Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Qualidade de Serviço Dinâmico para Diferentes Tipos de Fluxos em SDN

Alessandro Cordeiro de Lima

Dissertação apresentada como requisito parcial para conclusão do
Mestrado Profissional em Computação Aplicada

Orientador
Prof. Dr. Eduardo A. P. Alchieri

Brasília
2019

Ficha catalográfica elaborada automaticamente,
com os dados fornecidos pelo(a) autor(a)

CAC184q

Cordeiro de Lima, Alessandro
Qualidade de Serviço Dinâmico para Diferentes Tipos de Fluxos em SDN / Alessandro Cordeiro de Lima; orientador Eduardo Adilio Pelinson Alchieri. -- Brasília, 2019.
106 p.

Dissertação (Mestrado - Mestrado Profissional em Computação Aplicada) -- Universidade de Brasília, 2019.

1. SDN. 2. Qualidade de Serviço. 3. Fluxo Elefante. 4. Fluxo Guepardo. 5. Fluxo Alfa. I. Adilio Pelinson Alchieri, Eduardo , orient. II. Título.

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Qualidade de Serviço Dinâmico para Diferentes Tipos de Fluxos em SDN

Alessandro Cordeiro de Lima

Dissertação apresentada como requisito parcial para conclusão do
Mestrado Profissional em Computação Aplicada

Prof. Dr. Eduardo A. P. Alchieri (Orientador)
CIC/UnB

Prof. Dr. Marcos Fagundes Caetano
Universidade de Brasília

Prof. Dr. Edson Tavares de Camargo
Universidade Tecnológica Federal do Paraná

Profa. Dra. Aletéia Patrícia Favacho de Araújo
Coordenadora do Programa de Pós-graduação em Computação Aplicada

Brasília, 24 de Junho de 2019

Dedicatória

Primeiramente a Deus por todas as conquistas e pela fortaleza que tem sido na minha vida.

A toda minha família, especialmente ao meu pai Francisco, minha mãe Marluce, meus irmãos e minha querida esposa Thaís Oliveira por todo amor, compreensão e apoio ao longo dessa trajetória que compartilharam do esforço e compreenderam as (muitas) horas de dedicação.

Agradecimentos

Agradeço ao Prof. Dr. André Costa Drummond, que sugeriu o tema abordado nesta dissertação e também ao meu orientador e amigo Prof. Dr. Eduardo Adilo Pelinson Alchieri pela confiança, paciência e serenidade.

Agradeço aos meus colegas de trabalho da Faculdade de Ceilândia - FCE/UnB, Alisson Assis, Evilásio Marinho e Francisco Aírton por todo apoio prestado.

Agradeço especialmente ao PPCA (Programa de Pós-Graduação em Computação Aplicada) e a Universidade de Brasília - UnB pelo conhecimento e apoio ao longo de toda trajetória acadêmica. O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES), por meio do Acesso ao Portal de Periódicos.

Resumo

A estrutura de redes de computadores atual dificulta a implementação de Qualidade de Serviço (*Quality of Service (QoS)*) em fluxos distintos e nas aplicações em geral (ex: tráfego de vídeos *streaming*). Uma vez que a maioria das tecnologias de rede são proprietárias, o administrador de rede não detém o conhecimento técnico suficiente para configurar a qualidade de serviço nos equipamentos de rede com tecnologia proprietária. O paradigma das Redes Definidas por *Software* (*SDN*) surgiu para poder remover essas restrições separando o plano de controle do plano de dados. Essa separação proporciona aos administradores de rede o uso eficiente de recursos de rede e a facilidade de provisionamento de diversos serviços e novas aplicações desenvolvidas de acordo com a necessidade da rede (ex: *QoS* ou *firewall*). Porém, a própria tecnologia *SDN* ainda sofre com pouca documentação e a limitação de mecanismos sólidos para aplicação de *QoS*, principalmente em fluxos do tipo elefantes (tráfegos com volume de dados excessivos), guepardo (tráfegos que geram alta taxa de transferência na rede) e alfa (tráfego que geram diversas rajadas no tráfego). Visando preencher esta lacuna, este trabalho propõe um novo serviço que consegue trabalhar com o plano de controle e o plano de dados em *SDN*, chamado de QoS-Flux. Este serviço tem o objetivo de aplicar filtros para priorizar aplicativos sensíveis e executar algoritmos de *QoS* em diferentes fluxos em uma rede *SDN*. Resultados preliminares mostram que o QoS-Flux melhora significativamente o *QoS* em uma rede *SDN* nos parâmetros de atraso, *jitter*, perda de pacotes e largura de banda.

Palavras-chave: SDN, Qualidade de Serviço, Fluxo Elefante, Fluxo Alfa, Fluxo Guepardo

Abstract

The current computer network structure makes it difficult to implement Quality of Service (QoS) in distinct streams and in general applications (eg. streaming video traffic). Since most network technologies are proprietary, the network administrator does not have sufficient technical knowledge to configure quality of service on proprietary technology network equipment. The Software Defined Networks (SDN) paradigm has emerged in order to remove these constraints by separating the control plane from the data plane. This separation provides network administrators with the efficient use of network resources and the ease of provisioning of various services and new applications developed according to the need of the network (eg QoS or firewall). However, SDN technology itself still suffers from poor documentation and limited solid mechanisms for QoS application, especially in elephant type flows (traffic with excessive data volume), cheetah (traffic that generates high throughput in the network) and alpha (traffic that generates several bursts of traffic). In order to fill this gap, this work proposes a new service that can work with the control plan and data plan in SDN, called QoS-Flux. This service aims to apply filters to prioritize sensitive applications and execute QoS algorithms on different flows in an SDN network. Preliminary results show that QoS-Flux significantly improves the QoS in an SDN network in the parameters of delay, jitter, packet loss and bandwidth.

Keywords: SDN, Quality of Service, Elephant Flow, Alpha Flow, Cheetah Flow

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas e Siglas

API Application Programming Interface.

AQM Active Queue Management.

Codel Controlled Delay.

D-ITG Distributed Internet Traffic Generator.

DiffServ Differentiated Services.

DSCP Differentiated Services Code Point.

FIFO First In First Out.

FQ Codel Fair Queuing Controlled Delay.

HFSC Hierarchical Fair-Service Curve.

HTB Hierarchical Token Bucket.

IETF Internet Engineering Task Force.

IntServ Integrated Services.

MPLS Multi-Protocol Label Switching.

NFV Network Functions Virtualization.

OVSDB Open vSwitch Database Management Protocol.

PCMM PacketCable MultiMedia.

PHB Per-Hop Behavior.

QoS Quality of Service.

RED Random Early Detection.

RSVP Resource Reservation Protocol.

SDN Software Defined Networking.

SFQ Stochastic Fairness Queuing.

TE Traffic Engineering.

VoIP Voice over Internet Protocol.

Capítulo 1

Introdução

A *Internet* hoje se tornou um dos principais meios de comunicação entre as pessoas no mundo. No século passado, as pessoas acessavam basicamente aplicações de *e-mail*, *downloads* e verificavam as notícias do dia [6]. Dessa forma, todo o tráfego de *Internet* era tratado com a mesma prioridade. O modelo de serviço de melhor esforço em redes de *Internet Protocol* (*IP*) não oferecia garantias para parâmetros de desempenho de rede como atraso, *jitter*, confiabilidade e dentre outros. Porém, com o surgimento de novas aplicações em tempo real desenvolvidos para o nosso cotidiano (ex: videoconferência e aplicações bancárias *online*), requisitos como atraso, *jitter*, largura de banda e a perda de pacotes exercem um papel fundamental para o desempenho da rede, que antes era irrelevante[7].

