.200 on port 14916 -- from switch S2
src: 192.168.2.101 --> dst: 192.168.4.200 on port 13256 -- from switch S2
src: 192.168.2.101 --> dst: 192.168.4.200 on port 12178 -- from switch S2
src: 192.168.2.101 --> dst: 192.168.4.200 on port 19022 -- from switch S2
src: 192.168.2.101 --> dst: 192.168.4.200 on port 10550 -- from switch S2
src: 192.168.2.101 --> dst: 192.168.4.200 on port 11288 -- from switch S2
src: 192.168.2.101 --> dst: 192.168.4.200 on port 19900 -- from switch S2
src: 192.168.2.101 --> dst: 192.168.4.200 on port 14946 -- from switch S2
src: 192.168.2.101 --> dst: 192.168.4.200 on port 10854 -- from switch S2
src: 192.168.2.101 --> dst: 192.168.4.200 on port 18890 -- from switch S2
src: 192.168.2.101 --> dst: 192.168.4.200 on port 11260 -- from switch S2
src: 192.168.2.101 --> dst: 192.168.4.200 on port 16880 -- from switch S2
src: 192.168.2.101 --> dst: 192.168.4.200 on port 12770 -- from switch S2
src: 192.168.2.101 --> dst: 192.168.4.200 on port 10204 -- from switch S2
src: 192.168.2.101 --> dst: 192.168.4.200 on port 16358 -- from switch S2
src: 192.168.2.101 --> dst: 192.168.4.200 on port 15638 -- from switch S2
src: 192.168.2.101 --> dst: 192.168.4.200 on port 15482 -- from switch S2
src: 192.168.2.101 --> dst: 192.168.4.200 on port 10988 -- from switch S2
src: 192.168.2.101 --> dst: 192.168.4.200 on port 11600 -- from switch S2
src: 192.168.2.101 --> dst: 192.168.4.200 on port 14222 -- from switch S2
src: 192.168.2.101 --> dst: 192.168.4.200 on port 17034 -- from switch S2
src: 192.168.2.101 --> dst: 192.168.4.200 on port 18470 -- from switch S2
src: 192.168.2.101 --> dst: 192.168.4.200 on port 19994 -- from switch S2
src: 192.168.2.101 --> dst: 192.168.4.200 on port 14260 -- from switch S2
src: 192.168.2.101 --> dst: 192.168.4.200 on port 10208 -- from switch S2
```

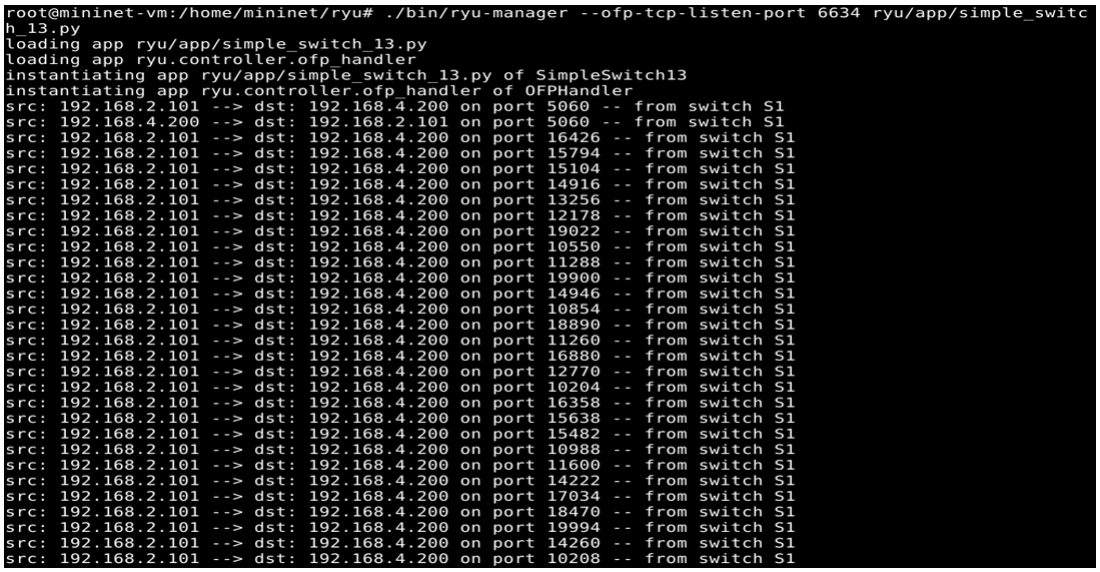
Fonte: elaborada pelo autor.

