Extensões na Arquitetura SDN para o Provisionamento de QoS Através do Monitoramento e Uso de Múltiplos Caminhos

Pedro Henrique Amorim Rezende

Universidade Federal de Uberlândia
Faculdade de Computação
Programa de Pós-Graduação em Ciência da Computação

Uberlândia
2016

Pedro Henrique Amorim Rezende

Extensões na Arquitetura SDN para o Provisionamento de QoS Através do Monitoramento e Uso de Múltiplos Caminhos

Dissertação de mestrado apresentada ao Programa de Pós-graduação da Faculdade de Computação da Universidade Federal de Uberlândia como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação.

Área de concentração: Ciência da Computação

Orientador: Rafael Pasquini
Coorientador: Lásaro Jonas Camargos

Uberlândia
2016

Dados Internacionais de Catalogação na Publicação (CIP)
Sistema de Bibliotecas da UFU, MG, Brasil.

R467e
2016

Rezende, Pedro Henrique Amorim, 1990-
Extensões na arquitetura SDN para o provisionamento de QoS através do monitoramento e uso de múltiplos caminhos / Pedro Henrique Amorim Rezende. - 2016.
132 f. : il.

Orientador: Rafael Pasquini.
Coorientador: Lásaro Jonas Camargos.
Dissertação (mestrado) - Universidade Federal de Uberlândia, Programa de Pós-Graduação em Ciência da Computação.
Inclui bibliografia.

1. Computação - Teses. 2. Redes de computadores - Gerência - Teses. 3. Multicasting (Redes de computadores) - Teses. 4. Redes de computadores - Software - Teses. I. Pasquini, Rafael. II. Camargos, Lásaro Jonas. III. Universidade Federal de Uberlândia, Programa de Pós-Graduação em Ciência da Computação. IV. Título.

CDU: 681.3

UNIVERSIDADE FEDERAL DE UBERLÂNDIA

FACULDADE DE COMPUTAÇÃO

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Os abaixo assinados, por meio deste, certificam que leram e recomendam para a Faculdade de Computação a aceitação da dissertação intitulada "**Extensões na Arquitetura SDN para o Provisionamento de QoS Através do Monitoramento e Uso de Múltiplos Caminhos**" por **Pedro Henrique Amorim Rezende** como parte dos requisitos exigidos para a obtenção do título de **Mestre em Ciência da Computação**.

Uberlândia, 17 de fevereiro de 2016.

Orientador: ____________________________________

Prof. Dr. Rafael Pasquini

Universidade Federal de Uberlândia

Coorientador: ____________________________________

Prof. Dr. Lásaro Jonas Camargos

Universidade Federal de Uberlândia

Banca Examinadora:

____

Prof. Dr. Maurício Ferreira Magalhães

FEEC/Unicamp

____

Prof. Dr. Paulo Roberto Guardieiro

FEELT/UFU

Este trabalho é dedicado aos meus pais.

Agradecimentos

Primeiramente, agradeço aos meus pais, Carlos e Sandra, pelo apoio e amor incondicional durante toda a minha vida.

Ao meu orientador, Rafael Pasquini, por toda a orientação, paciência e dedicação durante todo o mestrado. E, também, aos professores, Lásaro, Faina e Paulo por todos os conselhos e orientações nos trabalhos desenvolvidos.

Aos colegas de mestrado e doutorado que me deram sugestões, incentivo e ajuda durante todo este processo.

E, por fim, à Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) pela bolsa de mestrado.

"Sonhos determinam o que você quer. Ação determina o que você conquista."
(Aldo Novak)

Resumo

O aumento expressivo no número de aplicações ofertadas através das redes de computadores, bem como no volume de tráfego encaminhado pela rede, têm dificultado assegurar níveis adequados de serviço para os usuários. A oferta de Qualidade de Serviço (QoS), honrando parâmetros especificados em Acordos de Nível de Serviço (SLA), estabelecidos entre provedores de serviço e seus clientes, constitui uma tradicional e ampla área de pesquisa em redes de computadores. Inúmeras propostas de esquemas para provimento de QoS foram apresentadas nas últimas três décadas, mas o escopo de atuação destas propostas sempre é limitado devido a fatores diversos, incluindo o desenvolvimento restrito de hardware e software de rede, geralmente exclusivos a um fabricante. O surgimento das Redes Definidas por Software (SDN), junto com o amadurecimento de sua materialização principal, o protocolo OpenFlow, permitiu a separação entre hardware e software de rede, através de uma arquitetura que prevê um plano de controle e um plano de dados. Este cenário flexibiliza as redes de computadores, permitindo que novas abstrações sejam aplicadas ao hardware compondo o plano de dados, através do desenvolvimento de novas peças de software que são executadas no plano de controle. Esta dissertação investiga a oferta de QoS através do uso e da extensão da Arquitetura SDN. A partir da criação de dois novos módulos, um para efetuar o monitoramento do plano de dados, SDNMon, e um segundo, MP-Routing, desenvolvido para determinar o uso de múltiplos caminhos no encaminhamento de tráfego referente a um fluxo, demonstramos neste trabalho que algumas métricas de QoS especificadas em SLAs, como a largura de banda, podem ser atendidas. Ambos os módulos foram implementados e avaliados através de um protótipo. Os resultados de avaliações referentes a diversos aspectos de ambos os módulos propostos são apresentados nesta dissertação, demonstrando a precisão obtida pelo módulo de monitoramento SDNMon e os ganhos na utilização de multiplos caminhos definidos pelo MP-Routing ao encaminhar fluxos de dados através de uma SDN.

Palavras-chave: Acordo de Nível de Serviço (SLA), Monitoramento, Multicaminhos, OpenFlow, Qualidade de Serviço (QoS), Redes Definidas por Software.

Abstract

The substantial increase in the number of applications offered through the computer networks, as well as in the volume of traffic forwarded through the network, have hampered to assure adequate service level to users. The Quality of Service (QoS) offer, honoring specified parameters in Service Level Agreements (SLA), established between the service providers and their clients, composes a traditional and extensive computer networks' research area. Several schemes proposals for the provision of QoS were presented in the last three decades, but the acting scope of these proposals is always limited due to some factors, including the limited development of the network hardware and software, generally belonging to a single manufacturer. The advent of Software Defined Networking (SDN), along with the maturation of its main materialization, the OpenFlow protocol, allowed the decoupling between network hardware and software, through an architecture which provides a control plane and a data plane. This eases the computer networks scenario, allowing that new abstractions are applied in the hardware composing the data plane, through the development of new software pieces which are executed in the control plane. This dissertation investigates the QoS offer through the use and extension of the SDN architecture. Based on the proposal of two new modules, one to perform the data plane monitoring, SDNMon, and the second, MP-ROUTING, developed to determine the use of multiple paths in the forwarding of data referring to a flow, we demonstrated in this work that some QoS metrics specified in the SLAs, such as bandwidth, can be honored. Both modules were implemented and evaluated through a prototype. The evaluation results referring to several aspects of both proposed modules are presented in this dissertation, showing the obtained accuracy of the monitoring module SDNMon and the QoS gains due to the utilization of multiple paths defined by the MP-Routing, when forwarding data flow through the SDN.

Keywords: Monitoring, Multipath, OpenFlow, Quality of Service (QoS), Service Level Agreement (SLA), Software Defined Networking (SDN).

Lista de ilustrações

Figura 1 – Visão lógica da Arquitetura SDN. 34
Figura 2 – Principais campos de uma Entrada de Fluxo. 36
Figura 3 – Campos de uma Entrada de Grupo. 36
Figura 4 – Campos de uma Entrada de *Meter*. 37
Figura 5 – Comutador OpenFlow. 37
Figura 6 – Classificação da granularidade de divisão de tráfego. Extraída de (PRABHAVAT et al., 2012). 41
Figura 7 – Uma instância da Arquitetura SDN com as extensões de monitoramento. 60
Figura 8 – Arquitetura do SDNMon. 60
Figura 9 – Coleta através do sFlow. 65
Figura 10 – Cálculo da vazão. 65
Figura 11 – Cálculo do atraso. 66
Figura 12 – Ilustração da Aplicação de Monitoramento. 68
Figura 13 – Tabela de Fluxos e a Vazão do Fluxo e das Portas. 69
Figura 14 – Arquitetura MP-Routing. 70
Figura 15 – Fluxograma do Módulo MP-Routing. 72
Figura 16 – Extensão do Comutador OpenFlow. 78
Figura 17 – Escalonador - Encaminhamento no Nível do *Bucket* e do Grupo. . . . 78
Figura 18 – Visão de alto nível das extensões à Arquitetura SDN. 79
Figura 19 – Detalhes das extensões propostas no Plano de Controle. 80
Figura 20 – Cenário utilizado nas avaliações do SDNMon. 84
Figura 21 – Variação da vazão em diferentes intervalos de coleta do mecanismo de *polling*. 85
Figura 22 – Variação da vazão em função da taxa de amostragem N do sFlow. . . . 87
Figura 23 – Variação da vazão em diferentes aplicações legadas. 89
Figura 24 – Análise do atraso sem a ocorrência de descarte de pacotes. 90
Figura 25 – Análise do atraso quando há a ocorrência de descarte de pacotes. . . . 91

Figura 26 – Quadro Ethernet. 92
Figura 27 – Pacote IP contendo uma *Statistics Request*. 93
Figura 28 – Pacote IP contendo uma *Statistics Reply*. 93
Figura 29 – *Statistics Reply* contendo Estatísticas de Fluxos. 93
Figura 30 – Multicaminhos com 04 Saltos por Caminho vs Caminho Único. 98
Figura 31 – Multicaminhos com 05 Saltos por Caminho vs Caminho Único. 98
Figura 32 – Multicaminhos com Múltiplos Saltos por Caminho vs Caminho Único. 99
Figura 33 – Diagrama de Classe do SDNMon. 120
Figura 34 – Diagrama de Classe do MP-Routing. 121
Figura 35 – Exemplo de uma mensagem *Statistics Request*. 124
Figura 36 – Exemplo de uma mensagem *Statistics Reply*. 125

Lista de tabelas

Tabela 1 – Contadores do Fluxo. 39
Tabela 2 – Contadores da Porta. 39
Tabela 3 – Dados consolidados referentes à análise do mecanismo de *polling*. . . . 86
Tabela 4 – Dados consolidados referentes à análise do sFlow. 88
Tabela 5 – Dados consolidados referentes às transmissões UDP e TCP. 88
Tabela 6 – Custos das Estatísticas de Fluxo. 94
Tabela 7 – IPerf/SCP - Multicaminhos com 04 Saltos/Caminho + Tráfego de Fundo. 100
Tabela 8 – IPerf/SCP - Caminho Único vs Multicaminhos com 04 Saltos/Caminho. 101
Tabela 9 – IPerf/SCP - Multicaminhos com 05 Saltos/Caminho + Tráfego de Fundo. 102
Tabela 10 – IPerf/SCP - Caminho Único vs Multicaminhos com 05 Saltos/Caminho. 103
Tabela 11 – IPerf/SCP - Multicaminhos com 03 a 06 Saltos/Caminho + Tráfego de Fundo. 103
Tabela 12 – IPerf/SCP - Multicaminhos com 03 a 06 Saltos/Caminho. 104
Tabela 13 – IPerf/SCP - Multicaminhos com 03 a 06 Saltos/Caminho + Tráfego de Fundo + Perdas de Pacote de (1%). 105
Tabela 14 – IPerf/SCP - Multicaminhos com 03 a 06 Saltos/Caminho + Perda de Pacotes de 1%. 105
Tabela 15 – Entrada de Fluxo com novos contadores. 128

Lista de siglas

API *Application Programming Interface*

ARP *Address Resolution Protocol*

ATM *Asynchronous Transfer Mode*

AFLCMF *Adaptive Flow-Level Load Control Scheme for Multipath Forwarding*

DH *Direct Hashing*

DSCP *Differentiated Services Code Point*

ECMP *Equal Cost Multi Path*

EDPF *Earliest Delivery Path First*

EIGRP *Enhanced Interior Gateway Routing Protocol*

FCS *Frame Check Sequence*

FEC *Forwarding Equivalency Class*

FS *Fast Switching*

FLARE *Flowlet Aware Routing Engine*

GMPLS *Generalized Multi-Protocol Label Switching*

HTTP *Hypertext Transfer Protocol*

IETF *Internet Engineering Task Force*

IP *Internet Protocol*

JSON *JavaScript Object Notation*

LBPF *Load Balancing for Parallel Forwarding Protocol*

LLDP *Link Layer Discovery Protocol*

LDP *Label Distribution Protocol*

LDM *Load Distribution over Multipath*

LSR *Label Switch Router*

LSP *Label Switch Path*

LER *Label Edge Router*

MAC *Media Access Control*

MPLS *Multi-Protocol Label Switching*

MPTCP *MultiPath TCP*

MBD-/ADBR *Progressive Multiple Bin Disconnection with Absolute Difference Bin Reconnection*

NAT *Network Address Translation*

NOP *No-Operation*

NTP *Network Time Protocol*

OVS Open vSwitch

ofsoftswitch13 OpenFlow 1.3 *Software Switch*

OSI *Open Systems Interconnection*

OSPF *Open Shortest Path First*

PHB *Per-Hop Behavior*

PNNI *Private Network-to-Network Interface*

PBP-RR *Packet-By-Packet Round-Robin*

QoS *Quality of Service*

QUIC *Quick UDP Internet Connections*

RIP *Routing Information Protocol*

RSVP *Resource ReSerVation Protocol*

RSTP *Rapid Spanning Tree Protocol*

SCP *Secure Copy*

SDN *Software Defined Networking*

STP *Spanning Tree Protocol*

SLA *Service Level Agreement*

SSL *Secure Socket Layer*

SRR *Surplus Round Robin*

TCP *Transmission Control Protocol*

TLS *Transport Layer Security*

TOS *Type of Service*

TTL *Time to Live*

UDP *User Datagram Protocol*

URI *Uniform Resource Identifier*

WRR *Weighted Round Robin*

VLAN *Virtual Local Area Network*

VLC *Video Lan Client*

VMs *Virtual Machines*

Sumário

1 INTRODUÇÃO . . . 27
1.1 Motivação . . . 30
1.2 Objetivos e Desafios da Pesquisa . . . 31
1.3 Hipótese . . . 32
1.4 Contribuições . . . 32
1.5 Organização da Dissertação . . . 32

2 FUNDAMENTAÇÃO TEÓRICA . . . 33
2.1 Redes Definidas por Software . . . 33
2.2 OpenFlow . . . 34
2.2.1 Comutadores OpenFlow . . . 35
2.2.2 Fluxo de dados . . . 35
2.2.3 Contadores . . . 38
2.2.4 Controladores . . . 39
2.3 Multi-Caminhos . . . 39
2.3.1 Divisão de Tráfego . . . 41
2.3.2 Seleção de Caminhos . . . 42
2.3.3 Problemas de Desempenho . . . 42
2.4 Qualidade de Serviço(QoS) . . . 44
2.4.1 Propostas tradicionais de QoS apresentadas pelo IETF . . . 44
2.4.2 SDN e Protocolos Tradicionais . . . 46

3 TRABALHOS RELACIONADOS . . . 49
3.1 Monitoramento . . . 49
3.1.1 *OpenNetMon* . . . 49
3.1.2 *OpenSample* . . . 50
3.1.3 *Latency* . . . 50
3.1.4 *PayLess* . . . 50

3.2 Multi-Caminhos . . . 51
3.2.1 *Info-unaware* . . . 51
3.2.2 Informação do pacote . . . 52
3.2.3 Condição do tráfego . . . 52
3.2.4 Condição da rede . . . 53
3.2.5 Condição do tráfego e da rede . . . 54
3.2.6 Multi-caminhos com SDN . . . 55
3.2.7 QUIC . . . 56

4 SDNMON E MP-ROUTING . . . 59
4.1 SDNMon . . . 59
4.1.1 Arquitetura do Módulo SDNMon . . . 59
4.1.2 Mecanismo de Monitoramento baseado em *Polling* . . . 62
4.1.3 Mecanismo de Monitoramento baseado em *Push* . . . 64
4.1.4 Cálculo da Vazão . . . 64
4.1.5 Cálculo do Atraso . . . 65
4.1.6 Aplicação de Monitoramento . . . 67
4.2 MP-Routing . . . 68
4.2.1 Arquitetura do módulo MP-Routing . . . 69
4.2.2 Funcionamento do MP-Routing . . . 71
4.2.3 Extensão Open vSwitch . . . 75
4.3 Extensões propostas na Arquitetura SDN . . . 79

5 EXPERIMENTOS E ANÁLISE DOS RESULTADOS . . . 81
5.1 Ferramentas . . . 81
5.1.1 Floodlight . . . 81
5.1.2 Mininet . . . 82
5.1.3 Comutadores . . . 82
5.1.4 sFlow . . . 82
5.1.5 LoxiGen . . . 82
5.2 Avaliações do SDNMon . . . 83
5.2.1 Objetivos das Avaliações do SDNMon . . . 83
5.2.2 Ambiente e Cenário de Testes do SDNMon . . . 83
5.2.3 Experimentos com o SDNMon . . . 83
5.2.4 Resumo das Avaliações do SDNMon . . . 95
5.3 Avaliações do MP-Routing . . . 96
5.3.1 Objetivos das Avaliações do MP-Routing . . . 96
5.3.2 Ambiente de Testes do MP-Routing . . . 96
5.3.3 Cenários Avaliados no MP-Routing . . . 96
5.3.4 Experimentos com o MP-Routing . . . 99

5.3.5 Resumo das avaliações do MP-Routing 106

6 CONCLUSÃO 107
6.1 Principais Contribuições 107
6.2 Trabalhos Futuros 108
6.3 Contribuições em Produção Bibliográfica 109

REFERÊNCIAS 111

APÊNDICES 117

APÊNDICE A – DIAGRAMAS DE CLASSES 119

APÊNDICE B – PEDIDO E RESPOSTA DE ESTATÍSTICAS . 123

APÊNDICE C – EXTENSÃO DO PROTOCOLO OPENFLOW . 127
C.1 Adição de Contadores 128

Capítulo 1

Introdução

Redes de computadores são compostas por diversos tipos de equipamentos, desde roteadores e comutadores, até *middleboxes*, como *firewalls*, tradutores de endereços de Internet e servidores de balanceamento de carga. A demanda por mais recursos imposta por novas aplicações foi um dos fatores que sempre contribuiu para a inovação e o desenvolvimento de novos dispositivos e arquiteturas para as redes de computadores. A Internet, a principal rede de computadores, foi projetada para ser uma rede de melhor esforço, ou seja, ela não garantiria que o usuário da rede teria os seus requisitos de *Quality of Service* (QoS) atendidos. Com a padronização do *Internet Protocol* (IP) como protocolo principal de comunicação, foi possível aplicar algumas técnicas para ofertar QoS.

A oferta de QoS (CHEN; NAHRSTED, 1998) é uma peça fundamental e complexa nas redes de computadores. A noção de QoS vem sendo proposta para assegurar qualitativamente ou quantitativamente o contrato de desempenho entre a provedora de serviço e as aplicações dos usuários. O requisito de QoS em uma conexão é dado como um conjunto de restrições, como largura de banda e atraso. A restrição de largura de banda define que os enlaces que pertencem a um caminho devem assegurar uma banda mínima e a restrição de atraso garante que o atraso em um caminho não pode ser maior que um determinado limite.

As principais técnicas para prover QoS são: *over-provisioning* (HUANG; GUERIN, 2005), engenharia de tráfego (AKYILDIZ et al., 2014) e diferenciar o tratamento dos pacotes nos dispositivos de encaminhamento (VALI et al., 2004). A primeira técnica é usada para proteger o desempenho da rede contra variações de tráfego, como falhas ou oscilações temporárias. Esta técnica propõe que a largura de banda de uma aresta seja maior que o tráfego esperado mais as suas variações. A segunda técnica irá otimizar o desempenho da rede através da análise, provisionamento e regulação do comportamento do dado transmitido. A engenharia de tráfego é uma área bastante ampla, englobando os mecanismos de multi-caminhos, monitoramento de rede, análise, caracterização, gerenciamento de tráfego, entre outros.

A última técnica irá diferenciar o encaminhamento dos pacotes através de suas ca-

racterísticas, com isso os pacotes serão classificados, enfileirados e escalonados de formas diferentes dentro de um roteador. Aqui entra também o conceito de sinalização de QoS, onde recursos serão reservados de acordo com um pedido de serviço de QoS. Podemos citar dois modelos bastante conhecidos que utilizam mecanismo de sinalização, o Int-Serv (BRADEN; CLARK; SHENKER, 1994) e o *Multi-Protocol Label Switching* (MPLS) (ROSEN; VISWANATHAN; CALLON, 2001).

As redes de computadores podem possuir muitos caminhos redundantes, mas na grande maioria dos casos um mesmo tipo de tráfego irá passa por apenas um caminho. As principais razões desse encaminhamento incluem a baixa complexidade de roteamento e os pacotes chegarem mais ordenados no usuário final. Entretanto, caso um mesmo fluxo fosse enviado por múltiplos caminhos disjuntos, seria possível aproveitar melhor os recursos da rede e, ainda, ajudar no atendimento do Acordo de Nível de Serviço (ANS ou *Service Level Agreement* (SLA)) entre a provedora de serviço e o cliente (PRABHAVAT et al., 2012).

Algumas abordagens foram propostas para prover multicaminhos, como o *Equal Cost Multi Path* (ECMP) (HOPPS, 2000), o *MultiPath TCP* (MPTCP) (FORD et al., 2011) e o *Quick UDP Internet Connections* (QUIC) (JANA; SWETT, 2015). O ECMP usa um *hash* estatístico baseado em uma tupla de cinco campos e envia o tráfego pela interface determinada pela função de *hash*, até o fim do mesmo. Cada fluxo, de acordo com a respectiva tupla de cinco campos, poderá seguir por um caminho distinto através da rede, distribuindo os fluxos em multiplos caminhos. As limitações incluem o fato dos pacotes de um mesmo fluxo sempre passarem por um único caminho e, também, como a escolha da interface é baseada em uma função de *hash*, sempre que novos envios de fluxos com a mesma tupla de cinco campos forem efetuados, é possível que sempre passem por um mesmo caminho.

O protocolo MPTCP deriva um fluxo TCP em subfluxos e cada um destes será, em tese, transmitido através de um caminho distinto. O MPTCP pode realmente dividir um mesmo tráfego em ambientes controlados, por exemplo, uma interface do *host* pode estar em uma rede local cabeada, e uma outra, em uma rede local sem fio. Entretanto, como o MPTCP não possui controle direto sobre os dispositivos de rede, é possível que todos os subfluxos passem pelos mesmos dispositivos de rede, como no caso da Internet. Além disso, possui limitações por atuar apenas em tráfego TCP e atuar no modelo *host centric*, pois seria necessário alterar os usuários finais para permitir o uso do MPTCP.

O QUIC é um protocolo de transporte de *stream* multiplexado, construído em cima do protocolo *User Datagram Protocol* (UDP), que tem como principal objetivo reduzir a latência na Web. Embora não possua um mecanismo de encaminhamento de pacotes por múltiplos caminhos implementado, tal como, cada *stream* de uma mesma conexão passando por caminhos distintos, a natureza do QUIC já garante parcialmente o uso de múltiplos caminhos, com o campo *Connection ID* disponível em seu cabeçalho. O *Con-*

nection ID permite que um dispositivo mova livremente por diversas redes sem precisar reestabelecer uma nova conexão com o servidor, com isso, nada impede que um mesmo tráfego passe por diferentes caminhos durante toda a sua transmissão. Entretanto, o QUIC não garante que esta nova rede escolhida irá garantir o SLA do usuário.

Nos últimos anos, o tema Redes Definidas por Software (SDN ou *Software Defined Networking*, em inglês) (KREUTZ et al., 2014) atraiu atenção tanto da indústria como da academia. A Arquitetura SDN tem como principal objetivo desacoplar o Plano de Controle do Plano de Dados. O Plano de Dados contém os dispositivos de rede responsáveis pelo encaminhamento de pacotes. O Plano de Controle contém um ou mais controladores responsáveis pelo controle dos dispositivos, e também, pelo oferecimento às aplicações de serviços relacionados à rede.

O OpenFlow é o protocolo mais conhecido e utilizado na Arquitetura SDN. Atualmente ele permite que o controlador SDN comunique-se com os dispositivos de rede, definindo como o encaminhamento de pacotes deve acontecer e, também, consultando o estado dos dispositivos de rede. Estas consultas permitem coletar estatísticas dos comutadores ou roteadores em diversas granularidades, tais como por fluxo, por porta ou por fila (ONF, 2013).

Com o surgimento e a constante evolução do protocolo OpenFlow, junto com o aumento das pesquisas e da adoção das SDNs na indústria e na academia, foi possível que novos mecanismos fossem criados ou aprimorados para resolver problemas já conhecidos nas redes de computadores, como o monitoramento e o encaminhamento por múltiplos caminhos. Tendo em vista a importância, e as infinitas possibilidades de uso das SDNs, o presente trabalho propõe extensões à Arquitetura SDN visando atender a QoS em uma rede OpenFlow.

Devido a necessidade de auxiliar no atendimento do SLA do usuário e o uso mais eficiente dos recursos da rede, nós propomos uma plataforma que usa como base a Arquitetura SDN e adota técnicas de engenharia de tráfego para auxiliar na oferta de QoS. Com isso, a nossa proposta contempla a criação de uma plataforma composta por dois módulos principais, o módulo de monitoramento e o de multi-caminhos. A nossa plataforma está implementada no controlador Floodlight (FLOODLIGHT, 2015), um controlador baseado em Java, multi-tarefa, modular e de distribuição livre. O Floodlight oferece suporte completo para o OpenFlow 1.3 e é possível fornecer serviços às aplicações através de sua Restful API.

Cada módulo do Floodlight possui funções específicas, e à medida que haja a necessidade de inserir novas funcionalidades ao controlador, basta registrar no controlador um novo módulo. Portanto, a plataforma desenvolvida aproveita-se desta característica modular do controlador para criar os dois módulos necessários para o desenvolvimento deste trabalho.

O monitoramento é fundamental para uma rede, incluindo este novo cenário SDN.

As informações do estado da rede podem ser coletados usando algumas técnicas, tais como *push* e *polling*. Na primeira técnica, os dispositivos da rede enviam informações sobre seu estado quando acontece algum evento. Já na segunda técnica, um elemento é responsável por requisitar aos dispositivos de rede informações relativas ao seu estado, periodicamente. O monitoramento permite, por exemplo, que sejam coletadas informações referentes à banda remanescente, atraso de propagação e perdas de pacotes (BULUT et al., 2009)(AKYILDIZ et al., 2014). Existem muitos protocolos de comunicação entre o Plano de Dados e o Plano de Controle, tais como o NETCONF (ENNS et al., 2011), SNMP (CASE et al., 1990) e o OpenFlow (MCKEOWN et al., 2008).

O módulo de monitoramento proposto é responsável por coletar, processar e armazenar métricas importantes para a oferta de QoS, tais como a largura de banda, o atraso de propagação e as perdas de pacotes. Além disso, o módulo permite ao administrador escolher um mecanismo de monitoramento, podendo ser através do sFlow (PHAAL; PANCHEN; MCKEE, 2001)(SFLOW, 2015) ou de consultas explícitas, o primeiro faz *push* de informações e o segundo faz um *polling*. Com isso, o controlador terá uma visão mais enriquecida da rede e poderá tomar ações de maneira mais rápida e precisa.

O módulo de multi-caminhos proposto é responsável pelo roteamento. O algoritmo de multi-caminhos proposto permite a este módulo verificar por quais caminhos disjuntos é possível transmitir um determinado tráfego de modo que a largura de banda seja honrada durante toda a transmissão do fluxo. O módulo de multi-caminhos prioriza o uso de um caminho único, e na ausência deste caminho único, oferta um conjunto de caminhos, os impactos desta abordagem são avaliados no Capítulo 5.

Embora os dois módulos criados no Floodlight sejam as peças mais importantes do trabalho, o nosso trabalho não se limita apenas a eles. Os dispositivos do Plano de Dados foram estendidos com novos contadores e com um algoritmo de escalonamento baseado em grupos e *buckets*. Uma aplicação de monitoramento de fácil entendimento e interação também foi criada para o administrador de rede verificar o estado atual da rede.

1.1 Motivação

A oferta de QoS é um dos aspectos mais importantes das Redes de Computadores e, sem ela, seria impossível assegurar o desempenho da rede para os provedores e servidores de serviços, como também, para o usuário final. A provisão de QoS é uma área bastante ampla, composta por inúmeras técnicas e tecnologias, entretanto, a oferta de QoS tem um objetivo bem claro, que é assegurar qualitativamente ou quantitativamente o compromisso feito entre o fornecedor de QoS e o solicitante do serviço de QoS.

Desde os primórdios da oferta de QoS, até a atualidade, muitas técnicas e dispositivos foram desenvolvidos para assegurar que o acordo entre as partes envolvidas na comunicação fosse honrado. Entretanto, a oferta de QoS é algo bastante complexo, e nem sempre

é atendida completamente, pois são necessários dispositivos caros; protocolos complexos, dependentes um do outro e, em muitos casos, são proprietários, inviabilizando o acesso ao código fonte pela comunidade; há a troca constante de informação na rede; existe a necessidade de serem precisos; e, a cada dia que passa, mais dispositivos são conectados nas redes, alguns móveis, outros não, o que deixa o fardo da garantia de QoS ainda mais pesado.

A Arquitetura SDN surge como um caminho interessante para auxiliar na oferta de QoS, embora as SDNs não estejam limitadas apenas ao QoS, elas podem, sim, ser usadas neste contexto. Devido à separação do Plano de Controle do Plano de Dados, proposto nas SDNs, o controlador, além de ter uma visão completa rede, também será o cérebro da rede, ou seja, tomará todas as decisões relativas ao roteamento, segurança, acesso, balanceamento de carga, entre outros. Posteriormente, o controlador irá apenas inserir as funções nos dispositivos de rede. Com isso, é possível atacar diversos problemas da oferta de QoS, como os protocolos propretários, a troca constante de informação na rede e a precisão para alterar o estado da rede.

1.2 Objetivos e Desafios da Pesquisa

O objetivo principal da pesquisa é auxiliar na oferta de QoS dos usuários finais em SDN. Para atingir nosso objetivo, o trabalho proposto irá focar em dois mecanismos importantes que visa atender a QoS, o monitoramento da rede e o encaminhamento de tráfego por múltiplos caminhos.

Os objetivos secundários incluem: a criação de uma aplicação para a visualização de informações relativas à rede; a adição de novos contadores nos comutadores OpenFlow, para uma visão mais precisa e acurada da rede; e a adição de um escalonador nos comutadores OpenFlow, para realizar o encaminhamento preciso através de múltiplos caminhos.

As funcionalidades dos comutadores podem variar de acordo com a versão do OpenFlow, então um dos desafios é verificar uma versão do OpenFlow que forneça, por exemplo, diversos contadores de estatística do dispositivo, para alimentar o mecanismo de monitoramento; funcionalidades que permitam o uso de multi-caminhos no dispositivo, como os grupos; e que a extensão de alguma funcionalidade no comutador possa ser feita em um tempo razoável. Um outro desafio é que ambos os mecanismos propostos neste trabalho, monitoramento e multi-caminhos, devem funcionar, pois o segundo módulo depende do primeiro, e não é possível auxiliar na oferta de QoS apenas com o mecanismo de monitoramento.

1.3 Hipótese

Os mecanismos de monitoramento e multi-caminhos auxiliam na oferta de QoS na Arquitetura SDN.

1.4 Contribuições

1. Módulo de Monitoramento (SDNMon)
2. Módulo de Multi-Caminhos (MP-Routing)
3. Extensão do Comutador OpenFlow
    - 3.1. Novos contadores de estatísticas
    - 3.2. Algoritmo de Escalonamento
4. Aplicação de Monitoramento

1.5 Organização da Dissertação

O Capítulo 2 detalha os fundamentos teóricos relacionados a este trabalho, apresentando, primeiramente, o conceito de Redes Definidas por Software, dando mais ênfase ao protocolo OpenFlow. Posteriormente, discute sobre monitoramento, multi-caminhos e alguns protocolos, desenvolvidos pelo IETF, para a oferta da QoS.

O Capítulo 3 apresenta os trabalhos relacionados que foram considerados para o desenvolvimento desta dissertação, detalhando trabalhos relativos ao monitoramento e multi-caminhos.

O Capítulo 4 apresenta a proposta desta dissertação, como os módulos criados no controlador, a aplicação de monitoramento e as extensões realizadas nos dispositivos.

O Capítulo 5 detalha as ferramentas usadas para a construção de nossa proposta e apresenta os resultados obtidos com o uso de nossa proposta. São realizados experimentos nos dois módulos criados no controlador em conjunto com a extensão do comutador OpenFlow. Também são detalhadas as topologias e as máquinas usadas nos testes.

O Capítulo 6 apresenta as conclusões desta dissertação, resumindo as contribuições principais e apontando alguns dos trabalhos futuros.

Três apêndices finalizam o trabalho. O Apêndice A detalha os diagramas de classe dos módulos propostos neste trabalho. O Apêndice B apresenta mensagens do tipo *Statistic Request* e *Statistic Reply*. Por fim, o Apêndice C detalha a extensão do protocolo OpenFlow, com a inserção de novos contadores.

Capítulo 2

Fundamentação Teórica

Esta seção apresenta um breve resumo da literatura associada ao desenvolvimento desta dissertação. A Seção 2.1 descreve o conceito de Redes Definidas por Software, o ambiente central deste trabalho. A Seção 2.2 apresenta o protocolo mais conhecido e utilizado na Arquitetura SDN, o OpenFlow, descrevendo seus principais componentes e funcionalidades. A Seção 2.3 descreve o encaminhamento de pacotes em múltiplos caminhos. Finalmente, a Seção 2.4 apresenta alguns dos esforços do IETF para ofertar QoS, indicando brevemente como o OpenFlow adequa-se a estes esforços.

2.1 Redes Definidas por Software

As Redes Definidas por Software ou *Software Defined Networking* (SDN) (FEAMSTER; REXFORD; ZEGURA, 2014) tornaram-se mais populares e palpáveis nos últimos anos, principalmente com o surgimento do OpenFlow, entretanto muito de suas idéias já existem há mais de vinte anos. O SDN utiliza conceitos das antigas redes telefônicas e das *active networks* (TENNENHOUSE et al., 1997), onde na primeira era usado uma separação entre o Plano de Controle e o Plano de Dados para simplificar o gerenciamento da rede e o oferecimento de serviços, e na última eram injetados pacotes na rede para modificar o comportamento dos dispositivos.

O SDN está mudando o modo como redes são modeladas e gerenciadas. O SDN tem duas características principais, a primeira é que separa o Plano de Controle (que decide como encaminhar o tráfego) do Plano de Dados (que encaminha o tráfego de acordo com as decisões do Plano de Controle). A segunda é que o SDN consolida o Plano de Controle, pois um único programa de controle atua sobre múltiplos dispositivos de rede no Plano de Dados.

Os programadores têm usado a plataforma SDN para criar diversos tipos de aplicações, como controle de acesso dinâmico, servidor de balanceamento de carga, virtualização de rede e também para migração de máquinas virtuais e mobilidade do usuário. Além disso,

os controladores foram evoluindo e ficando cada vez mais robustos, precisos e oferecendo uma interface mais amigável ao programador e usuário final.

A Figura 1 mostra a Arquitetura Lógica SDN, que é composta pelas camadas de Infraestrutura, Controle e Aplicação. A camada de Infraestrutura é composta pelos dispositivos de rede e é responsável pelo encaminhamento de pacotes. A camada de Controle atua sobre os dispositivos de rede, através de uma *Application Programming Interface* (API) bem definida, e oferece serviços. A camada de Aplicação é responsável por acessar os recursos da camada de controle, processar e oferecer ao usuário final os recursos sob o formato de aplicações. A comunicação entre as camadas ocorre através de APIs, a API entre a camadas de Controle e Aplicação é chamada de *northbound* e entre as camadas de Controle e Infraestrutura é chamada de *southbound*.

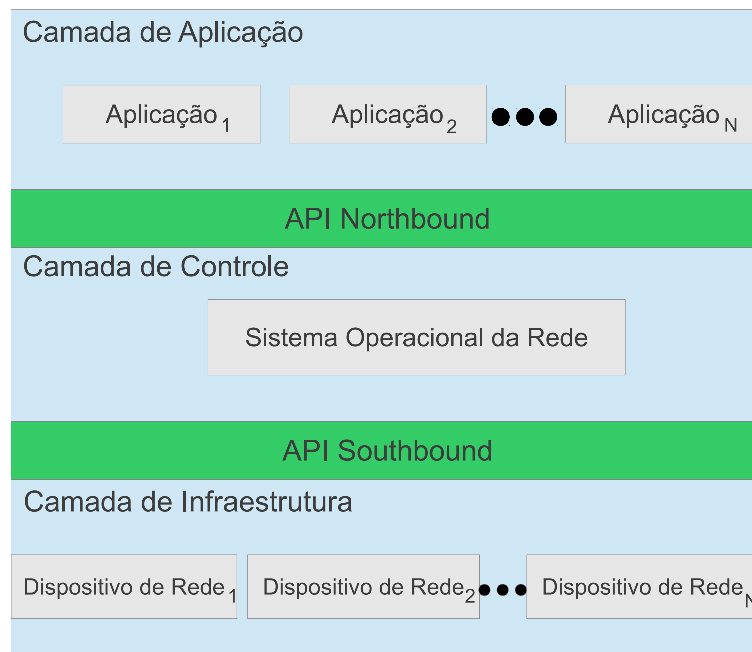


Figura 1 – Visão lógica da Arquitetura SDN.