Dentro deste contexto, é necessário buscar soluções que possam garantir Qualidade de Serviço (*Quality of Service – QoS*) na rede e consigam assegurar que os requisitos citados anteriormente sejam atingidos. Essas soluções devem lidar com os mais variados tipos de fluxos e priorizá-los de acordo com suas características. Estes fluxos podem ser classificados com base em seu volume (elefante ou rato) [8], rajadas (alfa ou beta) [9] e taxa de transferência (guepardo ou caracol) [10].

Os fluxos do tipo volume podem ser aqueles fluxos contínuos com grande quantidade de dados (elefantes) ou fluxos com pequena quantidade de dados (ratos), os quais possuem duração de tempo maior ou menor na rede [11]. Já os fluxos do tipo rajadas são aqueles que provocam muitas intermitências no tráfego (alfa) ou poucas intermitências de tráfego (beta) [12]. Por último, temos os fluxos que geram alta taxa de transferência (guepardos) ou baixa taxa de transferência (caracóis) [13].

Alguns trabalhos propuseram soluções para fornecer QoS em redes tradicionais sujeitas a esses tipos de fluxos [8, 9, 14]. Infelizmente, estas pesquisas usam soluções com algoritmos ou cálculos matemáticos complexos e não são viáveis sua implementação em uma rede de produção, uma vez que, requer conhecimento avançado por parte do ad-

ministrador de rede. Por isso, podemos superar essas limitações em *QoS* através da utilização do paradigma Redes Definidas por *Software* (*Software Defined Networking – SDN*), que consegue separar o plano de dados (responsável pela transmissão de dados) do plano de controle (responsável por configurar e gerenciar a rede). Os seguintes trabalhos [13, 15, 16, 17, 18, 19] abordam a tecnologia *SDN* aplicando meios de gerenciamento de *QoS* e concessão de prioridades para algum tipo de fluxo.

Entretanto, estas soluções existentes não conseguem realizar o gerenciamento de *QoS* de forma eficaz em mais de um tipo de fluxo na *Software Defined Networking (SDN).* Além disso, até o momento na literatura, não foi encontrado nenhuma solução que possa controlar e monitorar os fluxos do tipo volume, rajadas e taxa de transferência através da mesma aplicação em redes *SDN*, já que há diversas particularidades em cada fluxo. Por esse motivo, é necessário o desenvolvimento de alguma solução que possa garantir prioridades de tráfego e o gerenciamento desses fluxos com objetivo de reduzir os problemas que os mesmos possam gerar para uma rede *SDN*.

Neste contexto, foi desenvolvido um serviço chamado QoS-Flux que consegue aplicar engenharia de tráfego no plano de dados com o auxílio do plano de controle em *SDN*. Este serviço possui componentes que conseguem analisar as consequências que os fluxos do tipo volume, rajadas e taxa de transferência podem gerar em uma rede *SDN*. Além disso, será habilitado filtros de tráfego específicos para filas priorizadas de acordo com o tipo de aplicação e ativar dois algoritmos de *QoS* de forma dinâmica, os quais são acionados de acordo com a situação da rede.

O primeiro algoritmo é o *HFSC* [20], que trabalha com mecanismo de distribuição de links que especifica a taxa de transferência mínima; o segundo é o *FQ_Codel* [21], usa um modelo estocástico para classificar pacotes de entrada em fluxos diferentes e é usado para fornecer um padrão de enfileiramento de pacote justo. Quando o serviço é executado os dois algoritmos são ativados de forma automática e entram em modo de espera, inicialmente. No momento que ocorrer alguma situação de alta taxa de transferência, será executado o *HFSC*, caso ocorra atrasos, perda de pacotes e/ou *jitter*, o *FQ_Codel* é acionado, realizando a troca entre eles. Porém, dependendo da situação da rede, pode ocorrer dos dois algoritmos entrarem em funcionamento.

Alguns experimentos foram realizados para demonstrar as vantagens da implementação a partir de métricas de *QoS* em redes *SDN* realizando um comparativo entre algoritmos configurados manualmente e o serviço QoS-Flux configurado dinamicamente. Estes experimentos mostram que o QoS-Flux é capaz de melhorar o desempenho da rede no atraso, *jitter*, largura de banda e perda de pacotes.

1.1 Definição do Problema

Na última década, a *Internet Engineering Task Force* (*IETF*) analisou diversas arquiteturas de *QoS*, mas nenhuma foi verdadeiramente bem-sucedida e implementada [22]. Isso acontece porque as arquiteturas de *QoS*, como *IntServ* – Serviços Integrados e *Diffserv* – Serviços Diferenciados, foram desenvolvidas sobre arquitetura de roteamento salto a salto originado pela *Internet* antigamente, sem nenhuma perspectiva para redes do futuro. Além disso, ainda existe uma outra solução dada pela *IETF* que seria o tunelamento com comutação de etiquetas por multiprotocolo (*MPLS*). Porém, existe a interoperabilidade entre configurações nos diversos modelos de equipamentos de rede.

Segundo a *Open Networking Foundation* (*ONF*), com o surgimento do paradigma *SDN*, gerou uma grande revolução em redes de computadores que pode ajudar a reduzir os problemas principais em *QoS*, os quais não deixaram que as técnicas tradicionais citadas anteriormente fossem implementados efetivamente até hoje [23].

O trabalho elaborado por [24] explica que na tecnologia *SDN*, a versão do protocolo *OpenFlow* v1.0 habilita os dispositivos de encaminhamento para que enviem o fluxo de tráfego para somente uma saída em fila, porém, para que isso ocorra é necessário utilizar outro protocolo como o *Network Configuration* (*NETCONF*). Além disso, também podemos configurar a marcação de pacotes através do protocolo *Differentiated Services Code Point* (*DSCP*). Ainda segundo os autores, as versões mais recentes do *Openflow* (ex: v1.3 até v1.5) foram acrescentados alguns recursos para configurações de *QoS*, como por exemplo, classificar os fluxos de tráfego somente por limite de taxa de transferência.

Por fim, o *SDN* e o protocolo *Openflow* nas versões 1.0 até 1.5 não disponibiliza ainda de mecanismos para gerenciar e aplicar *QoS* em fluxos de forma efetiva em tráfegos de volume, rajada e taxa de transferência, ou seja, na literatura não foi encontrado nenhum trabalho que consiga aplicar tais mecanismos para os três fluxos ao mesmo tempo, somente para um ou outro. Esses fluxos podem ocasionar problemas de congestionamento, lentidão, excesso de *buffer* e uso excessivo de taxa de transferência na rede, sendo necessário outras técnicas combinadas com *SDN* para o controle desses fluxos, como é o caso do serviço QoS-Flux.

1.2 Justificativa do Tema

Na rede de dados das instituições públicas de ensino do governo federal e estadual (universidades, institutos técnicos e escolas técnicas) há um parque tecnológico grande de equipamentos em seus datacenters (ex: switches, roteadores, *firewalls*, servidores, *hubs* e placas de rede) administrados pelas equipes de tecnologia da informação. Esses equi-

pamentos que se encontram nestes centros de dados, geralmente são embarcados, alguns dispositivos antigos, sistemas operacionais distintos, modelos e/ou marcas diferentes, funcionando com configurações específicas, possuem custo elevado de suporte e dificuldade para ter escalabilidade. Além disso, geralmente, não há implementações de políticas de *QoS* bem como soluções de gerenciamento e monitoramento de fluxos nas infraestruturas de redes.