Tabela 6.9: *Delay* para chamadas VoIP atravessando *firewall* e IPS com múltiplos controladores

	Média	Mediana	Min	Max
Três controladores (ms)	4,863	3,698	1,036	244,4
Um controlador (ms)	5,577	4,460	1,143	130,00
Diferença (ms)	-0,714	-0,762	–	–

Nas avaliações anteriores, os resultados provinham de amostras pareadas por isso foi

Figura 6.13: Controlador c2 executando a aplicação responsável por configurar os *switches* da rede



Fonte: elaborada pelo autor.

Tabela 6.10: *Jitter* para chamadas VoIP atravessando *firewall* e IPS com múltiplos controladores

	Média	Mediana	Min	Max
Três controladores (ms)	0,522	0,5	0,280	2,22
Um controlador (ms)	0,541	0,49	0,32	2.71
Diferença (ms)	-0,019	0,01	–	–

usado o *Wilcoxon Signed-Rank Test*, contudo nesta avaliação os dados não são pareados, exigindo o uso de outro teste. Foi aplicado o *Wilcoxon Rank-Sum Test* (WILCOXON; KATTI; WILCOX, 1970) para dados não pareados (independentes) nos resultados apresentados nas Tabelas acima a fim de avaliar se a arquitetura é capaz de crescer mantendo um bom desempenho. Foram definidas as hipóteses:

$$H_0 : \tilde{\mu_0} = \tilde{\mu_1}$$
$$H_1 : \tilde{\mu_0} \neq \tilde{\mu_1}$$

em que $\tilde{\mu_0}$ é a mediana da amostra com um e $\tilde{\mu_1}$ a mediana da amostra com três controladores.

Comparando-se os resultados sobre o atraso das chamadas foi obtido um *p-value* de $2,2^{-16}$, levando à rejeição da hipótese nula. Entretanto, novamente foi realizado um segundo teste para avaliar se o protótipo apresenta um desempenho superior com as hipóteses:

$$H_0 : \tilde{\mu_0} > \tilde{\mu_1}$$
$$H_1 : \tilde{\mu_0} < \tilde{\mu_1}$$

e obtido um *p-value* de 1, que conduz à não rejeição da hipótese nula. Portanto, tendo o atraso como métrica, a arquitetura apresentou um desempenho melhor ao utilizar múltiplos controladores.

Para os dados sobre o *jitter* o teste WRST retornou um *p-value* de 0,2237 que, como é maior que $\alpha = 0,05$, não é possível rejeitar H_0 e portanto as medianas não são diferentes. Assim, não há diferença estatística no *jitter* quando a arquitetura utiliza um ou três controladores, mostrando-se capaz de crescer mantendo um bom desempenho.

6.3 Ameaças à validade dos resultados

Esta seção apresenta algumas ameaças à validade dos resultados obtidos nos experimentos (EASTERBROOK et al., 2008), e quais medidas foram tomadas para evitar tais ameaças. São elas:

- Interna:
  - Ameaça: a relação entre o tratamento e os resultados deve ser causal. Portanto, temos de assegurar que os resultados não sofrem influência de fatores externos;
  - Medidas: uma vez que o ambiente de rede é totalmente controlado, controlar os fatores ambientais não foi um problema. Nenhum fator externo ao ambiente usado na avaliação foi capaz de influenciar os resultados.
- Externa:
  - Ameaça: esta validade é alcançada apenas quando é possível generalizar os resultados para outros escopos além do inicial;
  - Medidas: procurou-se utilizar uma rede cujo comportamento fosse o mais próximo possível de uma rede real. Como foi necessário utilizar um ambiente de emulação por falta de uma rede real, entende-se que a rede usada nos experimentos não seja tão exata quanto uma rede física. Entretanto, espera-se que o *software* de emulação utilizado seja capaz de simular corretamente o comportamento de uma rede física, visto que é um *software* amplamente utilizado na literatura. Os *middleboxes* usados são amplamente utilizados em ambientes de produção e as condições de tráfego e topologia refletem ambientes reais. Nosso objetivo é que o ambiente experimental tenha características de redes tão próximas às de redes reais quanto possível. Assim, acreditamos que os resultados obtidos podem ser generalizados para diferentes contextos.
- Construto:
  - Ameaça: procura assegurar que existe uma relação entre teoria e observação;
  - Medidas: neste trabalho de mestrado, espera-se que não existam problemas com esta validade uma vez que o protótipo desenvolvido segue rigorosamente a teoria apresentada nesta dissertação.

- Conclusão:
  - Ameaça: esta validade está relacionada com a capacidade para assegurar que os resultados obtidos estão relacionados com o tratamento aplicado;
  - Medidas: testes estatísticos foram empregados na análise dos resultados para auxiliar na conclusão, reduzindo a possibilidade de conclusões incorretas a respeito dos resultados. Como o tratamento mostrou ser eficaz na solução do problema avaliado nesta dissertação e testes estatísticos atestaram que a arquitetura causa pouco ou nenhum impacto no desempenho da rede, os resultados são considerados válidos.