2.2 OpenFlow

O OpenFlow (MCKEOWN et al., 2008) é a materialização atual de SDN. O OpenFlow oferece o acesso ao Plano de Dados da rede ao mesmo tempo em que provê uma API simples e comum para o controlador da rede (ROTSOS et al., 2012). Sua origem tem vínculos com o projeto Ethane (CASADO et al., 2007), no ano de 2006, e provavelmente a característica mais marcante do OpenFlow é sua forte adoção na indústria. Atualmente, a versão mais estável e usada é a 1.3 (ONF, 2013). Com o passar dos anos, o SDN tem ganhado muita significância e popularidade na indústria, fazendo com que muitos comutadores comerciais suportem a API OpenFlow, tais como HP, NEC, Pronto e a brasileira Datacom.

Um comutador OpenFlow tem uma tabela de regras de encaminhamento de pacotes, onde cada regra tem um padrão (que faz um *match* nos bits do cabeçalho do pacote), uma lista de ações (como *drop*, *flood*, *forward* e *modify*), um conjunto de contadores (para registrar o número de bytes e pacotes) e uma prioridade (para diferenciar regras com padrões sobrepostos). Ao receber um pacote, um comutador OpenFlow identifica a regra de maior prioridade que faz um *match* neste pacote, realiza as ações associadas e incrementa os contadores.

2.2.1 Comutadores OpenFlow

O número de comutadores que aceitam o protocolo OpenFlow vem aumentando constantemente, permitindo o uso comercial e experimental do OpenFlow. No final de 2009, a especificação do protocolo OpenFlow foi lançada na sua primeira versão estável (1.0), sendo esta a primeira implementada nos dispositivos.

OpenFlow não está limitado apenas ao *hardware*. Alguns comutadores virtuais também aceitam o protocolo OpenFlow, tais como o Open vSwitch (OVS) (PFAFF et al., 2009) e o OpenFlow 1.3 *Software Switch* (ofsoftswitch13) (FERNANDES; ROTHENBERG, 2014), sendo que o primeiro é mais usado para pesquisa e também por empresas, já o segundo é um comutador brasileiro ainda não tão usado quanto o primeiro. Atualmente, os comutadores virtuais tendem a ser mais precisos e robustos que os físicos, além de que a maioria deles são *open-source*, suportando o avanço das pesquisas.

Uma alternativa para implementar redes com diferentes topologias é usar o Mininet (LANTZ; HELLER; MCKEOWN, 2010), um ambiente de virtualização no nível do sistema operacional. O Mininet gera de maneira rápida uma topologia composta por *hosts*, comutadores, enlaces ligando os dispositivos e também é possível configurar algumas métricas, como o atraso, banda máxima e a perda de pacotes de um enlace. Os comutadores gerados pelo Mininet incluem o Open vSwitch e o ofsoftswitch13, portanto suporta o protocolo OpenFlow (KREUTZ et al., 2014).

2.2.2 Fluxo de dados

O fluxo de dados detalha como é o processo de encaminhamento de um pacote dentro de um comutador OpenFlow(ONF, 2013). Antes de entrar em detalhes sobre o fluxo de dados, é importante descrever alguns componentes fundamentais de um comutador OpenFlow.

Um comutador OpenFlow possui três tipos de tabelas: A tabela de fluxo (*flow table*), de grupo (*group table*) e de *meter* (*meter table*). Um comutador OpenFlow pode ter uma ou mais tabelas de fluxo, onde cada tabela de fluxo contém múltiplas entradas de fluxo. Uma entrada de fluxo (*flow entry*) possui três campos principais: o campo de *match*, de instrução e os contadores de estatísticas, como mostrado na Figura 2. O campo de

match, ou regra, é usado para verificar em qual entrada de fluxo o pacote analisado se encaixa. Campos do cabeçalho do pacote, geralmente das camadas 1, 2 e 3 da Arquitetura TCP/IP, são extraídos e comparados com os valores do campo de *match* de cada entrada de fluxo, em ordem de prioridade de fluxo, até que uma regra seja idêntica aos valores extraídos do pacote. Ao achar a entrada de fluxo, o campo de instrução executa e define uma ou mais ações que são aplicadas ao pacote.

Figura 2 – Principais campos de uma Entrada de Fluxo.

A tabela de grupo é composta por entradas de grupos (*group entries*), onde cada uma destas é composta por quatro campos: o identificador de grupo, o tipo de grupo, contadores e um conjunto de *buckets*, como detalhado na Figura 3. O identificador de grupo é um inteiro positivo que identifica unicamente uma entrada de grupo, o tipo do grupo define a semântica do grupo e o conjunto dos *buckets* aplica as ações aos pacotes. Os tipos de grupo são detalhados a seguir:

- *Indirect*: O tipo mais simples de grupo, terá apenas um *bucket* e é usado para suportar uma convergência mais eficiente e rápida;

- *All*: Irá fazer cópias do pacote recebido e irá enviar cada cópia por um *bucket*, este tipo é mais adequado para fazer encaminhamento *multicast* e *broadcast* de pacotes;

- *Select*: Oferece a possibilidade de pacotes serem enviados por *buckets* diferentes durante toda a duração do fluxo. Para definir qual interface irá transmitir um determinado subfluxo, pode ser usado um algoritmo de seleção de caminhos, como o *round-robin*, juntamente com o peso dos *buckets*;

- *Fast failover*: O primeiro *bucket* ativo da entrada de grupo irá encaminhar o pacote. Este tipo é usada para implementar rotas *backup*.

Figura 3 – Campos de uma Entrada de Grupo.

A tabela de *meter* é composta por entradas de meter (*meter entries*), cada uma destas tem os seguintes campos: identificador da *meter*, as *meter bands* e os contadores, como mostrado na Figura 4. O identificador é um inteiro único de 32 bits usado para identificar a *meter*. Uma *meter* contém uma lista de *meter bands*, onde cada um destes é

responsável por especificar uma taxa de tráfego e definir como são processados os pacotes. O objetivo de uma *meter* é medir e controlar a taxa de transferência de pacotes. Sendo útil para fazer simples operações de QoS, tais como verificar a quantidade de *bytes* e pacotes transmitidos por um fluxo, e caso essa quantidade analisada ultrapasse um valor pré-definido, esse excesso é descartado.

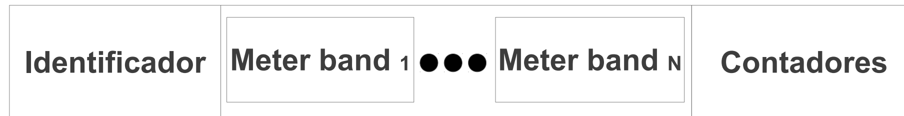


Figura 4 – Campos de uma Entrada de *Meter*.

A partir da descrição dos componentes acima, é possível entrar em detalhes de como é o encaminhamento de pacotes dentro de um comutador OpenFlow, conforme cenário da Figura 5. Quando um pacote entra no comutador, ele é enviado para a primeira tabela de fluxos do *pipeline*, esta tabela tem a numeração 0. Nessa tabela será verificado qual a entrada de fluxo que coincide com o cabeçalho do pacote.

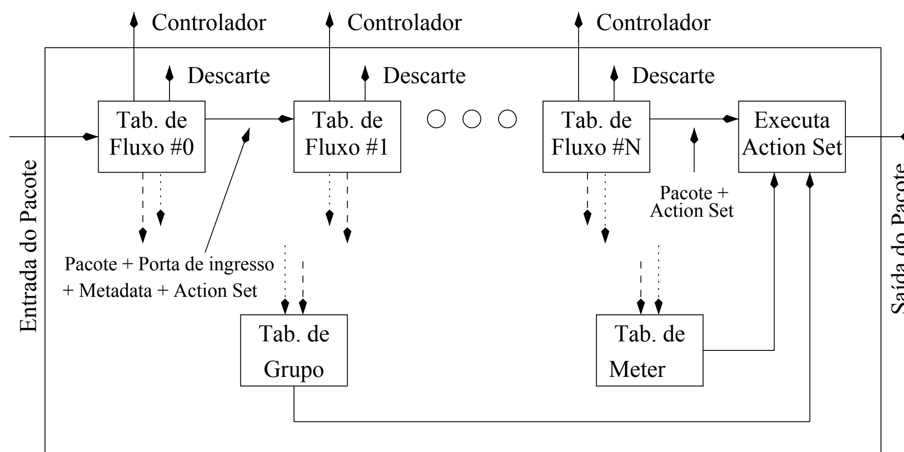


Figura 5 – Comutador OpenFlow.

Caso o pacote não case com nenhuma entrada de fluxo, cada implementação do comutador é livre para definir como agir nesse caso, definindo se o pacote é enviado para o controlador através de mensagens *packet-in*; se é descartado ou enviado para uma outra tabela de fluxo do *pipeline*, onde esta última alternativa é realizada através da instrução *Goto-Table* prevista na especificação.

Cada pacote tem uma lista chamada *Action Set*, e esta lista está vazia no começo do *pipeline*. Entretanto, é preenchida à medida que o pacote vai sendo encaminhado pelas tabelas de fluxo. Quando o cabeçalho do pacote coincide com a regra de uma entrada de fluxo, ações são inseridas dentro da *Action Set*, através da instrução *Write-Actions*. Estas ações inseridas são executadas apenas no final do *pipeline*, através do componente *Execute Action Set*. Além disso, a entrada de fluxo pode executar ações imediatamente no pacote através das instruções *Apply-Actions* ou *Meter*, a primeira instrução pode, por exemplo,

adicionar um cabeçalho *Virtual Local Area Network* (VLAN) no pacote, encaminhar uma cópia do pacote para um grupo ou para uma interface do comutador. Já a segunda envia o pacote para uma entrada da tabela *Meter*.

Após todo o processamento em uma entrada de fluxo, o pacote pode continuar no *pipeline*, ou seja, sendo processado por uma entrada em uma outra tabela de fluxo; ou sair do pipeline. Ao sair do pipeline, o *Action Set* do pacote é executado e o pacote pode ser modificado, ir para uma entrada de grupo, para uma *meter*, para uma interface de saída ou ser descartado.

Ao chegar e coincidir com uma entrada de grupo, o pacote é encaminhado de acordo com o tipo do grupo. Uma entrada de grupo pode conter vários *buckets*, responsáveis por encaminhar pacote(s) para uma ou mais interfaces do comutador. Quando o pacote chega e coincide com uma entrada de *meter*, é avaliado se o acumulado dos pacotes transmitidos, por esta *meter*, na atual janela de tempo, ultrapassa a taxa definida na *meter*. Caso não ultrapasse, apenas encaminhe para uma interface; senão descarte, ou então marque este pacote com um código *Differentiated Services Code Point* (DSCP)(NICHOLS et al., 1998) e o envie para uma interface.

2.2.3 Contadores

Os contadores de estatísticas são de extrema importância em um dispositivo OpenFlow, pois eles mantêm dados sobre o estado deste dispositivo em diferentes granularidades, tais como porta, fluxo, grupo, *bucket*, *meter* e fila. Os contadores são atualizados, em média, a cada 1 segundo em comutadores OpenFlow virtuais. As estatísticas dos contadores podem ser coletadas pelo controlador através do pedido de mensagens *Statistics Request*. As Tabelas 1 e 2, extraídas da especificação, apresentam apenas os contadores para as granularidades de fluxo e porta, respectivamente, baseado na versão 1.3 do OpenFlow, o restante dos contadores podem ser encontrados em (ONF, 2013).

A partir dos contadores definidos no OpenFlow, e do conjunto de mensagens que permitem obter estatísticas dos dispositivos de rede, em termos de quantidade de pacotes e *bytes*, podemos construir um mecanismo de monitoramento. Esse tipo de monitoramento é bastante preciso, entretanto pode sobrecarregar muito o processador dos comutadores e além disso fica limitado às atualizações dos contadores, que geralmente ocorrem a cada um segundo. A técnica de consultar explicitamente os dispositivos de rede periodicamente para coletar sua estatística chama-se *polling*.

Também é possível obter estatísticas dos comutadores com outras técnicas que não utilizam o protocolo OpenFlow. A amostragem de pacotes é uma técnica em que 1 pacote é escolhido dentre N, onde N é definido de acordo com o tráfego da interface e as limitações do comutador. O pacote amostrado é enviado para um coletor que irá estimar algumas métricas, como a quantidade de pacotes e *bytes*. Exemplos destas técnicas incluem algumas versões do NetFlow (CLAISE, 2004) e o sFlow (PHAAL; PANCHEN;

Tabela 1 – Contadores do Fluxo.

Contador	*Bits*
Pacotes Recebidos	64
Bytes Recebidos	64
Duração (segundos)	32
Duração (nanosegundos)	32

Tabela 2 – Contadores da Porta.

Contador	*Bits*
Pacotes Recebidos	64
Pacotes Transmitidos	64
Bytes Recebidos	64
Bytes Transmitidos	64
Descartes Recebidos	64
Descartes Transmitidos	64
Erros Recebidos	64
Erros Transmitidos	64
Erros de Alinhamento do *Frame* Recebidos	64
Erros de *Overrun* Recebidos	64
Erros CRC Recebidos	64
Colisões	64
Duração (segundos)	32
Duração (nanosegundos)	32

MCKEE, 2001). A técnica descrita nesse parágrafo chama-se *push*, onde o dispositivo envia os dados para uma estação de monitoramento quando esses estiverem disponíveis.

2.2.4 Controladores

Um controlador SDN tem duas funções principais, comunicar com os dispositivos de encaminhamento através da API *southbound* e oferecer serviços às aplicações através da API *northbound*. A maioria dos controladores suportam apenas o OpenFlow na API *southbound* e existem inúmeras APIs *northbound*, como APIs ad-hoc, APIs Restful, interfaces de programação multinível e sistemas de arquivos. Um dos objetivos futuros da comunidade SDN é padronizar a API *northbound*.

Existem mais de 25 controladores SDN, uns são semelhantes a outros, mas alguns possuem características únicas. Podemos ter controladores com arquitetura distribuída ou centralizada; que são *multi-threaded*; mais consistentes e mais tolerantes a falhas; que suportem a versão do OpenFlow 1.0, 1.1, 1.2, 1.3 ou 1.4; que são codificados em diversas linguagens de programação, como Java, C++, Python e C, e inúmeras outras características (KREUTZ et al., 2014).

2.3 Multi-Caminhos

A técnica de multi-caminhos (TSAI; MOORS, 2006) não é nova, ela já vem sendo usada em diversas áreas de redes há algum tempo. Nas redes tradicionais de comutação de circuitos, caminhos alternativos eram usados para reduzir a probabilidade de bloqueio de uma chamada. O menor caminho era usado até ele falhar ou ficar cheio, e então um novo caminho era escolhido para a transmissão.

Em redes de dados, o primeiro algoritmo distribuído de multi-caminhos foi proposto por Gallager (GALLAGER, 1977). Este algoritmo era usado para reduzir o atraso de convergência de uma rede, entretanto, ele apresentava algumas desvantagens, como o fato de ser de difícil implementação. Posteriormente, esse algoritmo foi estendido e aprimorado inúmeras outras vezes, como em (BERTSEKAS; GAFNI; GALLAGER, 1984) e (ZAUMEN; VUTUKURY; GARCIA-LUNA-ACEVES, 2000).

No padrão *Asynchronous Transfer Mode* (ATM) *Private Network-to-Network Interface* (PNNI)(ATM Forum, 2002), caminhos distintos podem ser estabelecidos durante o processo de reserva. Quando uma chamada falha por uma rota, um mecanismo *backtracking* chamado *crankback* (FELSTAINE; COHEN; HADER, 1999) é iniciado e inúmeros caminhos alternativos são procurados até uma rota ser estabelecida. Na Internet, alguns roteadores que implementam os protocolos de roteamento *Routing Information Protocol* (RIP) (MALKIN, 1998) ou *Open Shortest Path First* (OSPF) (MOY, 1991) suportam multi-caminhos, embora os caminhos devem ter o mesmo custo, para permitir o uso de múltiplos caminhos.

(TSAI; MOORS, 2006) detalha alguns benefícios de se usar a técnica de multi-caminhos, eles são detalhados abaixo:

- Balanceamento de carga: Os tráfegos podem ser distribuídos de modo que uma interface do dispositivo não fique sobrecarregada ou ociosa, com isso aumenta a eficiência da rede;

- Tolerância a falhas: Como o tráfego está passando por caminhos diferentes, no caso de uma falha haverá menor perda de pacotes, com isso a probabilidade da conexão ser comprometida é reduzida;

- Agregação de banda: Quando se envia partes de um mesmo fluxo por diversos caminhos é possível aproveitar melhor a banda residual de cada interface, com isso o tráfego não fica limitado a um caminho já congestionado;

- Redução do atraso: No caso de uma falha do caminho, não será necessário recalcular novos caminhos se existirem rotas alternativas ou rotas *backup*, com isso não haverá a necessidade de computar e inserir novas rotas.

Também é possível aumentar a segurança de uma rede usando multi-caminhos, como detalhado em (LOU; LIU; FANG, 2003). Os autores mencionam que é mais fácil interceptar o tráfego de um fluxo caso ele passe por apenas um caminho. Já em múltiplos caminhos seria necessário interceptar um roteador de cada caminho disjunto.

Para ser possível o encaminhamento usando multi-caminhos é necessário ter alguns componentes em um dispositivo de rede. (PRABHAVAT et al., 2012) detalha dois componentes fundamentais para o encaminhamento em multi-caminhos: a divisão de tráfego e a seleção de caminho.

2.3.1 Divisão de Tráfego

O componente de divisão de tráfego irá dividir o tráfego, passando em um dispositivo de rede, em muitas unidades de tráfego, onde cada unidade poderá ter características e tamanhos diferentes, dependendo da granularidade aplicada. (CALLADO et al., 2009) faz uma análise acerca da caracterização de tráfego, mostrando suas vantagens e desvantagens. A Figura 6 ilustra essa classificação. Os tipos de divisões são detalhados na sequência.

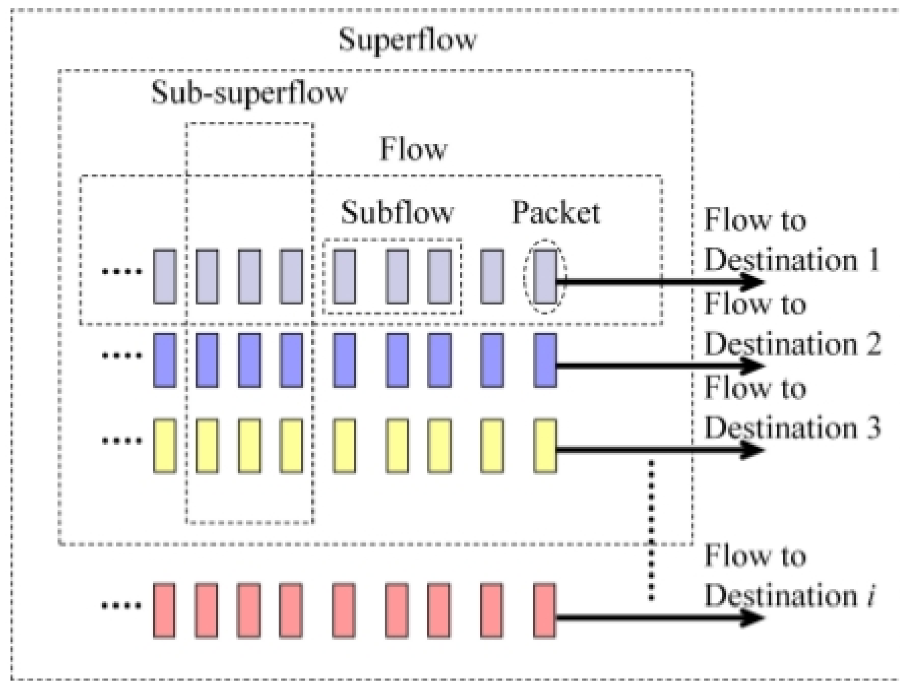


Figura 6 – Classificação da granularidade de divisão de tráfego. Extraída de (PRABHAVAT et al., 2012).

- Divisão em nível de pacote (*packet*): É a menor escala de divisão. Nessa divisão, cada pacote será tratado de maneira independente, mesmo que sejam de um mesmo fluxo. O fato de um pacote ser enviado por um caminho não garantirá que um próximo pacote do mesmo tráfego irá pelo mesmo caminho;

- Divisão em nível de fluxo(*flow*): Irá diferenciar o tráfego em fluxos, onde um fluxo pode ser definido como um tráfego que possui campos do cabeçalho dos pacotes semelhantes, como o mesmo IP de origem e IP de destino. Nessa divisão, os pacotes de um mesmo fluxo serão tratados da mesma maneira;

- Divisão em nível de subfluxo(*subflow*): Irá dividir um fluxo em fluxos menores (subfluxos) e cada um destes serão tratados de maneira diferente. Um subfluxo pode ser definido, por exemplo, de acordo com o tempo de chegada de pacotes;

- Divisões em nível de superfluxo(*superflow*) e em nível de sub-superfluxo(*sub-superflow*): são bastante comuns quando se usa uma função *hash* para definir a divisão de trá-

fego. Nessa função *hash*, campos dos cabeçalhos de pacotes são inseridos como argumento de entrada e a partir da saída dessa função será definido o processamento do pacote. Entretanto, como é possível ter cabeçalhos distintos com a mesma saída, é bem provável que fluxos completamente distintos recebam o mesmo tratamento. A primeira divisão irá tratar fluxos distintos de uma mesma maneira caso possuam o mesmo resultado da função *hash*, já na segunda divisão, subfluxos de fluxos distintos serão tratados da mesma maneira caso possuam a mesma saída da função *hash*.

2.3.2 Seleção de Caminhos

Este componente será responsável por determinar em qual caminho enviar os pacotes. A seleção será feita para todos os pacotes daquela unidade de tráfego, podendo ser por pacote, por fluxo, por subfluxo, superfluxo e sub-superfluxo. Os principais esquemas de seleção de caminho são descritos abaixo:

- ❏ Seletor *Round-Robin*: É um esquema de seleção de caminhos bastante simples. As unidades de tráfego serão enviadas, por um escalonador, para um caminho até que um critério de seleção expire. Critérios de seleção podem ser baseados em tempo de transmissão e número de pacotes, por exemplo. Após o término de envio por um caminho, o escalonador definirá, de um modo sequencial e circular, uma outra interface para enviar mais tráfego;
- ❏ Seletor baseado na informação do pacote: O seletor definirá a interface que será enviado o pacote a partir de uma função *hash*. Um identificador obtido através do cabeçalho do pacote, incluindo informações tais como IP de origem e destino, será argumento de entrada na função *hash*. Esta função irá ter como saída a interface onde o pacote será enviado;
- ❏ Seletor baseado na condição do tráfego: O seletor definirá a interface de acordo com as condições de tráfego no dispositivo. Aqui podemos citar, como exemplos, a carga, a taxa e o volume de tráfego nas interfaces, e também a quantidade de fluxos ativos no dispositivo;
- ❏ Seletor baseado na condição da rede: Será levado em consideração a condição da rede para definir em qual interface o seletor enviará o tráfego. Aqui pode ser usado como métrica, o tamanho da fila, o atraso, o *jitter* e a perda de pacotes de um caminho.

2.3.3 Problemas de Desempenho

Embora o encaminhamento baseado em multi-caminhos pode agregar bastante valor à rede, ele também pode causar efeitos colaterais caso não seja usado corretamente. Abaixo

serão detalhados alguns problemas comuns, como mostrado em (PRABHAVAT et al., 2012).

- Não balanceamento de carga: Acontece isso quando a carga de tráfego passando por uma interface é maior ou menor que o desejado, ou seja, há um desvio do tráfego esperado naquela interface. Com isso, é possível que hajam interfaces, no mesmo dispositivo, recebendo muito tráfego e outras recebendo pouco tráfego. Esse não balanceamento de carga pode fazer com que ocorram perdas de pacotes e aumento do atraso em interfaces muito sobrecarregadas, enquanto outras ficam subutilizadas;

- Grau de redistribuição dos fluxos: É comum acontecer em multi-caminhos, isso acontece quando é alterado a interface de saída de um pacote para um mesmo fluxo. Se for usado de forma correta pode ajudar no balanceamento de carga, pois pode adaptar a carga nas interfaces à medida que o número de tráfego e número de caminhos disponíveis aumentem ou diminuam. Entretanto, se for usado incorretamente, pode causar aumento do processamento do dispositivo, como também reordenação e perdas de pacotes;

- *Overhead* de comunicação: Esse caso acontece mais em seletores de caminhos mais especializados, como o seletor baseado na condição da rede. Neste caso, os dispositivos precisam trocar informações frequentemente entre si para verificar o estado atual da rede, esta análise é necessária para definir posteriormente o caminho de envio do fluxo. Isso acarreta um gasto adicional do uso de largura de banda e aumento do processamento dos dispositivos;

- Complexidade computacional: Alguns seletores de caminhos são mais simples, como os baseados no modelo *Round-Robin*, e isso gera um custo computacional bem baixo, geralmente O(1). Entretanto, algumas técnicas de seleção necessitam analisar cada interface do dispositivo e também precisam de um maior processamento para definir a interface de envio do pacote, com isso há um maior gasto computacional. Então, é necessário levar em consideração esse aspecto para desenvolver e implementar um algoritmo de seleção eficiente que seja, também, compatível com os dispositivos de rede;

- Inundação por tráfego *broadcast*: O tráfego *broadcast* pode causar um grande “estrago” em uma rede Ethernet, pois os pacotes poderão entrar em *loop* infinito. A principal técnica usada para contornar este problema é o uso do protocolo *Spanning Tree Protocol* (STP) (PERLMAN, 1985) e suas variações, tal como o *Rapid Spanning Tree Protocol* (RSTP) (Cisco Systems, 2006). De forma geral, estas técnicas irão desativar algumas interfaces dos comutadores para que o tráfego não passe por elas, com isso cria-se uma rede sem *loops*. Entretanto, isso inviabiliza a técnica de multi-caminhos, pois a rede não contará mais com caminhos reduntantes. Portanto,

será necessário desabilitar o STP e protocolos similares para fazer uso de múltiplos caminhos. Alguns trabalhos foram propostos para evitar o problema da inundação de pacotes durante um *broadcast*, tal como o trabalho proposto em(WANG et al., 2013). Os autores propõem que o controlador armazene informações relativas aos *hosts* da rede, tal como o endereço *Media Access Control* (MAC) e o endereço IP em uma tupla, para que quando um *host* envie uma mensagem *Address Resolution Protocol* (ARP) soliticitando o endereço MAC do destinatário, o comutador envie diretamente esta mensagem para o controlador. O controlador irá processar a mensagem de ARP recebida, consultará sua base de dados e responderá, através de uma mensagem *packet-out*, ao *host* solicitante.

2.4 Qualidade de Serviço(QoS)

Alguns esforços do *Internet Engineering Task Force* (IETF) para prover a QoS são descritos na Seção 2.4.1 (EL-GENDY; BOSE; SHIN, 2003)(SHARAFAT et al., 2011). A Seção 2.4.2 fará uma breve análise dos aspectos negativos dos mecanismos propostos pelo IETF, e as vantagens oferecidas pelas SDNs caso sejam usadas em conjunto ou como alternativa aos protocolos descritos na Seção 2.4.1 (EL-GENDY; BOSE; SHIN, 2003)(REID; KATCHABAW, 2004)(DAS; PARULKAR; MCKEOWN, 2012) (SHARAFAT et al., 2011).

2.4.1 Propostas tradicionais de QoS apresentadas pelo IETF

- IntServ: A Arquitetura de Serviços Integrados (IntServ), definida pelo IETF em (BRADEN; CLARK; SHENKER, 1994), foi desenvolvida devido à necessidade de garantir a QoS de aplicações em tempo real, tais como uma conferência multimedia e realidade virtual. Como o IntServ foi desenvolvido para prover QoS por fluxo, seria possível ter um controle maior sobre cada tráfego da rede, e esta alta granularidade é fundamental para prover QoS às aplicações em tempo real. Quando uma aplicação requisita uma determinada QoS, o IntServ usa o protocolo *Resource ReSerVation Protocol* (RSVP) (BRADEN; ZHANG, 1997) para reservar os recursos requisitados pela aplicação em todos os roteadores do caminho. O RSVP utiliza a mensagem PATH, que é enviada pelo cliente e passa pelos roteadores do caminho até chegar ao servidor, para definir a característica do tráfego enviado (T-spec), como também, a QoS requisitada (R-spec). Quando a mensagem PATH chega ao servidor, este irá enviar uma mensagem RESV ao cliente, e caso todos os roteadores do caminho aceitem o RESV, a QoS requisitada será alocada e o estado do fluxo armazenados nos roteadores. As reservas feitas pelo RSVP para um fluxo devem ser atualizadas a cada 30 segundos, através do envio de novas mensagens PATH e RESV, caso contrário, a reserva é desfeita;

- DiffServ: A Arquitetura de Serviços Diferenciados (DiffServ), proposta pelo IETF em (BLAKE et al., 1998), surgiu propondo uma abordagem mais simples e eficiente que o IntServ, onde não haveria a necessidade de sinalizar e reservar os recursos, e nem manter os estados dos fluxos da rede, fazendo com que o DiffServ fosse uma solução ideal para a Internet, já que o IntServ não seria escalável em *backbones* de milhares ou milhões de fluxos. A idéia do DiffServ é classificar os fluxos em classes através de uma marca DSCP, de 6 bits, feita no campo *Type of Service* (TOS) do datagrama IPv4 ou no campo Classe de Tráfego do datagrama IPv6. A partir dessa marca DSCP feita no pacote, seria possível atribuir este pacote a um comportamento de encaminhamento, também chamado de *Per-Hop Behavior* (PHB). A PHB definiria como escalonar o pacote, por exemplo, enviá-lo para uma fila de menor atraso, *jitter* e poucas perdas; de melhor esforço; de maior prioridade; e a fila, posteriormente, encaminharia o pacote para uma interface;

- MPLS: O MPLS, proposto pelo IETF em (ROSEN; VISWANATHAN; CALLON, 2001), é um esquema de encaminhamento avançado, ele estende o roteamento em relação ao encaminhamento de pacote e controle de caminho. Ele trabalha em uma camada intermediária entre as camadas 2 (Enlace) e 3 (Rede) do modelo *Open Systems Interconnection* (OSI). O pacote MPLS possui um cabeçalho de 32 *bits* usado para encapsular um pacote IP, sendo que o campo principal do cabeçalho MPLS é o de 20 *bits*, chamado de rótulo. Em uma rede MPLS é comum usarmos os seguintes termos: *Label Switch Router* (LSR), *Label Edge Router* (LER) e *Label Switch Path* (LSP). Os LSRs são os roteadores que estão em uma rede MPLS e são responsáveis por executar algoritmos de encaminhamento e manter a tabela de encaminhamento, e têm como função principal a substituição dos rótulos dos pacotes recebidos por outros rótulos e encaminhar os pacotes para a interface correta do dispositivo. Os LERs têm as mesmas funções dos LSRs, entretanto, eles se localizam na entrada do domínio MPLS, e possuem, como funcão adicional, definir uma *Forwarding Equivalency Class* (FEC) para um conjunto de pacotes, ou seja, os pacotes com características semelhantes, por exemplo, mesmo IP de destino, serão associados a uma FEC e encaminhados por um determinado LSP. O LSP é o caminho por onde pacotes, com as mesmas características, irão passar. O MPLS usa protocolos de sinalização para construir LSPs, tais como, o RSVP e o *Label Distribution Protocol* (LDP). Por fim, vale destacar que o MPLS é semelhante ao DiffServ devido à classificação de fluxos em classes, como também ao ATM(KESHAV, 1997) e ao Frame Relay(BRADLEY; BROWN; MALIS, 1992), devido ao uso de rótulos para comutação de pacotes;

- GMPLS: O *Generalized Multi-Protocol Label Switching* (GMPLS), proposto pelo IETF em (MANNIE, 2004), é uma extensão do MPLS para prover novos tipos de

comutações, tais como, por fibra, por tempo (TDM) e por comprimento de onda (*lambda*). Para que o GMPLS suportasse esses novos tipos de comutações foi necessário estender algumas funcionalidades presentes no MPLS, e em alguns casos, adicionar novas funcionalidades. Exemplos de tais funcionalidades são: propagação de erros, propagação de informações de sincronização entre os comutadores de ingresso e de egresso e requisição e distribuição de rótulos. Outras características do GMPLS é que não há uma restrição do modelo de interconexão utilizado entre as redes, há a separação entre o Plano de Controle (sinalização e roteamento) e o Plano de Dados, existe a possibilidade de estabelecer caminhos bidirecionais, entre outros.

2.4.2 SDN e Protocolos Tradicionais

Os protocolos tradicionais, citados na Seção 2.4.1, são bastante utilizados para a oferta de QoS dos usuários finais. Entretanto, eles possuem algumas desvantagens, e também podem ser usados em conjunto com a Arquitetura SDN para a oferta de QoS especificada pelo SLA do usuário.

Algumas desvantagens do IntServ são, todos os roteadores precisam ter noção do protocolo RSVP e serem capazes de sinalizar a QoS requerida; as reservas em cada dispositivo devem ser atualizadas periodicamente, ocasionando aumento do tráfego na rede e aumentando a chance de que as reservas possam expirar se os pacotes de atualização forem perdidos; manter os estados em cada roteador, combinando com controle de admissão em cada salto e aumento dos requisitos de memória para permitir um número maior de reservas, fazendo com que seja necessário roteadores mais robustos. Com o uso do protocolo OpenFlow, a QoS pode ser aplicada diretamento nos fluxos dos comutadores pelo controlador sem o uso do protocolo RSVP, sem a necessidade de atualização constante, sem envio de tráfego na rede e usando apenas dispositivos comoditizados.

Algumas desvantagens do DiffServ incluem, o gerenciamento e o monitoramento complexos, há a perda da granularidade, pois o QoS é aplicado na classe e há a necessidade de classificar o tráfego da rede em classes. Com o uso do protocolo OpenFlow é possível gerenciar e monitorar a rede com mensagens já pré-definidas ou desenvolvendo ferramentas de acordo com a vontade do usuário, não existe perda de granularidade, pois é possível trabalhar com fluxos individualmente e com seus campos de *match*, e não existe a necessidade de classificação de tráfego em classes.

Um dos maiores problemas do MPLS/GMPLS é que eles ficam limitados aos protocolos de Engenharia de Tráfego existentes, tais como o OSPF, LDP, RSVP-TE e I-BGP. Além disso, muitos desses protocolos foram estendidos para funcionar no MPLS/GMPLS. O RSVP, por exemplo, foi desenvolvido como um protocolo para sinalização de recursos em uma rede na Arquitetura IntServ, depois estendido para o uso no MPLS e depois para o GMPLS, gerando um protocolo mais complexo e de codificação maior, ocasionando

desperdício de recursos. O uso de protocolos distribuídos eleva o tráfego da rede quando há mudanças constantes na mesma, acarretando desperdício de banda e aumento do atraso do enlace, além do gasto adicional para recálculo de rotas e processamento de informações nos dispositivos de rede. A partir da versão 1.1 do OpenFlow, foram inseridas algumas funcionalidades do MPLS/GMPLS, como o *push*, *pop* e *swap* de rótulos. Com a Arquitetura SDN, o MPLS não ficará mais limitado a protocolos já existentes, pois um novo protocolo pode ser criado e modificado de acordo com o desejo de cada administrador de rede.

Portanto, é possível perceber que a SDN, e o OpenFlow, não vieram para exterminar os protocolos tradicionais de oferta de QoS, mas sim, como uma arquitetura que pode usá-los de uma forma mais eficiente, e que possa, também, propor futuras extensões aos mesmos. Essa idéia é reforçada no momento em que o próprio protocolo OpenFlow define mensagens que permitem a inserção de cabeçalho MPLS nos pacotes, como mencionado anteriormente; e também através das *meters*, é possível implementar operações simples de QoS, podendo ser combinado com as filas das portas para desenvolver *frameworks* mais complexos, tais como o DiffServ.

Vale ressaltar que, o OpenFlow é um protocolo aberto, e portanto, é possível adicionar novas funcionalidades aos dispositivos, já que não ficaríamos mais limitados aos protocolos proprietários, e qualquer grupo de pesquisa poderia, por exemplo, adicionar uma nova funcionalidade ao MPLS ou DiffServ. Esta funcionalidade poderia ser, no futuro, implementada em larga escala, e consequentemente, beneficiaria toda a indústria, empresas e o usuário final.

Capítulo 3

Trabalhos Relacionados

A Seção 3.1 apresenta os principais trabalhos associados ao monitoramento em redes OpenFlow. A Seção 3.2 detalha trabalhos relacionados sobre encaminhamento de tráfego por múltiplos caminhos. Em ambas as seções são feitas comparações pontuais entre os trabalhos relacionados e a nossa proposta.

3.1 Monitoramento

Nas Seções 3.1.1 a 3.1.4 são apresentados os principais trabalhos publicados recentemente sobre monitoramento em SDN, apresentando o diferencial entre as publicações e o módulo de monitoramento proposto neste trabalho.