Por isso, a justificativa do tema desse projeto tem como perspectiva o estudo dos dispositivos com fio (*switches*, roteadores e etc.), buscando redução de custos, solução dos problemas de monitoramento e gerenciamento de *QoS*, melhor distribuição dos fluxos em funcionamento, escalabilidade e a implantação de uma nova infraestrutura com soluções abertas, não importando a marca ou modelo do equipamento utilizado, sem gerar conflitos na rede em produção.

1.3 Perguntas de Pesquisa

Diversas perguntas de pesquisa foram definidas, os quais ajudarão no desenvolvimento deste estudo, onde serão listadas abaixo:

a) **Como se beneficiar das características de *SDN* para agilizar a implementação de *QoS* ?** O desenvolvimento de aplicações para dispositivos e controladores voltados para tecnologia *SDN* possuem capacidade de modificar a forma de monitoramento e o gerenciamento de *QoS*. Assim sendo, podemos observar novas condições para implementação de políticas e mecanismos de controle mais eficazes para *QoS*.

b) **Como gerenciar e reduzir os efeitos dos fluxos do tipo volume, rajadas e taxa de transferência em uma rede SDN ?** A escolha de algoritmos que consigam trabalhar com *QoS* e implementá-los em um rede *SDN*, sem que haja intervenção do administrador de rede é primordial. Este cenário contribui para redução de tempo na resolução de problemas na rede que esses fluxos possam causar bem como diminuição do custo com softwares de terceiros.

c) **Como o custo pode influenciar no método de configuração de algoritmos de *QoS* adotado para redes *SDN* ?** O tempo de resposta dado pelo administrador de rede para seus usuários pode influenciar bastante o custo da rede, principalmente com relação aos períodos de manutenção, suporte e resolução de problemas. Deste modo, é necessário realizar uma análise do tamanho da rede para decidir qual método de configuração consegue melhor atendê-lo, ou seja, configurações estáticas ou dinâmicas de *QoS* na rede *SDN*.

1.4 Objetivos

O objetivo geral deste estudo visa propor e avaliar um modelo de serviço híbrido de *QoS* em redes *SDN*, que considere fluxos diferentes (elefantes, guepardos e alfas) comparando com modelos tradicionais configurados estaticamente.

Para alcançar o objetivo geral deste trabalho, faz-se necessário atingir os seguintes objetivos específicos:

- Estudar os conceitos e os algoritmos que tenham propósito de aplicar *QoS* em uma rede *SDN*.
- Desenvolver e avaliar o serviço QoS-Flux utilizando somente *softwares* sem custo (*open-source*).
- Analisar o desempenho dos algoritmos selecionados que trabalhem com *QoS* e o serviço QoS-Flux em uma rede *SDN* virtualizada.

1.5 Organização do Trabalho

Este trabalho está dividido em 5 capítulos, contando com esta introdução. O Capítulo 2 apresenta a fundamentação teórica, abordando a Internet e os tipos de fluxos existentes, definições sobre Qualidade de Serviço (*QoS*) e Redes Definidas por *Software* (*SDN*), com suas características, desafios, modelos e protocolos. Além disso, este capítulo também apresenta os trabalhos desenvolvidos na literatura relacionados ao tema abordado.

O Capítulo 3 apresenta em detalhes a arquitetura do serviço QoS-Flux, desenvolvido para prover *QoS* dinâmico aplicado para redes *SDN*, com os seus componentes e requisitos necessários para monitoramento e gerenciamento de uma rede que trabalha com diferentes tipos de fluxos. Dessa forma, neste capítulo, também será ressaltado as ferramentas utilizadas para análise e desenvolvimento do serviço QoS-Flux, como o *Open Vswitch*, *Traffic Control Linux* (*TC Linux*) e o controlador *SDN Ryu*.

O Capítulo 4 apresenta uma série de experimentos realizados, bem como uma análise a cerca dos resultados dos respectivos algoritmos de *QoS* configurados estaticamente através da ferramenta *TC Linux* comparando com o serviço de *QoS* dinâmico QoS-Flux. Assim, neste capítulo será também apresentado os dispositivos que contribuíram para elaboração da topologia de rede, como a ferramenta *Mininet*, e para geração de carga de trabalho, como a ferramenta *D-ITG*.

Finalmente, o Capítulo 5 realiza várias ponderações com relação a proposta defendida a partir de uma visão geral, explicando também se foram cumpridos todos os objetivos específicos e, por fim, possibilidades de trabalhos futuros são discutidas.

Capítulo 2

Fundamentação Teórica e Trabalhos Relacionados

Este capítulo apresenta os conceitos fundamentais para este trabalho, os quais incluem os tipos de fluxos na *Internet*, *QoS* e redes *SDN*. Por fim, uma revisão do estado da arte é apresentada.

2.1 Internet e os Tipos de Fluxos

Um estudo realizado pela Cisco em 2016 mostra que o tráfego de vídeo na *Internet* tornou-se um dos principais serviços utilizados pelos usuários, o qual irá atingir 82% da *Internet* mundial até o ano de 2021, principalmente por serviços de *streaming*, *Video-on-Demand* (*VoD*), *Ultra-high-Definition* (*UHD*), *Internet Protocol Television* (*IPTV*), realidade virtual, vigilância por câmeras *online*, videoconferências em alta definição, dentre outros [25].

Outro estudo relata que existe uma grande popularidade de conteúdo multimídia em redes cabeadas (principalmente aqueles que utilizam transmissão de vídeo) [1]. De fato, nos últimos anos vem sendo notado a rápida expansão de serviços de entrega de conteúdo em larga escala, como o *YouTube*, *Netflix* e *Amazon Prime Video*. Ainda segundo a pesquisa, no continente americano se destaca o serviço de *Netflix*, no continente europeu o serviço de *Youtube* e no continente da Ásia/Oceania outros serviços de vídeos em *streaming HTTP* como os mais acessados pelos usuários entre todas as aplicações em 2018, como podemos observar na Figura 2.1.

Por isso, é primordial conhecer as particularidades do tráfego quando pretende-se desenvolver mecanismos de engenharia de tráfego para o controle da rede. Conhecer as características dos fluxos de *Internet* se torna também fundamental para fins de gerenciamento do tráfego [26].

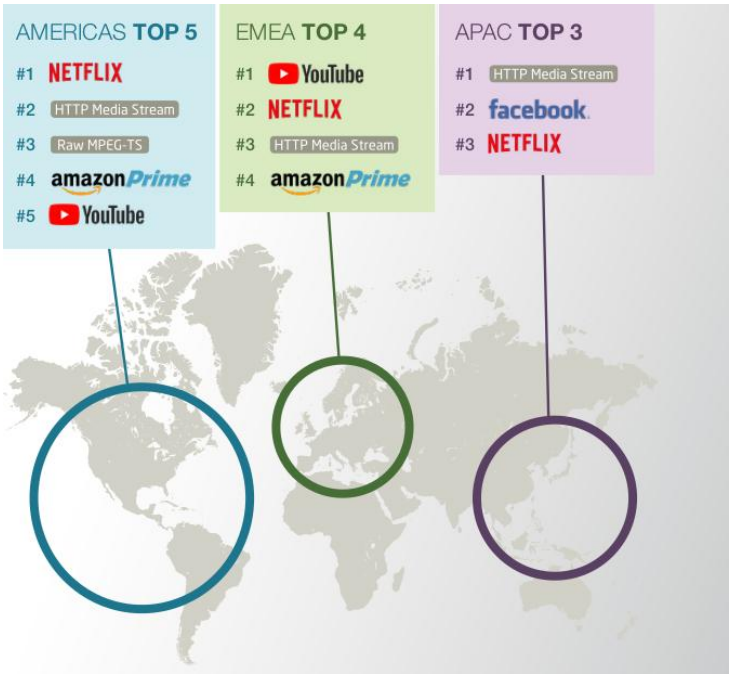

Figura 2.1: As principais aplicações mais utilizadas na internet pelo mundo em 2018 [1].