6.4 Considerações finais

Nesta seção, foram apresentadas a metodologia e o ambiente para avaliação do protótipo. Foram apresentados também os diferentes cenários usados na avaliação do protótipo e os resultados obtidos nesses experimentos. Na análise dos resultados, foram empregados testes estatísticos a fim de comprovar a eficiência do protótipo na aplicação de políticas de *middleboxes*. Além disso, foram descritas as ameaças aos resultados desta dissertação e quais medidas foram tomadas para evitá-las.

7 Conclusões

Nesta seção, a solução proposta apresentada neste trabalho de mestrado é brevemente revisitada. São apresentadas as considerações finais a respeito da solução proposta, assim como as conclusões sobre os resultados obtidos nos experimentos descritos na seção anterior. Por fim, são apresentadas alguns extensões para a arquitetura proposta nesta dissertação a serem implementadas em trabalhos futuros. Essas extensões vão do uso de criptografia ao uso da arquitetura proposta em um ambiente de *Internet* das coisas, passando pela integração com um banco de dados, garantir que políticas de *middleboxes* sejam aplicadas em tráfego modificado por esses dispositivos e configuração de caminhos fim a fim entre a origem e o destino dos dados.

7.1 Considerações finais

Neste trabalho foi apresentada uma arquitetura para aplicação dinâmica de políticas de *middleboxes* com o uso de *Software-Defined Networking*. Foi proposto o uso do controlador SDN para configurar políticas nos *middleboxes*, em resposta a requisitos de aplicações dinâmicas de rede. A comunicação do controlador com *middleboxes* é realizada através de uma API proposta neste trabalho, que possibilita ao controlador configurar e obter políticas desses dispositivos. Como o controlador é logicamente centralizado e possui uma visão global sobre o tráfego da rede que gerencia, *middleboxes* poderão ser configurados com base no tráfego corrente, de um ponto logicamente centralizado, simplificando a atual configuração feita de forma distribuída.

A partir da arquitetura proposta, foi desenvolvido um protótipo para aplicação de políticas de *middleboxes*. O protótipo obtém informações sobre o tráfego da rede a partir das mensagens *packet-in* enviadas ao controlador e informações sobre políticas aplicadas por *middleboxes* através da API proposta neste trabalho. Políticas são configuradas nos *middleboxes*, e os *switches* são configurados para encaminhar o tráfego tratado por essas políticas até o *middlebox* que aplicará estas políticas. Informações sobre políticas aplicadas por middleboxes podem ser utilizadas na configuração de seus pares, tornando a integração entre *middleboxes* mais simples.

Nas avaliações às quais o protótipo foi submetido, foram usados três *middleboxes* amplamente utilizados em ambientes de produção, permitindo-nos generalizar os resultados para outros contextos. Os *middleboxes* foram avaliados no tratamento de comunicação VoIP, uma aplicação muito popular e que enfrenta vários desafios na presença desses dispositivos, e comunicação HTTP, outra aplicação muito popular. Os resultados obtidos mostram que o protótipo é capaz de aplicar políticas de *middleboxes*, gerando um impacto mínimo, quando há. Portanto, os resultados mostram que o protótipo configura-se como uma ferramenta viável para aplicação de políticas de *middleboxes* com o uso da arquitetura SDN.

Os resultados mostram que a arquitetura pode ser facilmente ampliada através do uso de

múltiplos controladores, uma prática popular em redes com muitos dispositivos de encaminhamento. Logo, além ser facilmente expandida, não gera custos extras para escalar já que emprega um mecanismo de crescimento largamente adotado independentemente da sua utilização.

A arquitetura mostrou-se capaz de manter um bom desempenho utilizando apenas um controlador mesmo com o aumento na quantidade dispositivos gerenciados e, ao ser ampliada com múltiplos controladores manteve e até apresentou resultados melhores em relação ao uso de apenas um controlador. A arquitetura apresentou bons resultados tanto em uma topologia pequena, com quatro *switches*, quanto em uma topologia relativamente maior, com 50 *switches*. E, os resultados obtidos na topologia com 50 *switches* mantiveram as hipóteses sobre a arquitetura não gerar impacto ao desempenho da rede.

Como a comunicação entre controlador e *middleboxes* introduz tráfego e a configuração das políticas é realizada anteriormente ao encaminhamento dos dados, é natural que exista alguma interferência no atraso, mesmo sendo mínima, como mostrado pelos resultados. A arquitetura atua na liberação da chamadas VoIP, logo é esperado que exista pouca interferência sobre o *jitter* que ocorre nos pacotes transmitidos durante a chamada. O tráfego gerado pela arquitetura é mínimo, não sobrecarrega os enlaces nem os *switches*, assim como o controlador e *middleboxes* não têm seu desempenho comprometido. Logo, é esperado que não exista perda de pacotes.