3.1.1 *OpenNetMon*

Em (ADRICHEM; DOERR; KUIPERS, 2014), os autores utilizam um mecanismo de *polling* para coletar dados referentes aos fluxos, permitindo apresentar medidas referentes à largura de banda consumida por fluxo, à taxa de pacotes perdidos e o atraso fim-a-fim. O principal diferencial entre o trabalho apresentado em (ADRICHEM; DOERR; KUIPERS, 2014) e o módulo proposto refere-se à quantidade de detalhes capturados pelo módulo de monitoramento. Em (ADRICHEM; DOERR; KUIPERS, 2014), os autores fazem leitura apenas nos dispositivos de rede de ingresso e egresso, enquanto no módulo proposto as leituras são feitas em todos os dispositivos de rede, permitindo o monitoramento do fluxo em cada um dos enlaces por onde é encaminhado.

No cálculo do atraso em (ADRICHEM; DOERR; KUIPERS, 2014), é injetado pelo controlador um pacote no primeiro comutador do caminho e quando o pacote chega ao último comutador, esse manda o pacote para o controlador. A partir da diferença do tempo de saída e chegada do pacote, e da subtração do tempo de atraso de envio comutador-controlador, obtém-se o atraso. O método para o cálculo do atraso proposto nesta dis-

sertação de mestrado não utiliza a injeção de pacotes na rede, sendo feita através dos próprios pacotes de cada fluxo, independente do protocolo sendo encaminhado.

3.1.2 *OpenSample*

Em (SUH et al., 2014), os autores modificam o sFlow (PHAAL; PANCHEN; MCKEE, 2001) para inspecionar o número de sequência do *Transmission Control Protocol* (TCP) presente nos pacotes, como forma de aumentar a precisão das amostragens feitas pelo sFlow. A partir dos dados analisados e processados pelo sFlow, os autores detectam fluxos classificados como "elefantes", estimam a utilização de cada porta do comutador, calculam a vazão de cada fluxo e aplicam engenharia de tráfego na rede.

De fato, o mecanismo apresentado em (SUH et al., 2014) melhora o funcionamento do sFlow, porém restringe as amostragens apenas a transmissões TCP, sendo esta a principal diferença em relação ao módulo proposto, que é capaz de monitorar fluxos independente dos protocolos sendo transportados nos pacotes.

3.1.3 *Latency*

Em (PHEMIUS; BOUET, 2013), os autores utilizam algumas mensagens especificadas pelo OpenFlow como forma de medir a latência experimentada em cada enlace. Basicamente, os autores propõem a injeção de mensagens via controlador em um comutador da malha de dispositivos de rede, e a posterior recuperação destas mensagens em um próximo comutador adjacente. Através do monitoramento do tempo necessário ao retorno das mensagens ao controlador, os autores apresentam os valores de latência dos enlaces. Conforme mencionado anteriormente, no módulo proposto não são utilizadas mensagens injetadas pelo controlador, as medições são feitas considerando pacotes pertencentes aos fluxos monitorados, comparando o volume de dados encaminhado em cada um dos comutadores.

3.1.4 *PayLess*

Em (CHOWDHURY et al., 2014), os autores apresentam um *framework* de monitoramento externo ao controlador da SDN capaz de solicitar ao controlador, através da *northbound API*, a coleta de dados dos dispositivos de rede através de *polling*. O módulo proposto difere do trabalho apresentado em (CHOWDHURY et al., 2014) devido ao fato de estendermos o controlador Floodlight (FLOODLIGHT, 2015), introduzindo neste controlador o módulo de monitoramento, utilizando a sua natureza *multi-thread*. Desta forma, o módulo proposto é capaz de adequar a quantidade de dispositivos de monitoramento ao número de dispositivos de rede e a real carga experimentada na rede, além de ter acesso direto à API Java do controlador.

3.2 Multi-Caminhos

Nesta seção serão descritos trabalhos sobre múltiplos caminhos. As técnicas descritas nas Seções 3.2.1 a 3.2.5 são baseadas no trabalho (PRABHAVAT et al., 2012). Este trabalho detalha algumas pesquisas relativas ao encaminhamento baseado em multi-caminhos, onde todos os modelos de seleção foram contemplados. As técnicas podem ser divididas em cinco grupos: *info-unaware* (*Round-Robin*), baseados na informação do pacote, condição do tráfego, na condição da rede e, por fim, na condição do tráfego e da rede. A Seção 3.2.6 detalha alguns trabalhos sobre encaminhamento de tráfego em múltiplos caminhos usando a Arquitetura SDN. Finalmente, a Seção 3.2.7 apresenta um protocolo proposto pelo Google que pode ser consolidado, futuramente, como um protocolo que provê o roteamento de pacotes em múltiplos caminhos.

3.2.1 *Info-unaware*

São técnicas bastante simples que não analisam informação alguma sobre a condição do tráfego e da rede. Além disso, quando analisam um pacote, fazem isso de forma bem superficial.

- *Packet-By-Packet Round-Robin* (PBP-RR): É uma técnica bastante simples onde cada pacote é enviado por um caminho diferente. Todos os caminhos recebem a mesma quantidade de pacotes e sua complexidade é bem baixa, mas a quantidade de reordenações e retransmissões de pacotes é bem alta. Além disso, a discrepância da largura de banda das interfaces não é levada em consideração nesta abordagem. É bastante usada em algumas implementações do ECMP, na multiplexação inversa e em algumas extensões de protocolos de roteamento, como em (VILLAMIZAR, 1999);

- *Weighted Round Robin* (WRR): Nesta abordagem será atribuído um peso à cada caminho do fluxo. Com isso, cada caminho receberá, de acordo com seu peso, um número de pacotes ou uma fatia de tempo, para transmitir. É uma abordagem bem superior à anterior, pois será definido um peso de acordo com a banda remanescente da interface e haverá menor perda de pacotes, pois mais pacotes serão enviados por um mesmo caminho, evitando desordenações de pacotes. A nossa proposta foi inspirada no WRR, onde são definidos pesos que delimitam o tráfego recebido, por cada caminho. Alguns exemplos de seu uso podem ser encontrados no *Enhanced Interior Gateway Routing Protocol* (EIGRP) (SLICE et al., 2013), em (PAREKH; GALLAGER, 1993) e (Cisco Systems, 2009);

- *Surplus Round Robin* (SRR): É uma técnica bem semelhante à WRR, mas tem como diferencial um contador de bytes, analisando o tamanho de cada pacote antes de

decidir se muda ou não de caminho. Desta forma o SRR consegue ter um balanceamento melhor da carga, pois não fica limitado apenas à quantidade de pacotes, mas ao seu tamanho também. (ADISESHU; PARULKAR; VARGHESE, 1996) detalha uma implementação desta técnica.

3.2.2 Informação do pacote

Este modelo irá fazer uma análise de cada pacote, onde será coletado algumas informações, tais como endereço de origem e destino, para o roteamento do pacote.

- *Fast Switching* (FS): É uma técnica baseada na divisão do tráfico em nível de fluxo (ZININ, 2002). Nesta técnica, todos os pacotes de um fluxo irão por apenas um caminho até o seu término. Quando surgir um novo fluxo, este irá ter como destino um próximo caminho, similiar ao modo *round-robin*. O número de pacotes desordenados de cada fluxo cai bastante nessa abordagem, pois eles passam por apenas um caminho durante toda a transmissão, entretanto um problema que pode ocorrer é o não balanceamento de carga, porque como não se sabe a quantidade e o tamanho de cada pacote, é possível que algumas interfaces fiquem mais sobrecarregadas que as outras;

- *Direct Hashing* (DH): É uma técnica de baixo custo computacional e bastante implementada, baseada na divisão de tráfego em nível de fluxo. Antes de um pacote ser enviado para uma interface de saída, campos do seu cabeçalho, geralmente [IP de origem, IP de destino, tipo do protocolo, porta de origem e porta de destino] (XI; LIU; CHAO, 2011), serão capturados e fornecidos como argumento em uma função *hash*, onde esta função também levará em consideração o número de múltiplos caminhos para o cálculo da saída. Após o cálculo, essa função terá como saída um caminho para o pacote. O principal problema dessa técnica é que o balanceamento de tráfego pode ficar comprometido, pois muitos fluxos podem ser mapeados em um único caminho, já que a função *hash* considera apenas o cabeçalho do pacote e o número de multi-caminhos para o escalonamento do tráfego. Está técnica pode ser encontrada na literatura em (VILLAMIZAR, 1999), (THALER; HOPPS, 2000) e no ECMP (HOPPS, 2000).

3.2.3 Condição do tráfego

Este modelo irá verificar as condições do tráfego para definir o escalonamento dos pacotes no dispositivo. Aqui entra o tempo de chegada entre os pacotes, carga e volume do tráfego, entre outros.

- *Adaptive Flow-Level Load Control Scheme for Multipath Forwarding* (AFLCMF) (LEE; CHOI, 2001): Essa técnica irá distribuir os pacotes nos múltiplos caminhos

de acordo com a chegada de pacotes no dispositivo e também através de uma razão pré- definida. Através da chegada de pacotes e dessa razão pré-definida, o algoritmo gerará periodicamente uma taxa limite usada para o escalonamento de pacotes. Cada fluxo será classificado de acordo com a taxa de chegada de pacotes e a partir dessa taxa será atribuído um caminho para ele. Por exemplo, caso a taxa de pacotes recebidos seja maior que a taxa limite, então envie os pacotes para o caminho 1; senão envie para o caminho 2. Esta taxa limite será modificada de acordo com a variação do comportamento do tráfego, permitindo que os fluxos sejam realocados de maneira mais eficiente. A principal vantagem dessa técnica é que consegue adaptar-se ao tráfego transmitido. Entretanto, os fluxos podem ter seus caminhos alterados frequentemente, caso o tráfego não seja relativamente uniforme, isto pode causar aumento da reordenação de pacotes e aumento do processamento do dispositivo;

- *Progressive Multiple Bin Disconnection with Absolute Difference Bin Reconnection* (MBD-/ADBR) (MARTIN; MENTH; HEMMKEPPLER, 2006): É uma abordagem que frequentemente analisa as interfaces do dispositivo em busca de caminhos pouco e excessivamente utilizados. Ao verificar que a quantidade de pacotes passando por um caminho está no limite da taxa suportada pela interface, fluxos desta interface serão realocados para interfaces pouco utilizadas e isso balanceará a carga do dispositivo de rede. Entretanto, o custo computacional é relativamente alto, pois é necessário monitorar todas as interfaces frequentemente e além disso redistribuir os fluxos à medida que as interfaces ficam sobrecarregadas.

3.2.4 Condição da rede

Este modelo irá verificar as condições da rede para definir o escalonamento dos pacotes no dispositivo. Algumas condições analisadas são: largura de banda da interface, número de saltos, o atraso, *jitter* e perdas de pacotes. A nossa proposta de multicaminhos também é inspirada neste modelo, pois avalia condições de rede, no momento, apenas a largura de banda, para definir os pesos e os múltiplos caminhos do tráfego.

- *Earliest Delivery Path First* (EDPF) (CHEBROLU; RAO, 2006): Esta técnica irá escalonar um pacote em um caminho baseado no menor tempo estimado de entrega. O EDPF irá analisar algumas métricas da rede, tais como atraso e largura de banda, entre os dispositivos de encaminhamento da origem até o destino, e a partir dessa análise escolher qual o caminho irá entregar o pacote mais rapidamente. É uma técnica bem eficiente, pois, teoricamente, envia os pacotes no menor tempo possível e reduz as reordenações de pacotes, porque um pacote de um fluxo irá pegar o menor caminho e o próximo pacote do mesmo fluxo irá, provavelmente, pegar o mesmo caminho do pacote anterior. Entretanto, para se ter uma visão bem precisa do estado da rede, seria necessário trocar muitas informações entre os dispositivos

de rede frequentemente, o que aumentaria o gasto da largura de banda com a troca dessas informações, além do aumento do uso do processador para processar essas métricas e definir o melhor caminho;

- *Load Distribution over Multipath* (LDM) (SONG et al., 2003): Esta técnica é semelhante à descrita acima, analisando algumas métricas de rede para definir o escalonamento do fluxo. Diferentemente do EDPF, os pacotes de um mesmo fluxo irão passar pelo mesmo caminho. Este será escolhido através da análise de duas métricas, a primeira será a partir da observação do tráfego de cada caminho e a segunda será observando o número de saltos de cada caminho. A partir desses dois parâmetros será escolhido o caminho com o menor número de saltos e menos utilizado. Esta técnica gasta menos processamento que a anterior, pois a granularidade aqui é por fluxo e não por pacote e consequentemente não precisaria coletar o estado da rede tão frequentemente. Entretanto, se a transmissão de dados dos fluxos não for uniforme, ou seja, o tráfego pode aumentar e diminuir consideravelmente, então poderá ocorrer utilização excessiva de algumas interfaces.

3.2.5 Condição do tráfego e da rede

Este modelo irá verificar as condições do tráfego e da rede para definir o escalonamento dos pacotes no dispositivo.

- *Load Balancing for Parallel Forwarding Protocol* (LBPF) (SHI; MACGREGOR; GBURZYNSKI, 2005): Esta técnica é um melhoramento das técnicas de multicaminhos baseado em uma função *hash*. Uma das principais desvantagens de se usar uma função *hash* para definir um caminho é que não se leva em consideração as características do tráfego. O funcionamento do LBPF é bem semelhante à uma função *hash*, onde a partir de um cabeçalho de pacote será definido o caminho, mas ele também analisa duas variáveis: a quantidade de pacotes passando por cada fluxo e a carga das interfaces, a primeira será usada para verificar se o fluxo requer ou não muita banda e a segunda será usada para ver se o caminho está ou não sobrecarregado. Os fluxos que transmitirem mais pacotes que um valor pré-definido, dentro de uma janela de observação, serão classificados como fluxos agressivos. Quando o sistema verifica que as cargas das interfaces não estão balanceadas, um algoritmo adaptativo será executado e os pacotes dos fluxos agressivos serão transmitidos pela interface menos utilizada no momento. É uma técnica bastante eficiente, mas ainda tem algumas deficiências, como o fato de que o não balanceamento de carga pode ser causado pelos fluxos não agressivos, a reordenação de pacotes e o custo computacional, embora este último não seja tão alto;

- *Flowlet Aware Routing Engine* (FLARE) (KANDULA et al., 2007): Foi proposto como uma forma de balancear o tráfego do dispositivo e ao mesmo tempo reduzir o número de pacotes reordenados. Para que o algoritmo funcione de forma adequada, será realizado uma estimativa do atraso de ida e volta de cada caminho, semelhante ao comando *ping*. A partir desses valores será calculado a diferença de atraso máximo entre os caminhos paralelos, obtendo no final um valor usado para definir o escalonamento de pacotes, este valor será chamado de limite máximo. Esta técnica define um termo chamado *flowlet*, onde cada *flowlet* será um grupo de pacotes na qual o tempo de chegada deles é menor que o limite máximo (calculado acima). Quando um pacote chega no dispositivo, será verificado se o tempo de chegada dele é menor que o limite máximo, caso seja, ele pertencerá à este *flowlet*, portanto será enviado pela mesma interface que os pacotes enviados anteriormente e que também pertencem a esse mesmo *flowlet*; se o tempo não for menor, este pacote será a cabeça do novo *flowlet* e todos os pacotes pertecente a este passarão por uma nova interface. É uma técnica que ajuda a balancear a carga e reduzir a reordenação de pacotes, mas deve-se ter uma rede bem estável para calcular este limite máximo, pois se este for muito baixo, irá aumentar o número de reordenações; se o limite máximo for muito alto, não terá um balanceamento eficiente de carga.

3.2.6 Multi-caminhos com SDN

- (SANDRI et al., 2015) propõe o uso do MPTCP junto com a Arquitetura SDN. O MPTCP é uma aplicação usada para dividir um fluxo TCP em múltiplos subfluxos TCP, sendo que todo o processo de encaminhamento de tráfego ocorre por apenas uma conexão TCP. Cada subfluxo terá seu próprio mecanismo de controle e pode passar por caminhos completamente diferentes de outros subfluxos. Na abordagem proposta pelos autores, um controlador SDN será usado para garantir que os subfluxos gerados pelo MPTCP sempre passem por caminhos disjuntos, com isso o tráfego gerado pelo MPTCP usará de forma mais eficiente a banda da rede e o tráfego não será tão afetado em caso de falhas de interfaces nos dispositivos de encaminhamento. É feito uma comparação usando o MPTCP, com múltiplos caminhos, e o TCP padrão, onde este irá usar apenas um único caminho para transmitir o tráfego. A partir dos resultados obtidos pelos autores, é possível perceber que o MPTCP tem resultados bem superiores em comparação com o TCP, caso o MPTCP realmente tenha seus subfluxos passando por caminhos diferentes. Entretanto, existem alguns problemas relativos, principalmente, à natureza do MPTCP, que limita-se ao protocolo TCP. Além disso, embora derive um fluxo TCP em subfluxos TCP, é uma abordagem centrada no *host*, pois será necessário que todos os hosts tenham o MPTCP instalado. Embora o artigo usado como referência use um controlador para definir caminhos disjuntos, o *host* precisa definir à *priori* quantos subfluxos

serão usados pelo MPTCP, pois caso gere mais subfluxos do que número de caminhos disjuntos, o MPTCP vai prejudicar a rede, pois gerará mais *overhead* na rede e, consequentemente, reduzirá a largura de banda livre, aumentando o gasto em processamento. O artigo propõe a idéia de que caso o MPTCP requisite um novo caminho e a rede não possa disponibilizar mais caminhos disjuntos, o controlador irá descartar essa requisição, mas mesmo assim gerará um *overhead* entre o dispositivo OpenFlow e o controlador para cada requisição. A nossa proposta difere do trabalho apresentado em (SANDRI et al., 2015) por ser independente de protocolo, suportando o tratamento de multicaminhos para todos os fluxos que utilizam a rede OpenFlow, sem que haja a necessidade de alteração na pilha de protocolos dos *hosts*. Além disso, a definição dos caminhos a serem utilizados é automaticamente obtida pelo módulo multicaminhos, MP-Routing, através de informações precisas sobre o Plano de Dados, obtidas pelo módulo de monitoramento, SDNMon.

- (JINYAO et al., 2015) propõe um *framework*, chamado de HiQoS, para prover encaminhamento de dados sobre múltiplos caminhos usando a Arquitetura SDN. O *framework* possui dois componentes principais: o de serviços diferenciados e o de roteamento em multiplos caminhos. O primeiro componente será responsável por classificar o tráfego das aplicações de acordo com o uso estimado de largura de banda, onde os tráfegos podem ser classificados em: vídeo com alto uso de banda, vídeo/audio com baixo atraso e um fluxo de dados como serviços de melhor esforço. Já o módulo de roteamento irá consultar os dispositivos periodicamente para coletar algumas estatísticas, neste caso a largura de banda de cada fila das interfaces, e a partir dessa métrica coletada irá verificar qual a fila menos utilizada para aplicar o novo fluxo no dispositivo. O fato de escolher sempre a fila com menor largura de banda ajuda a balancear a carga e aumentar a eficiência da rede, como mostrado no artigo. Diferentemente da nossa proposta, a escolha do melhor caminho em (JINYAO et al., 2015) baseia-se na divisão em nível de fluxo, ou seja, um fluxo é encaminhado pelo mesmo caminho até o seu término, não havendo a divisão de um mesmo fluxo em múltiplos subfluxos.

3.2.7 QUIC

O QUIC é um protocolo proposto pela Google no ano de 2012, construído em cima do protocolo UDP, e que possui como objetivo principal reduzir o tempo de acesso a uma página Web. O QUIC tem funcionalidades semelhantes ao protocolo TCP, tais como um serviço orientado à conexão, um controle de congestionamento, e também semelhantes ao protocolo *Transport Layer Security* (TLS)/*Secure Socket Layer* (SSL), como a criptografia e a autenticação dos dados transmitidos. Além disso, permite que uma conexão possua muitas *streams* de dados, através de uma multiplexação de *streams*, possibilitando que

cada *stream* tenha o seu próprio tráfego e um sistema de controle.

O trabalho proposto em (CARLUCCI; CICCO; MASCOLO, 2015) faz uma descrição do QUIC e o compara com a implementação Cubic (HA; RHEE; XU, 2008) do TCP, realizando experimentos em diferentes condições de rede e utilizando o protocolo HTTP/1.1. No final dos testes, os autores mencionam que o QUIC pode ser realmente melhor que o TCP em determinadas situações. O trabalho também menciona que o QUIC tem um suporte nativo à multi-caminhos, em decorrência do campo *Connection ID* em seu cabeçalho. No protocolo TCP, uma conexão é identificada pela seguinte tupla: [IP de origem, IP de destino, Porta de Origem e Porta de Destino], entretanto, quando um dispositivo move-se para uma outra rede, ou o *Network Address Translation* (NAT) do roteador é alterado, por exemplo, os endereços IPs ou as Portas podem ser modificadas, inviabilizando a conexão. Isso não acontece no QUIC, pois como a identificação da conexão ocorre através do campo *Connection ID*, os endereços da camada de rede ou as portas da aplicação podem ser alteradas sem afetar a conexão, e isto, indiretamente, garante que um mesmo fluxo pode passar por diferentes caminhos durante sua transmissão.

No futuro, o QUIC poderia implementar um mecanismo de multi-caminhos, onde diferentes *streams* podem passar por diferentes caminhos, a partir da observação da característica de cada tráfego ou da rede, por exemplo. Isso poderia contribuir para um melhor balanceamento da rede, utilizar de forma mais eficientes os recursos disponíveis, entre outras vantagens do uso de multi-caminhos, como descrito na Seção 2.3.

Capítulo 4

SDNMon e MP-Routing

Este capítulo descreve a arquitetura utilizada e as extensões propostas para a realização deste trabalho. A Seção 4.1 descreverá o módulo de monitoramento, chamado de SDNMon, juntamente com seu funcionamento e as técnicas desenvolvidas para determinar métricas de QoS da rede, tais como, largura de banda e atraso. A Seção 4.2 detalha o módulo de encaminhamento de tráfego por múltiplos caminhos, chamado de MP-Routing. Finalmente, a Seção 4.3 apresenta a visão global da Arquitetura SDN e do controlador Floodlight com as nossas contribuições.

4.1 SDNMon

Esta seção apresenta o módulo de monitoramento, chamado de SDNMon. A Seção 4.1.1 descreve a sua arquitetura. A Seção 4.1.2 descreve o mecanismo de *polling*. A Seção 4.1.3 apresenta o mecanismo de *push*. A Seção 4.1.4 detalha o cálculo da vazão. A Seção 4.1.5 detalha o cálculo do atraso. Finalmente, a Seção 4.1.6 apresenta a Aplicação de Monitoramento. O SDNMon foi publicado na trilha principal do SBRC 2015 (REZENDE et al., 2015).

4.1.1 Arquitetura do Módulo SDNMon

Conforme ilustra a Figura 7, o desenvolvimento do módulo de monitoramento SDNMon é feito sob o cenário de SDN, composta por dispositivos de rede OpenFlow, sendo os dispositivos de rede responsáveis pela composição do Plano de Dados e ao menos um controlador responsável pela composição do Plano de Controle. Além disso, todas as informações coletadas pelo módulo de monitoramento estão disponíveis utilizando o protocolo RESTful, através de uma extensão à API *northbound* do controlador. Nesta interface, a Aplicação de Monitoramento (Seção 4.1.6) proposta consegue interagir com o Módulo de Monitoramento proposto (SDNMon), obtendo dados para apresentação em uma interface

gráfica e, também, indicando qual o nível de granularidade desejado. O diagrama de classes do SDNMon está detalhado no Apêndice A.

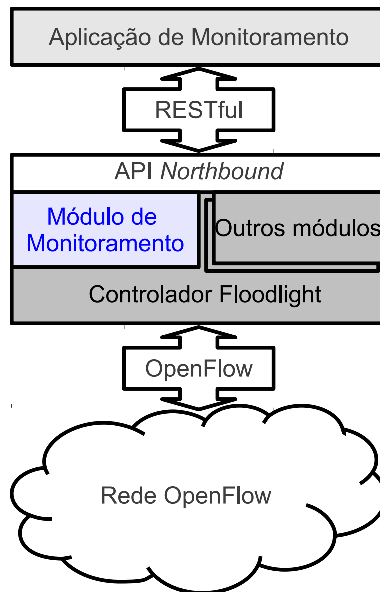


Figura 7 – Uma instância da Arquitetura SDN com as extensões de monitoramento.

O SDNMon é desenvolvido como uma extensão ao controlador Floodlight, sendo que o módulo SDNMon proposto é responsável por coletar, processar e armazenar os dados processados em um banco de dados. A Figura 8 ilustra a arquitetura do SDNMon juntamente com sua interação com outros módulos do controlador Floodlight e com um Banco de Dados. Os componentes do SDNMon e do Floodlight estão detalhados abaixo:

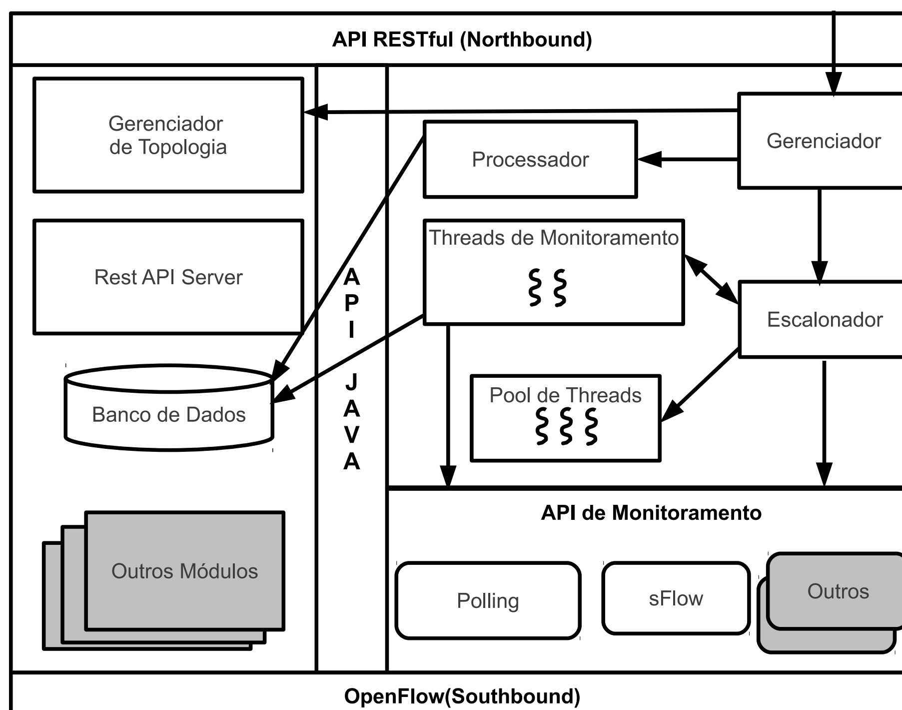


Figura 8 – Arquitetura do SDNMon.

Componentes do SDNMon:

- **Gerenciador**: Responsável por fazer o gerenciamento do módulo SDNMon junto ao Floodlight, definir a ordem de recebimento de mensagens; o nome; o registro e o fornecimento de serviços oferecidos pelo módulo, tanto via API RESTful, como pela API Java. O Gerenciador é, também, responsável por instanciar o Escalonador e o Processador;

- **Processador**: Coleta os dados do Banco de Dados armazenados previamente pelas *Threads* de monitoramento, e os processa para deixar em um formato padronizado para uso futuro. Com isso, o processador permite que dados coletados, indepedentemente do mecanismo de monitoramento e da granularidade da *Thread* de Monitoramento, estejam em um formato adequado para o cálculo de algumas métricas de QoS da rede, tais como vazão, atraso de propagação e perdas de pacote. Estes dados formatados são importantes também para o oferecimento destas métricas para aplicações, através da API Restful, ou para outros módulos, através da API Java;

- **Escalonador**: Responsável por consultar a topologia da rede, acessando o módulo Gerenciador de Topologia fornecido pelo componente Gerenciador, e, inicialmente, criar uma *thread* de monitoramento para cada comutador da rede. Este módulo também é responsável por analisar a carga coletada pelas Threads de Monitoramento e à medida que o tráfego da rede vai variando, este componente é responsável por instanciar, destruir ou realocar novas *threads* do *Pool* de *Threads*, como também, definir o escopo das mesmas, para o monitoramento da rede;

- ***Thread* de Monitoramento**: Responsável por coletar as estatísticas dos comutadores, com o mecanismo de monitoramento (*polling* ou *push*) e através do escopo (por porta, grupo, fluxo ou fila) recebido do Escalonador;

- ***Pool* de *Threads***: Mantém as *threads* que foram instanciadas e estão inutilizadas no momento. Quando o escalonador percebe que é necessário utilizar alguma *thread* para o monitoramento, ele ativa esta *thread*, que provavelmente está "dormindo", e define um escopo de monitoramento para ela. Com o *pool* de *threads*, reduz-se o aumento do gasto computacional para criar e deletar *threads*;

- **Componentes de Monitoramento**: São componentes responsáveis por coletar as estatísticas da rede. Para a coleta das estatísticas, é usado um mecanismo de monitoramento, como o *polling*, o sFlow ou algum outro, futuramente implementado. Após a coleta da estatística, estes dados são armazenados no Banco de Dados;

- **Banco de Dados**: O Banco de Dados armazena dados relativos ao monitoramento (SDNMon) e ao módulo de multi-caminhos (MP-Routing), este último módulo é detalhado na Seção 4.2. A base de dados de monitoramento é povoado da seguinte

forma: todos os dados coletados pelo SDNMon são armazenados na Base de Dados do tipo chave-valor pelas *threads*. A chave refere-se a um dispositivo de rede, uma porta, uma fila ou a um fluxo. No caso de um dispositivo de rede, das portas ou das filas, o campo valor armazena dados referentes ao volume agregado de tráfego sendo experimentado neste componente. No caso de um fluxo, o valor do objeto é o conjunto de arestas do caminho percorrido pelo fluxo; para cada aresta há uma outra entrada na base de dados, cuja chave associa o fluxo ao par de comutadores que compõem a aresta e o valor é o conjunto de informações relativas à mesma. Desta forma, as *threads* podem atualizar as arestas concorrentemente.

Componentes do Floodlight:

- ***Rest API Server***: Todos os módulos que desejam disponibilizar seus serviços através da API RESTful devem registrar-se no *Rest API Server*. Este módulo expõe uma API Java onde o SDNMon, ou qualquer outro módulo, pode definir a *Uniform Resource Identifier* (URI) para acessar os serviços expostos e qual o método usado para os pedidos *Hypertext Transfer Protocol* (HTTP), tais como o *Get* e o *Put*. Lembrando que os dados ofertados através da RESTful API estão no formato *JavaScript Object Notation* (JSON), por padrão;

- **Gerenciador de Topologia**: Este módulo é responsável por manter a topologia da rede, sendo que a topologia é atualizada sempre que acontece algum evento na rede, por exemplo, quando alguma interface do comutador é desligada ou ligada. Este módulo disponibiliza a topologia da rede através de sua API Java, onde iremos, posteriormente, acessar esta API para retornar a topologia da rede que é usada para alimentar o escalonador do SDNMon, possibilitando ao escalonador criar, deletar e atualizar as *threads*;

- **Outros módulos**: O Floodlight possui mais de uma dúzia de módulos e descreveremos apenas os dois módulos que mais interagem com o SDNMon. O Floodlight fornece alguns módulos interessantes, como o *Forwarding*, para o roteamento de pacotes; um módulo de *Firewall*; um módulo para o descobrimento de arestas, *LinkDiscoveryManager*, através do envio de mensagens *Link Layer Discovery Protocol* (LLDP); entre outros.

4.1.2 Mecanismo de Monitoramento baseado em *Polling*

Esta seção detalha como funciona o mecanismo de monitoramento chamado *polling*, juntamente com a adaptabilidade de *threads* proposto no SDNMon, sendo esta adaptabilidade um dos diferenciais em relação aos trabalhos relacionados de monitoramento.

Conforme mencionado anteriormente, o Escalonador consulta informações topológicas através do módulo Gerenciador de Topologia e instancia uma *thread* de monitoramento

para cada um dos dispositivos de rede compondo a SDN. Na sequência, a partir das informações coletadas via mecanismo de *polling* em cada um dos dispositivos de rede, as *threads* de monitoramento realimentam o escalonador, levando a adaptações no número de *threads* instanciadas.

Os dados são coletados através do envio de mensagens de *Statistic Request* a partir de cada uma das *threads* aos respectivos dispositivos de rede. O teor das solicitações enviadas nas mensagens é definido pelo escalonador de acordo com o escopo de monitoramento atribuído a cada uma das *threads*. Uma vez que inicialmente há uma *thread* para cada dispositivo de rede, as mensagens utilizam coringas (*wildcards*) em todos os campos previstos na estrutura da mensagem de *Statistic Request*. Ao receber uma requisição como esta, o dispositivo de rede devolve uma mensagem do tipo *Statistic Reply*, contendo todos os valores contabilizados.

Ao receber o retorno de um dispositivo de rede, a *thread*, além de armazenar as informações na base de dados, analisa o volume de informações retornado e comunica ao escalonador para que este decida sobre a necessidade de novas *threads*, para eliminar gargalos de monitoramento. Utilizando desta técnica de *feedback*, o escalonador é capaz de instanciar novas *threads* e definir o escopo de monitoramento atuando nos campos solicitados nas mensagens de *Statistic Request*, ou seja, o escalonador determina o escopo de monitoramento através da definição dos coringas em cada uma das *threads* instanciadas.

O Processador consulta, periodicamente, o Banco de Dados para verificar se há novos dados. Em caso positivo, ele processa e formata os dados de acordo com os requisitos do usuário, ou seja, alguns usuários podem preferir, por exemplo, a vazão representada em Mbits/s, já outros podem preferir em MBytes/s. Após o processamento e formatação dos dados, estes são armazenados no Banco de Dados, e quando um usuário requisitar alguma métrica, através da API RESTful, o Processador busca os dados e retorna ao usuário.

A partir da especificação atual dos coringas presente no OpenFlow, a SDNMon consegue instanciar *threads* responsáveis pelo monitoramento na granularidade de dispositivos de rede, portas individuais dos dispositivos de rede, filas ou fluxos individuais. Como investigação futura, um dos itens de interesse é o suporte por parte do OpenFlow para a definição de coringas que representam intervalos, permitindo a instanciação de *threads* responsáveis, por exemplo, pelo monitoramento de um conjunto de portas dos dispositivos de rede ou um conjunto de filas. Outro aspecto aberto para investigação futura está relacionado ao sincronismo entre as *threads*, o impacto que o sincronismo causa no volume de tráfego de controle e a sobrecarga de processamento aplicada nos dispositivos compondo o Plano de Dados da SDN.

Alguns exemplos de mensagens OpenFlow *Statistics Request* e *Statistics Reply* foram colocados no Apêndice B.

4.1.3 Mecanismo de Monitoramento baseado em *Push*

Como descrito na Seção 2.2.3, existem alguns mecanismos de monitoramento baseado no modelo *push* de dados, como o NetFlow e o sFlow. No módulo SDNMon, implementamos um mecanismo para coletar os dados do protocolo sFlow. A Figura 9 detalha como ocorre a coleta dos dados através da técnica sFlow. Os componentes do sFlow são descritos abaixo:

- **Agente sFlow**: O Agente sFlow é um processo executado em um comutador ou roteador, responsável por inspecionar 1 entre N pacotes que passam nas interfaces do dispositivo. Ao inspecionar um pacote, o Agente sFlow extrai seu cabeçalho, e adiciona informações deste cabeçalho juntamente com algumas informações relativas ao dispositivo, tais como interface de entrada e saída do pacote, em um datagrama sFlow. Cada amostra de pacote inserida em um datagrama sFlow equivale a 7% (HP, 2007) do tamanho de um pacote normal, e portanto, é possível inserir várias amostras de pacotes em um mesmo datagrama sFlow. Após o datagrama sFlow atingir um certo número de amostras, ele é encapsulado em um pacote UDP e enviado ao coletor;

- **Coletor sFlow**: Este processo, geralmente executado em um servidor, é o cerébro do sFlow, pois todo o processamento dos dados ocorre nele, deixando o Agente sFlow apenas com a responsabilidade de inspecionar, preencher os datagramas sFlow e enviar ao Coletor sFlow. O Coletor sFlow recebe os datagramas sFlow e produz uma visão do estado da rede, mostrando os enlaces que estão recebendo ou enviando mais tráfego, a vazão de cada fluxo capturado, entre outros. Essas informações processadas e estimadas são armazenadas em uma Base de Dados e disponibilizadas às aplicações através da API Restful do sFlow.

O Escalonador do SDNMon, ao verificar que o monitoramento sFlow foi iniciado, atribui a uma *thread* a responsabilidade de consultar, periodicamente, o Coletor sFlow, como também armazenar estes dados consultados no Banco de Dados do SDNMon. Posteriormente, o Processador trabalha os dados do sFlow e os deixa no formato requisitado pelos usuários. Como os dados do sFlow já vem pré-processados pelo Coletor sFlow, não é necessário calcular, por exemplo, a vazão usada por cada fluxo e o processamento fica restrito a pequenas formatações dos dados.