Dessa forma, diversos autores [9, 27, 28] explicam que os fluxos de *internet* podem ser caracterizados pelo volume (elefantes ou ratos), taxa de transferência (guepardo ou caracol) e rajadas (alfa ou beta).

Para entender melhor esta caracterização de fluxos da *Internet* dada pelos autores acima, também será necessário compreender o que seria exatamente um fluxo na *Internet*. Conforme explicado por [29], os fluxos de *Internet* podem ser considerados como uma sequência de pacotes que usam as mesmas cinco tuplas (5-tuplas): endereço *IP* de origem e destino, portas *IP* de origem e destino bem como o protocolo de rede. Além disso, os fluxos podem ser unidirecionais ou bidirecionais.

Nesse contexto, conhecer as particularidades do tráfego de *internet* e compreender suas razões implícitas é primordial, principalmente para desenvolver mecanismos de engenharia de tráfego com foco na otimização do desempenho de sistemas para fluxos [26].

As seções seguintes caracterizam os tipos de fluxos existentes.

2.1.1 Volume

Os fluxos do tipo volume são classificados de acordo com a quantidade de dados que trafegam e sua duração [11], sendo:

1) Ratos - Pode ser considerado como aqueles fluxos de *Internet* que possuem quantidade de dados com valor pequeno (ex: 1 byte) e duração de tempo reduzida (ex: 1 segundo);

2) Elefantes - Pode ser apontado como aqueles fluxos de *Internet* que possuem volumes de dados grande (1 Gigabyte) e duração de tempo elevada (ex: 1 hora). Além disso, os fluxos elefantes podem ser definidos de forma matemática. Estes fluxos possuem um volume $flow_s$ maior do que a média ($\bar{x} = \frac{1}{N}\sum_{i=1}^{N} x_i$) mais três vezes o desvio padrão dos dados amostrados ($\sigma = \sqrt{\frac{\sum_{i=1}^{n}(x_i - \bar{x})^2}{n-1}}$), ou seja,

$$Elephant = flow_s > (\bar{x} + 3 * \sigma) \tag{2.1}$$

de todos os fluxos. Caso um fluxo $flow_s$ seja menor do que $\bar{x} + 3 * \sigma$, então é considerado um fluxo rato.

Os fluxos elefantes podem ser definidos também com base em seu tamanho ou laragura de banda [30]. Na definição de tamanho, um fluxo é considerado elefante caso seu tamanho total seja maior que um limite, ou seja, um valor fixo (ex: 20 pacotes ou 1000 pacotes), diferente disso, é considerado fluxo rato. A definição de largura de banda, é definida por um valor determinado (ex: 100 *KBytes*) ou baseado no tráfego (ex: 5% do tráfego total ou 10% de tráfego total), diferente disso, será considerado fluxo rato.

Os usuários exigem dos fluxos ratos um tempo de resposta muito curto nas aplicações, enquanto os fluxos elefantes devem consumir um tempo maior de resposta [8]. Além disso, os fluxos elefantes trabalham com grandes massas de dados que podem gerar congestionamentos, atrasos e perda de pacotes nos fluxos ratos em uma rede. Por isso, é necessário desenvolver aplicações que possam minimizar esses problemas.

Geralmente, os fluxos considerados como ratos abrangem páginas *web*, *sites* de pesquisas na *web*, *email*, *DNS* e etc. Já os fluxos considerados como elefantes, abrangem serviços de *FTP*, *backup* de dados, banco de dados, atualizações de sistemas operacionais, *streaming* e etc [31].

2.1.2 Rajadas

Os fluxos do tipo rajadas são caracterizados na literatura como [32]:

1) Alfa (α): São fluxos de *Internet* com bastante rajadas no tráfego em certos intervalos de tempo, podendo ser um período grande ou pequeno (ex: 1 hora ou 30 minutos).

2) Beta (θ): Podem ser considerados fluxos de *Internet* com pequenas rajadas no tráfego em intervalos de tempo reduzido (ex: 30 segundos).

Existe outra caracterização para os fluxos em rajadas dada por [11], o qual explicam que esses fluxos podem ser divididos como porco-espinho (fluxo α) ou arraias (fluxo θ). Os fluxos porco-espinho podem ser aqueles matematicamente relacionados como fluxos de

rajadas ($flow_b$) maiores do que a média ($\bar{x} = \frac{1}{N}\sum_{i=1}^{N} x_i$) mais três vezes o desvio padrão ($\sigma = \sqrt{\frac{\sum_{i=1}^{n}(x_i-\bar{x})^2}{n-1}}$) de dados amostrados, ou seja,

$$Porcupine = flow_b > (\bar{x} + 3 * \sigma) \tag{2.2}$$

de todos os fluxos. Caso um fluxo $flow_b$ seja menor do que $\bar{x} + 3 * \sigma$, então é considerado fluxo arraia.

O tráfego em rajadas com ocorrência frequente, geralmente os fluxos alfa, podem gerar o aumento da perda de pacotes e o atraso em uma rede [33]. Essas perdas de pacotes e atrasos são encontradas com maior frequência na borda da rede. Além disso, o tráfego altamente intermitente (*On/Off*) pode gerar injustiça no espaço de armazenamento em *buffer* naqueles *switches* que utilizam memória compartilhada, também conhecido como *switches* que utilizam comutação por *software*, ou seja, ocorre quando um pacote, depois de recebido, é armazenado em uma memória compartilhada (*buffer*) e depois de analisado é enviado para porta de destino.

Existem outros problemas que os fluxos em rajadas podem causar em uma rede [34]. Entre eles, podemos destacar: moderação de interrupção (*Coalescing*), que dimunui a carga da *CPU* e acrescenta taxas de processamento em *switches* ou roteadores ocasionando pacotes em lotes, transportando rajadas no tráfego ao usuário; grandes transferências de dados de uma vez por aplicações na *Internet* também podem gerar bastante intermitência (*On/Off*) em uma rede.

Em outros trabalhos [35, 36], explica-se que os fluxos do tipo rajadas normalmente abrangem diversas aplicações, por exemplo, os fluxos alfas podem ser gerados em aplicações de Sistema de Arquivo em Rede (*NFS*), *P2P* (*torrent*), *streaming* de alta definição, aplicações *Big Data* (ex: *Hadoop MapReduce*) e etc.; os fluxos beta podem ser considerados como páginas *web* com gráficos pesados, pesquisas em páginas *web HTTP* e etc.

2.1.3 Taxa de Transferência

Os fluxos caracterizado como taxa de transferência podem ser considerados, seguindo uma definição geral na literatura, como [11]:

1) Guepardos: Aqueles fluxos de *Internet* que são característicos de alta taxa de transferência em um determinado intervalo de tempo, podendo este intervalo ser grande ou pequeno (ex: 1 hora ou 30 minutos), acima de 101 *KBit/s*.