Neste trabalho, foi mostrado que a arquitetura SDN pode ser estendida para tornar a aplicação de políticas de *middleboxes* dinâmica, impondo pouco ou nenhum impacto ao desempenho da rede.

A resposta para a questão de pesquisa: "como SDN contribui na aplicação dinâmica de políticas de *middleboxes*?"é o uso de características como controle centralizado e programabilidade dos dispositivos de rede, presentes na arquitetura SDN. Baseado nos resultados obtidos, é possível afirmar que a hipótese nula: a arquitetura habilita aplicação dinâmica de políticas, não deve ser rejeitada. Logo, a arquitetura proposta neste trabalho é capaz de dinamicamente aplicar políticas de *middleboxes*.

A principal contribuição deste trabalho para o estado da arte foi mostrar de forma clara, que a arquitetura SDN poderá ser empregada na aplicação de políticas de *middleboxes*. Foi mostrado que características da arquitetura SDN, como controle centralizado e programabilidade dos dispositivos da rede, são valiosas para estender as capacidades providas por *middleboxes*, habilitando novas formas de configuração e aplicação de suas políticas.

7.2 Trabalhos futuros

Nesta seção são apresentados alguns trabalhos que apresentam potencial de desenvolvimento em futuras melhorias do protótipo.

7.2.1 Comunicação entre *middleboxes* e controlador criptografada

A comunicação entre controlador e *middleboxes* deve ser protegida contra possíveis invasores ou usuários não autorizados a acessar tais informações. Políticas aplicadas por *middleboxes* não devem ser conhecidas pelos usuários da rede (SHERRY et al., 2015). Uma alternativa é a utilização de criptografia fim a fim entre controlador e *middleboxes*, para ocultar dos usuários informações sobre as políticas. Adicionalmente, podem ser usados certificados digitais para

autenticação entre *middleboxes* e controlador, garantido que apenas dispositivos autorizados possam participar da comunicação.

7.2.2 Armazenamento dos dados em um banco de dados

Apesar do bom desempenho do protótipo utilizando o controlador para armazenar políticas, o uso de um banco de dados apresenta-se com uma alternativa interessante para o armazenamento dessas informações. Com informações armazenadas externamente, o controlador poderá reduzir seu uso de memória, um recurso limitado e essencial. Caso o controlador apresente alguma falha, se as políticas estiverem em um banco de dados, poderão ser facilmente recuperadas. Como trabalho futuro, espera-se encontrar alternativas na implementação do protótipo, como o uso de um banco de dados para armazenamento das políticas. Inicialmente será utilizado um banco de dados *Not Only Structured Query Language* (NoSQL) devido ao desempenho superior desse tipo de banco de dados (HAN et al., 2011).

7.2.3 Tratar modificações geradas por *middleboxes*

Modificações no tráfego causadas por *middleboxes* geram vários desafios ao funcionamento da rede (FAYAZBAKHSH et al., 2014; NGO; KIM, 2014). Como o protótipo é capaz de obter informações sobre políticas aplicadas, é possível conhecer as modificações geradas pelos *middleboxes* no tráfego da rede, possibilitando o uso dessas informações na configuração de outros *middleboxes*, para que apliquem corretamente suas políticas. Em testes iniciais, o protótipo se mostrou capaz de conhecer modificações geradas por um balanceador de carga, e configurar em um *firewall* as políticas para lidar com o tráfego modificado. Contudo, o *firewall* não foi capaz de aplicar corretamente as políticas recebidas do controlador. Esse problema ocorre porque o tráfego chega ao *firewall* antes das políticas serem configuradas, que o leva a aplicar políticas incorretas. É preciso garantir que as políticas sejam configuradas no *middlebox* antes que o tráfego por elas tratado seja recebido.

7.2.4 Configuração de caminho fim a fim entre a origem e o destino

Em uma rede SDN, o envio do primeiro pacote de um fluxo do *switch* ao controlador é um processo que atrapalha o desempenho da rede. Logo, é importante reduzir ao máximo essa comunicação entre controlador e *switches*. Uma forma de evitar o envio do primeiro pacote ao controlador pelo *switch* é previamente configurar os *switches* para que encaminhem os dados.

Para que o controlador saiba quais *switches* devem ser configurados é preciso calcular o caminho a ser percorrido pelos dados da sua origem até o seu destino. Uma opção é calcular o caminho mais curto entre a origem e o destino dos dados. Isso pode ser feito construindo um grafo a partir das informações da topologia da rede, em que *switches* são os vértices e os enlaces são as arestas. Depois basta calcular o caminho mais curto. E, por fim é feita a configuração dos *switches* que compõem o caminho para que encaminhem os dados até seu destino.