4.1.4 Cálculo da Vazão

A Figura 10 apresenta o método proposto neste trabalho para o cálculo da vazão a partir do total de dados transmitidos. O método pode ser aplicado independentemente da granularidade dos dados obtidos pelo mecanismo de *polling*. Considerando a coleta de dados referente a um fluxo em um mesmo dispositivo de rede, temos que a Série 1,

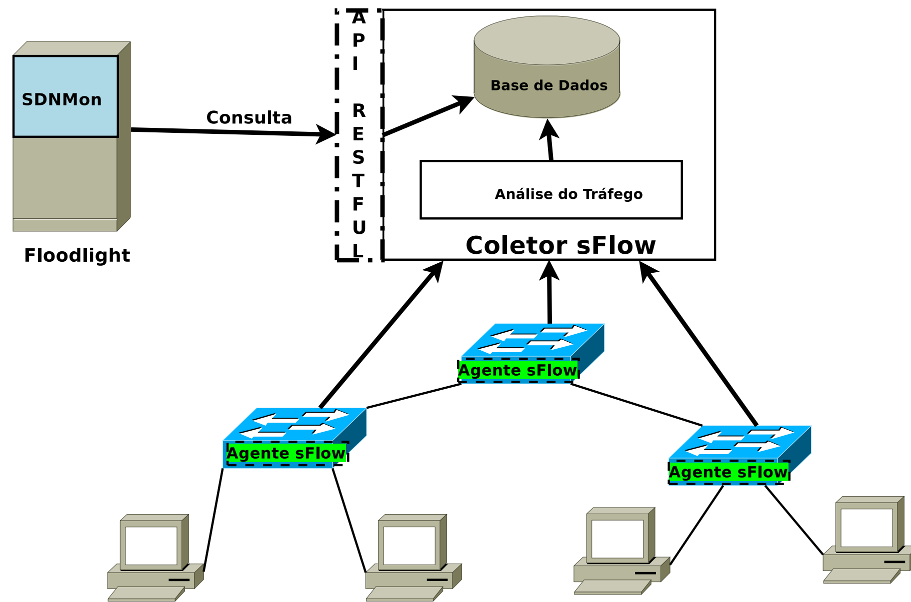


Figura 9 – Coleta através do sFlow.

apresentada na Figura 10, corresponde ao total de dados transmitidos no fluxo em questão para este dispositivo de rede. Através da coleta de dois pontos consecutivos é possível obter a vazão referente ao fluxo neste dispositivo de rede.

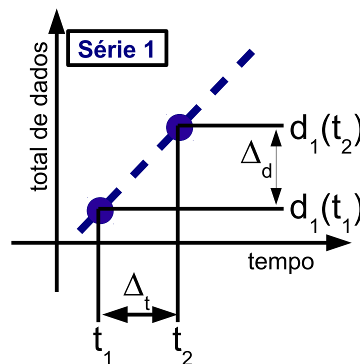


Figura 10 – Cálculo da vazão.

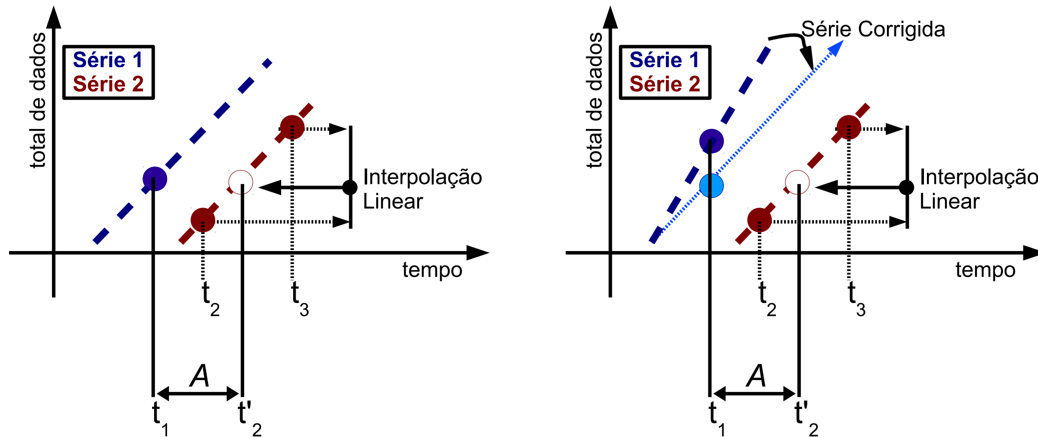
Conforme ilustrado na Figura 10, é possível obter o total de dados transmitidos entre os dois pontos coletados utilizando a fórmula $\Delta_d = d_1(t_2) - d_1(t_1)$, onde $d_x(t_y)$ representa o total de dados transmitidos na Série x em um determinado instante de tempo y. Ainda considerando estes dois pontos, é possível obter o intervalo de tempo entre eles, utilizando a fórmula $\Delta_t = t_2 - t_1$, onde t_y corresponde ao instante no qual as respectivas medições foram efetuadas. A partir destes valores, é possível obter a vazão $T^1_{t1,t2} = \Delta_d/\Delta_t$ da Série 1 no intervalo t_1, t_2.

4.1.5 Cálculo do Atraso

É comum encontrarmos trabalhos na literatura, por exemplo (PHEMIUS; BOUET, 2013) apresentado na Seção 3.1.3, que injetam sondas (pacotes de *probe*) na rede para

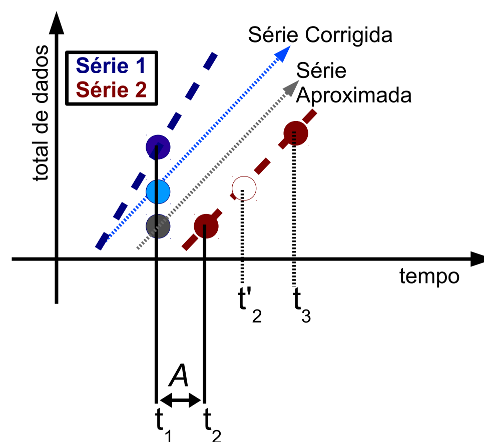
determinar o atraso sendo experimentado. Este tipo de abordagem é bastante simples de ser implementada, mas pode não representar precisamente o atraso sendo experimentado por um determinado fluxo. Atualmente, é comum utilizarmos diversas filas de encaminhamento em uma única porta de um dispositivo de rede, fato este que exigiria a inserção de diversas sondas, específicas para cada uma das possíveis filas ao longo do caminho.

Em nossa abordagem, propomos a utilização de informações referentes ao total de dados transmitidos em cada um dos dispositivos de rede, de tal forma a efetuar o cálculo de atraso em diferentes granularidades, de acordo com os valores obtidos nos contadores especificados pelo OpenFlow. A Figura 11(a) apresenta o método proposto em um cenário livre de interferências, sem a ocorrência de descartes de dados, consistindo essencialmente em estimar o intervalo de tempo necessário para que cada elemento receba todos os dados enviados pelo anterior.



(a) Séries sem a ocorrência de descarte de dados.

(b) Utilizando um contador de total de dados descartados.



(c) Sem a utilização de um contador de dados descartados.

Figura 11 – Cálculo do atraso.

Considerando o cálculo de atraso para um determinado fluxo, a Figura 11(a) apresenta as Séries 1 e 2, coletadas em dois dispositivos de rede adjacentes, através dos quais o

fluxo em questão é encaminhado. O método consiste em determinar t_1, t_2 e t_3, tal que $d_2(t_2) \leq d_1(t_1) \leq d_2(t_3)$.

O próximo passo é determinar $T^2_{t2,t3}$, conforme definido na Seção 4.1.4. Finalmente, determinar t'_2, tal que $d_1(t_1) = d_2(t'_2)$, utilizando uma interpolação linear, para obter o atraso $A = t'_2 - t_1$.

O contador de total de bytes transmitidos do OpenFlow, mantido pelos dispositivos de rede, contabiliza o total bruto de bytes transmitidos, incluindo os bytes referentes aos pacotes efetivamente transmitidos e, também, os bytes referentes aos pacotes descartados. Desta forma, a Figura 11(b) ilustra como esta condição interfere no cálculo do atraso experimentado. Para determinar o atraso real, é preciso subtrair da Série 1 o total de dados descartados, obtendo o total de dados efetivamente transmitidos no instante de tempo em questão, representado pela Série Corrigida na Figura 11(b). Após a correção da série de dados, aplica-se o mesmo método descrito anteriormente para obter o valor de atraso.

Entretanto, a especificação do OpenFlow contempla apenas o contador referente aos bytes descartados por porta, não sendo possível obter informações referentes aos bytes descartados por fila ou fluxo. Sendo assim, a Figura 11(c) apresenta um cenário alternativo, onde o ponto da Série 1 obtido no instante t_1 é aproximado para o correspondente em total de dados do ponto da Série 2 obtido no instante t_2, considerando $d_{ap}(t_1) \approx d_2(t_2)$. Neste caso, descarta-se a interpolação linear, sendo possível obter o atraso $A = t_2 - t_1$. Entretanto, como pode ser observando na Figura 11(c), esta metodologia compromete a precisão do cálculo de atraso. A Seção 5.1.3.4 apresenta uma análise sobre o nível de precisão perdido neste caso.

O Apêndice C detalha a adição dos novos contadores nos comutadores e no controlador Openflow, que nos permitiram fazer experimentos considerando a série corrigida referente a Figura 11(b).

4.1.6 Aplicação de Monitoramento

O SDNMon contempla a criação de uma aplicação, desenvolvida em Javascript, para permitir que o administrador de rede possa ter uma visão completa da rede. A partir desta aplicação, o administrador pode visualizar possíveis gargalos na rede, em diferentes granularidades, quais *hosts* estão sendo utilizados, gerar gráficos dos tráfegos, entre outras funcionalidades, de maneira intuitiva, precisa e de fácil customização.

A aplicação proposta comunica-se com o controlador Floodlight através da API RESTful do controlador. Algumas informações relativas à rede já estão disponíveis, por padrão, no Floodlight, e portanto a aplicação as acessa via o protocolo HTTP. Entretanto, algumas informações, tal como métricas de monitoramento, são disponibilizadas pela API RESTful do controlador, através do módulo SDNMon. As Figuras 12 e 13 ilustram a Aplicação de Monitoramento desenvolvida.

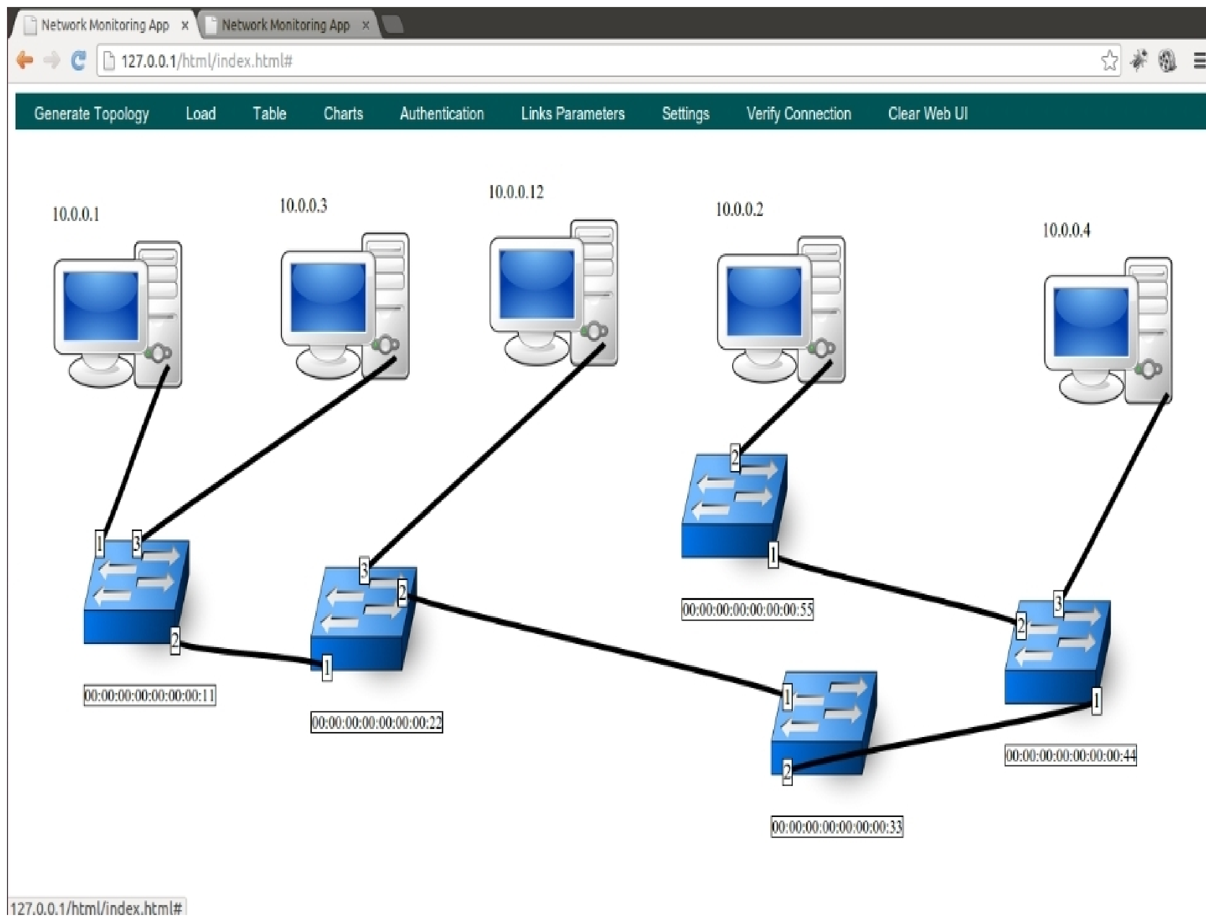


Figura 12 – Ilustração da Aplicação de Monitoramento.

A Figura 12 detalha uma topologia com cinco comutadores e cinco *hosts* juntamente com as arestas. É possível notar que acima da topologia há um menu com algumas funcionalidades da aplicação. Neste menu é possível selecionar opções para mostrar informações do estado da rede, tal como os fluxos, as portas, as filas e seus respectivos tráfegos, além disso, há informações sobre o estado do controlador, como o gasto atual de memória RAM e por quanto tempo ele está ativo.

A Figura 13 apresenta a tabela de fluxos no canto superior direito, junto com o tráfego de um fluxo escolhido, sendo mostrado nos rótulos das arestas, e o tráfego agregado das interfaces dos dispositivos sendo mostrado embaixo do comutador.

4.2 MP-Routing

Esta seção descreve o módulo de multi-caminhos, MP-Routing, proposto neste trabalho. A Seção 4.2.1 detalha a arquitetura do MP-Routing. A Seção 4.2.2 descreve o funcionamento do mecanismo de encaminhamento de pacotes proposto. Finalmente, a Seção 4.2.3 descreve a extensão necessária para a correta interação do MP-Routing com os dispositivos de rede. O MP-Routing encontra-se, atualmente, submetido à trilha principal do SBRC 2016 (REZENDE et al., 2016).

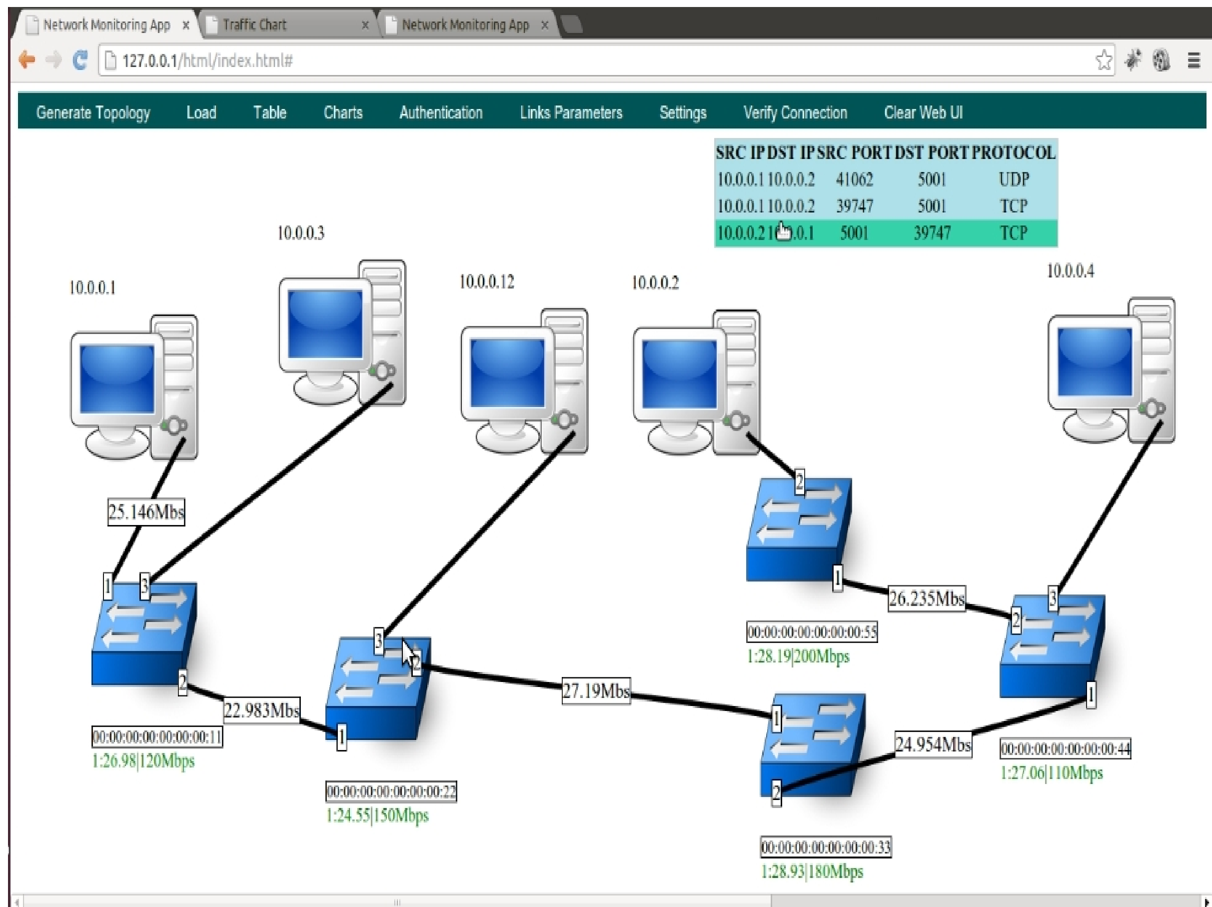


Figura 13 – Tabela de Fluxos e a Vazão do Fluxo e das Portas.

4.2.1 Arquitetura do módulo MP-Routing

Confome ilustra a Figura 14, o módulo MP-Routing constitui uma extensão do controlador Floodlight, e como tal, usufrui de todos os benefícios, como o acesso à outros módulos através da Java API, oferecimento de serviços pela API Restful, e acesso ao dispositivo de rede através do protocolo OpenFlow. O diagrama de classes do MP-Routing pode ser visualizado no Apêndice A. O MP-Routing é composto por três componentes, descritos a seguir:

- **Gerenciador**: Responsável por comunicar com o componente Seletor, com os outros módulos do controlador e com o Banco de Dados. Esta interação visa atender pedidos de requisição feitas pela API RESTful. Além disso, o Gerenciador repassa alguns serviços necessários ao Seletor e Construtor, como o serviço de enviar mensagens aos dispositivos OpenFlow. A comunicação com os outros módulos é fundamental para que o MP-Routing acesse os serviços do controlador, exemplos de tais serviços são o acesso à topologia atualizada da rede, através do módulo Gerenciador de Topologia, e também o registro do MP-Routing no REST API Server do controlador. O acesso ao Banco de Dados é necessário para obter informações relativas às métricas de QoS da rede;

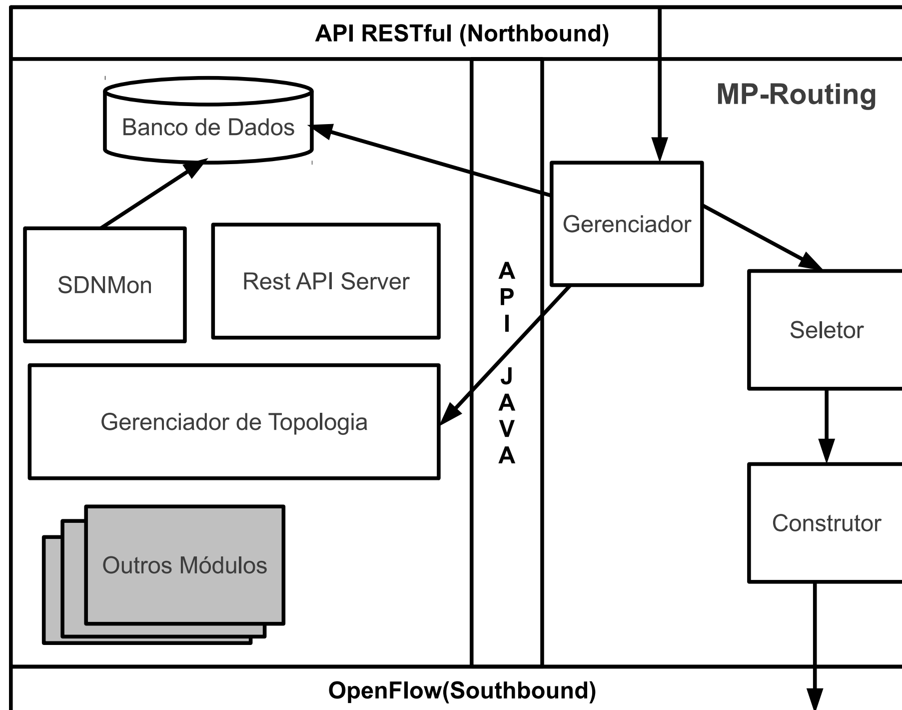


Figura 14 – Arquitetura MP-Routing.

- **Seletor**: Responsável por definir as rotas que serão aplicadas, posteriormente, nos dispositivos de rede. Este componente é executado quando o Gerenciador requisita rotas para um novo fluxo. Ao requisitar uma nova rota, o Gerenciador envia ao Seletor os seguintes parâmetros: cabeçalho do pacote, extraído através da mensagem *Packet IN*; métricas de QoS representando o estado atual da rede; SLA do usuário; e a topologia da rede. Através destes parâmetros, o Seletor procura por rotas que satisfaçam o SLA do usuário, subsequentemente, enviando um *feedback* ao Gerenciador e alimentando o Construtor;

- **Construtor**: Tem a responsabilidade de formatar as rotas e inserí-las nos comutadores. Ao encontrar as rotas de encaminhamento de um determinado fluxo, o Seletor passa as rotas para o Construtor, e este tem a responsabilidade de instanciar mensagens OpenFlow e preenchê-las de acordo com as rotas, para que sejam enviadas, imediatamente, aos dispositivos. As mensagens OpenFlow enviadas incluem a criação de entradas de grupo na Tabela de Grupo e, também, de entradas de fluxo na Tabela de Fluxo dos comutadores.

O comutador, ao receber pacotes que não casam com nenhuma das regras das entradas de fluxos, envia, simultaneamente, muitas mensagens *Packet-Ins* ao controlador. Devido à natureza *multithreaded* do Floodlight, estas mensagens requisitam, simultaneamente, novos pedidos de inserção de fluxos ao Gerenciador. Desta forma, o Gerenciador mantém uma fila para armazenar essas requisições e controlar o repasse destas, individualmente, ao Seletor à medida que as requisições são atendidas.

As inúmeras requisições não podem ser atendidas simultaneamente, pois elas iriam acessar as mesmas métricas da rede. À medida que um fluxo vai sendo criado, as requisições atendidas simultaneamente, não saberiam que um novo fluxo foi criado, e portanto, não estariam usando métricas de QoS atualizadas da topologia e isso comprometeria atender a QoS.

4.2.2 Funcionamento do MP-Routing

O funcionamento do MP-Routing é detalhado na Figura 15 e tem como referência o protocolo OpenFlow (ONF, 2013). Um comutador OpenFlow pode gerar uma mensagem *Packet-In* nas seguintes situações: quando o pacote processado não casa com nenhum fluxo da Tabela de fluxos; quando a Instrução da entrada de fluxo tiver uma ação explícita de enviar para o controlador; ou se o pacote processado tiver um *Time to Live* (TTL) inválido. Uma mensagem *Packet-In* enviada pelo comutador pode conter o pacote completo, partes do pacote, ou nenhuma informação do pacote que foi responsável por gerar essa *Packet-In* em seu *payload*. O *Packet-In* tem em seu cabeçalho informações importantes, tais como a razão pela qual o *Packet-In* foi gerado e o cabeçalho do pacote.

Quando um controlador recebe uma mensagem *Packet-In*, esta mensagem é distribuída, em uma ordem de prioridade definida no controlador, aos módulos que registraram-se para receber o *Packet-In*. Quando o MP-Routing recebe uma mensagem *Packet-In*, o componente que manipula esta mensagem é o Gerenciador, verificando se o campo razão, do cabeçalho *Packet-In*, tem o valor igual a 0, pois este valor significa que o *Packet-In* foi gerado devido ao não casamento dos cabeçalhos do pacote com as regras das entradas de fluxo do comutador. Caso o valor seja 0, o MP-Routing trata este *Packet-In*, senão, o MP-Routing o rejeita.

Ao verificar que o *Packet-In* tem o valor da razão 0, os seguintes argumentos são enviados, pelo Gerenciador, ao Seletor: campos do cabeçalho do pacote que identificam os endereços IP de origem e destino, os números das Portas de Origem e de Destino da aplicação; as métricas da rede, tais como a largura de banda e perdas de pacotes; os SLAs do usuário e; finalmente, o objeto referente à topologia da rede.

Ao receber os parâmetros do Gerenciador, o Seletor, primeiramente, procura por um único caminho, o menor caminho disponível na rede que satisfaça o SLA. Para procurar este caminho, o Seletor usa um algoritmo baseado no algoritmo de Dijkstra(DIJKSTRA, 1959), com algumas modificações, pois precisa considerar o SLA do usuário e também os recursos disponíveis na rede. Todo este processo está detalhado no Algoritmo 1, composto por dois procedimentos. O primeiro, chamado de Principal, recebe os seguintes argumentos de entrada: IP de origem; IP de destino; topologia da rede; métricas da rede; e SLA do usuário. O objetivo desse procedimento é retornar o menor caminho, em número de saltos, e que ao mesmo tempo garanta o SLA do usuário. O procedimento `Remover_Arestas`, remove as arestas da topologia que não satisfazem todas as métricas do SLA do usuário.

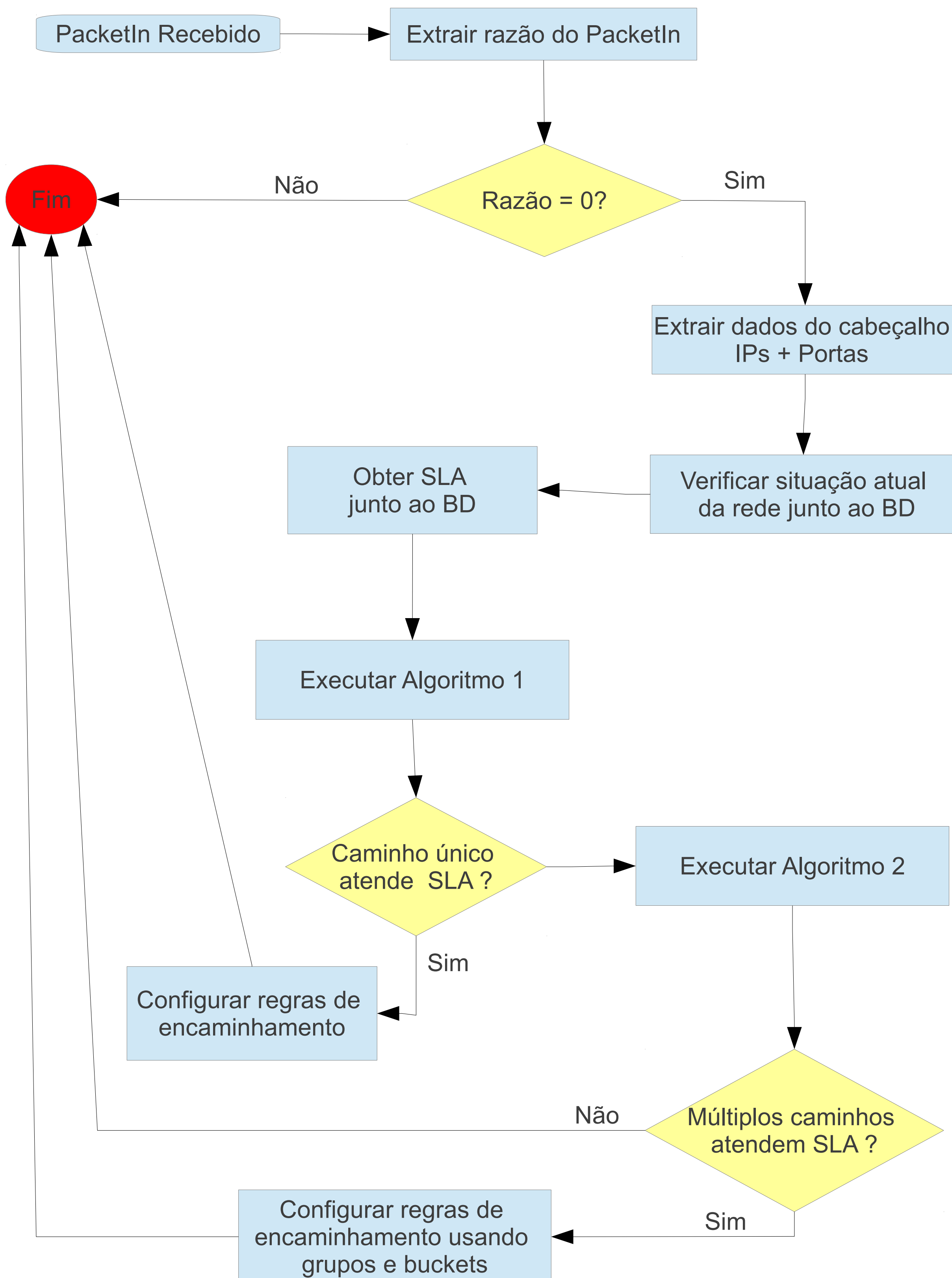


Figura 15 – Fluxograma do Módulo MP-Routing.

A Linha 2 do Algoritmo 1 faz uma cópia profunda (*deep copy*) da topologia, ou seja, toda a topologia é clonada para um outro espaço de memória, e referenciada, apenas, pela variável `copia_topologia`. É necessário clonar a topologia, pois senão qualquer mudança feita pelo controlador na topologia iria afetar todos os módulos que acessam a topologia. Na Linha 3, são removidas todas as arestas da `copia_topologia` que não honrem o SLA do usuário. Após a remoção das arestas na Linha 3, a Linha 4 executa o algoritmo de Dijkstra na topologia resultante, procurando o menor caminho que garanta o SLA do usuário. Caso algum caminho exista, o algoritmo de Dijkstra o retorna, senão retorna `NULL`.

O procedimento `Remover_Arestas`, percorre todas as arestas da topologia (Linha 8). Em cada iteração, recebe as métricas da aresta atual (Linha 10) e em cada iteração das Linhas (11-18), uma métrica (largura de banda, atraso, entre outros) da aresta é verificada, para ver se é possível satisfazer à respectiva métrica do SLA do usuário (Linhas 11-16). Caso alguma métrica não satisfaça o SLA requisitado pelo usuário, ou seja, a variável `valor_aresta` não tem um valor aceitável para a variável `valor_sla`, em uma determinada métrica, então remova a aresta (Linha 17) e mude para uma outra aresta (Linha 18), pois, todas as métricas de uma aresta devem garantir os requisitos do usuário. Após percorrer todas as arestas, um dos argumentos de entrada, `topologia`, terá apenas as arestas que honrem o SLA do usuário, e portanto, o procedimento retornará (Linha 19) ao procedimento `Principal`.

Algoritmo 1 Cálculo para achar um único caminho.

```
 1: procedure PRINCIPAL(host_inicial, host_final, topologia, metricas_rede, sla_usuario)
 2:     topologia_copia ← Clonar topologia
 3:     REMOVER_ARESTAS(topologia_copia, metricas_rede, sla_usuario)
 4:     caminho ← Dijkstra(host_inicial, host_final, topologia_copia)
 5:     return caminho
 6:
 7: procedure REMOVER_ARESTAS(topologia, metricas_rede, sla_usuario)
 8:     for all aresta em topologia do
 9:         aresta ← topologia.getAresta(i)
10:         metricas_aresta ← metricas_rede.getAresta(aresta)
11:         for all metrica em sla_usuario do
12:             metrica_requisitada ← sla_usuario.getMetrica(j)
13:             metrica_nome ← metrica_requisitada.getNome()
14:             valor_sla ← metrica_requisitada.getValor()
15:             valor_aresta ← metricas_aresta.getValor(metrica_nome)
16:             if garanteSLA(valor_sla, valor_aresta, metrica_nome) = false then
17:                 topologia.remove(aresta)
18:                 break
19:     return
```

Continuando o Fluxograma na Figura 15, ao término da execução do procedimento `Principal` do Algoritmo 1, é verificado se existe uma rota capaz de honrar o SLA. Caso exista, o componente Construtor atua nos comutadores, inserindo a rota no Plano de Dados e, o processamento do MP-Routing para este fluxo é concluído. Caso o `caminho`

não tenha nenhuma rota, então tente achar múltiplos caminhos que satisfaçam o SLA do usuário.

O Algoritmo 2 detalha o encaminhamento por múltiplos caminhos. Os argumentos de entrada do procedimento `Principal` são os mesmos do procedimento, de mesmo nome, no Algoritmo 1. São feitas cópias profundas das métricas de rede, do SLA do usuário e da topologia (Linha 2-4), porque senão, essas manipulações das estruturas de dados iriam causar, certamente, inconsistência nos outros módulos, já que muitos módulos usam o mesmo serviço.

Na Linha 5, as arestas da topologia que não garantirem o SLA do usuário são removidas. Neste caso, apenas o atraso de propagação e a perda de pacotes de uma aresta são consideradas, entretanto, futuramente, este procedimento pode ser estendido para levar em consideração outras métricas, tais como *jitter*, taxas de erros, disponibilidade, entre outros. O motivo pelo qual a largura de banda não foi contemplada na Linha 5, é devido à sua garantia através de múltiplos caminhos, necessitando de uma visão agregada das arestas, ao invés de uma visão individual. A Linha 6 verifica se a largura de banda do SLA do usuário foi honrada, em caso positivo, vai para Linha 14; senão, repete o laço do *While*. A linha 7 executa o algoritmo Dijkstra, retornando o menor caminho entre o `host_inicial` e o `host_final`.

Se a rota retornada pelo algoritmo de Dijkstra for `NULL`, significa que não existe caminho remanescente entre os dois `hosts`, e portanto, o agregado dos caminhos não garante a largura de banda requisitada pelo usuário e é atribuído o valor `NULL` aos `caminhos` (Linha 9), além de encerrar a execução do laço *While* (Linha 10). O Seletor então envia uma mensagem ao Gerenciador avisando que não foi possível garantir, o encaminhamento do fluxo, por múltiplos caminhos.

Por outro lado, se a rota retornada pelo algoritmo de Dijkstra for diferente de `NULL`, existe um outro caminho remanescente entre os dois `hosts`, que é adicionado à lista de caminhos (Linha 11). A variável `caminhos` pode já conter alguns caminhos, adicionados anteriormente devido às execuções anteriores do laço *While*; ou também pode estar vazio, significando que este caminho encontrado nesta iteração atual do laço *While* será a cabeça da lista.

O procedimento `Decrementa_LB` (Linha 12) atualiza as métricas da rede e o restante da largura de banda que deve ser garantida no SLA do usuário, tendo como base os caminhos já encontrados. O procedimento `Remover_Arestas` (Linha 13) remove as arestas da topologia que não possuem mais largura de banda residual. Finalmente, o Seletor retorna os caminhos (Linha 14) para o Construtor efetuar as devidas configurações no Plano de Dados.

O procedimento `Situacao` (Linhas 16-20) tem como argumento de entrada o SLA do usuário. Primeiramente, é atribuído a largura de banda do usuário que ainda precisa ser honrada à variável `largura_de_banda` (Linha 17), e verifica-se na Linha 18 se o valor da

`largura_de_banda` é igual ou diferente de 0. Se for 0, isso quer dizer que a largura de banda requisitada pelo usuário foi garantida, e consequentemente, retorna-se `true` (Linha 20); senão, retorna-se `false` (Linha 19), significando que a largura de banda ainda não foi completamente garantida.

O procedimento `Decrementa_LB` (Linhas 22-28) tem como argumento de entrada as métricas da rede, o remanescente do SLA do usuário que necessita ser garantido, e as rotas entre os dois *hosts* encontradas pelo algoritmo de Dijsktra. A Linha 23 atribui a menor largura de banda disponível no conjunto `rotas` na variável `menor_valor`, o motivo pelo qual foi escolhido o menor valor é que durante uma transmissão de tráfego por um caminho, a aresta com menor largura de banda disponível define a vazão que passa no caminho, ou seja, o tráfego pode passar por arestas com largura de banda superiores, mas ao chegar numa aresta com pouca banda disponível, o tráfego seria enfileirado ou descartado no dispositivo, ocasionando alguns efeitos colaterais, tais como aumento dos atrasos de enfileiramento, processamento e transmissão no dispositivo, como também retransmissões ou perdas definitivas de pacotes.