2) Caracol: São considerados fluxos de internet característicos de baixa taxa de transferência em um determinado intervalo de tempo, podendo este intervalo ser grande ou pequeno (ex: 30 segundos ou 1 minuto), abaixo de 101 *KBit/s*.

Em outra definição sugerida por [37], são considerados fluxos guepardos somente aqueles fluxos que satisfaçam a fórmula matemática:

$$r_j > \bar{r} + 3\sqrt{\frac{\sum_{i=1}^{n}(r_i - \bar{r})^2}{n-1}} \tag{2.3}$$

Onde, n é o número de fluxos, r_j é a taxa do fluxo j $(j = 1, 2, ..., n)$ e $\bar{r} = \frac{1}{n}\sum_{i=1}^{n} r_i$ é considerado a taxa média do fluxo. O fluxo será guepardo caso satisfaça a condição r_j maior que $\bar{r} + 3\sqrt{\frac{\sum_{i=1}^{n}(r_i-\bar{r})^2}{n-1}}$, caso contrário será considerado fluxo caracol.

Os fluxos guepardos em uma rede podem preencher todos os *buffers* associados às portas do *switch* ou roteador, gerando efeitos danosos para outros fluxos. Em dispositivos de rede que possuem configurações de grandes *buffers* [38], geralmente encontrado nos Provedores de Serviço de *Internet* (*ISP*), também conhecido como fenômeno de *bufferbloat*, os fluxos guepardos podem gerar atrasos de pacotes para aplicações em tempo real. Para dispositivos com pequenos *buffers* (ex: *switches* em redes menores), os fluxos guepardos podem gerar perdas de pacotes [39].

Além disso, aqueles tráfegos de áudio/vídeo interativos de velocidade reduzida e que são sensíveis a atrasos (fluxos caracóis) coincidem com transferências de arquivos grandes (fluxos elefantes) e de alta velocidade (fluxos guepardos) [13]. Em comutadores de rede com *buffer* grande, um fluxo com bastante rajada (fluxo alfa) em uma situação de download de arquivo com alta taxa de transferência poderia gerar latências muito grande para os pacotes com tráfegos de aplicações interativas. Já aqueles comutadores que utilizam *buffer* pequeno, os fluxos com bastante rajadas de pacotes (fluxo alfa) em aplicações de downloads de arquivos com alta taxa de transferência, poderia gerar transbordamento de *buffer* e, consequentemente, perdas de pacotes, afetando a taxa de transferência de outras aplicações na *Internet*.

2.2 Qualidade de Serviço

Com o avanço da *Internet*, vários aplicativos e serviços de rede (ex: navegação na Web, mensagens de texto, *email*, jogos *on-line* e *e-commerce* e etc.) foram desenvolvidos para os usuários finais [40]. Porém, existem aplicações que possuem particularidades, como por exemplo, aplicações de *streaming* que necessitam de uma porcentagem maior de taxa de transferência para seus fluxos, diferente de aplicativos como o *Voice over Internet Protocol (VoIP)*, que utilizam um valor menor de taxa de transferência e são mais sensíveis ao atraso em uma rede.

Neste contexto, a tecnologia de Qualidade de Serviço (*QoS*) pode ser utilizada. A *Internet Engineering Task Force (IETF)* apresenta três modelos de serviços, sendo eles:

o *Integrated Services (IntServ)* recomendado pela *RFC* 2998, o *Differentiated Services (DiffServ)* recomendado pela *RFC* 2474 e a *Multiprotocol Label Switching* (*MPLS*) recomendado pela *RFC 3031*.

2.2.1 IntServ

O serviço *IntServ* foi proposto para fornecer garantia fim-a-fim de *QoS* a apartir do núcleo da rede usando o protocolo *Resource Reservation Protocol (RSVP)*, onde o mesmo foi padronizado pela *RFC* 2205. Apresentado para garantir redução de atraso nos aplicativos que são intolerantes, o protocolo *RSVP* tem como característica reservar recursos para um fluxo de tráfego em todos os nós começando do host remetente para o host do receptor [41].

Além disso, o protocolo *RSVP* consegue assegurar alguns recursos, tais como taxa de transferência para fluxos ao longo do caminho de toda rede [42]. Essa tecnologia pode ser implantada em redes grandes ou menor escala, fornecendo *QoS* de qualidade relativamente alta. Porém, conforme o número de fluxos cresce e a escala da rede aumenta, o consumo de recursos nos roteadores de uma rede também deve crescer bastante devido à complexidade de configuração do serviço *IntServ*.

O *IntServ* possui duas classes de serviços: serviço garantido ou serviço de carga controlada [6].

a) Serviço garantido: tem como função tolerar aplicações de mídia em tempo real com mínimo de atraso garantido, como por exemplo, tráfego de videoconferência ou *VoIP*.

b) Serviço de carga controlada: foi desenvolvido para aplicações com mínimo de perda de pacotes e que não se preocupam com atraso na rede, por exemplo, aplicações de *email* e *backup*.

2.2.2 DiffServ

O modelo *Diffserv* foi introduzido pelo *IETF* para superar as limitações do *Intserv* com basicamente duas mudanças fundamentais [6]. Na primeira mudança, o processamento principal foi retirado do núcleo da rede e transportado para borda da rede. Na outra mudança, o serviço que era realizado por fluxo foi modificado para serviço por classe, onde os roteadores enviam os pacotes baseado na escolha da classe de serviço solucionando o problema de restrição do tipo de serviço.

O serviço *DiffServ* oferece escalabilidade sem a obrigação de sinalização e estado de fluxo. Isso quer dizer que ele não precisa confirmar a largura de banda ou atrasar informações para os roteadores antes de enviar o pacote. Como o *DiffServ* lida apenas com os roteadores de borda, não existe a obrigação de realizar configurações complexas no núcleo

da rede, apesar do roteador de borda conseguir enviar pacotes usando as prioridades e a classificação de cada pacote [43].

Uma das maneiras como o *QoS* pode ser definido no *DiffServ* inclui configurações de *bit* nos cabeçalhos *IP* dos endereços de origem e destino [15]. O *DiffServ* também usa as ferramentas de QoS, como classificação, marcação, policiamento, formatação e enfileiramento inteligente.

Segundo a *RFC* 4594, o *Diffserv* utiliza a classificação de tráfego e a marcação de pacotes *IP* através do campo *Differentiated Services* (*DS*). Esse campo está localizado no cabeçalho do pacote *IP*. Os primeiros três *bits* do campo *DS* encontramos a classe de tráfego e outros três estabelecem a probabilidade que um pacote pode ser rejeitado, além dos dois últimos *bits* que não é utilizado. Todos esses 6 *bits* geram o conhecido *Differentiated Services Code Point (DSCP).*

Além disso, os fluxos de tráfego de pacotes *IP* marcados obtem um comportamento específico por salto, conhecido como *Per-Hop Behavior (PHB)* [44]. Cada *PHB* é associado a um *DSCP* o qual define uma reserva de quantidade de recursos de encaminhamento (ex: espaço de buffer e taxa de transferência) para esses fluxos de tráfego ao longo da rede. Os pacotes marcados podem pertencer a um dos quatro grupos básicos de *PHB*, dependendo dos valores de *DSCP*. Esses grupos de *PHB* são respectivamente: *Best Effort* (*BE*), *Assured Forwarding* (*AF*), *Expedited Forwarding* (*EF*), *Class Selector* (*CS*).