Como trabalho futuro espera-se desenvolver um algoritmo que calcule um caminho mais curto entre a origem e o destino dos dados e configure os *switches* que compõem esse caminho para que encaminhem os dados até o destino. Dessa forma, apenas o primeiro *switch* do caminho enviaria o pacote ao controlador que calcularia o caminho mais curto e configuraria todos os outros *switches* desse caminha. Logo, os demais *switches* desse caminho ao receberem

dados os encaminharia sem precisar consultar o controlador. Portanto, com esse algoritmo os *switches* serão menos dependentes do controlador, a carga de trabalho do controlador será reduzida melhorando seu desempenho. O desempenho da rede deverá melhorar também.

7.2.5 Aplicação de políticas de *Middleboxes* na *Internet* das coisas

Internet das coisas (do inglês, *Internet of Things*, IoT) (ATZORI; IERA; MORABITO, 2010) tem como ideia básica o conceito de onipresença de objetos ou coisas, tais como *Radio-Frequency IDentification* (RFID) *tags* (WEIS et al., 2004), atuadores, *smartphones*, *tablets*, sensores, etc., que, através de esquemas de endereçamentos únicos, são capazes de interagir entre si e cooperar para alcançar objetivos comuns (ATZORI; IERA; MORABITO, 2010). O paradigma de IoT terá grande impacto no cotidiano de seus potenciais usuários, pois objetos estarão presentes em diversos ambientes do dia a dia, em domínios como carros conectados (LU et al., 2014), industrial, saúde, casas e cidades inteligentes (ATZORI; IERA; MORABITO, 2010).

Estimativas apontam que aproximadamente 50 bilhões de objetos estarão conectados à *Internet* até 2020 (EVANS, 2011). A natureza mutável e imprevisível do ecossistema IoT colocará a infraestrutura da rede em seu limite (OMNES et al., 2015). Atualmente, *middleboxes* são configurados estaticamente para tratar eventos e tráfego previstos pelo operador de rede. Contudo, o tráfego gerado pela IoT será enorme e com crescimento exponencial, impondo inúmeros desafios às redes e aos *middleboxes*. Considerando a multiplicidade dos objetos na IoT e tráfego gerado por eles, torna-se praticamente impossível prever quais objetos estão conectados a rede e como seu tráfego deve ser tratado pelos *middleboxes*.

A rede e os serviços por ela oferecidos deverão ser mais dinâmicos para garantir os requisitos dos objetos da IoT (OMNES et al., 2015). Logo, novas propostas têm surgido com objetivo de prover serviços para IoT através de *middleboxes* (OMNES et al., 2015).

Em um ambiente de rede dinâmico como é exigido pela IoT, para que *middleboxes* sejam capazes de prover serviços, estes dispositivos deverão ter a capacidade de dinamicamente aplicar suas políticas em resposta a requisitos dos objetos que compõem a IoT. Nesse cenário, em que IoT exigirá dinamicidade por parte dos *middleboxes* na aplicação de suas políticas, parece-nos uma excelente oportunidade para expandir a pesquisa realizada durante este mestrado para o contexto da *Internet* das coisas, na medida em que o ecossistema IoT tende a crescer muito nos próximos anos,

Como trabalho futuro, espera-se estender a arquitetura proposta nesta dissertação com o objetivo de tornar os *middleboxes* capazes de lidar com os requisitos das aplicações IoT.

Referências

AKYILDIZ, I. F. et al. A Roadmap for Traffic Engineering in SDN-OpenFlow Networks. **Comput. Netw.**, New York, NY, USA, v.71, p.1–30, October 2014.

ALHOMOUD, A. et al. Performance Evaluation Study of Intrusion Detection Systems. **Procedia Computer Science**, [S.l.], v.5, p.173 – 180, 2011.

ATZORI, L.; IERA, A.; MORABITO, G. The Internet of Things: a survey. **Computer Networks**, [S.l.], v.54, n.15, p.2787 – 2805, 2010.

BRAY, T. **The JavaScript Object Notation (JSON) Data Interchange Format**. [S.l.]: IETF, 2014. n.7159. (Request for Comments).

BREMLER-BARR, A.; HARCHOL, Y.; HAY, D. OpenBox: enabling innovation in middlebox applications. In: ACM SIGCOMM WORKSHOP ON HOT TOPICS IN MIDDLEBOXES AND NETWORK FUNCTION VIRTUALIZATION, 2015., New York, NY, USA. **Proceedings...** ACM, 2015. p.67–72. (HotMiddlebox '15).

CARPENTER, B.; BRIM, S. **Middleboxes**: taxonomy and issues. [S.l.]: IETF, 2002. n.3234. (Request for Comments).

CHIU, H.-W.; WANG, S.-Y. Boosting the OpenFlow control-plane message exchange performance of OpenvSwitch. In: COMMUNICATIONS (ICC), 2015 IEEE INTERNATIONAL CONFERENCE ON. **Anais...** IEEE, 2015. p.5284–5289.