As (Linhas 24-27) fazem uma iteração sobre cada aresta contida em `rotas`, o motivo dessa iteração é fazer a subtração da largura de banda da aresta pelo `menor_valor` (Linha 26), como também subtrair o `menor_valor` na largura de banda do SLA do usuário (Linha 27). A Linha 28 encerra a execução deste procedimento.

O procedimento `Remover_Arestas` (Linhas 30-37) tem os seguintes argumentos de entrada: topologia da rede e as métricas da rede. O procedimento itera sobre cada aresta da topologia (Linhas 31-35), e para cada iteração, verifica se a largura de banda remanescente da aresta é igual a 0 (Linha 34). Caso seja igual a 0, então isso significa que a aresta não pode mais ser utilizada para o cálculo de novos caminhos, então é necessário remove-la da topologia (Linha 35); senão, mantenha a aresta na topologia. A Linha 36 encerra o procedimento.

4.2.3 Extensão Open vSwitch

Além da extensão do controlador Floodlight, é necessário estender o Open vSwitch para suportar o uso de múltiplos caminhos, pois ele, atualmente, não possui um escalonador. A nossa principal contribuição nesse aspecto é o algoritmo de seleção de pacotes. O algoritmo de seleção é um escalonador de pacotes responsável por definir para qual interface é enviado um subfluxo de pacotes, baseado em um intervalo de tempo definido pelo peso dos *buckets*.

O algoritmo Seletor de *Bucket* é descrito no Algoritmo 3. O algoritmo é composto por dois procedimentos, o `Principal` e o `Auxiliar`. O procedimento `Principal` utiliza dois argumentos de entrada: a entrada de grupo, chamada no procedimento de `grupo`; e também, um vetor de tamanho dois, chamado de `par`. O primeiro elemento do vetor `par` é o *bucket* usado, no momento, para transmissão de dados em um grupo, e o segundo ele-

Algoritmo 2 Cálculo para achar múltiplos caminhos.

```
1: procedure Principal(host_inicial, host_final, topologia, metricas_rede, sla_usuario)
2:     metricas_rede_clone ← Clonar metricas_rede
3:     sla_clone ← Clonar sla_usuario
4:     topologia_clone ← Clonar topologia
5:     Remove_Arestas_Metricas_Insatisfeitas(sla_usuario, metricas_rede.getAtraso(),
   metricas_rede.getPerdaPacotes(), topologia)
6:     while Situacao(sla_clone) ≠ true do
7:         rotas ← Djikstra(host_inicial, host_final, topologia_clone)
8:         if rotas = NULL then
9:             caminhos ← NULL
10:            break
11:        caminhos ← caminhos ∪ rotas
12:        Decremeta_LB(metricas_rede_clone, sla_clone, rotas)
13:        Remover_Arestas(topologia_clone,metricas_rede_clone, sla_clone)
14:    return caminhos
15:
16: procedure Situacao(sla_clone)
17:    largura_de_banda ← sla_clone.getLarguraBanda()
18:    if largura_de_banda ≠ 0 then
19:        return false
20:    return true
21:
22: procedure Decremeta_LB(metricas_rede, sla_usuario, rotas)
23:    menor_valor ← rotas.getMenorLarguraBanda(rotas)
24:    for all aresta em rotas do
25:        metricas_aresta ← metricas_rede.getAresta(aresta)
26:        metrica_aresta.setLarguraBanda(metrica_aresta.getLarguraBanda() − menor_valor)
27:        sla_usuario.setLarguraBanda(sla_usuario.getLarguraBanda() − menor_valor)
28:    return
29:
30: procedure Remover_Arestas(topologia, metricas_rede)
31:    for all aresta em topologia do
32:        aresta ← topologia.getAresta(i)
33:        metricas_aresta ← metricas_rede.getAresta(aresta)
34:        if metricas_aresta.getMetrica(largura_banda) == 0 then
35:            topologia.remove(aresta)
36:    return
```

mento delimita até qual horário, definido em milisegundos, o *bucket* atual pode transmitir os pacotes. O procedimento `Principal` é chamado apenas quando um pacote passar por uma entrada de grupo que seja do tipo *Select*, e vale ressaltar que cada `grupo` do tipo *Select* precisa ter um vetor `par`, pois os grupos são independentes.

A variável ***bucket*** `_atual` recebe o primeiro valor de `par` (Linha 02) e na sequência verifica se o valor (Linha 03 - `NULL` ou não `NULL`). Se `NULL`, então é o primeiro pacote do fluxo e, assim, escolhe-se o primeiro *bucket* do grupo. Na sequência atualiza-se o tempo máximo que o *bucket* pode transmitir, bem como retorna o *bucket* (Linhas 04-06).

Se o valor não é `NULL` (Linha 03), isto significa que o pacote sendo processado não é o primeiro pacote do fluxo, então este terá o tempo do `par(1)` (Linha 08). Na sequência é comparado se o tempo corrente, coletado na Linha 9, é menor que o tempo do `par(1)` (Linha 10). Se verdadeiro, então o *bucket* corrente pode ainda enviar o pacote (Linha 11).

Algoritmo 3 Seletor de *Bucket*.

```
 1: procedure Principal(grupo, par)
 2:     bucket_atual ← par(0)
 3:     if bucket_atual = NULL then
 4:         bucket_atual ← Receber primeiro bucket do grupo
 5:         Auxiliar(grupo, par, bucket_atual)
 6:         return bucket_atual
 7:     else
 8:         tempo_permitido ← par(1)
 9:         tempo_atual ← Retornar o horário atual em milisegundos
10:         if tempo_permitido > tempo_atual then
11:             return par(0)
12:         else
13:             bucket_atual ← Receber o próximo bucket do grupo
14:             Auxiliar(grupo, par, bucket_atual)
15:             return bucket_atual
16:
17: procedure Auxiliar(grupo, par, bucket_atual)
18:     tempo_atual ← Retornar o horário atual em milisegundos
19:     peso_bucket ← Retornar o peso do bucket_atual
20:     tempo_permitido ← tempo_atual + peso_bucket
21:     par(0) ← bucket_atual
22:     par(1) ← tempo_permitido
```

Se falso, pega-se o próximo *bucket* do `grupo`, atualiza o `par` do grupo, onde a primeira posição é o *bucket* escolhido e a segunda posição o tempo máximo para transmissão, bem como retorna o *bucket* (Linhas 13-15).

O procedimento `Auxiliar` possui 03 argumentos de entrada: `grupo`; `par`; e `bucket_atual`. Este procedimento obtém o tempo corrente (Linha 18) e o peso da estrutura `bucket_atual` (Linha 19). Em seguida obtém-se o `tempo_permitido` (Linha 20). Esta variável representa o máximo de tempo, em milisegundos, que um *bucket* tem permissão para transmitir. Na sequência, armazena o *bucket* corrente na posição 0 (Linha 21) e o tempo calculado na posição 1 do vetor `par` (Linha 22).

A Figura 16 detalha o fluxo de dados entre os componentes de um comutador OpenFlow. O componente acinzentado é a nossa extensão no dispositivo. Este Escalonador atua nos pacotes recebidos pela entrada de grupo do tipo *SELECT*, e os envia às interfaces do dispositivo.

A Figura 17 detalha o funcionamento do escalonador proposto, inspirado no algoritmo *Round Robin*. Neste exemplo, a Tabela de Grupo contém três entradas identificadas como $\#A$, $\#B$ e $\#C$, onde o número do grupo aparece após $G\#$. Já o valor que aparece após $B\#$ representa o número do *bucket* e o número entre parêntesis representa o peso do *bucket*. Por fim, o número que segue a letra I representa o número da interface de saída. Neste exemplo, o Grupo $G\#1$ tem dois *buckets*, $B\#1$ e $B\#2$, com pesos 3 e 2, respectivamente.

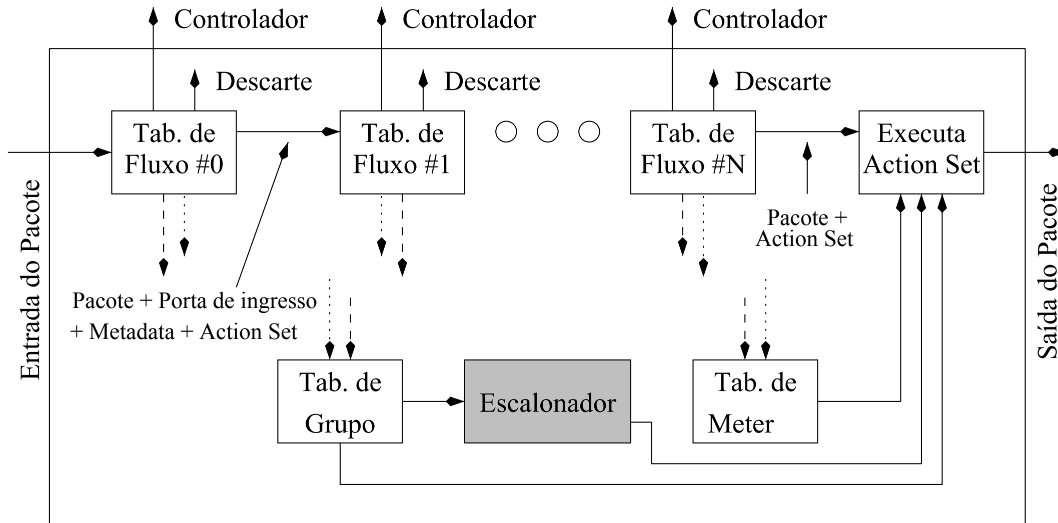


Figura 16 – Extensão do Comutador OpenFlow.

O primeiro *bucket* aponta para a interface $I1$ e o segundo para a interface $I2$. Uma estruturação similar foi definida na Figura 17 para os Grupos $G\#2$ e $G\#3$.

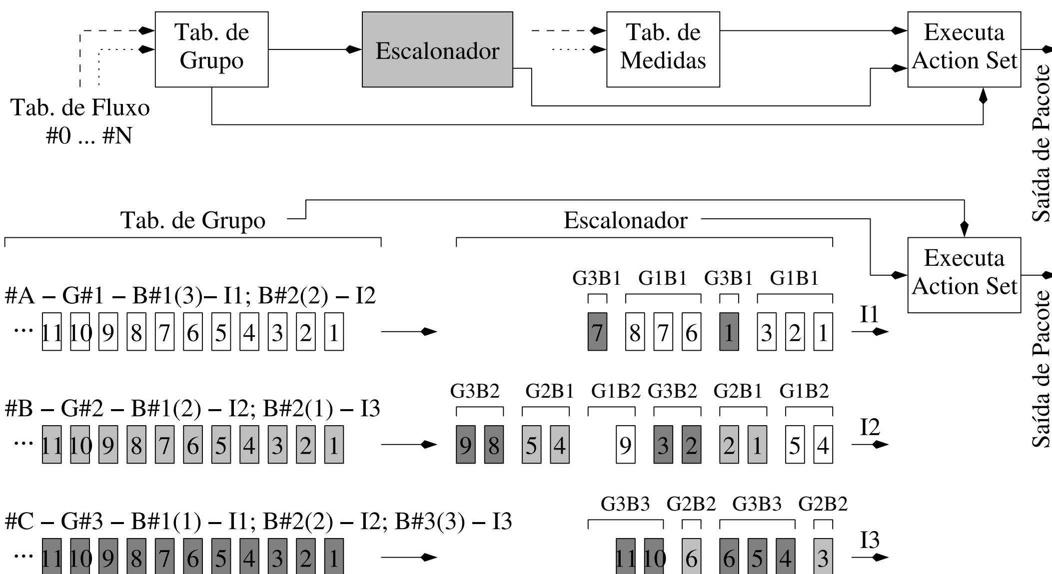


Figura 17 – Escalonador - Encaminhamento no Nível do *Bucket* e do Grupo.

A distribuição dos pacotes depende dos pesos dos *buckets*, o peso pode indicar um intervalo de tempo ou uma quantidade de pacotes. Por exemplo, um peso 3 pode indicar que todos os pacotes que chegarem dentro de um intervalo de tempo de 30 milisegundos irão passar por um mesmo *bucket*; como também, dizer que 3 pacotes irão passar pelo mesmo *bucket*. Embora o Algoritmo 3, proposto nesta seção, atue apenas no intervalo de tempo, seria relativamente fácil modificá-lo para fazer o escalonamento baseado na quantidade de pacotes.

O tratamento dispensado para todos os pacotes recebidos em todos os três grupos está ilustrado na Figura 17. No exemplo, o escalonador envia os três primeiros pacotes

do $G\#1$ para o *Bucket* $B\#1$ e os dois pacotes seguintes para $B\#2$; a seguir envia os dois primeiros pacotes do $G\#2$ para o *Bucket* $B\#1$ e o próximo pacote para o *Bucket* $B\#2$; e, o primeiro pacote do $G\#3$ para o *Bucket* $B\#1$ e os dois próximos para o *Bucket* $B\#2$. Este processo é mantido, até que todos os pacotes presentes na figura sejam escalonados.

É importante salientar que os pacotes recebidos pelo escalonador são fornecidos pelas filas do Open vSwitch, e à medida em que os pacotes vão sendo disponibilizado pelas filas, após o indexamento em uma entrada de grupo, eles são escalonados. Embora os pacotes sejam escalonados de acordo com os pesos dos *buckets*, nada impede que os pacotes sejam fornecidos ao escalonador numa ordem diferente da Figura 17, ou seja, os pacotes do Grupo #3 podem chegar antes do Grupo #1 no escalonador, por exemplo. Consequentemente, os pacotes do $G\#3$ serão escalonados primeiro. Também vale reforçar que os *buckets* de um grupo são completamente independentes de um outro, ou seja, o *Bucket* $B\#1$ do $G\#1$ não tem nenhuma relação com o *Bucket* $B\#1$ do $G\#2$.

4.3 Extensões propostas na Arquitetura SDN

As Figuras 18 e 19 detalham a Arquitetura SDN juntamente com as nossas extensões representadas pelos componentes acinzentados. A Figura 18 apresenta uma visão mais genérica da Arquitetura SDN. A Figura 19 apresenta enfoque no controlador Floodlight, ou seja, destaca as extensões feitas no Plano de Controle. Vale ressaltar que a Aplicação de multicaminhos, representada em ambas as figuras, ainda não está completamente pronta, e por isso, não foi descrita neste capítulo. Esta aplicação é responsável por fornecer uma interface amigável ao usuário para que este requisite ou compre o seu SLA. Atualmente os SLAs dos clientes são entradas estáticas no Banco de Dados do Floodlight.

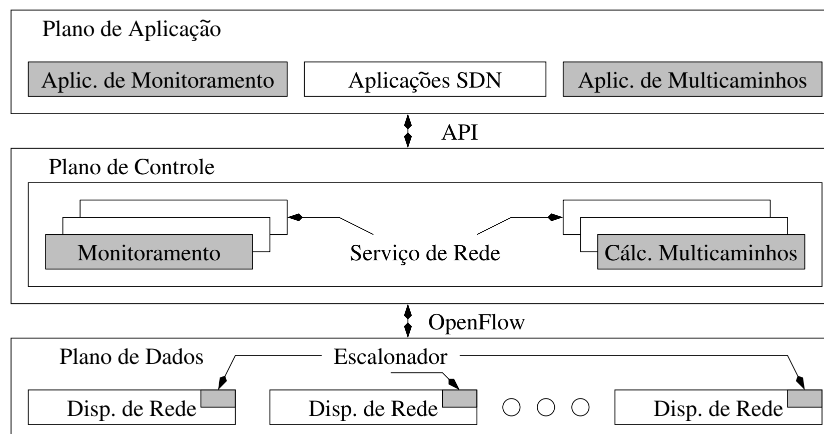


Figura 18 – Visão de alto nível das extensões à Arquitetura SDN.

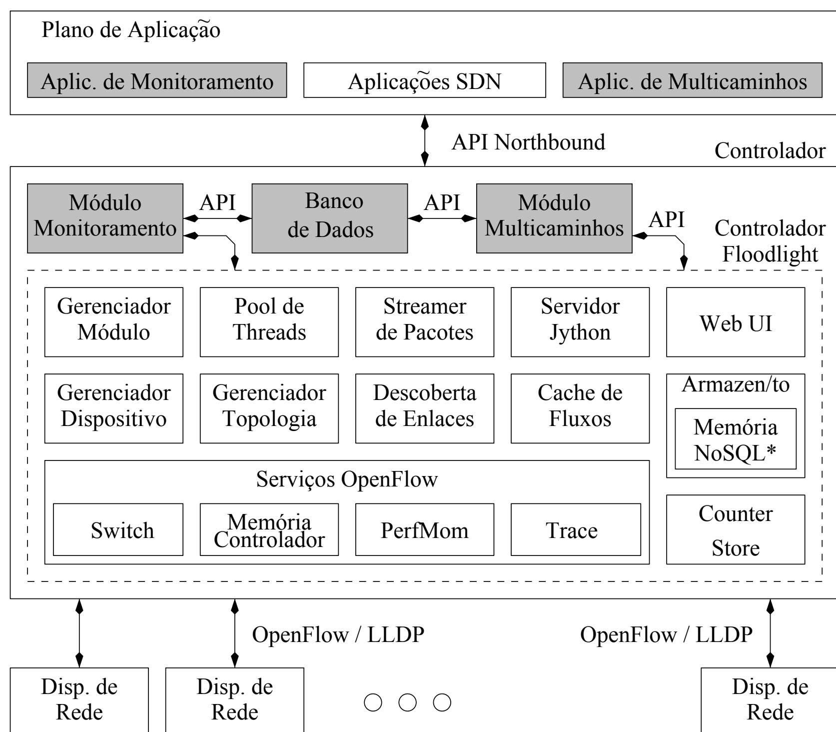


Figura 19 – Detalhes das extensões propostas no Plano de Controle.

Capítulo 5

Experimentos e Análise dos Resultados

Neste capítulo são detalhadas as ferramentas e os experimentos realizados para a validação dos dois módulos desenvolvidos. A Seção 5.1 detalha as ferramentas usadas para desenvolver o nosso trabalho. A Seção 5.2 avalia o SDNMon e a Seção 5.3 avalia o MP-Routing.

5.1 Ferramentas

As ferramentas usadas para desenvolver o nosso trabalho foram: o controlador OpenFlow Floodlight (FLOODLIGHT, 2015), um emulador de redes virtuais, chamado de Mininet (LANTZ; HELLER; MCKEOWN, 2010), comutadores virtuais com suporte ao OpenFlow: Open vSwitch(PFAFF et al., 2009) e o OpenFlow Software Switch 1.3 (FERNANDES; ROTHENBERG, 2014), uma ferramenta de monitoramento bastante conhecida na indústria de comutadores, o sFlow (PHAAL; PANCHEN; MCKEE, 2001), e também, o LoxiGen (LANE et al., 2015), responsável por gerar uma nova biblioteca OpenFlow para o Floodlight.

5.1.1 Floodlight

O controlador Floodlight foi usado pelos seguintes motivos: possui alta performance, é modular, multi-tarefas, *open source* e baseado na linguagem de programação Java. Como o Floodlight possui uma arquitetura modular, cada módulo é responsável por uma funcionalidade. Alguns módulos oferecem serviços aos outros módulos através de uma API Java, e também expõem serviços para aplicações através da API Restful (*API Northbound*). O protocolo usado para a comunicação entre os dispositivos de rede e o controlador é o OpenFlow (*API Southbound*). O Floodight suporta completamente as versões 1.0 e 1.3 do OpenFlow.

O Floodlight também possui uma documentação bastante detalhada e uma comunidade muito ativa na Internet, com isso qualquer dúvida sobre a implementação e funcio-

namento do controlador pode ser rapidamente solucionada.

5.1.2 Mininet

Como descrito na Seção 2.2.1, o Mininet é uma ferramenta usada para gerar topologias virtuais, permitindo a criação, de forma rápida e com baixo custo computacional, de redes compostas por dispositivos de características distintas, tais como atraso e largura de banda diferentes. O módulo SDNMon foi testado nas redes geradas pelo Mininet, pois não havia, no ano de 2014, servidores físicos disponíveis para a realização dos experimentos de monitoramento.

5.1.3 Comutadores

Para a realização do trabalho foram avaliados dois comutadores virtuais, o Open vSwitch e o ofsoftswitch13. A intenção era usar apenas o ofsoftswitch13 durante toda a pesquisa, pois é um dispositivo virtual de uma empresa brasileira, o CPqD, entretanto tivemos uma série de problemas com este dispositivo durante o primeiro ano de pesquisa, sendo que as principais complicações aconteceram durante os testes com o protocolo TCP, onde os dispositivos travavam, e consequentamente era necessário reiniciar os comutadores, além de conflitos de instalação do ofsoftswitch13 com algumas versões do Linux. Desta forma, decidimos realizar os experimentos apenas no Open vSwitch, permitindo a realização com sucesso e, em casos de dúvidas, a comunidade do Open vSwitch na Internet rapidamente atendia ao questionamento.

Vale ressaltar que também tentamos usar um comutador físico da HP com suporte ao OpenFlow 1.0, mas tivemos alguns problemas que comprometiam a realização do trabalho, tais como problemas na inserção de fluxos no dispositivo físico e os contadores de estatísticas não eram atualizados frequentemente. Problemas semelhantes ocorreram durante a emulação do protocolo OpenFlow 1.3 na distribuição Linux OpenWrt, no dispositivo TP-Link TL-WR1043ND.

5.1.4 sFlow

Uma das ferramentas usadas no nosso trabalho para o monitoramento do Plano de Dados foi o sFlow-RT. O motivo de seu uso foi o fato de ter uma comunidade *online* para tirar dúvidas, por ser uma ferramenta bastante conhecida na indústria e por ter uma API Restful bem estruturada para coleta das métricas do dispositivo da rede.

5.1.5 LoxiGen

O Loxigen é um *software* livre e de código aberto que permite a geração de bibliotecas OpenFlow para diversas linguagens, como C, Python e Java, sendo esta última usada

no Floodlight. A atual versão do LoxiGen suporta as seguintes versões do protocolo OpenFlow: 1.0, 1.1, 1.2 e 1.3.

5.2 Avaliações do SDNMon

5.2.1 Objetivos das Avaliações do SDNMon

Os experimentos realizados na Seção 5.2.3 verificam se o SDNMon é capaz de obter os contadores de estatísticas dos comutadores OpenFlow, usando o método *polling*, e, também, obter informações da rede através da amostragem de pacotes, utilizando o protocolo sFlow. Após a coleta dos dados, é verificado se o SDNMon consegue extrair métricas da rede a partir dos dados coletados, tais como, vazão e atraso, na granularidade de fluxo. Além disso, é avaliado qual a precisão de cada abordagem, *polling* e sFlow, e se a adição de novos contadores nos comutadores, como os de pacotes e *bytes* descartados, agregam valor a uma análise mais precisa do estado da rede.

5.2.2 Ambiente e Cenário de Testes do SDNMon

A Figura 20 apresenta o cenário utilizado para as análises do SDNMon. Foram consideradas a comunicação entre dois hospedeiros (H1 e H2) através de uma rede OpenFlow composta por cinco dispositivos de rede (SW1, SW2, SW3, SW4 e SW5). A infraestrutura de rede foi instanciada no Mininet versão 2.1.0, executando o Open vSwitch versão 2.0.2 nos dispositivos de rede e o controlador utilizado para a implementação do módulo de monitoramento é o Floodlight versão 0.90. Todos os experimentos foram efetuados em um Intel Core i3-2310M CPU 2.10GHz x 4 com 4 GB de memória RAM e sistema operacional Ubuntu 12.04. A sincronia dos relógios entre a máquina rodando o Floodlight e a máquina executando o Mininet foi feita através do protocolo *Network Time Protocol* (NTP)(MILLS et al., 2010).

5.2.3 Experimentos com o SDNMon

Conforme mencionado anteriormente, o módulo de monitoramento da SDNMon foi implementado contando com dois mecanismos de monitoramento. O primeiro é o mecanismo proposto de *polling*, que efetua a coleta de valores armazenados pelos contadores previstos na especificação OpenFlow, através de consultas explícitas usando as mensagens de *Statistic Request* e *Statistic Reply*. O segundo mecanismo utiliza o protocolo sFlow, baseado em agentes executados nos dispositivos de rede que enviam as amostragens a um coletor central. As amostragens consideram um pacote dentro de um conjunto N de pacotes transmitidos, na granularidade das portas. O módulo de monitoramento comunica-se com o protocolo na versão sFlow-RT via RESTful.

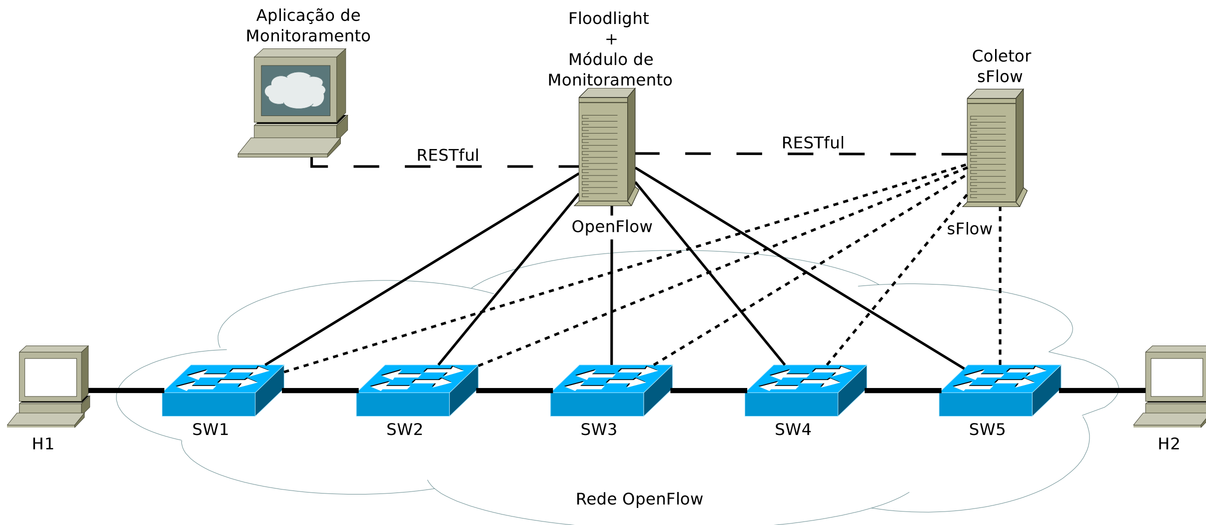


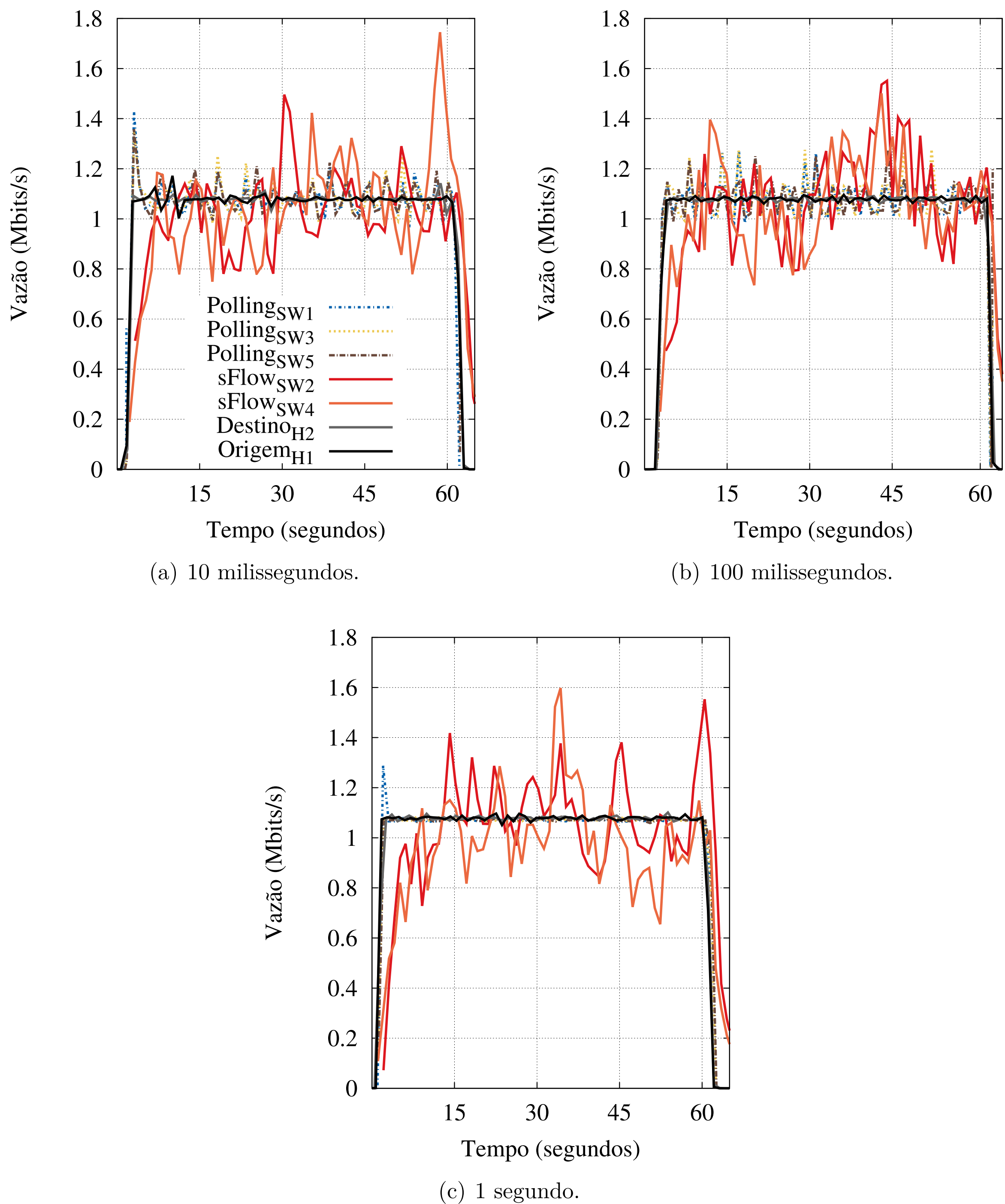
Figura 20 – Cenário utilizado nas avaliações do SDNMon.

5.2.3.1 Análise do Mecanismo de *Polling*

A Figura 21 apresenta uma análise referente às variações na taxa de coletas do módulo de monitoramento no caso do mecanismo de *polling*, objeto principal da análise efetuada nesta seção. O fluxo analisado foi gerado utilizando o *Iperf*, transmitindo UDP à taxa de 1 Mbps durante aproximadamente um minuto e utilizando pacotes com MTU de 1500 bytes. Para permitir uma visão completa sobre o mecanismo de monitoramento em todos os dispositivos de rede e permitir uma análise comparativa, são apresentados os dados de monitoramento do mecanismo de *polling* coletados nos dispositivos de rede SW1, SW3 e SW5 e os dados referentes ao sFlow coletados nos dispositivos de rede SW2 e SW4. Estes dados do mecanismo de monitoramento são comparados à transmissão real observada diretamente nas placas de rede dos hospedeiros H1 e H2, sendo H1 o hospedeiro de origem do tráfego analisado.

A metodologia foi variar a frequência entre leituras, considerando 10 milissegundos (Figura 21(a)), 100 milissegundos (Figura 21(b)), 1 segundo (Figura 21(c)) e armazenando apenas os valores observados quando havia mudança nos contadores. Em todas as amostragens da Figura 21, o mecanismo sFlow efetuou a amostragem considerando $N = 10$, ou seja, eram considerados para amostragem um pacote a cada dez transmitidos.

A Tabela 3 apresenta os dados consolidados referentes à média e o desvio das análises apresentadas na Figura 21. Os valores apresentados na Tabela 3 consideram as observações feitas em todos os dispositivos de rede (SW1 a SW5) para ambas as metodologias de *polling* e sFlow. Como pode ser observado, o mecanismo de *polling* apresenta maior precisão em todos os cenários avaliados, apresentando sempre média próxima aos valores de H1 e H2 com um baixo desvio padrão.



(a) 10 milissegundos.

(b) 100 milissegundos.

(c) 1 segundo.

Figura 21 – Variação da vazão em diferentes intervalos de coleta do mecanismo de *polling*.

5.2.3.2 Análise do Mecanismo de Amostragem sFlow

A Figura 22 apresenta as variações da vazão em função da taxa N de amostragem do sFlow. Por se tratar de um mecanismo de amostragem, o principal problema desta abordagem é ser utilizada em situações onde os pacotes amostrados possuem tamanho variado, uma característica comum nas redes de comutação por pacotes. Desta forma, a metodologia aplicada nestes testes foi a geração de 15 fluxos UDP entre H1 e H2 a uma

Tabela 3 – Dados consolidados referentes à análise do mecanismo de *polling*.

Análise	Média e Desvio Padrão (em Mbits/s)			
	$Origem_{H1}$	$Destino_{H2}$	$Polling_{SW1..5}$	$sFlow_{SW1..5}$
Fig. 21(a) (10ms)	1,073 +/- 0,044	1,075 +/- 0,029	1,086 +/- 0,069	1,055 +/- 0,191
Fig. 21(b) (100ms)	1,072 +/- 0,048	1,070 +/- 0,041	1,065 +/- 0,155	1,081 +/- 0,176
Fig. 21(c) (1s)	1,072 +/- 0,048	1,068 +/- 0,057	1,068 +/- 0,029	1,060 +/- 0,170

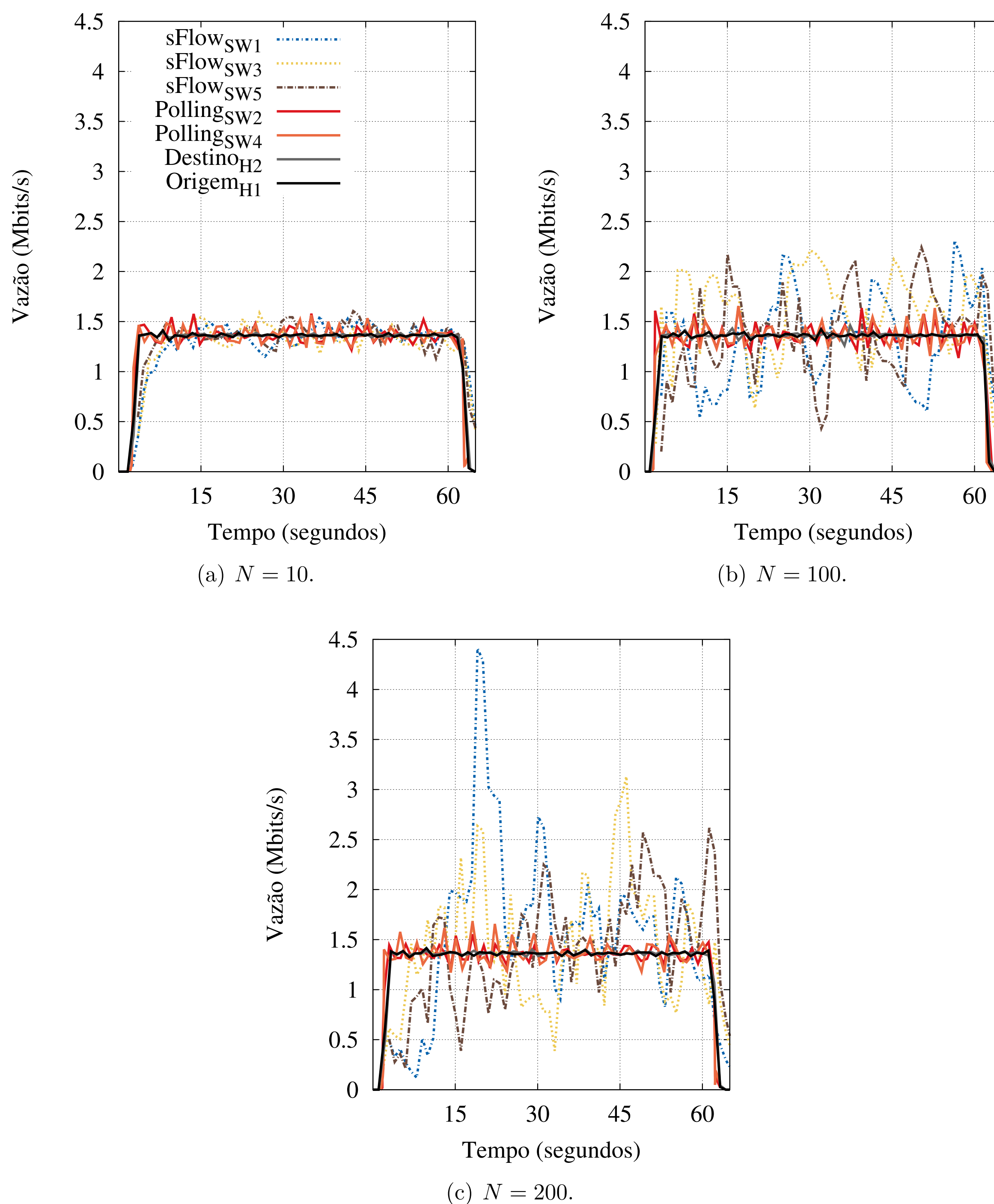
taxa de 100 kbps. Cada fluxo possui MTU distinta, variando entre 100 e 1500 bytes, em incrementos de 100 bytes.