2.2.3 MPLS

A tecnologia *Multi-Protocol Label Switching (MPLS)* é uma técnica emergente que utiliza rótulos de 32 *bits* em vez de protocolos de roteamento por cabeçalho *IP* (ex: *OSPF* ou *BGP*), porém, nada impede dele funcionar em conjunto com esses protocolos [45].

Esta tecnologia, geralmente, pode ser encontrada em redes de provedores *ISPs* para fornecer largura de banda eficiente e provisionamento de *QoS* [46]. O *MPLS* baseia-se especialmente em um rótulo (número) introduzido entre a camada 2 (camada de enlace de dados) e a camada 3 (camada de rede) no modelo *OSI*. Devido à variedade de estruturas de rede subjacentes, o *MPLS* consegue estabelecer conexões *IP* de ponta a ponta com diversas particularidades de *QoS* associadas aos múltiplos meios de transporte.

Outra maneira de adicionar *QoS* no protocolo *MPLS* seria usar o *RSVP-TE*, sendo uma extensão do protocolo *RSVP* do modelo *IntServ*, propondo um plano de controle entre domínios conhecido como *Traffic Engineering (TE)* [47]. Ele consegue suportar provisionamento de recursos em toda a rede. Este é uma das extensões mais configuradas para fornecer *QoS* em uma rede com *MPLS*.

Também temos o *MPLS-TE* que introduz modelos de *QoS* modificados, como o *Diff-Serv TE* (*DS-TE*), que é configurado junto com o *RSVP-TE*, o qual pode ser oferecido

em redes de comutação central para fluxos de tráfego em *switches Cores* (switches layer 3)([48]. O modelo *DS-TE* oferece melhor escalabilidade do que os modelos *DiffServ* e *IntServ* baseados em *IP*, mas ainda é limitado pelo uso do *RSVP-TE*. Em vista disso, na prática, seu uso é reduzido para fluxos trabalhando em *switches Cores* (*switches layer* 3), independentemente do tipo de agregação.

2.2.4 Problemas na Adoção de QoS Atualmente

Nos últimos anos, o *IETF* apresentou muitos modelos e serviços com diversas características, mas todos eles apresentaram algumas deficiências.

O modelo *IntServ* tem como desvantagem o aumento do tempo de conexão, reservas ineficientes de taxa de transferência e problemas de escalabilidade com o *RSVP*, sendo que o último gera aumento de processamento e uso de memória dos roteadores. No modelo *Diffserv*, quando há uma taxa de transferência constante, a utilização do mecanismo de prioridade começa a perder seu propósito, pois, a rede começa trabalhar com a mesma prioridade em todo tráfego de *Internet*, ou seja, no modo *Best Effort*. Além disso, o descarte seletivo de pacotes durante períodos de picos na rede gera uma alta probabilidade de ocorrer falha no serviço de conexões com baixa prioridade [49].

A tecnologia *MPLS* também apresenta alguns problemas, como por exemplo: uma camada adicional na rede tem que ser desenvolvida, interoperabilidade nas configurações dos equipamentos, não oferece nenhuma proteção de dados no tráfego de pacotes, dentre outros [50]. Outra deficiência encontrada pode ser observado no *RSVP-TE* [51]. Essa extensão pode tornar-se inflexível para ser configurada, pois, o prazo para o provisionamento pode chegar a vários dias para que os provedores de serviços de *Internet* disponibilizem ou atualizem a conectividade de circuitos virtuais (túneis em uma rede IP) na tecnologia *MPLS*, onde o custo pode se tornar vantajoso somente para empresas de grande porte que tenham foco em utilizar *QoS* a longo prazo.

Na tabela 2.1 foi realizado um resumo das principais características dos serviços de QoS tradicionais bem como suas vantagens e desvantagens para implementação em uma rede.

Tabela 2.1: Tabela com os Tipos de Serviços de QoS em Redes Tradicionais

Tipos de QoS	**Características Principais**	**Vantagem**	**Desvantagem**
DiffServ	Reserva de recursos através do protocolo RSVP	Robustez e funcionamento no núcleo da rede.	Aumento do tempo de conexão entre equipamentos, perda de desempenho nos equipamentos e etc.
IntServ	Classificação, filtragem e policiamento de pacotes	Escalabilidade e funcionamento na borda da rede	O serviço pode trabalhar com a mesma prioridade em todo tráfego de Internet (modo Best Effort) e o descarte de pacotes seletivos podem gerar falhas em conexões de baixa prioridade.
MPLS ou MPLS-TE	Técnicas de QoS através dos serviços DiffServ-TE e do protocolo RSVP-TE	Escalabilidade	Interoperabilidade nas configurações dos equipamentos, não oferece nenhuma proteção de dados no tráfego de pacotes e etc.

Por isso, segundo a *Open Networking Foundation* (*ONF*), o surgimento do paradigma *SDN* é uma grande revolução em redes de computadores que pode ser uma alternativa para reduzir os problemas principais em *QoS*, que não deixaram que as técnicas tradicionais citadas anteriormente fossem implementadas efetivamente em todos os *datacenters* até hoje [23].

2.3 Redes Definidas por Software

As Redes Definidas por *Software* (*SDN*) apareceram como um novo paradigma que consegue separar o plano de controle (gerenciamento) do plano de dados nos dispositivos de rede [52]. O controle de roteadores/*switches* é tradicionalmente executado em um dispositivo central, conhecido por controlador. Os equipamentos com tecnologia *SDN* possuem uma interface de programação que consegue trabalhar com o protocolo *OpenFlow*.

O estudo realizado por [53] explica os principais termos utilizados para redes *SDN*, como pode ser visualizado na Figura 2.2 e explicado a seguir:

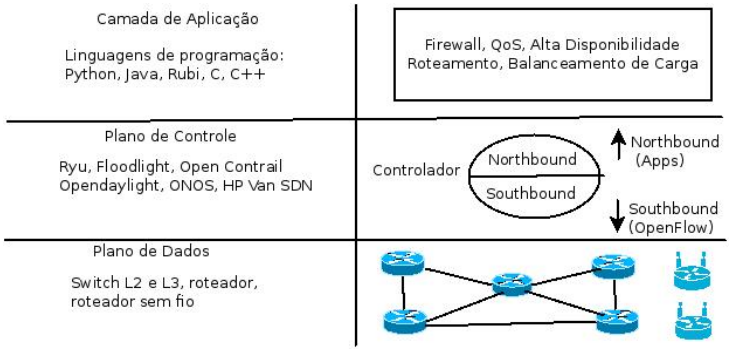

Figura 2.2: Arquitetura SDN.

a) Dispositivos de encaminhamento: São todos os ativos de rede, como os *switches*, roteadores, *firewall*/roteador, *gateways* virtuais, dentre outros, onde temos diversos conjuntos de instruções bem definidas, por exemplo, regras de fluxo que são usadas para executar ações de redirecionamento para os pacotes recebidos (enviar dados para portas específicas, enviar dados para o controlador e etc.). Essas regras são estabelecidas pela *Interface Southbound* e reenviadas pelo protocolo *Openflow* (ou outro protocolo *SDN*) aos dispositivos de encaminhamento através dos controladores *SDN*.