CRAVEN, R.; BEVERLY, R.; ALLMAN, M. A Middlebox-cooperative TCP for a Non End-to-end Internet. In: ACM CONFERENCE ON SIGCOMM, 2014., New York, NY, USA. **Proceedings...** ACM, 2014. p.151–162. (SIGCOMM '14).

DIXIT, A. A. et al. ElastiCon: an elastic distributed sdn controller. In: TENTH ACM/IEEE SYMPOSIUM ON ARCHITECTURES FOR NETWORKING AND COMMUNICATIONS SYSTEMS, New York, NY, USA. **Proceedings...** ACM, 2014. p.17–28. (ANCS '14).

EASTERBROOK, S. et al. Selecting Empirical Methods for Software Engineering Research. In: SHULL, F.; SINGER, J.; SJøBERG, D. (Ed.). **Guide to Advanced Empirical Software Engineering**. [S.l.]: Springer London, 2008. p.285–311.

EDELINE, K.; DONNET, B. Towards a middlebox policy taxonomy: path impairments. In: COMPUTER COMMUNICATIONS WORKSHOPS (INFOCOM WKSHPS), 2015 IEEE CONFERENCE ON. **Anais...** IEEE, 2015. p.402–407.

EVANS, D. The internet of things: how the next evolution of the internet is changing everything. **CISCO white paper**, [S.l.], v.1, p.14, 2011.

FAYAZBAKHSH, S. K. et al. Enforcing Network-Wide Policies in the Presence of Dynamic Middlebox Actions using FlowTags. In: USENIX SYMPOSIUM ON NETWORKED SYSTEMS DESIGN AND IMPLEMENTATION (NSDI 14), 11., Seattle, WA. **Anais...** USENIX Association, 2014. p.543–546.

FOSSATI, T.; GURBANI, V. K.; KOLESNIKOV, V. Love All, Trust Few: on trusting intermediaries in http. In: ACM SIGCOMM WORKSHOP ON HOT TOPICS IN MIDDLEBOXES AND NETWORK FUNCTION VIRTUALIZATION, 2015., New York, NY, USA. **Proceedings...** ACM, 2015. p.1–6. (HotMiddlebox '15).

G711A. Disponível em: `http://sourceforge.net/p/sipp`, Acesso em: 29/10/2015.

GARG, A.; BAGGA, S. An autonomic approach for fault tolerance using scaling, replication and monitoring in cloud computing. In: MOOCS, INNOVATION AND TECHNOLOGY IN EDUCATION (MITE), 2015 IEEE 3RD INTERNATIONAL CONFERENCE ON. **Anais...** IEEE, 2015. p.129–134.

GEMBER, A. et al. Toward Software-defined Middlebox Networking. In: ACM WORKSHOP ON HOT TOPICS IN NETWORKS, 11., Redmond, Washington. **Proceedings...** ACM, 2012. p.7–12. (HotNets-XI).

GEMBER-JACOBSON, A.; AKELLA, A. Improving the Safety, Scalability, and Efficiency of Network Function State Transfers. In: ACM SIGCOMM WORKSHOP ON HOT TOPICS IN MIDDLEBOXES AND NETWORK FUNCTION VIRTUALIZATION, 2015., New York, NY, USA. **Proceedings...** ACM, 2015. p.43–48. (HotMiddlebox '15).

GEMBER-JACOBSON, A. et al. OpenNF: enabling innovation in network function control. In: ACM CONFERENCE ON SIGCOMM, 2014., New York, NY, USA. **Proceedings...** ACM, 2014. p.163–174. (SIGCOMM '14).

GONDIM, E. B.; PINHEIRO, A. J.; CAMPELO, D. R. Monitoramento de Desempenho com Middleboxes em Redes Definidas por Software. In: XXXIII SIMPOSIO BRASILEIRO DE TELECOMUNICAÇÕES. **Anais...** [S.l.: s.n.], 2015. p.189–193.

GRINBERG, M. **Flask Web Development**: developing web applications with python. [S.l.]: "O'Reilly Media, Inc.", 2014.

GUDE, N. et al. NOX: towards an operating system for networks. **SIGCOMM Comput. Commun. Rev.**, New York, NY, USA, v.38, n.3, p.105–110, July 2008.

HAN, J. et al. Survey on NoSQL database. In: PERVASIVE COMPUTING AND APPLICATIONS (ICPCA), 2011 6TH INTERNATIONAL CONFERENCE ON. **Anais...** [S.l.: s.n.], 2011. p.363–366.

HANDIGOL, N. et al. Reproducible Network Experiments Using Container-based Emulation. In: INTERNATIONAL CONFERENCE ON EMERGING NETWORKING EXPERIMENTS AND TECHNOLOGIES, 8., New York, NY, USA. **Proceedings...** ACM, 2012. p.253–264. (CoNEXT '12).