Neste caso, como o sFlow é o objeto principal de avaliação, são apresentados os dados de monitoramento da amostragem sFlow referentes aos dispositivos de rede SW1, SW3 e SW5 e, para comparação, são apresentados os dados referentes ao mecanismo de *polling* coletados nos dispositivos de rede SW2 e SW4. Em todas as análises da Figura 22, o mecanismo de *polling* efetuou medições em intervalos de um segundo.

Como pode ser observado nos gráficos da Figura 22, mudanças na taxa de amostragem N comprometem a precisão do protocolo sFlow no caso de transmissões com pacotes de tamanho variado. Além disso, a taxa de amostragem, conforme mencionado, é associada à porta dos dispositivos de rede, não possuindo caráter adaptativo, ou seja, não reage às mudanças no volume de tráfego. Por exemplo, uma taxa de amostragem $N = 256$, ou seja, 1 em cada 256 pacotes é amostrado, é o valor recomendado para portas de 10 Mbps, $N = 512$ para 100 Mbps, $N = 1024$ para 1 Gbps, $N = 2048$ para 10 Gbps e $N = 8192$ para 100 Gbps (SFLOW, 2015). Embora o valor de amostragem da Figura 22(c) seja o recomendado para portas de 10 Mbps, a incidência de um volume inferior de tráfego também ocasiona perda de precisão.

Considerando as características dos fluxos UDP transmitidos, temos uma taxa de geração superior a quatrocentos pacotes por segundo, levando o agente sFlow a amostrar mais de quarenta pacotes por segundo no cenário com $N = 10$. Neste mesmo cenário, a coleta de dados dos contadores no mecanismo de *polling* está ocorrendo à frequência de uma coleta por segundo. Nestes experimentos, o mecanismo de *polling* está atuando na granularidade da porta de rede, da mesma forma como o sFlow atua. Pode-se ajustar o mecanismo de *polling* para atuar na granularidade de fluxos, exibindo dados individuais referentes a cada um dos quinze fluxos do experimento, uma característica não possível de ser implementada pelo sFlow.

A Tabela 4 apresenta a consolidação dos dados referentes ao sFlow. Especificamente sobre os resultados apresentados na Figura 22(a), é possível notar que a precisão do sFlow está bem próxima aos valores apresentados pelo mecanismo de *polling*, mas ainda assim indicando vantagem do mecanismo de *polling*. Ao verificar os resultados obtidos nas Figuras 22(b) e 22(c), é possível perceber que o sFlow perde bastante sua precisão à



(a) $N = 10$.

(b) $N = 100$.

(c) $N = 200$.

Figura 22 – Variação da vazão em função da taxa de amostragem N do sFlow.

medida que a taxa de amostragem N aumenta.

Tabela 4 – Dados consolidados referentes à análise do sFlow.

Análise	Média e Desvio Padrão (em Mbits/s)			
	$Origem_{H1}$	$Destino_{H2}$	$Polling_{SW1..5}$	$sFlow_{SW1..5}$
Fig. 22(a) ($N = 10$)	1,356 +/- 0,045	1,352 +/- 0,072	1,359 +/- 0,109	1,327 +/- 0,185
Fig. 22(b) ($N = 100$)	1,360 +/- 0,023	1,362 +/- 0,027	1,360 +/- 0,123	1,431 +/- 0,436
Fig. 22(c) ($N = 256$)	1,354 +/- 0,064	1,357 +/- 0,042	1,357 +/- 0,122	1,493 +/- 0,636

5.2.3.3 Avaliação da Vazão em Aplicações Legadas

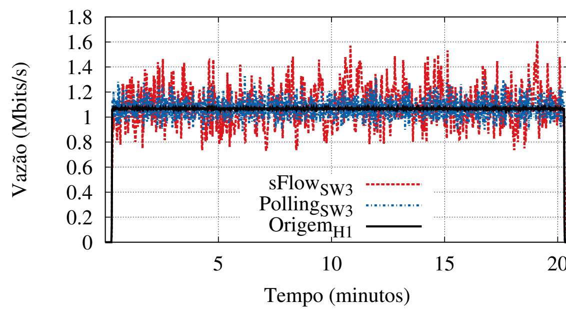
A Figura 23 apresenta uma análise com três aplicações legadas, incluindo transmissão UDP feita a partir do gerador de tráfego *Iperf* (IPERF, 2015) à taxa de 1 Mbps, transmissão TCP feita utilizando a aplicação *Secure Copy* (SCP) limitada à vazão aproximada de 1Mbps durante a cópia de um arquivo, e transmissão de vídeo utilizando o *Video Lan Client* (VLC) (VLC, 2015) sem qualquer controle referente à vazão. Em todas as três aplicações legadas avaliadas, o período de análise considera um intervalo aproximado de vinte minutos de transmissão. Nestes experimentos, os mecanismos de monitoramento foram configurados da seguinte forma: a) mecanismo de *polling* efetuando coletas em intervalos de cem milissegundos e b) sFlow com taxa de amostragem $N = 10$. Ambas as taxas de leitura elevadas, de tal forma a melhorar a precisão das leituras efetuadas.

Por se tratar de experimentos distintos, as escalas dos gráficos foram definidas individualmente, de tal forma a permitir uma melhor avaliação das séries de dados. Devido ao longo tempo de duração, as séries foram plotadas utilizando uma ordem que permitia uma melhor visualização das sobreposições, includindo apenas a série observada no hospedeiro H1 (origem do tráfego) e as séries do mecanismo de *polling* e sFlow observadas no dispositivo de rede SW3, localizado no centro do cenário avaliado.

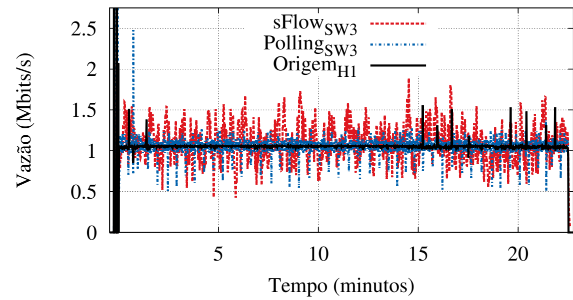
Nas Figuras 23(a) e 23(b) é possível observar a maior precisão do mecanismo de *polling* frente ao mecanismo de amostragem do sFlow. A Tabela 5 apresenta os dados consolidados referentes às transmissões UDP e TCP, evidenciando no caso do mecanismo de *polling* observações de taxa média similares aos valores de H1 e H2, além de apresentar um valor de desvio padrão inferior ao sFlow.

Tabela 5 – Dados consolidados referentes às transmissões UDP e TCP.

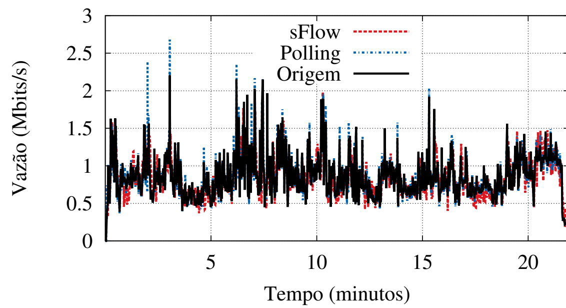
Análise	Média e Desvio Padrão (em Mbits/s)			
	$Origem_{H1}$	$Destino_{H2}$	$Polling_{SW1..5}$	$sFlow_{SW1..5}$
Figura 23(a)	1,064 +/- 0,038	1,064 +/- 0,041	1,067 +/- 0,090	1,076 +/- 0,168
Figura 23(b)	1,045 +/- 0,102	1,045 +/- 0,104	1,054 +/- 0,126	1,086 +/- 0,237



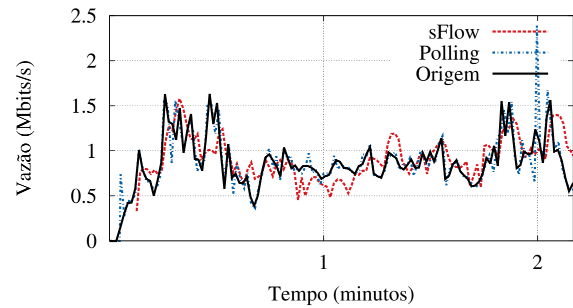
(a) UDP gerado com *Iperf*.



(b) TCP utilizando *SCP*.



(c) *Streaming* de vídeo utilizando VLC.



(d) Zoom na coleta do *streaming*.

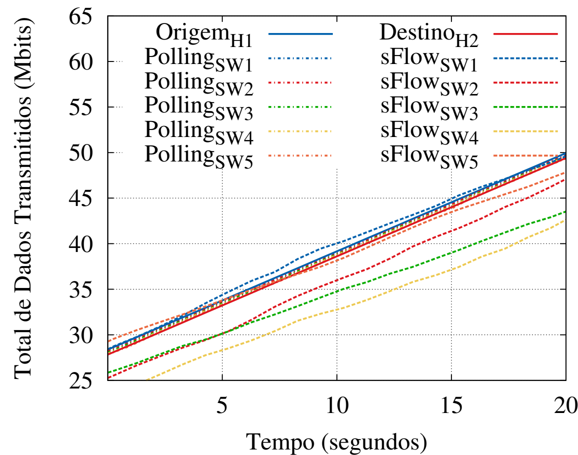
Figura 23 – Variação da vazão em diferentes aplicações legadas.

Especificamente para o *streaming* de vídeo apresentado na Figura 23(c), a natureza da aplicação possui variação frequente na vazão, não justificando as análises de média e desvio padrão. Desta forma, a Figura 23(d) apresenta um intervalo de aproximadamente três minutos, a contar do início do experimento, onde fica caracterizado comportamento similar ao observado nas análises com UDP e TCP, evidenciando a maior precisão no caso do mecanismo de *polling*.

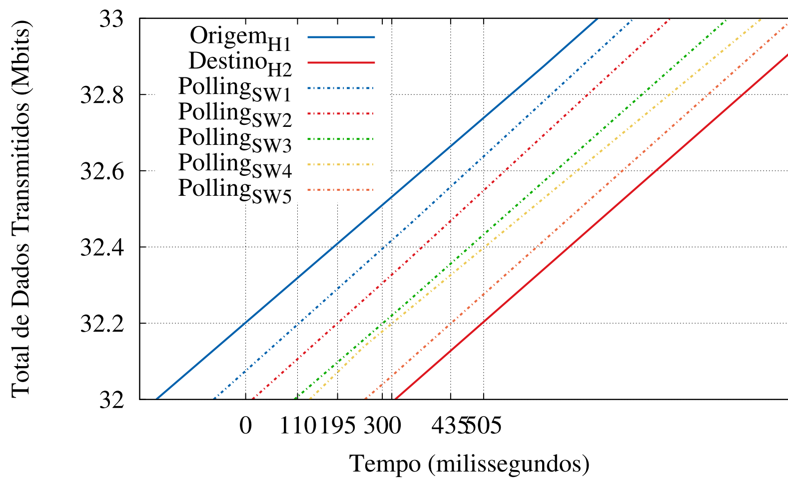
5.2.3.4 Análise do Atraso

A Figura 24 apresenta as séries temporais referentes ao total de dados transmitidos, coletadas durante uma transmissão UDP à taxa de 1 Mbps feita pelo gerador de tráfego *Iperf*, em um cenário sem a ocorrência de descarte de pacotes. As medições do mecanismo de *polling* ocorreram em intervalos de cem milissegundos e a taxa de amostragem utilizada para o sFlow foi $N = 10$.

Neste experimento, o Mininet foi configurado para aplicar um atraso fixo de cem milissegundos em cada um dos enlaces entre os dispositivos de rede OpenFlow, ou seja, espera-se que o tráfego experimente em torno de seiscentos milissegundos de atraso fim-a-fim. De maneira geral, o Mininet não conseguiu aplicar o tempo esperado em cada um dos enlaces, realizando um atraso fim-a-fim da ordem de quinhentos milissegundos. Esta configuração foi utilizada para demonstrar a metodologia de cálculo de atraso proposta, uma vez que sem esta configuração o Mininet apresenta atraso próximo a zero.



(a) Todas as séries temporais.



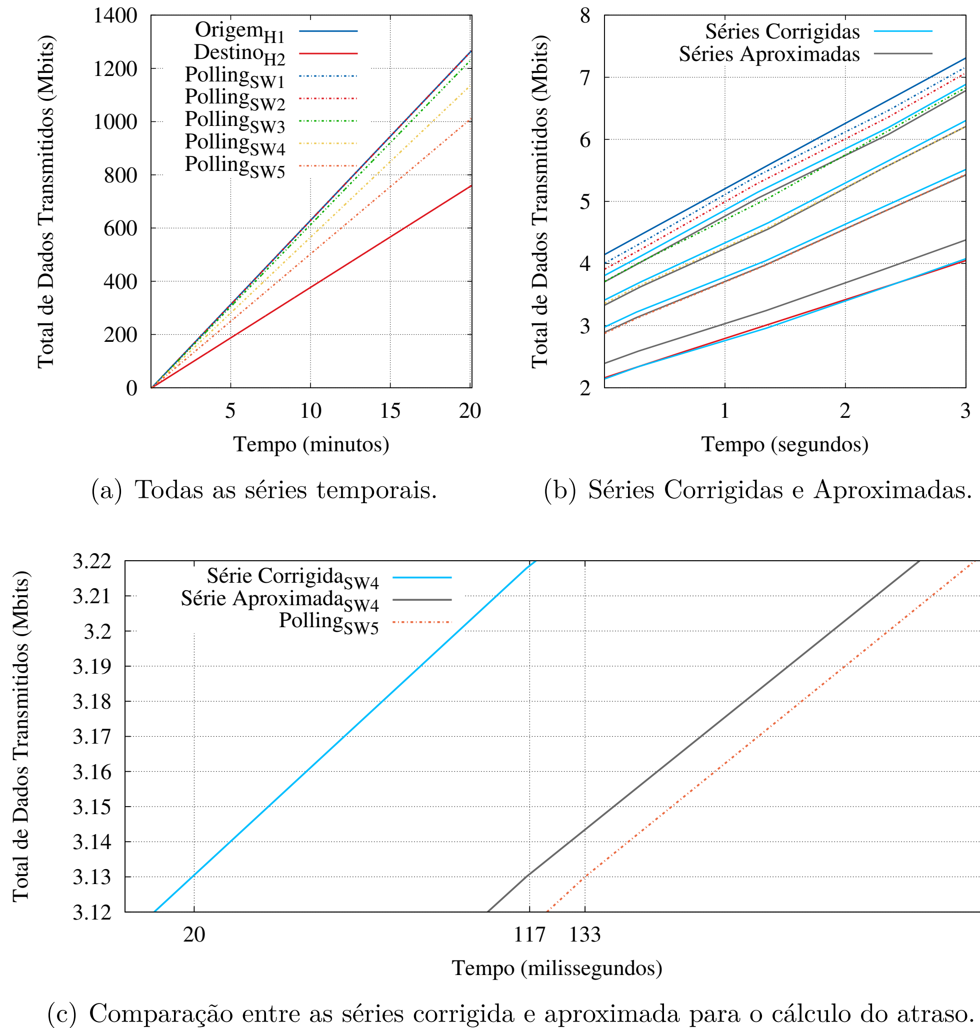
(b) Zoom nas séries do mecanismo de *polling*.

Figura 24 – Análise do atraso sem a ocorrência de descarte de pacotes.

A Figura 24(a) evidencia a inviabilidade do mecanismo de amostragem do sFlow para efetuar a análise de atraso apresentada na Seção 4.1.5, uma vez que o total de Mbits apresentado nas séries temporais do sFlow são estimativas obtidas através das amostras enviadas ao coletor pelos agentes. Porém, na Figura 24(b) é possível observar o sequenciamento das séries temporais coletadas pelo mecanismo de *polling*, partindo do hospedeiro de origem H1, atravessando sequencialmente os dispositivos de rede de SW1 a SW5 até atingir o hospedeiro de destino H2. Os valores destacados no eixo X permitem visualizar o atraso entre as séries temporais observadas na altura de 32.2 Mbits transmitidos, sendo 110 ms, 85 ms, 105 ms, 10 ms, 125 ms, 70 ms, respectivamente.

A Figura 25 apresenta as séries temporais coletadas em um cenário onde há descarte de pacotes. Basicamente, foram transmitidos pacotes com MTU de tamanho variado e foram definidos no Mininet enlaces com MTU decrescente entre SW1 (1500 bytes) e SW5 (1496 bytes) para forçar o descarte de pacotes. Foram enviados cinco fluxos UDP com

o gerador de tráfego *Iperf*, de tal forma que não haveria nenhum descarte no SW1, uma pequena ocorrência de descarte no SW2, gradualmente aumentando até permitir a vazão de um volume de tráfego no SW5. As medições do mecanismo de *polling* ocorreram em intervalos de cem milissegundos.



(a) Todas as séries temporais.

(b) Séries Corrigidas e Aproximadas.

(c) Comparação entre as séries corrigida e aproximada para o cálculo do atraso.

Figura 25 – Análise do atraso quando há a ocorrência de descarte de pacotes.

A Figura 25(a) apresenta todas as séries temporais durante a transmissão de aproximadamente vinte minutos, deixando evidente o distanciamento entre as séries ocasionado pelos descartes. A Figura 25(b) apresenta as séries corrigidas e aproximadas observadas em um período de três segundos, conforme discutido na Seção 4.1.5. A Figura 25(c) apresenta ambas as abordagens, detalhando a série temporal coletada do SW5 e as séries corrigida e aproximada referentes ao SW4. É possível observar a significativa perda de precisão do mecanismo aproximado, apresentando um atraso na ordem de 16 milissegundos, sendo que a série corrigida apresenta um atraso na ordem de 113 milissegundos.

A coleta referente aos dados descartados foi feita utilizando o contador *Transmit Drops* previsto no OpenFlow apenas para a granularidade em nível de portas. Este contador

mantém apenas o número de pacotes descartados. Os resultados da Figura 25 reforçam a sugestão feita na Seção 4.1.5, para que seja feita a inserção de contadores de total de bytes e total de pacotes descartados. Devido à necessidade de novos contadores no comutador OpenFlow, este trabalho também contempla a inserção dos contadores de pacotes e *bytes* descartados, na granularidade de fluxo, no comutador ofsoftswitch13. A extensão está detalhada no Apêndice C.

5.2.3.5 Análise das Mensagens de Estatística de Fluxo

Para coletar as estatísticas mantidas nos contadores dos comutadores é necessário enviar uma mensagem OpenFlow, chamada de *Statistics Request*, para o comutador, e o mesmo irá retornar uma *Statistics Reply* contendo informações, tais como o agregado de pacotes e *bytes* transmitidos, como também, o valor dos campos de *match*. As mensagens *Statistics Request* e *Statistics Reply* podem carregar estatísticas de diversas granularidades, como porta, fluxo individual, fluxo agregado, grupo, entre outros. A análise das trocas de mensagens desta seção, foi limitada à granularidade de fluxo individual, ou seja, do tipo *individual flow statistics*.

Nas versões mais recentes do protocolo OpenFlow, as mensagens *Statistics Request* e *Statistics Reply* são chamadas de: `OFPT_Multipart_Request` e `OFPT_Multipart_Reply`, respectivamente. Para a coleta dos dados desta seção, foi de fundamental importância o uso do Wireshark (WIRESHARK, 2015).

A comunicação entre o controlador e o dispositivo OpenFlow acontece através do protocolo TCP, um protocolo orientado a conexão, e, também, é possível estabelecer uma conexão segura, SSL/TLS, entre o controlador e o comutador, entretanto, o Floodlight não suporta o mesmo, e, por isso, o *overhead* de SSL/TLS não foi considerado na análise abaixo. A Figura 26 exemplifica um quadro (*frame*) Ethernet, encapsulando um datagrama IP, um segmento TCP, o Payload, e, por fim, o *Frame Check Sequence* (FCS).

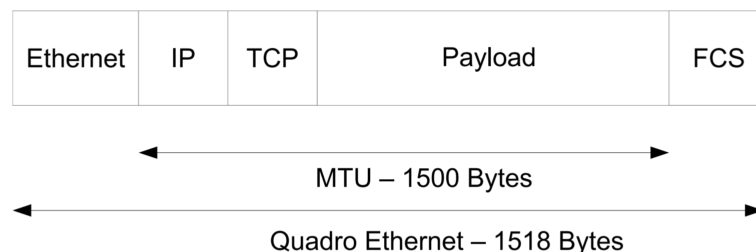


Figura 26 – Quadro Ethernet.

Como mostrado na Figura 26, o quadro Ethernet tem 1518 *bytes*, onde 14 *bytes* são do cabeçalho e 4 *bytes* são do FCS. O restante do pacote possui um MTU de 1500 *bytes*. A mensagem OpenFlow *Statistics Request* tem um tamanho relativamente pequeno, apenas 56 *bytes*, e cabe perfeitamente em um pacote IP, como mostrado na Figura 27. O cabeçalho IP tem 20 *bytes* e o TCP tem 32 *bytes*, ressaltando que embora o tamanho do cabeçalho

TCP tenha, por padrão, 20 *bytes*, neste caso está sendo usado algumas funcionalidades do TCP *Options*, dos quais 10 *bytes* são de *Timestamps* e 2 *bytes* são devido ao uso de 2 *No-Operation* (NOP), o que incrementa, no total, 12 *bytes* por pacote, o cabeçalho TCP.

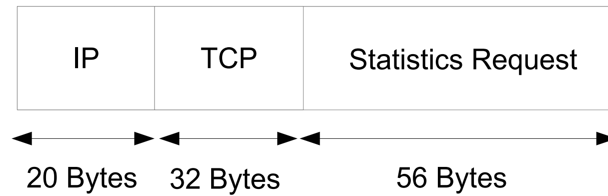


Figura 27 – Pacote IP contendo uma *Statistics Request*.

A Figura 28 detalha a mensagem *Statistics Reply* contida em pacote IP. Conforme detalhes na figura, apenas 1448 *bytes* são para uso de uma *Statistics Reply*, descontando os cabeçalhos IP e TCP. A Figura 29 mostra apenas a mensagem *Statistics Reply*, onde 16 *bytes* são usados para o cabeçalho e 1432 *bytes* transmitem as estatísticas.

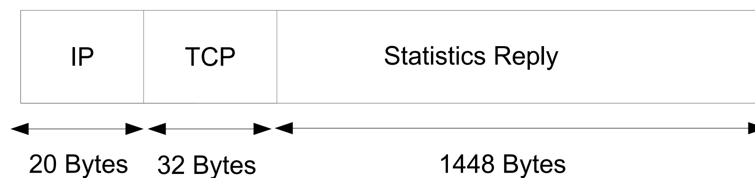


Figura 28 – Pacote IP contendo uma *Statistics Reply*.

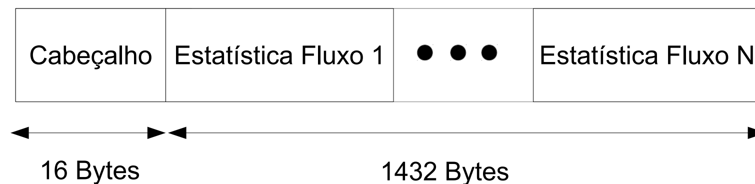


Figura 29 – *Statistics Reply* contendo Estatísticas de Fluxos.

Como detalhado na Figura 29, uma mensagem *Statistics Reply* carrega *N* Estatísticas de Fluxos, onde *N* depende do tamanho de cada Estatística de Fluxo. Como os fluxos são estabelecidos com características diferentes pelo controlador OpenFlow, por exemplo, um fluxo faz um *match* apenas no cabeçalho IP do pacote, e um outro faz um *match* no cabeçalho IP e também, no TCP, então, para este último caso, seria necessário armazenar mais informações nas entradas de fluxos da Tabela de Fluxo do comutador.

A Tabela 6 detalha o tamanho gasto para armazenar estatísticas de um fluxo, em uma Estatística de Fluxo, de acordo com os cabeçalhos usados para fazer o *match* nas entradas de fluxo. Como é possível perceber, quanto mais geral um fluxo, menor a quantidade de *bytes* gastos para armazená-lo em uma Estatística de Fluxo, como exemplo, o *Table-miss*, que significa, caso nenhum pacote coincida com alguma entrada de fluxo, envie para o controlador. Entretanto, quanto mais especifíco um fluxo, como no caso de fazer um *match*

nos cabeçalhos IP e TCP, maior é a quantidade de espaço necessária para armazenar o fluxo na Estatística de Fluxo.

Tabela 6 – Custos das Estatísticas de Fluxo.

Campos no *match*	***bytes***
Nenhum*	80
ARP	88
IP	104
IP + TCP	120
IP + UDP	120

***Table-miss*, usado apenas quando não existir nenhuma entrada de fluxo que coincide com os cabeçalhos do pacote. Na maioria dos dispositivos, a ação, neste caso, é enviar o pacote para o controlador.

Um mecanismo que deseja monitorar uma rede deve levar esse aspecto das trocas de mensagens em consideração para desenvolver uma maneira de adaptar-se ao estado da rede. Por exemplo, considere o cenário hipotético no qual um comutador OpenFlow esteja tratando 80 fluxos distintos. Desses 80 fluxos, 1 refere-se a um *Table-miss*, 5 referem-se a pacotes ARP, 10 referem-se a pacotes com match apenas pelo IP, 34 referem-se a pacotes com *matching* pelos cabeçalhos IP+TCP e 30 referem-se a pacotes com *matching* pelos cabeçalhos de IP+UDP.

Caso o controlador envie uma mensagem *Statistics Request* com todos os campos *wildcarded*, ou seja, requisitando todas as estatísticas de fluxos do comutador, então seria necessário enviar 9240 *bytes*, conforme calculado em (1). Desta forma, a primeira mensagem carregaria 1432 bytes de conteúdo, já que temos o cabeçalho do *Statistics Reply* no primeiro pacote, e ainda, seriam necessários mais 7 pacotes, conforme demonstrado em (2). Portanto, seriam necessários 8 pacotes TCP/IP para fazer o envio completo dos dados ao controlador.

$$(1 * 80) + (5 * 88) + (10 * 104) + (34 * 120) + (30 * 120) = 9240 \tag{1}$$

$$(9240 - 1432) \div 1448 = 6,38 \tag{2}$$

Conforme os dados obtidos nesta seção, é de fundamental importância que o controlador selecione os escopos que serão usados nas mensagens *Statistics Request*. Ao definir um escopo na mensagem *Statistics Request*, o comutador envia informações relativas aos fluxos delimitados pelo escopo, tal como os fluxos que tem como porta de saída uma determinada interface. Com isso, é necessário menos pacotes TCP/IP para carregar toda

a informação contida na *Statistics Reply*, reduzindo o tráfego no canal de controle e os dados processados no controlador.

O SDNMon consegue adaptar-se ao estado da rede ao definir um escopo de consulta para cada *thread* de monitoramento, com isso, cada *thread* consulta apenas os fluxos definidos *a priori* pelo escalonador, e à medida que é aumentado ou reduzido o tráfego na rede, o escalonador pode escalonar novas threads do *Poll* de *Threads* para adaptar-se à carga da rede. Esta adaptabilidade é detalhada na Seção 4.1.2.

Um outro fator que aumenta o uso do canal de controle entre os comutadores e o controlador é a quantidade de mensagens *Statistic Request* enviadas ao comutador por intervalo de tempo, pois quanto mais mensagens enviadas em um período de tempo para coleta de estatísticas, maior é a ocupação do canal de controle com mensagens *Statistic Reply*. No entanto, o envio constante de mensagens *Statistics Request* para coleta de estatísticas permite ao controlador obter uma informação mais atualizada do estado da rede.

Por fim, vale destacar o conceito de gerenciamento *out-of-band* e *in-band*, pois isto afeta diretamente o monitoramento de rede. O gerenciamento *out-of-band* permite que um canal de controle dedicado seja usado para a comunicação do controlador com o dispositivo OpenFlow, e embora seja uma abordagem mais cara, é possível ter um controle maior sobre o canal de controle pra que ele não comprometa a comunicação entre os dispositivos e o controlador. Já no gerenciamento *in-band* a comunicação entre o dispositivo OpenFlow e o controlador é compartilhado com o tráfego da rede e, se não forem tomadas medidas proativas, é possível que durante um pico de tráfego a comunicação entre os dispositivos e o controlador seja comprometida.

5.2.4 Resumo das Avaliações do SDNMon

Os resultados obtidos nesta seção demonstram que o SDNMon consegue obter estatísticas da rede através de duas abordagens de monitoramento, *polling* e *push*. No *polling*, o SDNMon consegue extrair duas métricas de QoS da rede, a vazão e o atraso, já no sFlow, é possível obter apenas a vazão. O SDNMon armazena as informações por ele coletadas e processadas no Banco de Dados e permite o acesso aos serviços do SDNMon através da API Java e API Restful. Devido à sua característica *multithreaded*, ele consegue ir alocando o monitoramento da rede a novas *threads*, para que cada *thread* tenha um escopo de monitoramento, permitindo que apenas alguns fluxos sejam coletados em um determinado instante e, consequentemente, controlando o uso do canal de controle OpenFlow.

A adição de novos contadores, em diferentes granularidades, pode ser muito útil para obter um estado mais preciso da rede, tal como os contadores de pacotes e *bytes* proposto e implementado neste trabalho no ofsoftswitch13. Por fim, algo que precisa ser mais bem avaliado em trabalhos futuros é o custo x benefício do envio de mensagens para coleta de

estatísticas. O comutador Open vSwitch atualiza seus contadores, em média, a cada 500 milisegundos e, portanto, pode-se desenvolver estratégias para coletar estas estatísticas de forma mais precisa, com menos gasto de troca de mensagens e com o menor uso do canal de controle entre o comutador e o controlador, através da definição de coringas nas mensagens *Statistics Request*.

5.3 Avaliações do MP-Routing

5.3.1 Objetivos das Avaliações do MP-Routing

Os experimentos realizados nesta seção têm como objetivo principal verificar se é possível garantir o SLA do usuário utilizando o módulo MP-Routing, apresentando os ganhos e custos da utilização do MP-Routing. A metodologia adotada efetua experimentos comparando o encaminhamento de pacotes por um único caminho e por múltiplos caminhos, na ausência e presença de tráfego concorrente.

5.3.2 Ambiente de Testes do MP-Routing

Os experimentos foram conduzidos utilizando-se um servidor de *rack* com 02 processadores *quad-core* de 2.4 GHz, 64 GB de memória RAM e HD de 300 GB. Todos os cenários são executados utilizando-se o *VMware ESXi* como *hypervisor*. Todos os *hosts*, comutadores e o controlador são *Virtual Machines* (VMs) assim como também são virtuais os enlaces que os ligam. Utilizou-se o *Open vSwtich* na versão 2.3.2 como comutador virtual e o Floodlight como o controlador. Cada VM utiliza 1 Processador Virtual, 8 GB de HD e 1 GB de RAM.

5.3.3 Cenários Avaliados no MP-Routing

Duas aplicações distintas, SCP e IPerf, foram consideradas na avaliação, ambas com serviço orientado por conexão. Isto é, o TCP é o responsável por prover transferência confiável e ordenada para ambas aplicações. Conforme mencionado anteriormente, o Módulo Multicaminhos é capaz de tratar todos os fluxos que utilizam o Plano de Dados, sendo a opção pelo SCP e IPerf devido aos dados estatísticos apresentados pelo TCP, essenciais para as análises apresentadas nesta seção.

O IPerf gera um tráfego de acordo com as especificações do IPerf, já no SCP os dados estão em arquivo, bastando apenas o encaminhamento pela rede. Através do IPerf é possível avaliar quanto se transfere de dados num dado intervalo fixo de tempo (90 segundos), a uma taxa de 100 Mbps, limitada pela interface do *host*. Já com o SCP, avalia-se o tempo de transferência de um arquivo de tamanho fixo (1GB), a uma taxa

de 100 Mbps, limitada pela interface do *host*. Desta forma, temos duas aplicações que permitem análises complementares sobre o uso de multicaminhos.

As investigações comparam experimentos em topologias multicaminhos *versus* caminho único, na presença e ausência de tráfego concorrente, isto é, tráfego não gerido pelo módulo de multicaminhos e que concorre com a aplicação avaliada (SCP ou IPerf). Os cenários avaliados são os seguintes:

(i) multicaminhos *versus* caminho único + tráfego concorrente, onde todos os caminhos apresentam 4 saltos;

(ii) multicaminhos *versus* caminho único, onde todos os caminhos apresentam 4 saltos;

(iii) multicaminhos *versus* caminho único + tráfego concorrente, onde todos os caminhos apresentam 5 saltos;

(iv) multicaminhos *versus* caminho único, onde todos os caminhos apresentam 5 saltos;

(v) multicaminhos *versus* caminho único + tráfego concorrente, onde cada caminho apresenta um número de saltos variado;

(vi) multicaminhos, onde cada caminho apresenta um número de saltos variado;

(vii) multicaminhos *versus* caminho único + tráfego concorrente, onde cada caminho apresenta um número de saltos variado e há descarte de pacotes em um dos caminhos; e,

(viii) multicaminhos, onde cada caminho apresenta um número de saltos variado e há descarte de pacotes em um dos caminhos;

Em todos os cenários, esboçados nas Figuras 30, 31 e 32, as interfaces de ingresso e egresso dos comutadores e dos *hosts* têm largura de banda de 100 Mbps. Os *hosts* indicados por círculos em linha contínua geram o tráfego para a aplicação avaliada (SCP e IPerf), conforme características definidas anteriormente para tempo, tamanho de arquivo e largura de banda. Os *hosts* indicados por círculos em linha pontilhada geram o tráfego concorrente, que insere carga adicional nos caminhos compondo a topologia. Em todos os cenários, o tráfego concorrente foi gerado utilizando o IPerf, para o protocolo UDP, sendo executado durante todo o tempo do experimento, e sempre a uma vazão fixa de 60 Mbps entre cada par de *hosts* origem-destino.

A Figura 30 apresenta a topologia utilizada na avaliação dos cenários (i) e (ii). Como pode ser observado nesta figura, todos os caminhos são compostos por 4 saltos (3 comutadores). No caminho único (topo), o tráfego da aplicação avaliada é gerado entre os *hosts* H_1 e H_2, conforme características mencionadas anteriormente, e o tráfego concorrente é gerado a 60 Mbps usando o IPerf entre os *hosts* H_3 e H_4. Para a topologia de multicaminhos (parte inferior da figura), o tráfego da aplicação avaliada também é gerado entre H_1

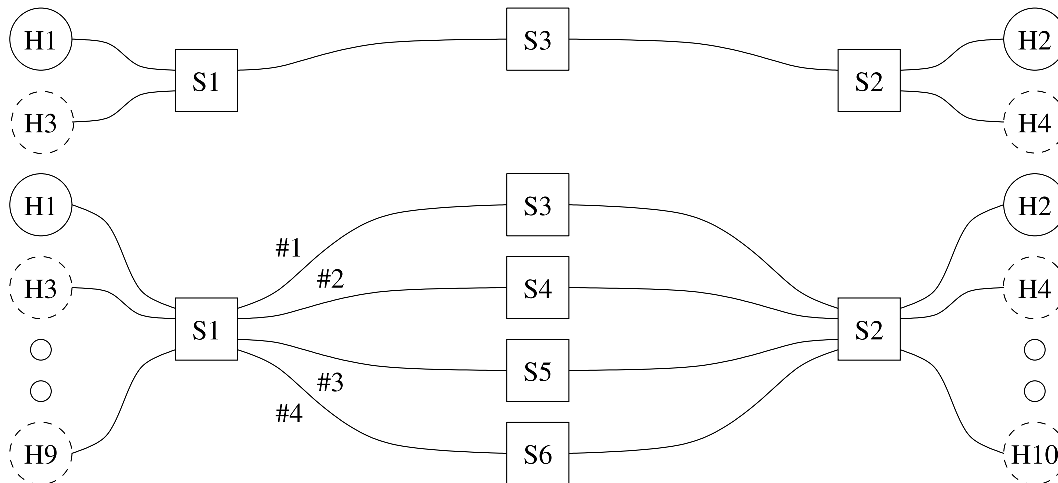


Figura 30 – Multicaminhos com 04 Saltos por Caminho vs Caminho Único.

e H_2, e cada um dos quatro caminhos recebe um tráfego concorrente de 60 Mbps, entre os pares $\{H_3, H_4\}$, $\{H_5, H_6\}$, $\{H_7, H_8\}$ e $\{H_9, H_{10}\}$.

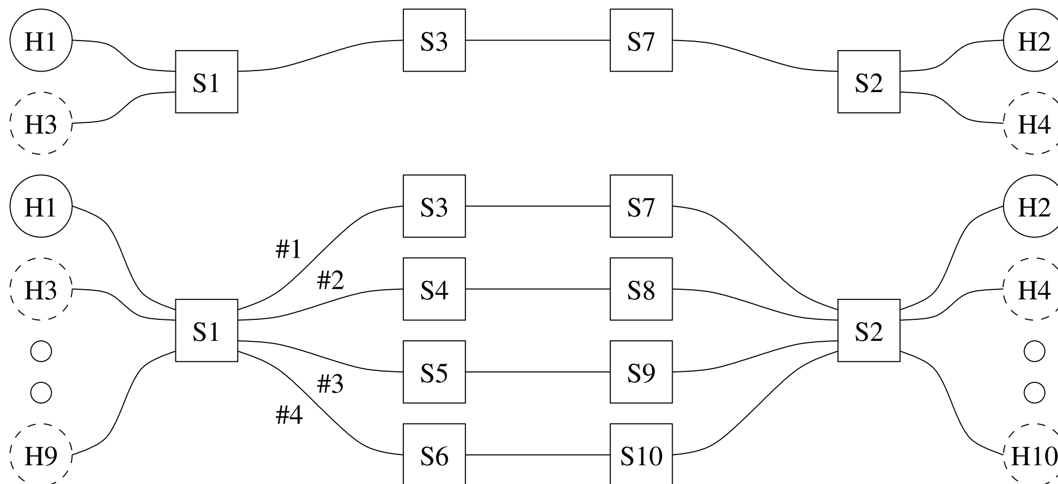


Figura 31 – Multicaminhos com 05 Saltos por Caminho vs Caminho Único.