b) Plano de dados ou *Data Plane*: os dispositivos de encaminhamento são interligados utilizando a tecnologia sem fio ou cabeamento com fio. Toda infraestrutura de rede que compõe os dispositivos de encaminhamento interligados constitui o plano de dados.

c) *Interface* Sul ou *Southbound Interface*: o conjunto de instruções dos dispositivos de encaminhamento é estabelecido por uma aplicação de alto nível que faz parte da *Interface*

Southbound. Além disso, a *Interface Southbound* também é o principal responsável pelo protocolo de comunicação (por exemplo, o *Openflow*) entre os dispositivos de encaminhamento e os componentes do plano de controle.

d) Plano de controle ou *Control Plane*: os dispositivos de encaminhamento são programados por componentes do plano de controle ou controlador por meio da *Interface Southbound* de forma bem definida. O plano de controle pode ser entendido como a inteligência de toda rede *SDN*, o qual será melhor explicado em seções posteriores.

e) *Interface* Norte ou *Northbound Interface*: Essa *interface* trabalha diretamente com a Camada de Aplicação e o controlador, disponibilizando uma plataforma para desenvolvimento a partir de uma *Application Programming Interface (API)* em uma rede *SDN* para diversos serviços. As *APIs* desenvolvidas na *Northbound Interface* se comunicam com a *Interface Southbound* e os dispositivos de encaminhamento em uma rede *SDN*.

f) Camada de Aplicação ou *Application Layer*: Esta camada pode ser entendida como um grupo de aplicações que trabalham diretamente na *Interface Northbound* com o objetivo de executar as *APIs* desenvolvidas por alguma linguagem de programação nessa *interface*. Isso inclui aplicativos desenvolvidos para roteamento, *firewalls*, balanceadores de carga, *QoS*, dentre outros.

2.3.1 O OpenFlow

Os dispositivos de encaminhamento habilitados com *OpenFlow* possui uma ou mais tabelas de fluxo e uma camada de abstração, que podem conversar entre si com segurança utilizando o protocolo *Transport Layer Security* (*TLS*) através de um controlador *SDN* por meio do protocolo *OpenFlow* [2]. Podemos observar na Figura 2.3 a arquitetura de funcionamento do protocolo *Openflow* dentro de um dispositivo de encaminhamento, conforme será explicado a seguir:

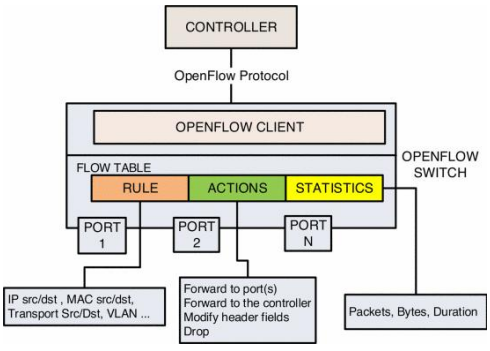

Figura 2.3: Arquitetura de Funcionamento do Openflow [2].

1) Regras de correspondência (*rule*) - São utilizadas para corresponder aos pacotes recebidos; esses campos possuem informações do cabeçalho do pacote, porta de entrada e saída, *Vlans*, dentre outros.

2) Contadores (*statistics*)- São usados para recolher estatísticas para o fluxo específico, como número de pacotes recebidos, número de *bytes* e duração do fluxo;

3) Ações (*actions*) - É um conjunto de instruções ou comandos a serem aplicadas no fluxo. As instruções determinam como lidar com os pacotes no tráfego da rede.

Existem diversas versões do *OpenFlow* atualmente [54]. A versão *OpenFlow* 1.0.0 define a tabela de fluxo com 12 tuplas, que contém informações como endereço *IP* e *MAC* de origem/destino, suporte a *DSCP* e etc. No *OpenFlow* 1.1.0 foi acrescentado algumas regras e suporte a múltiplas tabelas de fluxo. A partir do *OpenFlow* 1.2.0, adicionaram o *IPv6* e o suporte a controladores distribuídos. Mecanismos de congestionamento para *TCP* e o suporte para métricas básicas de *QoS* foi atribuído na versão 1.3.0. Na versão 1.4.0, foi proposto o protocolo *OpenFlow-Configuration* (*OFCONFIG*) como um protocolo de configuração. O *OpenFlow* 1.5.0 adiciona a nova tabela de saída, ou seja, permitindo maior agilidade para o pacote conseguir sair exatamente em sua porta correspondente.

2.3.2 Controladores SDN

O controlador tem a função de fornecer uma *interface* programática através de uma *API* bem definida para redes *SDN* [2]. O controlador pode trabalhar de forma centralizada ou distribuída e os aplicativos são fornecidos como se a rede fosse um único sistema para o usuário. Além disso, os controladores permitem que o modelo *SDN* seja aplicado em diversas aplicações, como tecnologia de redes heterogêneas e nas mídias físicas (ex: redes sem fio (*IEEE* 802.11, *IEEE* 802.16 e etc.), redes com fio (ex: *Ethernet*) e redes ópticas.

Existem diversos controladores disponíveis no mercado, porém, nem todos tem suporte para habilitar *QoS* em uma rede *SDN*, o qual será o propósito do nosso trabalho. Existem alguns controladores *Open Source* que disponibiliza essa tecnologia [40]:

1) *OpenDaylight Project* (*ODL*) - É um controlador *open source* baseado na linguagem de programação *Java*, desenvolvido pelo projeto colaborativo da *Linux Foundation* para promover o uso de *SDN*. O projeto *ODL* consiste em muitos outros subprojetos e *plugins*, como o *PacketCable MultiMedia (PCMM)*, que pode ser utilizado para acionar *QoS* através de políticas. Além disso, existe outro *plugin* conhecido como *Open vSwitch Database Management Protocol (OVSDB)* na *Interface Southbound*, encontrado na versão 1.3 do *OpenFlow*, que tem a função de gerenciar e configurar filas em *switches* para o controle de taxa de transferência.

2) *Open Network Operating System* (*ONOS*) - É um controlador *SDN open source* baseado na linguagem de programação *Java*, desenvolvido pela Open Networking Lab, o

qual utiliza tecnologia distribuída. Com relação a *QoS*, o *ONOS* suporta o mecanismo de medição para o protocolo *OpenFlow*, porém, essa funcionalidade poucas vezes é implementada em *switches* existentes. Esse controlador também tem suporte para filas configuradas na função *set_queue* para o controle de taxa de transferência, essa função trabalha na versão 1.3 do *OpenFlow*.

3) *Floodlight Project* - É um controlador de código aberto desenvolvido pela Big Switch Networks, baseado na linguagem de programação *Java*, sendo bem aceito pela comunidade *SDN*. Existem projetos gerados pela comunidade do *Floodlight* que facilita a integração, atualização de módulos novos e existentes. Entre eles, temos um módulo de *QoS* externo implementado para o controlador *Floodlight*, que visa fornecer configurações de filas para habilitar políticas e controlar a taxa de transferência [55]. Esta função é configurada na versão 1.0.0 do protocolo *OpenFlow*.

4) *Ryu Project* - É um controlador desenvolvido pela *NTT e OSRG group* elaborado em linguagem de programação *Python* [56]. Esse controlador tem suporte para as versões mais recentes do protocolo *OpenFlow*, além de oferecer novos aplicativos para gerenciamento, controle de rede, *QoS*, *firewall*, roteamento, dentre outros. A documentação sobre a tecnologia *QoS* oferecida pelo controlador *Ryu* [57] diz que ele é elaborado para trabalhar com a versão 1.3 do protocolo *OpenFlow*, disponibiliza somente configurações de filas que podem ser configuradas para métricas de controle de taxa de transferência.