HTTPING. Disponível em: `https://www.vanheusden.com/httping/`, Acesso em: 07/12/2015.

IDSTOOLS. Disponível em: `https://pypi.python.org/pypi/idstools`, Acesso em: 06/12/2015.

KARAPANTAZIS, S.; PAVLIDOU, F.-N. VoIP: a comprehensive survey on a promising technology. **Computer Networks**, [S.l.], v.53, n.12, p.2050 – 2090, 2009.

KETI, F.; ASKAR, S. Emulation of Software Defined Networks Using Mininet in Different Simulation Environments. In: INTELLIGENT SYSTEMS, MODELLING AND SIMULATION (ISMS), 2015 6TH INTERNATIONAL CONFERENCE ON. **Anais...** IEEE, 2015. p.205–210.

KHLIFI, H.; GREGOIRE, J.; PHILLIPS, J. VoIP and NAT/firewalls: issues, traversal techniques, and a real-world solution. **Communications Magazine, IEEE**, [S.l.], v.44, n.7, p.93–99, July 2006.

KRISHNAMURTHY, A.; CHANDRABOSE, S. P.; GEMBER-JACOBSON, A. Pratyaastha: an efficient elastic distributed sdn control plane. In: THIRD WORKSHOP ON HOT TOPICS IN SOFTWARE DEFINED NETWORKING, New York, NY, USA. **Proceedings...** ACM, 2014. p.133–138. (HotSDN '14).

LANTZ, B.; HELLER, B.; MCKEOWN, N. A Network in a Laptop: rapid prototyping for software-defined networks. In: ACM SIGCOMM WORKSHOP ON HOT TOPICS IN NETWORKS, 9., New York, NY, USA. **Proceedings...** ACM, 2010. p.19:1–19:6. (Hotnets-IX).

LARA, A.; KOLASANI, A.; RAMAMURTHY, B. Network Innovation using OpenFlow: a survey. **Communications Surveys Tutorials, IEEE**, [S.l.], v.16, n.1, p.493–512, First 2014.

LU, N. et al. Connected Vehicles: solutions and challenges. **Internet of Things Journal, IEEE**, [S.l.], v.1, n.4, p.289–299, Aug 2014.

MCKEOWN, N. et al. OpenFlow: enabling innovation in campus networks. **SIGCOMM Comput. Commun. Rev.**, New York, NY, USA, v.38, n.2, p.69–74, Mar. 2008.

MEDVED, J. et al. OpenDaylight: towards a model-driven sdn controller architecture. In: WORLD OF WIRELESS, MOBILE AND MULTIMEDIA NETWORKS (WOWMOM), 2014 IEEE 15TH INTERNATIONAL SYMPOSIUM ON A, Sydney, SY, Australia. **Anais...** IEEE, 2014. p.1–6.

MININET Overview. Disponível em: `http://mininet.org/overview/`, Acesso em: 22/11/2015.

MONTGOMERY, D. C. **Applied Statistics and Probability for Engineers**. 6.ed. [S.l.]: Wiley, 2014.

NAOUS, J. et al. Implementing an OpenFlow Switch on the NetFPGA Platform. In: ACM/IEEE SYMPOSIUM ON ARCHITECTURES FOR NETWORKING AND COMMUNICATIONS SYSTEMS, 4., New York, NY, USA. **Proceedings...** ACM, 2008. p.1–9. (ANCS '08).

NGO, M.-T.; KIM, Y. A policy enforcement architecture for a set of connected middleboxes. In: INFORMATION AND COMMUNICATION TECHNOLOGY CONVERGENCE (ICTC), 2014 INTERNATIONAL CONFERENCE ON. **Anais...** IEEE, 2014. p.790–791.

OMNES, N. et al. A programmable and virtualized network IT infrastructure for the internet of things: how can nfv sdn help for facing the upcoming challenges. In: INTELLIGENCE IN NEXT GENERATION NETWORKS (ICIN), 2015 18TH INTERNATIONAL CONFERENCE ON. **Anais...** IEEE, 2015. p.64–69.

OREBAUGH, A.; RAMIREZ, G.; BEALE, J. **Wireshark & Ethereal network protocol analyzer toolkit**. [S.l.]: Syngress, 2006.

PFLEEGER, S. Experimental design and analysis in software engineering. **Annals of Software Engineering**, [S.l.], v.1, n.1, p.219–253, 1995.

PYTHON-IPTABLES'S. Disponível em: `http://ldx.github.io/python-iptables/`, Acesso em: 05/12/2015.

ROBUCK, M. **ONF Debuts Northbound Interfaces for Intent-Based Networking**. Disponível em: `https://www.sdxcentral.com/articles/news/onf-debuts-northbound-interfaces-for-intent-based-networking/2015/09/`, Acesso em: 30/10/2015.