A Figura 31 apresenta a topologia utilizada na avaliação dos cenários (iii) e (iv). Como pode ser observado nesta figura, todos os caminhos são compostos por 5 saltos (4 comutadores). No caminho único (topo), o tráfego da aplicação avaliada é gerado entre os *hosts* H_1 e H_2, conforme características mencionadas anteriormente, e o tráfego concorrente é gerado a 60 Mbps usando o IPerf entre os *hosts* H_3 e H_4. Para a topologia de multicaminhos (parte inferior da figura), o tráfego da aplicação avaliada também é gerado entre H_1 e H_2, e cada um dos quatro caminhos recebe um tráfego concorrente de 60 Mbps, entre os pares $\{H_3, H_4\}$, $\{H_5, H_6\}$, $\{H_7, H_8\}$ e $\{H_9, H_{10}\}$.

A Figura 32 contempla múltiplos caminhos entre fonte e destino, onde o número de saltos varia por caminho, ou seja, 03 saltos pelo caminho #1 até 06 saltos pelo caminho #4, incluindo caminhos menores e maiores (em número de saltos) quando comparados com as topologias simétricas (Figuras 30 e 31). A topologia da Figura 32 é usada nas

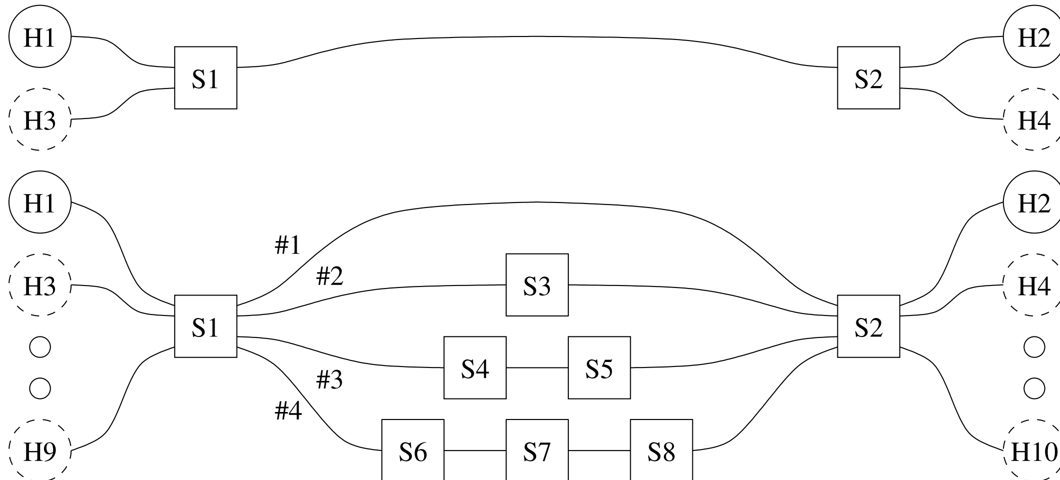


Figura 32 – Multicaminhos com Múltiplos Saltos por Caminho vs Caminho Único.

avaliações referentes aos cenários (v), (vi), (vii) e (viii). De maneira equivalente aos cenários anteriores, no caminho único (topo), o tráfego da aplicação avaliada é gerado entre os *hosts* H_1 e H_2, e o tráfego concorrente é gerado entre os *hosts* H_3 e H_4. Para a topologia de multicaminhos (parte inferior), o tráfego da aplicação avaliada também é gerado entre H_1 e H_2, e cada um dos quatro caminhos recebe um tráfego concorrente entre os pares $\{H_3,\ H_4\}$, $\{H_5,\ H_6\}$, $\{H_7,\ H_8\}$ e $\{H_9,\ H_{10}\}$. Diferentemente dos outros cenários, os cenários (vii) e (viii) possuem uma característica única, onde o caminho único e o caminho #1 da topologia de multicaminhos apresentam um descarte de 1% dos pacotes na interface de saída do comutador s1, ou seja, dentre 100 pacotes enviados pelo comutador S1, 1 deles é aleatoriamente descartado.

5.3.4 Experimentos com o MP-Routing

A Tabela 7 detalha o volume total de dados transferidos entre os *hosts* em MBytes, a vazão experimentada em Mbps, o tempo total para a transferência em segundos e dados sobre o total de pacotes e total de retransmissões do TCP. Esta tabela refere-se aos experimentos do cenário (i), e os valores apresentados, são a média dos valores coletados em 5 execuções de cada experimento. Nas avaliações considerando o uso de multicaminhos, o escalonador foi ajustado com uma taxa de 25% do tráfego encaminhado através de cada um dos quatro caminhos disponíveis na topologia da Figura 30.

Vale ressaltar que o algoritmo proposto na Subseção 4.3.2 geraria uma solução na qual, por exemplo, alocaria a vazão dos caminhos #1, #2, #3 como 40%, 40% e 20%, respectivamente, garantindo o SLA com o menor números de caminhos. Entretanto, foi feita uma alteração simples no algoritmo do MP-Routing para que todos os caminhos pudessem transmitir tráfego, permitindo uma comparação com o conjunto de pesos {25%, 25%, 25%, 25%}, do encaminhamento por múltiplos caminhos sem tráfego de fundo.

Observa-se que para o IPerf, o multicaminhos possibilita a tranferência de 1024 MB

Tabela 7 – IPerf/SCP - Multicaminhos com 04 Saltos/Caminho + Tráfego de Fundo.

Dados (MBytes)	Vazão (Mbits/seg)	Tempo (seg)	SEGs Enviados	SEGs* REXMIT	SEGs(%) REXMIT
IPerf - Caminho Único + Tráfego de Fundo					
415,40	38,70	90,00	300765	486;00;00	0,1923%
IPerf - Multicaminhos com 04 Saltos + Tráfego de Fundo					
1024,00	95,52	90,00	742122	351;00;02	0,0475%
SCP - Caminho Único + Tráfego de Fundo					
1024,00	36,80	222,00	742441	1216;00;00	0,1635%
SCP - Multicaminhos com 04 Saltos+ Tráfego de Fundo					
1024,00	92,36	93,00	742598	343;20;01	0,0490%

* REXMIT = SEGs Retransmitidos [Fast; Forward; Slow Start].

em 90 segundos, enquanto que no caminho único, apenas 415,40 MB são enviados no mesmo período. Há ganho explícito de vazão na rede, experimentando 38,7 Mbps pelo caminho único e atingindo 95,52 Mpbs ao distribuir o mesmo tráfego entre os múltiplos caminhos. Ainda referente à análise do IPerf, observa-se que o caminho único, por estar saturado entre a aplicação avaliada e o tráfego concorrente, leva a um total de 0,1923% de pacotes retransmitidos, e este volume de pacotes retransmitidos é reduzido para 0,0475% ao utilizarmos os multicaminhos, onde os caminhos não estão saturados.

Os resultados apresentados na avaliação do SCP na Tabela 7, onde o tamanho do arquivo a ser transferido é fixo em 1GB (1024MB), observa-se que o uso de um caminho único, saturado pela existência do tráfego concorrente, eleva o tempo total de transmissão para 222 segundos, ao passo que ao recorrermos ao uso de multicaminhos, este tempo total de transmissão é reduzido para 93 segundos. Esta variação no tempo total, também gera uma variação na taxa média de transmissão, saindo de 36,80 Mbps de taxa efetiva no caminho único, para 92,36 Mbps no multicaminhos. Os resultados referentes ao volume de retransmissões indicam 0,1635% para o caminho único e este valor cai para 0,0490% ao utilizarmos o multicaminhos.

Ainda para o cenário da Figura 30, as Tabelas 8 (a) e (b) detalham os dados coletados na ausência de tráfego concorrente. Estes experimentos representam o cenário (ii) e possuem o objetivo de avaliar qual o impacto do uso de multicaminhos, quando na verdade um único caminho seria capaz de atender o SLA solicitado. Desta forma, definimos quatro conjuntos de pesos distintos para o escalonador: (a) {85%, 5%, 5%, 5%}; (b) {65%, 20%, 10%, 5%}; (c) {45%, 30%, 15%, 10%}; e (d) {25%, 25%, 25%, 25%} . Os dados apresentados referem-se à média dos valores coletados de 05 execuções em caminho único e 05 execuções para cada um dos conjuntos de pesos para o caso multicaminhos, totalizando 25 execuções.

Como observado na Tabela 8 (a), o IPerf consegue gerar praticamente o mesmo volume de tráfego dentro da janela de 90 segundos, resultando em uma vazão praticamente estável, independente do fato de usarmos caminho único ou diferentes pesos para multicamihos. Entretanto, é possível observar que o uso de multicaminhos gera uma elevação no

Tabela 8 – IPerf/SCP - Caminho Único vs Multicaminhos com 04 Saltos/Caminho.

(a) IPerf - Caminho Único vs Multicaminhos com 04 Saltos/Caminho.

Dados (MBytes)	Vazão (Mbits/seg)	Tempo (seg)	SEGs Enviados	SEGs* REXMIT	SEGs(%) REXMIT
Caso #1 - Caminho Único com 04 Saltos: #1=100					
1024,00	95,70	90,00	743658	116;00;01	0,0158%
Caso #2 - Multicaminhos com 04 Saltos: #1=85; #2=05 #3=05; #4=05					
1024,00	95,70	90,00	742720	303;01;01	0,0410%
Caso #3 - Multicaminhos com 04 Saltos: #1=65; #2=20; #3=10; #4=05					
1024,00	95,70	90,00	742050	349;01;02	0,0473%
Caso #4 - Multicaminhos com 04 Saltos: #1=45; #2=30; #3=15; #4=10					
1023,00	95,70	90,00	742030	329;00;02	0,0447%
Caso #5 - Multicaminhos com 04 Saltos: #1=25; #2=25; #3=25; #4=25					
1024,00	95,70	90,00	742720	330; 02; 02	0,0450%

* REXMIT = SEGs Retransmitidos [Fast; Forward; Slow Start].

(b) SCP - Caminho Único vs Multicaminhos com 04 Saltos/Caminho.

Dados (MBytes)	Vazão (Mbits/seg)	Tempo (seg)	SEGs Enviados	SEGs* REXMIT	SEGs(%) REXMIT
Caso #1 - Caminho Único com 04 Saltos: #1=100					
1024,00	92,36	93,00	742580	150;00;01	0,0203%
Caso #2 - Multicaminhos com 04 Saltoss: #1=85; #2=05; #3=05; #4=05					
1024,00	92,36	93,00	742628	351;06;02	0,0482%
Caso #3 - Multicaminhos com 04 Saltos: #1=65; #2=20; #3=10; #4=05					
1024,00	92,36	93,00	742591	347;09;01	0,0408%
Caso #4 - Multicaminhos com 04 Saltos: #1=45; #2=30; #3=15; #4=10					
1024,00	92,36	93,00	742581	370;12;01	0,0515%
Caso #5- Multicaminhos com 04 Saltos: #1=25; #2=25; #3=25; #4=25					
1024,00	92,36	93,00	742611	369;12;01	0,0513%

* REXMIT = SEGs Retransmitidos [Fast; Forward; Slow Start].

número de retransmissões, variando entre 2,59 a 2,99 vezes o número de retransmissões identificados no caminho único.

O mesmo tipo de comportamento é observado para o SCP na Tabela 8 (b), onde o tempo total para a transmissão do mesmo arquivo de 1024MB é estável, frente ao uso de qualquer configuração. O número de retransmissões no multicaminhos varia de 2 a 2,51 vezes o número no caminho único, um valor ligeiramente melhor do que os apresentados nas avaliações para o IPerf (Tabela 8 (a)).

A variação observada nas retransmissões quando utilizados os multiplos caminhos, entre os diferentes pesos, é bastante sutil, sendo praticamente estável, e, além disso, não interferiu no tempo gasto e nem na vazão durante a transferência do arquivo. Porém, indicam que o uso de multicaminhos leva a mais retransmissões, quando comparado ao uso de caminho único.

Basicamente, ao segmentarmos o fluxo em quatro subfluxos, o TCP torna-se mais suscetível ao envio de mensagens de reconhecimento (ACKs) duplicados, pois os segmentos chegam fora de ordem ao destinatário. Como pode ser observado, a maior taxa de retransmissão refere-se a *Fast Retransmissions*, que ocorrem quando o contador de ACKs duplicados atinge o total de três ACKs com mesmo número de reconhecimento. Ou seja, os resultados indicam que na existência de um caminho único capaz de honrar o SLA, a solução multicaminhos não é a mais indicada. Desta forma, conforme descrito na Seção

4.2.2, o MP-Routing procura primeiramente atender ao SLA usando um caminho único, e na ausência deste caminho, determina o conjunto de múltiplos caminhos a ser usado.

Tabela 9 – IPerf/SCP - Multicaminhos com 05 Saltos/Caminho + Tráfego de Fundo.

Dados (MBytes)	Vazão (Mbps)	Tempo (seg)	SEGs Enviados	SEGs* REXMIT	SEGs(%) REXMIT
IPerf - Caminho Único + Tráfego de Fundo					
416,00	38,70	90,00	300929	382;00;00	0,1538%
IPerf - Multicaminhos com 05 Saltos + Tráfego de Fundo					
1022,00	95,22	90,00	739896	374;04;01	0,0512%
SCP - Caminho Único + Tráfego de Fundo					
1024,00	36,80	222,00	742458	855;00;00	0,1150%
SCP - Multicaminhos com 05 Saltos + Tráfego de Fundo					
1024,00	92,36	93,00	742647	409;24;01	0,0584%

* REXMIT = SEGs Retransmitidos [Fast; Forward; Slow Start].

Os resultados da Tabela 9 referem-se ao cenário (iii). Os resultados são bem semelhantes ao do cenário (i), onde o uso de múltiplos caminhos aumenta o ganho da vazão em ambas as abordagens, IPerf e SCP. No IPerf, é gerado 1022 MBytes de dados efetivos, contra 416 MBytes no caminho único, ou seja, um ganho de 2,45 vezes. Além disso, o número de segmentos retransmitidos no multicaminhos é 3 vezes menor, 0,0512%, contra 0,1538% do caminho único. No SCP, o tempo gasto para a transferência de um mesmo arquivo no multicaminhos é 2,38 vezes menor do que no caminho único, 93 segundos contra 222 segundos, e o número de segmentos retransmitidos também é menor no multicaminhos, 0,0584% contra 0,1150% no caminho único.

Entretanto, o aumento de 1 salto por caminho no cenário (iii), apresenta um ligeiro impacto na utilização de multicaminhos, elevando o número de retransmissões em ambos os casos, IPerf e SCP. No IPerf, na topologia de 4 saltos os resultados indicaram 351 retransmissões (Tabela 7) e na topologia de 5 saltos 374 retransmissões (Tabela 9). No SCP, 4 saltos resultaram em 343 retransmissões (Tabela 7) e 5 saltos em 409 retransmissões (Tabela 9). Confirmando que há impacto no uso de caminhos mais longos uma vez que a chegada desordenada de pacotes no destinatário é ampliada.

A Tabela 10 detalha os resultados do cenário (iv). Os resultados são semelhantes ao do cenário (ii). E, confome comportamento identificado ao aumentarmos o número de saltos da topologia, observa-se na Tabela 10 um pequeno aumento no número de retransmissões em todos os casos, quando comparado com os valores apresentados na Tabela 8. O caminho único, quando disponível e capaz de atender ao SLA, continua sendo a melhor opção em ambas as aplicações. No IPerf, o multicaminhos tem de 3,07 a 3,37 vezes mais retransmissões. No SCP, o multicaminhos tem de 2,69 a 2,82 vezes mais retransmissões.

Os resultados apresentados na Tabela 11 correspondem ao cenário (v). Os resultados são semelhantes aos cenários (i) e (iii), embora os caminhos tenham número de saltos distintos nos multicaminhos, variando de 3 a 6 saltos. Novamente, o uso de multicaminhos, na ausência de caminho único capaz de honrar o SLA, melhora consideravelmente a

Tabela 10 – IPerf/SCP - Caminho Único vs Multicaminhos com 05 Saltos/Caminho.

(a) IPerf - Caminho Único vs Multicaminhos com 05 Saltos/Caminho.

Dados (MBytes)	Vazão (Mbps)	Tempo (seg)	SEGs Enviados	SEGs* REXMIT	SEGs(%) REXMIT
Caso #1 - Caminho Único com 05 Saltos: #1=100					
1024,00	95,62	90,00	743024	122;00;01	0,0166%
Caso #2 - Multicaminhos com 05 Saltos: #1=85; #2=05; #3=05; #4=05					
1022,00	95,22	90,00	740005	377;06;01	0,0519%
Caso #3 - Multicaminhos com 05 Saltos: #1=65; #2=20; #3=10; #4=05					
1023,00	95,34	90,00	740945	376;02;01	0,0511%
Caso #4 - Multicaminhos com 05 Saltos: #1=45; #2=30; #3=15; #4=10					
1024,00	95,50	90,00	742394	386;01;01	0,0522%
Caso #5 - Multicaminhos com 05 Saltos: #1=25; #2=25; #3=25; #4=25					
1024,00	95,30	90,00	740710	410;04;01	0,0560%

* REXMIT = SEGs Retransmitidos [Fast; Forward; Slow Start].

(b) SCP - Caminho Único vs Multicaminhos com 05 Saltos/Caminho.

Dados (MBytes)	Vazão (Mbps)	Tempo (seg)	SEGs Enviados	SEGs* REXMIT	SEGs(%) REXMIT
Caso #1 - Caminho Único com 05 Saltos: #1=100					
1024,00	92,36	93,00	742579	150;00;01	0,0203%
Caso #2 - Multicaminhos com 05 Saltos: #1=85; #2=05; #3=05; #4=05					
1024,00	92,36	93,00	742619	393;11;01	0,0546%
Caso #3 - Multicaminhos com 05 Saltos: #1=65; #2=20; #3=10; #4=05					
1024,00	92,36	93,00	742581	397;18;01	0,0560%
Caso #4 - Multicaminhos com 05 Saltos: #1=45; #2=30; #3=15; #4=10					
1024,00	92,36	93,00	742616	404;19;01	0,0571%
Caso #5 - Multicaminhos com 05 Saltos: #1=25; #2=25; #3=25; #4=25					
1024,00	92,36	93,00	742584	406;19;01	0,0573%

* REXMIT = SEGs Retransmitidos [Fast; Forward; Slow Start].

quantidade de tráfego transmitido pelo IPerf, em comparação com o caminho único, 1023 MBytes contra 416 MBytes, ou seja, 2,45 vezes mais dados. O número de retransmissões também é menor no multicaminhos, 0,0527% contra 0.1427%. No SCP, o multicaminhos também é bem superior ao caminho único, tanto no tempo gasto para a transmissão de um mesmo arquivo, 90 segundos contra 222, como também, no número de retransmissões de segmentos, 0,0554% contra 0,1275%.

Tabela 11 – IPerf/SCP - Multicaminhos com 03 a 06 Saltos/Caminho + Tráfego de Fundo.

Dados (MBytes)	Vazão (Mbps)	Tempo (seg)	SEGs Enviados	SEGs* REXMIT	SEGs(%) REXMIT
IPerf - Caminho Único + Tráfego de Fundo					
416,00	38,68	90,00	300894	430;00;00	0,1427%
IPerf - Multicaminhos com 03 a 06 Saltos + Tráfego de Fundo					
1023,00	95,38	90,00	741256	387;02;01	0,0527%
SCP - Caminho Único + Tráfego de Fundo					
1024,00	38,59	222,00	742448	948;00;00	0,1275%
SCP - Multicaminhos com 03 a 06 Saltos + Tráfego de Fundo					
1024,00	92,36	93,00	742636	394;17;01	0,0554%

* REXMIT = SEGs Retransmitidos [Fast; Forward; Slow Start].

A Tabela 12 mostra os resultados do IPerf e do SCP referentes ao cenário (vi), equivalentes aos cenários (ii) e (iv), onde os diferentes pesos foram avaliados na ausência de tráfego concorrente. Novamente, a mudança dos pesos não interfere diretamente no número de retransmissões ou na quantidade de dados tranferidos.

Tabela 12 – IPerf/SCP - Multicaminhos com 03 a 06 Saltos/Caminho.

(a) IPerf - Multicaminhos com 03 a 06 Saltos por Caminho.

Dados (MBytes)	Vazão (Mbps)	Tempo (seg)	SEGs Enviados	SEGs* REXMIT	SEGs(%) REXMIT
Caso #1 - Multicaminhos com 03 a 06 Saltos: #1=85; #2=05; #3=05; #4=05					
1023,00	95,38	90,00	741270	285;04;01	0,0359%
Caso #2 - Multicaminhos com 03 a 06 Saltos: #1=65; #2=20; #3=10; #4=05					
1023,00	95,36	90,00	741163	323;02;01	0,0442%
Caso #3 - Multicaminhos com 03 a 06 Saltos: #1=45; #2=30; #3=15; #4=10					
1024,00	95,58	90,00	742774	321;02;01	0,0436%
Caso #4 - Multicaminhos com 03 a 06 Saltos: #1=25; #2=25; #3=25; #4=25					
1024,00	95,24	90,00	742448	341;02;01	0,0463%

* REXMIT = SEGs Retransmitidos [Fast; Forward; Slow Start].

(b) SCP - Multicaminhos com 03 a 06 Saltos por Caminho.

Dados (MBytes)	Vazão (Mbps)	Tempo (seg)	SEGs Enviados	SEGs* REXMIT	SEGs(%) REXMIT
Caso #1 - Multicaminhos com 03 a 06 Saltos: #1=85; #2=05; #3=05; #4=05					
1024,00	92,36	93,00	742602	288;22;01	0,0419%
Caso #2 - Multicaminhos com 03 a 06 Saltos: #1=65; #2=20; #3=10; #4=05					
1024,00	92,36	93,00	742629	352;23;01	0,0506%
Caso #3 - Multicaminhos com 03 a 06 Saltos: #1=45; #2=30; #3=15; #4=10					
1024,00	92,36	93,00	742584	319;07;01	0,0440%
Caso #4 - Multicaminhos com 03 a 06 Saltos: #1=25; #2=25; #3=25; #4=25					
1024,00	92,36	93,00	742599	318;05;01	0,0436%

* REXMIT = SEGs Retransmitidos [Fast; Forward; Slow Start].

Os cenários (vii) e (viii) utilizam a mesma topologia dos cenários (v) e (vi), apresentada na Figura 32. Conforme mencionado anteriormente, o diferencial é a inserção de descarte aleatório de 1% dos pacotes, na interface de saída do comutador s1, no caminho único, como também, no caminho #1 do multicaminhos. A Tabela 13 detalha os resultados obtidos no cenário (vii). Novamente, o uso de multicaminhos apresenta-se superior ao uso de um único caminho, e isto se torna ainda mais discrepante, em relação aos cenários de (i), (iii) e (v), devido ao fato de o tráfego passar por um caminho que descarta 1 de 100 pacotes. Uma característica interessante deste resultado refere-se à diluição dos impactos causados à transmissão fim-a-fim devido a problemas em partes da rede. Nas soluções tradicionais, onde um único caminho é usado, todos os pacotes compartilham as condições deste caminho, um conceito que pode ser definido como *fate sharing*. O uso de multicaminhos melhora significativamente o impacto deste *fate sharing*, como pode ser observado na Tabela 14, o IPerf no multicaminhos, transfere 2,66 vezes mais dados que o caminho único. O número de segmentos retransmitidos representa 0,2673% do total contra 1,0067% no caso do caminho único. No SCP, o multicaminhos transfere a mesma quantidade de dados em um tempo 2,56 vezes menor que o caminho único, e o número de segmentos retransmitidos no multicaminhos é de 0,2690%, contra 0,9954% do caminho único.

Os resultados obtidos no cenário (viii) são detalhados na Tabela 14, onde é possível observar o impacto na utilização dos diferentes arranjos de pesos no escalonador. Os resultados comprovam os impactos do *fate sharing*, como pode ser observado ao comparar

Tabela 13 – IPerf/SCP - Multicaminhos com 03 a 06 Saltos/Caminho + Tráfego de Fundo + Perdas de Pacote de (1%).

Dados (MBytes)	Vazão (Mbps)	Tempo (seg)	SEGs Enviados	SEGs* REXMIT	SEGs(%) REXMIT
IPerf - Caminho Único + Tráfego de Fundo + Perda de Pacotes.					
382,00	35,6	90,00	276995	2817;00;00	1,0067%
IPerf - Multicaminhos com 03 a 06 Saltos + Tráfego de Fundo + Perda de Pacotes.					
1019,00	94,93	90,00	737622	1976;00;01	0,2673%
SCP - Caminho Único + Tráfego de Fundo + Perda de Pacotes.					
1024,00	34,4	241,00	742380	7464;00;00	0,9954%
SCP - Multicaminhos com 03 a 06 Saltos + Tráfego de Fundo + Perda de Pacotes.					
1024,00	91,44	94,00	742603	1996;05;02	0,2690%

* REXMIT = SEGs Retransmitidos [Fast; Forward; Slow Start].

os valores para diferentes frações de tráfego através do Caminho #1.

No caso do IPerf, o Caso #1 tem 3,2 vezes mais retransmissões de segmentos e transmite apenas 89% do tráfego transmitido no Caso #4. À medida que o peso dado ao caminho #1 vai sendo diminuído, o número de retransmissões diminui, como também, aumenta a taxa efetiva de dados transmitidos.

Já no caso do SCP, o Caso #1 tem 3,1 vezes mais retransmissões de segmentos e o tempo gasto, para transmitir um mesmo arquivo de 1024 MBytes, é 1,11 vezes maior que o Caso #4. Similar ao IPerf, à medida que o peso dado ao caminho #1 vai sendo diminuído, o número de retransmissões diminui, como também, diminui o tempo gasto para transmitir o arquivo.

Tabela 14 – IPerf/SCP - Multicaminhos com 03 a 06 Saltos/Caminho + Perda de Pacotes de 1%.

(a) IPerf - Multicaminhos com 03 a 06 Saltos por Caminho.

Dados (MBytes)	Vazão (Mbps)	Tempo (seg)	SEGs Enviados	SEGs* REXMIT	SEGs(%) REXMIT
Caso #1 - Multicaminhos e Descarte de Pacotes: #1=85; #2=05; #3=05; #4=05					
910,00	84,80	90,00	658900	5412;09;67	0,8260%
Caso #2 - Multicaminhos e Descarte de Pacotes: #1=65; #2=20; #3=10; #4=05					
968,00	90,20	90,00	700470	4584;05;34	0,6557%
Caso #3 - Multicaminhos e Descarte de Pacotes: #1=45; #2=30; #3=15; #4=10					
1007,00	93,86	90,00	729430	3305;00;10	0,4524%
Caso #4 -Multicaminhos e Descarte de Pacotes: #1=25; #2=25; #3=25; #4=25					
1024,00	95,56	90,00	742292	1910;01;07	0,2577%

* REXMIT = SEGs Retransmitidos [Fast; Forward; Slow Start].

(b) SCP - Multicaminhos com 03 a 06 Saltos por Caminho.

Dados (MBytes)	Vazão (Mbps)	Tempo (seg)	SEGs Enviados	SEGs* REXMIT	SEGs(%) REXMIT
Caso #1 - Multicaminhos e Descarte de Pacotes: #1=85; #2=05; #3=05; #4=05					
1024,00	83,05	104,00	742545	6209;14;71	0,8405%
Caso #2 - Multicaminhos e Descarte de Pacotes: #1=65; #2=20; #3=10; #4=05					
1024,00	88,08	98,00	742656	4731;07;30	0,6379%
Caso #3 - Multicaminhos e Descarte de Pacotes: #1=45; #2=30; #3=15; #4=10					
1024,00	92,36	93,00	742729	3399;02;08	0,4569%
Caso #4 - Multicaminhos e Descarte de Pacotes: #1=25; #2=25; #3=25; #4=25					
1024,00	92,36	93,00	742704	1993;02;01	0,2680%

* REXMIT = SEGs Retransmitidos [Fast; Forward; Slow Start].

5.3.5 Resumo das avaliações do MP-Routing

O uso de multicaminhos, na existência de caminho único capaz de atender o SLA, não é recomendado, uma vez que acarreta a retransmissão de um número maior de segmentos, conforme resultados dos cenários (i), (iii) e (v).

Porém, na ausência de caminho único capaz de atender ao SLA, ou na existência de caminhos que apresentem mau comportamento, como descarte de pacotes, o uso de multicaminhos é ideal, reduzindo o número de retransmissões e ampliando a vazão, conforme resultados dos cenários (ii), (iv), (vi), (vii) e (viii).

Capítulo 6

Conclusão

Este capítulo apresenta as conclusões deste trabalho, alguns trabalhos futuros e as contribuições científicas que foram produzidas a partir deste trabalho.

6.1 Principais Contribuições

Neste trabalho foram propostas e avaliadas extensões à Arquitetura SDN para auxiliar no provisionamento de QoS através do monitoramento e uso de múltiplos caminhos. Foram propostos, como contribuição principal, os módulos de monitoramento, SDNMon, e de multicaminhos, MP-Routing, implementados como uma extensão do controlador Floodlight. A inserção destes dois módulos no Floodlight permite que reajam mais rapidamente às mudanças da rede, facilitando o oferecimento de serviços aos outros módulos ou para as aplicações.

Os testes realizados no SDNMon mostram que este é capaz de calcular a taxa de transmissão e atraso de um tráfego, inclusive na granularidade de fluxo. O SDNMon também consegue adaptar-se à carga da rede, pois através do uso do *Pool* de *threads* proposto, o componente escalonador do SDNMon define escopos para cada *thread* e permite o monitoramento mais preciso da rede. Foram feitas comparações entre o mecanismo proposto, *polling*, e o sFlow, indicando o melhor nível de precisão obtido pelo mecanismo de *polling*. Esta precisão referente à situação de todo o Plano de Dados, em diferentes granularidades é essencial para o funcionamento do MP-Routing.

Os testes efetuados no MP-Routing mostraram que este é capaz de atender o SLA do usuário através de múltiplos caminhos. Através das análises conduzidas em 8 diferentes cenários, fica evidente os ganhos ao segmentarmos um fluxo, em subfluxos, aglutinando fatias menores de largura de banda disponíveis em toda a rede, como forma de atender ao nível de QoS solicitado. Os experimentos deixam, inclusive, evidente os custos desta segmentação. O MP-Routing busca, primeiramente, honrar o SLA do usuário através de um único caminho e, caso não seja possível atender desta forma, o MP-Routing busca

um conjunto, múltiplos caminhos, para atender o SLA do usuário através da análise da largura de banda residual dos enlaces.

Esta dissertação inclui objetivos secundários, alcançados através do desenvolvimento de uma aplicação de fácil entendimento e que possibilita ao administrador de rede visualizar o estado da rede em tempo real. Também foram desenvolvidas duas extensões aos comutadores OpenFlow, a primeira adiciona dois contadores na granularidade de fluxo, o contador de pacotes e de bytes descartados, permitindo uma visão mais precisa do estado da rede; a segunda extensão, adiciona uma função que permite o escalonamento de pacotes de um mesmo tráfego por diferentes caminhos, ou seja, a divisão de um fluxo em subfluxos, conforme proposta do MP-Routing.

Como mostrado ao longo do trabalho, a Arquitetura SDN junto com sua materialização principal, o protocolo OpenFlow, agrega muito valor à oferta de QoS. Os grandes problemas da arquitetura das redes tradicionais, tais como o uso de protocolos proprietários e complexos, a troca de mensagens constantes entre os dispositivos de rede e a dificuldade de se atuar nas redes de maneira centralizada, tornam a oferta de QoS cada vez mais difícil.

6.2 Trabalhos Futuros

Trabalhos futuros relacionados ao SDNMon:

1. Investigar algoritmos alternativos para consultar métricas do Plano de Dados, priorizando soluções que efetuem o envio periódico de uma quantidade mínima de mensagens OpenFlow, permitindo uma visualização completa e com baixo peso de sinalização (*overhead*);

2. Implementar novos contadores aos dispositivos e ao protocolo OpenFlow, incluindo os descartes em todas as granularidades, possibilitando ao controlador uma visão mais precisa da rede;

3. Investigar a implementação de um mecanismo de *push* nos comutadores OpenFlow, permitindo o envio das estatísticas no momento que são atualizadas nos contadores do dispositivo. Nesta abordagem, não seria necessário o envio frequente de mensagens pelo controlador requisitando estatísticas ao comutador.

Trabalhos futuros relacionados ao MP-Routing:

1. Implementar extensões propostas na versão 1.5 do protocolo OpenFlow para os dispositivos de rede, tais como a possibilidade de remover ou adicionar um *bucket* sem haver a necessidade de remover o grupo, do qual o *bucket* está associado;

2. Investigar novas heurísticas para o algoritmo de definição de pesos para cada *bucket*, possibilitando a transmissão mais eficiente do tráfego;

3. Investigar novas heurísticas para a seleção de rotas, incluindo aspectos como quais caminhos, com quais características, devem ser escolhidos dentre aqueles que satisfaçam um dado nível de serviço. Estas investigações podem incluir o número de saltos, variações de atraso, número máximo de caminhos e, outros parâmetros. Todas estas questões apresentam elevado grau de complexidade.

6.3 Contribuições em Produção Bibliográfica

1. Pedro H. A. Rezende, Paulo R. S. L. Coelho, Luís F. Faina, Lásaro J. Camargos, Rafael Pasquini. "*Plataforma para Monitoramento de Métricas de Nível de Serviço em Redes Definidas por Software*". Publicado no XXXIII Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos - SBRC 2015, Vitória.

2. Pedro H. A. Rezende, Luís F. Faina, Lásaro J. Camargos, Rafael Pasquini. "*Roteamento Multicaminhos em Redes Definidas por Software*". Submetido em dezembro de 2015 ao XXXIV Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos - SBRC 2016, Salvador.

3. Pedro H. A. Rezende, Paulo R. S. L. Coelho, Luís F. Faina, Lásaro J. Camargos, Rafael Pasquini. "*Monitoring and Multipath Routing in OpenFlow*". Submetido em fevereiro de 2016 ao journal IEEE Transactions on Network and Service Management - TNSM.

Referências

ADISESHU, H.; PARULKAR, G.; VARGHESE, G. A reliable and scalable striping protocol. In: **Conference Proceedings on Applications, Technologies, Architectures, and Protocols for Computer Communications**. New York, NY, USA: ACM, 1996. (SIGCOMM '96), p. 131–141. ISBN 0-89791-790-1. Disponível em: <http://doi.acm.org/10.1145/248156.248169>.

ADRICHEM, N. L. M. van; DOERR, C.; KUIPERS, F. A. OpenNetMon: Network monitoring in OpenFlow Software-Defined Networks. In: **2014 IEEE Network Operations and Management Symposium, NOMS 2014, Krakow, Poland, May 5-9, 2014**. [s.n.], 2014. p. 1–8. Disponível em: <http://dx.doi.org/10.1109/NOMS.2014.6838228>.

AKYILDIZ, I. F. et al. A roadmap for traffic engineering in sdn-openflow networks. **Comput. Netw.**, Elsevier North-Holland, Inc., New York, NY, USA, v. 71, p. 1–30, out. 2014. ISSN 1389-1286. Disponível em: <http://dx.doi.org/10.1016/j.comnet.2014.06.002>.

ATM Forum. **Private Network-Network Interface Specification v.1.1**. 2002.

BERTSEKAS, D.; GAFNI, E.; GALLAGER, R. Second derivative algorithms for minimum delay distributed routing in networks. **Communications, IEEE Transactions on**, v. 32, n. 8, p. 911–919, Aug 1984. ISSN 0090-6778.

BLAKE, S. et al. **An Architecture for Differentiated Service**. United States, 1998.

BRADEN, R.; CLARK, D.; SHENKER, S. **Integrated Services in the Internet Architecture: An Overview**. United States, 1994.

BRADEN, R.; ZHANG, L. **Resource ReSerVation Protocol (RSVP) – Version 1 Message Processing Rules**. United States, 1997.

BRADLEY, T.; BROWN, C.; MALIS, A. **Multiprotocol Interconnect over Frame Relay**. [S.l.], 1992.

BULUT, A. et al. Optimization techniques for reactive network monitoring. **Knowledge and Data Engineering, IEEE Transactions on**, v. 21, n. 9, p. 1343–1357, Sept 2009. ISSN 1041-4347.

CALLADO, A. et al. A survey on internet traffic identification. **Communications Surveys Tutorials, IEEE**, v. 11, n. 3, p. 37–52, rd 2009. ISSN 1553-877X.