RYU SDN Framework. Disponível em: `https://osrg.github.io/ryu/`, Acesso em: 5/12/2015.

SALTZER, J. H.; REED, D. P.; CLARK, D. D. End-to-end Arguments in System Design. **ACM Trans. Comput. Syst.**, New York, NY, USA, v.2, n.4, p.277–288, Nov. 1984.

SEKAR, V. et al. Design and Implementation of a Consolidated Middlebox Architecture. In: USENIX CONFERENCE ON NETWORKED SYSTEMS DESIGN AND IMPLEMENTATION, 9., Berkeley, CA, USA. **Proceedings...** USENIX Association, 2012. p.24–24. (NSDI'12).

SHALIMOV, A. et al. Advanced Study of SDN/OpenFlow Controllers. In: CENTRAL AND EASTERN EUROPEAN SOFTWARE ENGINEERING CONFERENCE IN RUSSIA, 9., New York, NY, USA. **Proceedings...** ACM, 2013. p.1:1–1:6. (CEE-SECR '13).

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

SHERRY, J. et al. Making Middleboxes Someone else's Problem: network processing as a cloud service. **SIGCOMM Comput. Commun. Rev.**, New York, NY, USA, v.42, n.4, p.13–24, Aug. 2012.

SHERRY, J. et al. BlindBox: deep packet inspection over encrypted traffic. In: ACM CONFERENCE ON SPECIAL INTEREST GROUP ON DATA COMMUNICATION, 2015., New York, NY, USA. **Proceedings...** ACM, 2015. p.213–226. (SIGCOMM '15).

SIP performance. Disponível em: `http://sipp.sourceforge.net/`, Acesso em: 07/12/2015.

STIEMERLING, M.; QUITTEK, J.; CADAR, C. **NEC's Simple Middlebox Configuration (SIMCO) Protocol Version 3.0**. [S.l.]: IETF, 2006. n.4540. (Request for Comments).

STIEMERLING, M.; QUITTEK, J.; TAYLOR, T. **Middlebox Communication (MIDCOM) Protocol Semantics**. [S.l.]: IETF, 2008. n.5189. (Request for Comments).

TSHARK. `https://www.wireshark.org/docs/man-pages/tshark.html`, Acessado em: 29/10/2015.

VAN MEGGELEN, J.; MADSEN, L.; SMITH, J. **Asterisk**: the future of telephony. [S.l.]: O'Reilly Media, Inc., 2007.

WANG, A. et al. Scotch: elastically scaling up sdn control-plane using vswitch based overlay. In: ACM INTERNATIONAL ON CONFERENCE ON EMERGING NETWORKING EXPERIMENTS AND TECHNOLOGIES, 10., New York, NY, USA. **Proceedings...** ACM, 2014. p.403–414. (CoNEXT '14).

WATCHDOG documentation. Disponível em: `https://pypi.python.org/pypi/watchdog`, Acesso em: 06/12/2015.

WEIS, S. A. et al. Security and privacy aspects of low-cost radio frequency identification systems. In: **Security in pervasive computing**. [S.l.]: Springer, 2004. p.201–212.

WILCOXON, F.; KATTI, S.; WILCOX, R. A. **Critical values and probability levels for the Wilcoxon rank sum test and the Wilcoxon signed rank test**. [S.l.]: Markham Publishing Co. Chicago, 1970. 171–259p. v.1.

XUAN, L. fei; WU, P. fei. The Optimization and Implementation of Iptables Rules Set on Linux. In: INFORMATION SCIENCE AND CONTROL ENGINEERING (ICISCE), 2015 2ND INTERNATIONAL CONFERENCE ON. **Anais...** IEEE, 2015. p.988–991.

YANG, H.-Y.; LEE, K.-H.; KO, S.-J. Communication quality of voice over TCP used for firewall traversal. In: MULTIMEDIA AND EXPO, 2008 IEEE INTERNATIONAL CONFERENCE ON. **Anais...** IEEE, 2008. p.29–32.

ZHOU, D.; HUANG, B.; MO, Y. Distributed architecture of VOIP for firewall/NAT Traversing. In: WIRELESS COMMUNICATIONS, NETWORKING AND MOBILE COMPUTING, 2005. PROCEEDINGS. 2005 INTERNATIONAL CONFERENCE ON. **Anais...** IEEE, 2005. v.2, p.1160–1163.

ZHOU, W. et al. REST API Design Patterns for SDN Northbound API. In: ADVANCED INFORMATION NETWORKING AND APPLICATIONS WORKSHOPS (WAINA), 2014 28TH INTERNATIONAL CONFERENCE ON. **Anais...** IEEE, 2014. p.358–365.