CARLUCCI, G.; CICCO, L. D.; MASCOLO, S. Http over udp: An experimental investigation of quic. In: **Proceedings of the 30th Annual ACM Symposium on Applied Computing**. New York, NY, USA: ACM, 2015. (SAC '15), p. 609–614. ISBN 978-1-4503-3196-8. Disponível em: <http://doi.acm.org/10.1145/2695664.2695706>.

CASADO, M. et al. Ethane: Taking Control of the Enterprise. In: **Proceedings of the 2007 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications**. New York, NY, USA: ACM, 2007. (SIGCOMM '07), p. 1–12. ISBN 978-1-59593-713-1. Disponível em: <http://doi.acm.org/10.1145/1282380.1282382>.

CASE, J. D. et al. **Simple Network Management Protocol (SNMP)**. [S.l.], 1990. <http://www.rfc-editor.org/rfc/rfc1157.txt>. Disponível em: <http://www.rfc-editor.org/rfc/rfc1157.txt>.

CHEBROLU, K.; RAO, R. Bandwidth aggregation for real-time applications in heterogeneous wireless networks. **Mobile Computing, IEEE Transactions on**, v. 5, n. 4, p. 388–403, April 2006. ISSN 1536-1233.

CHEN, S.; NAHRSTED, K. An overview of quality of service routing for next-generation high-speed networks: problems and solutions. **Network, IEEE**, v. 12, n. 6, p. 64–79, Nov 1998. ISSN 0890-8044.

CHOWDHURY, S. R. et al. PayLess: A Low Cost Network Monitoring Framework for Software Defined Networks. In: **IEEE/IFIP Network Operations and Management Symposium (NOMS)**. [S.l.: s.n.], 2014.

Cisco Systems. **Understanding Rapid Spanning Tree Protocol**. 2006. <http://www.cisco.com/c/en/us/support/docs/lan-switching/spanning-tree-protocol/24062-146.html/>. [Online; accessed 17-Novembro-2015].

_____. **How Does Unequal Cost Path Load Balancing (Variance) Work in IGRP and EIGRP?** 2009. <http://www.cisco.com/c/en/us/support/docs/ip/enhanced-interior-gateway-routing-protocol-eigrp/13677-19.html/>. [Online; accessed 10-Novembro-2015].

CLAISE, B. **Cisco Systems NetFlow Services Export Version 9**. [S.l.], 2004. Disponível em: <http://www.ietf.org/rfc/rfc3954.txt>.

DAS, S.; PARULKAR, G.; MCKEOWN, N. Why openflow/sdn can succeed where gmpls failed. In: **Optical Communications (ECOC), 2012 38th European Conference and Exhibition on**. [S.l.: s.n.], 2012. p. 1–3.

DIJKSTRA, E. W. A note on two problems in connexion with graphs. **NUMERISCHE MATHEMATIK**, v. 1, n. 1, p. 269–271, 1959.

EL-GENDY, M.; BOSE, A.; SHIN, K. Evolution of the internet qos and support for soft real-time applications. **Proceedings of the IEEE**, v. 91, n. 7, p. 1086–1104, July 2003. ISSN 0018-9219.

ENNS, R. et al. **Network Configuration Protocol (NETCONF)**. [S.l.], 2011. <http://www.rfc-editor.org/rfc/rfc6241.txt>. Disponível em: <http://www.rfc-editor.org/rfc/rfc6241.txt>.

FEAMSTER, N.; REXFORD, J.; ZEGURA, E. The Road to SDN: An Intellectual History of Programmable Networks. In: **ACM SIGCOMM Computer Communication Review archive Volume 44 Issue 2, April 2014, Pages 87-98**. [S.l.: s.n.], 2014.

FELSTAINE, E.; COHEN, R.; HADER, O. Crankback prediction in hierarchical atm networks. In: **INFOCOM '99. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE**. [S.l.: s.n.], 1999. v. 2, p. 671–679 vol.2. ISSN 0743-166X.

FERNANDES, E. L.; ROTHENBERG, C. E. OpenFlow 1.3 software switch. In: **Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos**. [s.n.], 2014. Disponível em: <https://github.com/CPqD/ofsoftswitch13>.

FLOODLIGHT, P. **<http://www.projectfloodlight.org>**. 2015. Disponível em: <http://www.projectfloodlight.org>.

FORD, A. et al. **Architectural Guidelines for Multipath TCP Development**. [S.l.], 2011. ISSN 2070-1721. Disponível em: <https://tools.ietf.org/rfc/rfc6182.txt>.

GALLAGER, R. A minimum delay routing algorithm using distributed computation. **Communications, IEEE Transactions on**, v. 25, n. 1, p. 73–85, Jan 1977. ISSN 0090-6778.

HA, S.; RHEE, I.; XU, L. Cubic: A new tcp-friendly high-speed tcp variant. **SIGOPS Oper. Syst. Rev.**, ACM, New York, NY, USA, v. 42, n. 5, p. 64–74, jul. 2008. ISSN 0163-5980. Disponível em: <http://doi.acm.org/10.1145/1400097.1400105>.

HOPPS, C. **Analysis of an Equal-Cost Multi-Path Algorithm**. [S.l.], 2000.

HP. **sFlow Agent**. 2007. <ftp://ftp.hp.com/pub/networking/software/17-C14-sflow.pdf/>. [Online; accessed 26-Novembro-2015].

HUANG, Y.; GUERIN, R. Does over-provisioning become more or less efficient as networks grow larger? In: **Network Protocols, 2005. ICNP 2005. 13th IEEE International Conference on**. [S.l.: s.n.], 2005. p. 11 pp.–235.

IPERF. **<http://iperf.fr>**. 2015. Disponível em: <http://iperf.fr>.

JANA; SWETT, I. **QUIC: A UDP-Based Secure and Reliable Transport for HTTP/2**. [S.l.], 2015. <http://www.ietf.org/internet-drafts/draft-tsvwg-quic-protocol-01.txt>. Disponível em: <http://www.ietf.org/internet-drafts/draft-tsvwg-quic-protocol-01.txt>.

JINYAO, Y. et al. Hiqos: An sdn-based multipath qos solution. **Communications, China**, v. 12, n. 5, p. 123–133, May 2015. ISSN 1673-5447.

KANDULA, S. et al. Dynamic load balancing without packet reordering. **SIGCOMM Comput. Commun. Rev.**, ACM, New York, NY, USA, v. 37, n. 2, p. 51–62, mar. 2007. ISSN 0146-4833. Disponível em: <http://doi.acm.org/10.1145/1232919.1232925>.

KESHAV, S. **An Engineering Approach to Computer Networking: ATM Networks, the Internet, and the Telephone Network**. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1997. ISBN 0-201-63442-2.

KREUTZ, D. et al. Software-defined networking: A comprehensive survey. **CoRR**, abs/1406.0440, 2014. Disponível em: <http://arxiv.org/abs/1406.0440>.

LANE, R. et al. **Loxigen**. [S.l.]: GitHub, 2015. <https://github.com/floodlight/loxigen>.

LANTZ, B.; HELLER, B.; MCKEOWN, N. A network in a laptop: rapid prototyping for software-defined networks. In: **Hotnets-IX Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks Article No. 19**. [S.l.: s.n.], 2010.

LEE, Y.; CHOI, Y. An adaptive flow-level load control scheme for multipath forwarding. In: LORENZ, P. (Ed.). **Networking — ICN 2001**. Springer Berlin Heidelberg, 2001, (Lecture Notes in Computer Science, v. 2093). p. 771–779. ISBN 978-3-540-42302-7. Disponível em: <http://dx.doi.org/10.1007/3-540-47728-4_76>.

LOU, W.; LIU, W.; FANG, Y. Spread: improving network security by multipath routing. In: **Military Communications Conference, 2003. MILCOM '03. 2003 IEEE**. [S.l.: s.n.], 2003. v. 2, p. 808–813 Vol.2.

MALKIN, G. S. **RIP Version 2**. [S.l.], 1998. <http://www.rfc-editor.org/rfc/rfc2453.txt>. Disponível em: <http://www.rfc-editor.org/rfc/rfc2453.txt>.

MANNIE, E. **Generalized Multi-Protocol Label Switching (GMPLS) Architecture**. United States, 2004.

MARTIN, R.; MENTH, M.; HEMMKEPPLER, M. Accuracy and dynamics of hash-based load balancing algorithms for multipath internet routing. In: **Broadband Communications, Networks and Systems, 2006. BROADNETS 2006. 3rd International Conference on**. [S.l.: s.n.], 2006. p. 1–10.

MCKEOWN, N. et al. Openflow: Enabling innovation in campus networks. **SIGCOMM Comput. Commun. Rev.**, ACM, New York, NY, USA, v. 38, n. 2, p. 69–74, mar. 2008. ISSN 0146-4833. Disponível em: <http://doi.acm.org/10.1145/1355734.1355746>.

MILLS, D. et al. **Network Time Protocol Version 4: Protocol and Algorithms Specification**. [S.l.], 2010. <http://www.rfc-editor.org/rfc/rfc5905.txt>. Disponível em: <http://www.rfc-editor.org/rfc/rfc5905.txt>.

MOY, J. **OSPF Version 2**. [S.l.], 1991. <http://www.rfc-editor.org/rfc/rfc1247.txt>. Disponível em: <http://www.rfc-editor.org/rfc/rfc1247.txt>.

NICHOLS, K. et al. **Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers**. United States, 1998.

ONF. OpenFlow Switch Specification 1.3.2. 2013. Disponível em: <https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.3.2.pdf>.

PAREKH, A.; GALLAGER, R. A generalized processor sharing approach to flow control in integrated services networks: the single-node case. **Networking, IEEE/ACM Transactions on**, v. 1, n. 3, p. 344–357, Jun 1993. ISSN 1063-6692.

PERLMAN, R. An algorithm for distributed computation of a spanningtree in an extended lan. **SIGCOMM Comput. Commun. Rev.**, ACM, New York, NY, USA, v. 15, n. 4, p. 44–53, set. 1985. ISSN 0146-4833. Disponível em: <http://doi.acm.org/10.1145/318951.319004>.

PFAFF, B. et al. Extending networking into the virtualization layer. In: **Proceedings of the Workshop on Hot Topics in Networks (HotNets '09), New York, NY, USA, October 2009**. [S.l.: s.n.], 2009.

PHAAL, P.; PANCHEN, S.; MCKEE, N. **InMon Corporation's sFlow: A Method for Monitoring Traffic in Switched and Routed Networks**. IETF, 2001. RFC 3176 (Informational). (Request for Comments, 3176). Disponível em: <http://www.ietf.org/rfc/rfc3176.txt>.

PHEMIUS, K.; BOUET, M. Monitoring latency with OpenFlow. In: **Proceedings of the 9th International Conference on Network and Service Management, CNSM 2013, Zurich, Switzerland, October 14-18, 2013**. [s.n.], 2013. p. 122–125. Disponível em: <http://dx.doi.org/10.1109/CNSM.2013.6727820>.

PRABHAVAT, S. et al. On load distribution over multipath networks. **Communications Surveys Tutorials, IEEE**, v. 14, n. 3, p. 662–680, Third 2012. ISSN 1553-877X.

REID, D.; KATCHABAW, M. J. **Internet QoS: Past, Present, and Future**. [S.l.: s.n.], 2004.

REZENDE, P. et al. A platform for monitoring service-level metrics in software defined networks. In: **Computer Networks and Distributed Systems (SBRC), 2015 XXXIII Brazilian Symposium on**. [S.l.: s.n.], 2015. p. 19–30.

______. Roteamento multicaminhos em redes definidas por software. In: **Submetido ao Computer Networks and Distributed Systems (SBRC), 2016 XXXIV Brazilian Symposium on**. [S.l.: s.n.], 2016.

ROSEN, E.; VISWANATHAN, A.; CALLON, R. **Multiprotocol Label Switching Architecture**. United States, 2001.

ROTSOS, C. et al. OFLOPS: an open framework for openflow switch evaluation. In: **PAM'12 Proceedings of the 13th international conference on Passive and Active Measurement, Pages 85-95**. [S.l.: s.n.], 2012.

SANDRI, M. et al. On the benefits of using multipath tcp and openflow in shared bottlenecks. In: **Advanced Information Networking and Applications (AINA), 2015 IEEE 29th International Conference on**. [S.l.: s.n.], 2015. p. 9–16. ISSN 1550-445X.

SFLOW. **<http://sflow.org>**. 2015. Disponível em: <http://sflow.org>.

SHARAFAT, A. R. et al. Mpls-te and mpls vpns with openflow. **SIGCOMM Comput. Commun. Rev.**, ACM, New York, NY, USA, v. 41, n. 4, p. 452–453, ago. 2011. ISSN 0146-4833. Disponível em: <http://doi.acm.org/10.1145/2043164.2018516>.

SHI, W.; MACGREGOR, M.; GBURZYNSKI, P. Load balancing for parallel forwarding. **Networking, IEEE/ACM Transactions on**, v. 13, n. 4, p. 790–801, Aug 2005. ISSN 1063-6692.

SLICE, D. et al. **Enhanced Interior Gateway Routing Protocol**. [S.l.], 2013. <http://www.ietf.org/internet-drafts/draft-savage-eigrp-00.txt>. Disponível em: <http://www.ietf.org/internet-drafts/draft-savage-eigrp-00.txt>.

SONG, J. et al. Adaptive load distribution over multipath in nepls networks. In: **Communications, 2003. ICC '03. IEEE International Conference on**. [S.l.: s.n.], 2003. v. 1, p. 233–237 vol.1.

SUH, J. et al. OpenSample: A Low-Latency, Sampling-Based Measurement Platform for Commodity SDN. In: **IEEE 34th International Conference on Distributed Computing Systems, ICDCS 2014, Madrid, Spain, June 30 - July 3, 2014**. [s.n.], 2014. p. 228–237. Disponível em: <http://dx.doi.org/10.1109/ICDCS.2014.31>.

TENNENHOUSE, D. et al. A survey of active network research. In: . [S.l.: s.n.], 1997. v. 35, n. 1, p. 80–86. ISSN 0163-6804.

THALER, D.; HOPPS, C. **Multipath Issues in Unicast and Multicast Next-Hop Selection**. [S.l.], 2000. <http://www.rfc-editor.org/rfc/rfc2991.txt>. Disponível em: <http://www.rfc-editor.org/rfc/rfc2991.txt>.

TSAI, J.; MOORS, T. **A review of multipath routing protocols: From wireless ad hoc to mesh networks**. [S.l.]: ACoRN Early Career Researcher Workshop on Wireless Multihop Networking, 2006.

VALI, D. et al. A survey of internet qos signaling. **Communications Surveys Tutorials, IEEE**, v. 6, n. 4, p. 32–43, Fourth 2004. ISSN 1553-877X.

VILLAMIZAR, C. **Ospf optimized multipath (ospf-omp)**. [S.l.], 1999. Disponível em: <https://tools.ietf.org/html/draft-ietf-ospf-omp-02>.

VLC. **<http://www.videolan.org/vlc/>**. 2015. Disponível em: <http://www.videolan.org/vlc/>.

WANG, J. et al. Fsdm: Floodless service discovery model based on software-defined network. In: **Communications Workshops (ICC), 2013 IEEE International Conference on**. [S.l.: s.n.], 2013. p. 230–234.

WIRESHARK. **<http://www.wireshark.org/>**. 2015. Disponível em: <http://www.wireshark.org/>.

XI, K.; LIU, Y.; CHAO, H. Enabling flow-based routing control in data center networks using probe and ecmp. In: **Computer Communications Workshops (INFOCOM WKSHPS), 2011 IEEE Conference on**. [S.l.: s.n.], 2011. p. 608–613.

ZAUMEN, W.; VUTUKURY, S.; GARCIA-LUNA-ACEVES, J. Load-balanced anycast routing in computer networks. In: **Computers and Communications, 2000. Proceedings. ISCC 2000. Fifth IEEE Symposium on**. [S.l.: s.n.], 2000. p. 566–574.

ZININ, A. **Cisco IP Routing: Packet Forwarding and Intra-domain Routing Protocols**. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002. ISBN 0-201-60473-6.

Apêndices

APÊNDICE A

Diagramas de Classes

Este apêndice detalha os diagramas de classes dos dois módulos desenvolvidos neste trabalho. A Figura 33 detalha a especificação do módulo de monitoramento, SDNMon, e a Figura 34 detalha a especificação do módulo de multi-caminhos, MP-Routing.

O SDNMon é composto por 7 classes. A classe MonitoriaGerenciador é responsável por gerenciar informações, como os *PacketIns* recebidos; cadastrar e oferecer serviços através das APIs Java e RESTful; e instanciar o Escalonador e o Processador. A classe Escalonador gerencia as *threads* usadas no monitoramento, tanto no sFlow como no *Polling*. Ao verificar que as *threads* não estão sendo eficientes no monitoramento, o Escalonador pode definir novos escopos ou alocar novas threads do *Poll* de *Threads* para ajudar no monitoramento. O Escopo e o EscopoBuilder são classes usadas pelo Escalonador para definir novos escopos que serão usados nas *threads* de monitoramento. As classes *Polling* e sFlow definem a API usada para comunicar com os dispositivos de rede, ou no coletor, no caso do sFlow. As *threads*, instanciadas nestas classes, mas definidas pelo Escalonador, usam os recursos oferecidos pelas classes *Polling* e sFlow para consultar, processar e armazenar os dados coletados. Por fim, o Processador é responsável por processar e formatar os dados de acordo com regras pré-estabelecidas com os clientes, e, posteriormente, armazená-los no Banco de Dados.

O MP-Routing é composto por 3 classes. A classe Gerenciador é responsável por gerenciar as informações; cadastrar e oferecer serviços através das APIs Java e RESTful; gerenciar mecanismos internos do MP-Routing, como a fila de *Packet-Ins*, repassando, à medida do possível, novos pedidos de caminhos, métricas da rede e o SLA do usuário ao Seletor; e atualiza o Banco de Dados. A classe Seletor recebe o pedido de novos caminhos para um determinado tráfego do Gerenciador, calcula caminhos que garantem o SLA do usuário e repassa os caminhos calculados ao Construtor, ao mesmo tempo que manda um *feedback* ao Gerenciador. A classe Construtor é a mais simples, ao receber os caminhos do Seletor, ele apenas cria, formata e envia as mensagens OpenFlow aos dispositivos de rede.

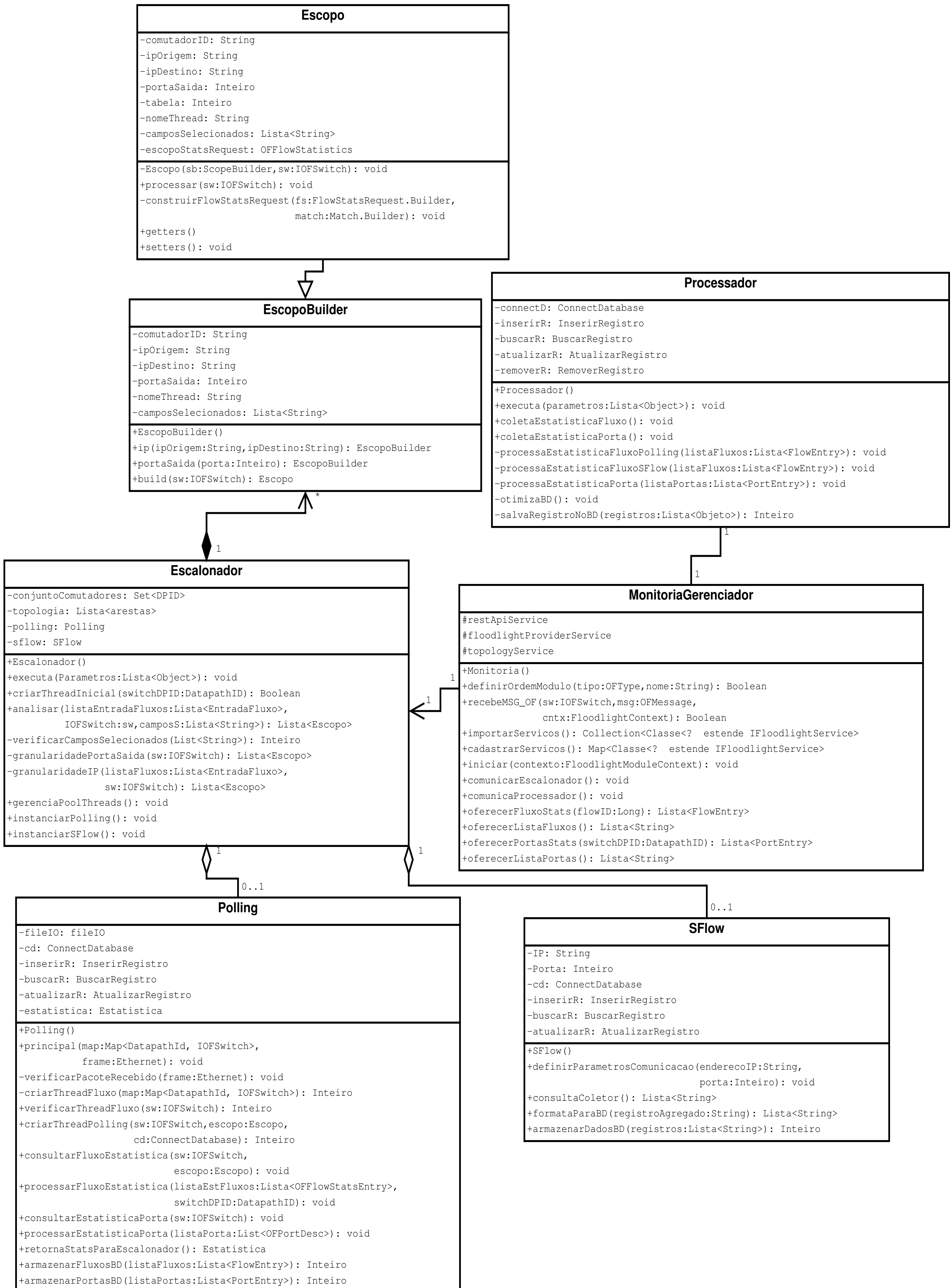


Figura 33 – Diagrama de Classe do SDNMon.

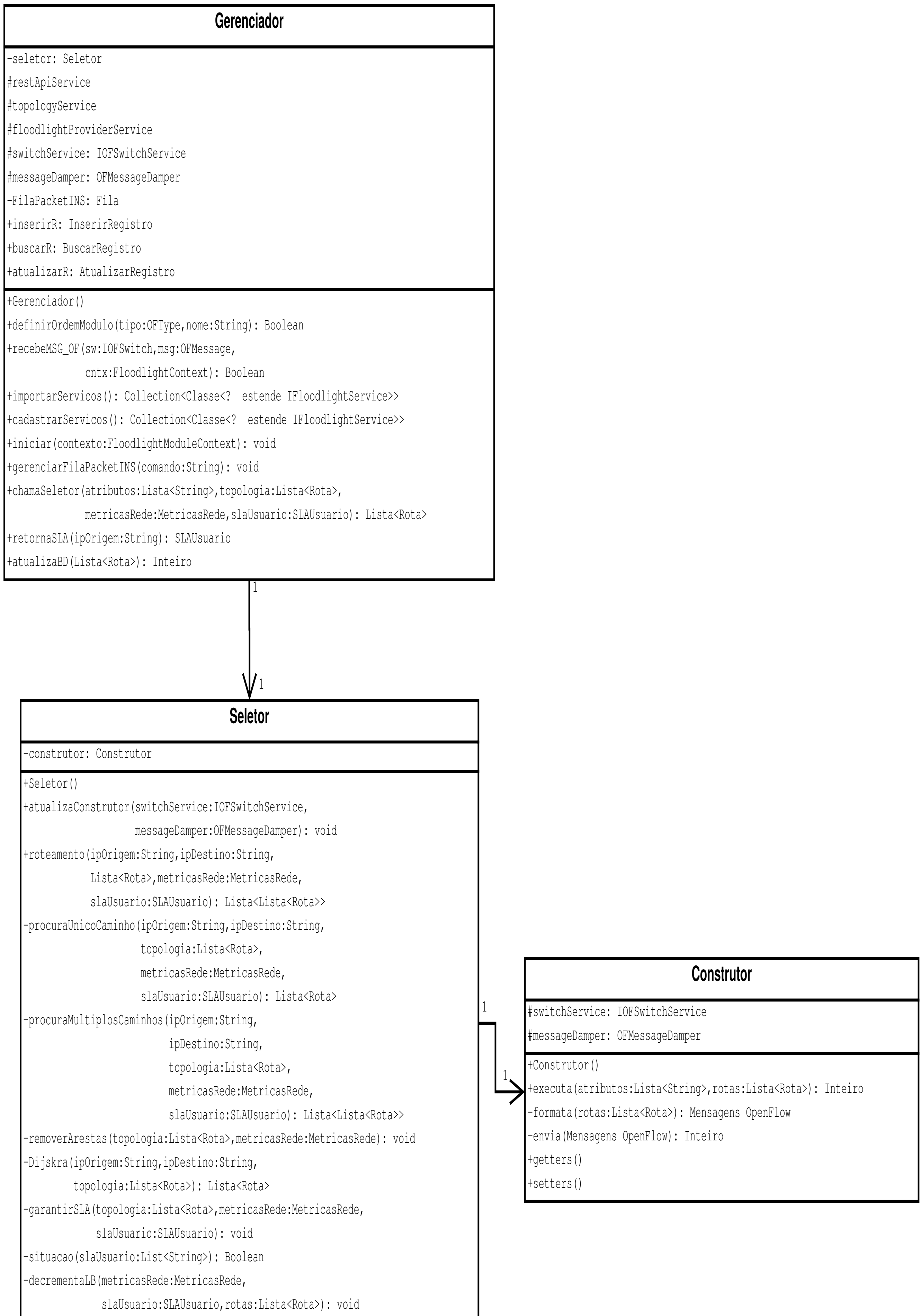


Figura 34 – Diagrama de Classe do MP-Routing.

APÊNDICE B

Pedido e Resposta de Estatísticas

Este apêndice mostra, na Figura 35, um exemplo de mensagem OpenFlow *Statistics Request*, do tipo *Individual Flow Statistics* com todos os campos *wildcarded*, ou seja, requisita todas as estatísticas individuais de fluxos do comutador. A mensagem OpenFlow *Statistics Reply* é apresentada na Figura 36. Está mensagem é enviada pelo comutador ao controlador, em resposta ao *Statistics Request*. Como mostrado na Figura 36, esta *Statistics Reply* contém apenas uma Estatística de Fluxo (*Flow Stats*); os campos setados nas regras são: Porta de Entrada, ID da VLAN, Tipo do Quadro, Protocolo de Transporte, Endereços IP de Origem e Destino, Portas de Origem e Destino da Aplicação; é possível observar alguns contadores, como os contadores de pacotes (*Packet Count*) e de *bytes* (*Byte Count*) e, também, qual a instrução deste fluxo, neste caso, enviar os pacotes para a porta de egresso (*Output Port*) 1. A interceptação das mensagens foi feita através do Wireshark.

```
OpenFlow Protocol
    Header
        Version: 0x01
        Type: Stats Request (CSM) (16)
        Length: 56
        Transaction ID: 52746
    Stats Request
        Type: Individual flow statistics (0x0001)
        Flags: 0x0000
        Flow Stats Request
            Match
                Match Types
                    .... .... .... .... .... .... .... ...1 =  Input port: Wildcard (1)
                    .... .... .... .... .... .... .... ..1. =  VLAN ID: Wildcard (1)
                    .... .... .... .... .... .... .... .1.. =  Ethernet Src Addr: Wildcard (1)
                    .... .... .... .... .... .... .... 1... =  Ethernet Dst Addr: Wildcard (1)
                    .... .... .... .... .... .... ...1 .... =  Ethernet Type: Wildcard (1)
                    .... .... .... .... .... .... ..1. .... =  IP Protocol: Wildcard (1)
                    .... .... .... .... .... .... .1.. .... =  TCP/UDP Src Port: Wildcard (1)
                    .... .... .... .... .... .... 1... .... =  TCP/UDP Dst Port: Wildcard (1)
                    .... .... .... .... ..11 1111 .... .... =  IP Src Addr Mask: /0 (63)
                    .... .... .... 1111 11.. .... .... .... =  IP Dst Addr Mask: /0 (63)
                    .... .... ...1 .... .... .... .... .... =  VLAN priority: Wildcard (1)
                    .... .... ..1. .... .... .... .... .... =  IPv4 DSCP: Wildcard (1)
            Table ID: All Tables
            Out Port: None  (not associated with a physical port)
```

Figura 35 – Exemplo de uma mensagem *Statistics Request*.

```
OpenFlow Protocol
    Header
        Version: 0x01
        Type: Stats Reply (CSM) (17)
        Length: 396
        Transaction ID: 52746
    Stats Reply
        Type: Individual flow statistics (0x0001)
        Flags: 0
        Flow Stats Reply
            Table ID: 0
            Match
                Match Types
                    .... .... .... .... .... .... .... ...0 =   Input port: Exact (0)
                    .... .... .... .... .... .... .... ..0. =   VLAN ID: Exact (0)
                    .... .... .... .... .... .... .... .1.. =   Ethernet Src Addr: Wildcard (1)
                    .... .... .... .... .... .... .... 1... =   Ethernet Dst Addr: Wildcard (1)
                    .... .... .... .... .... .... ...0 .... =   Ethernet Type: Exact (0)
                    .... .... .... .... .... .... ..0. .... =   IP Protocol: Exact (0)
                    .... .... .... .... .... .... .0.. .... =   TCP/UDP Src Port: Exact (0)
                    .... .... .... .... .... .... 0... .... =   TCP/UDP Dst Port: Exact (0)
                    .... .... .... .... ..00 0000 .... .... =   IP Src Addr Mask: /32 (0)
                    .... .... .... 0000 00.. .... .... .... =   IP Dst Addr Mask: /32 (0)
                    .... .... ...1 .... .... .... .... .... =   VLAN priority: Wildcard (1)
                    .... .... ..1. .... .... .... .... .... =   IPv4 DSCP: Wildcard (1)
                Input Port: 2
                Input VLAN ID: 65535
                Ethernet Type: IP (0x0800)
                Protocol: TCP (0x06)
                IP Src Addr: 10.0.0.4 (10.0.0.4)
                IP Dst Addr: 10.0.0.1 (10.0.0.1)
                TCP/UDP Src Port: commplex-link (5001)
                TCP/UDP Dst Port: 54453 (54453)
            Flow Duration (sec): 11
            Flow Duration (nsec): 929000000
            Priority: 2
            Number of seconds idle before expiration: 10
            Number of seconds before expiration: 0
            Cookie: 0x0020000000000000
            Packet Count: 8704
            Byte Count: 577504
            Output Action(s)
                Action
                    Type: Output to switch port (0)
                    Len: 8
                    Output port: 1
                    Max Bytes to Send: 65535
                # of Actions: 1
```

Figura 36 – Exemplo de uma mensagem *Statistics Reply*.

APÊNDICE C

Extensão do protocolo OpenFlow

Uma das grandes vantagens de se usar o protocolo OpenFlow é que pode-se adicionar novas funcionalidades a ele. Com isso, qualquer pessoa que tenha um mínimo de conhecimento sobre programação e da arquitetura SDN, pode fazer modificações nos comutadores e no controlador para torná-los mais completos e eficientes. Na arquitetura tradicional de redes, os comutadores e os roteadores são fechados, ou seja, apenas a empresa responsável pelo desenvolvimento destes podem modificá-los.

Embora o OpenFlow defina uma série de contadores em diversos níveis de granularidade, dois contadores poderiam ser adicionados na granularidade de nível de fluxo, o contador de pacotes descartados e *bytes* descartados. Quando um pacote é descartado pelo comutador OpenFlow, os contadores, na granularidade de porta, de pacotes e *bytes* descartados são incrementados, entretanto não é possível saber a qual fluxo pertencia o pacote descartado. Além disso, um pacote pode passar em um fluxo e, posteriormente, ser descartado por uma fila de saída, entretanto o fluxo não saberá que aquele pacote foi descartado e isso pode dar a falsa sensação que o fluxo está transmitindo seus pacotes corretamente. Caso o fluxo tivesse contadores de pacotes e *bytes*, a fila de saída do dispostivo poderia ser a entidade responsável por atualizar esses contadores de pacotes/*bytes* descartados, com isso teríamos uma visão mais precisa do funcionamento da rede.

A Tabela 15 detalha a proposta de inserção de dois novos contadores na granularidade de fluxo. Como é possível notar, os contadores de Pacotes Descartados e *Bytes* Descartados, em itálico, foram adicionados, onde cada contador utiliza 8 *bytes*, garantindo que cada contador possa representar um valor de 0 a $2^{64} - 1$.

Para adicionar novos contadores é necessário alterar os comutadores e também os controladores OpenFlow. Uma vez que existem inúmeras versões de comutadores e controladores, cada um precisará de modificações diferentes no seu código. A Seção C.1 apresenta as modificações feitas no ofsoftswitch13.

Além da inserção de ambos os contadores no comutador virtual, é necessário realizar uma modificação no controlador, no nosso caso a modificação ocorreu no Floodlight, para que ele reconheça esta alteração feita na mensagem OpenFlow. Para isso utilizamos a

Tabela 15 – Entrada de Fluxo com novos contadores.

Contador	***Bits***
Pacotes Recebidos	64
Bytes Recebidos	64
Pacotes Descartados	64
Bytes Descartados	64
Duração (segundos)	32
Duração (nanosegundos)	32

ferramenta LoxiGen.

Dentro do diretório do LoxiGen existe um conjunto de arquivos usados para gerar a biblioteca OpenFlow, as *structs* contidas nestes arquivos são semelhantes às contidas na especificação do OpenFlow(ONF, 2013). Basta adicionar os dois contadores e o tamanho dos mesmos em um desses arquivos, no caso do OpenFlow 1.3, o arquivo chama "standard-1.3", e então, basta compilar e gerar a biblioteca. Após gerar a biblioteca, basta importá-la no Floodlight.

C.1 Adição de Contadores

Para adicionar os contadores de pacotes e *bytes* descartados no OpenFlow Software Switch devem ser adicionadas as seguintes linhas no arquivo a seguir:

1. `oflib/ofl-structs.h`
   Linha 160: uint64_t dropped_packet_counter;
   Linha 161: uint64_t dropped_bytes_counter;

2. `oflib/ofl-messages-pack.c`
   Linha 189: ofr→dropped_packet_counter =
   hton64(msg→stats→dropped_packet_counter);
   Linha 190: ofr→dropped_bytes_counter =
   hton64(msg→stats→dropped_bytes_counter);

3. `oflib/ofl-messages-unpack.c`
   Linha 305: dr→stats→dropped_packet_counter =
   ntoh64(sr→dropped_packet_counter);
   Linha 306: dr→stats→dropped_bytes_counter =
   ntoh64(sr→dropped_bytes_counter);

4. `oflib/ofl-structs-print.c`
   Adicionar na chamada da função `fprint` os seguintes parâmetros:
   Linha 551: d_pkt_cnt=\"%"PRIu64"\", d_byte_cnt=\"%"PRIu64"\"
   Linha 554: s→dropped_packet_counter, s→dropped_bytes_counter

5. `oflib/ofl-structs-pack.c`

   Linha 550: flow_stats→dropped_packet_counter =
   hton64(src→dropped_packet_counter);
   Linha 551: flow_stats→dropped_bytes_counter =
   hton64(src→dropped_bytes_counter);

6. `oflib/ofl-structs-unpack.c`
   Linha 528: s→dropped_packet_counter =
   ntoh64(src→dropped_packet_counter);
   Linha 529: s→dropped_bytes_counter =
   ntoh64(src→dropped_bytes_counter);

7. `include/openflow/openflow.h`
   Linha 1042: uint64_t dropped_packet_counter;
   Linha 1043: uint64_t dropped_packet_counter;
   Linha 1493: uint64_t dropped_packet_counter;
   Linha 1494: uint64_t dropped_bytes_counter;

8. `udatapath/flow_entry.c`
   Linha 359: entry→stats→dropped_packet_counter = 0;
   Linha 360: entry→stats→dropped_bytes_counter = 0;

Os contadores estão contidos na granularidade de fluxo após a adição das linhas anteriores. O próximo passa detalha como atualizar os contadores quando um pacote é descartado na porta de saída do comutador. Adicionar as seguintes linhas em `udatapath/dp_ports.c`
Linha 636: struct pipeline *pl;
Linha 637: struct flow_table *table;
Linha 638: struct packet *pkt;
Linha 639: pl = dp→pipeline;
Linha 640: table = pl→tables[0];
Linha 641: pkt = packet_create(dp, 0, buffer, false);
Linha 642: dp_ports_flow_entry_update(table, pkt);

Adicionar o Algoritmo 4 no arquivo `udatapath/dp_ports.c`.

Algoritmo 4 Atualizar Contadores de Descarte.

```
1: function DP_PORTS_FLOW_ENTRY_UPDATE(struct flow_table * table, struct packet * pkt)
2:     struct flow_entry *entry;
3:     LIST_FOR_EACH(entry, struct flow_entry, match_node, &table→match_entries){
4:         struct ofl_match_header *m;
5:         m = entry→match == NULL ? entry→stats→match : entry→match;
6:         switch (m→type){
7:             case (OFPMT_OXM): {
8:                 if (packet_handle_std_match(pkt→handle_std,
9:                   (struct ofl_match *)m)){
10:                    entry→stats→dropped_packet_counter++;
11:                    entry→stats→dropped_bytes_counter += pkt→buffer→size;
12:                    return 0;
13:                }
14:            }
15:        }
16:    }
17:    return 1;
```