2.4 Trabalhos Relacionados

2.4.1 Análise e gerenciamento de Fluxos em Redes SDN sem Aplicação de QoS

Foram realizados pesquisas para identificação de fluxos elefantes em redes *SDN*. Existem artigos que utilizaram para detecção deste tipo de fluxo a ferramenta *SFlow* [58, 59]. Porém, esta ferramenta utiliza monitoramento por amostragem, o qual não reflete todos os fluxos, gerando brechas. Dessa forma, não consegue detectar problemas que estão ocorrendo em toda rede *SDN*.

Alguns trabalhos realizaram pesquisas sobre gerenciamento de fluxos elefantes em SDN. Há trabalhos que utilizam um gerenciamento de fluxos elefantes utilizando um esquema de engenharia de tráfego escalável para categorizar fluxos [60] ou balanceamento de carga para distribuição dos fluxos [61]. Contudo, os autores realizam a seleção de fluxo restrito somente pela taxa de transferência consumida, podendo gerar falsos positivos, caso possa aparecer um fluxo guepardo na rede, por exemplo.

Existem trabalhos que utilizam a identificação e gerenciamento de fluxos guepardos em redes *SDN* [10]. Os autores desenvolveram uma implementação conhecida como *CFINF/CFTES*, baseado em *Network Functions Virtualization (NFV)* e *SDN* para realizar o reconhecimento desses fluxos a partir do espelhamento de pacotes. Entretanto, a proposta utiliza algoritmos complexos que podem dificultar sua implementação em uma rede de produção.

Outros projetos são voltados para identificação de fluxos alfa [62]. O artigo tem como propósito identificar este tipo de fluxo na rede através de espelhamentos de *switches SDN*. A solução desenvolvida pelos autores, se utilizado em redes de *datacenter*, pode gerar sobrecarga considerável para transmitir e processar todo o tráfego visualizado.

Também existem outros artigos relacionadas ao gerenciamento de fluxos alfa [63]. Naquele trabalho foi proposto um método para colocar os controladores em pontos quentes (*hot spot*) onde os *switches* carregam mais fluxos. Os *switches* com fluxos baixos podem migrar dinamicamente de um controlador sobrecarregado devido a fluxos alfa para outro controlador, reduzindo seus efeitos na rede. Os custos de migração são inevitáveis e os custos de troca de mensagens não podem ser insignificantes durante a migração do *switch*.

2.4.2 Identificação e Aplicação de QoS em Redes SDN para Fluxos Distintos

Existem alguns trabalhos que usam algoritmos de *QoS* configurados internamente nos controladores *SDN*, para fluxos elefantes [15], para fluxos alfa a proposta *QAMO-SDN* [16] e uma extensão do projeto *CFTES/CFINF* para fluxos guepardos [13]. Estes artigos utilizam a técnicas para reduzir a perda de pacotes [15, 16] ou engenharia de tráfego [13] aplicando redirecionamento de filas para controle da largura de banda e perda de pacotes. Os autores abordam somente algumas métricas de *QoS*, deixando de lado outras que são consideradas importantes, como por exemplo, o atraso ou jitter.

Outros trabalhos também se concentram em fornecer *QoS* em *SDN* através de configurações internas em *switches*, como o *Open Vswitch* (*OVS*) para fluxos elefantes [17] e fluxos alfa [18] ou placas *PCI* (*NetFPGA*) para fluxos com alta taxa de transferência (fluxo guepardo) [19]. As pesquisas realizadas utilizam algoritmos complexos para fluxos elefantes, modelos analíticos para fluxos alfas ou configurações de *QoS* específicas para determinado *hardware* em fluxos guepardos.

Diferente dos outros artigos, existem módulos externos que habilita *QoS* em *SDN* configurados em *switches* ou controladores. Em controladores, temos aqueles que utilizam um *framework* habilitado no *OpenDaylight* [64], o *Queue Pusher* no controlador *Floodlight* [55] e o controlador *HiQoS* [42]. Para *switches SDN*, temos os projetos implementando rote-

amento junto com *QoS* [65] e o *QoSFlow* [66], ambos com intuito de limitar a taxa de transferência na rede. Todos os módulos desenvolvidos são dependentes da linguagem de programação *Java* nos controladores ou com suporte somente para *switches OVS*.

Como observado nas pesquisas, todos os artigos em suas abordagens conseguem se aproximar da nossa proposta, porém, distinguindo como será desenvolvido. O objetivo do nosso trabalho é o desenvolvimento e análise do módulo dinâmico QoS-Flux, com objetivo de gerenciar e monitorar os fluxos elefantes, guepardos e/ou alfa na rede *SDN*, através da configuração dinâmica de algoritmos aplicados através do módulo *TC Linux* para melhoria de características da rede com relação a latência, *jitter*, perda de pacotes e a largura de banda.

2.5 Considerações Finais

Neste capítulo, apresentou os conceitos necessários relacionados a *Internet* e os principais fluxos existentes, a tecnologia QoS e os principais modelos de serviço adotados em uma rede tradicional, o paradigma SDN bem como o protocolo OpenFlow e os principais controladores existentes que possuem suporte a QoS. Estes conceitos são indispensável para compreensão e desenvolvimento deste trabalho. Além disso, este Capítulo apresentou os trabalhos relacionados ao tema desta dissertação. O capítulo 3 apresentará a proposta de um novo serviço de QoS dinâmico utilizando a tecnologia SDN, chamado de QoS-Flux.

Capítulo 3

Qualidade de Serviço Dinâmico para Diferentes Tipos de Fluxos em SDN

Como não existe uma solução de *QoS* que consiga prover o melhor desempenho para todos os principais tipos de fluxos (volume, taxa de transferência e rajadas), existe a necessidade de desenvolver alguma solução que amenize os problemas existentes atualmente em QoS para *SDN*, como explicado nas seções anteriores. Por isso, foi projetado uma solução chamada de QoS-Flux que consegue alinhar o paradigma *SDN* e a tecnologia de *QoS* através de técnicas de *DiffServ* [67]. Motivo da implementação é contribuir para o gerenciamento e controle dinâmico dos fluxos elefantes (volume), guepardos (taxa de transferência) e alfas (rajadas).

3.1 QoS-Flux

O serviço QoS-Flux tem como propósito realizar o monitoramento e controle de fluxos em redes SDN, especialmente aqueles tipos específicos, ou seja, fluxos elefantes, guepardos e/ou alfas. Além disso, também é utilizado algumas funções, como: definição de métricas e a porcentagem de taxa de transferência (restringe o limite máximo de taxa de transferência pela porta do dispositivo), filtros para aplicações (portas e/ou protocolos com análise se existe ou não prioridade) e o emprego dos algoritmos *Hierarchical Fair-Service Curve* – HFSC [20] e o *Fair Queuing Controlled Delay* – FQ_Codel [21] de forma dinâmica (acionados de acordo com estado da rede *SDN*). Estes componentes serão melhores explicados na seção 3.1.1.

Os pré-requisitos para instalação, configuração e a montagem do ambiente de testes para o serviço QoS-Flux podem ser verificados no site do próprio projeto na plataforma *Github* [67].

3.1.1 Funcionamento do QoS-Flux

Na figura 3.1, pode ser visualizado a arquitetura básica do serviço QoS-Flux. Este serviço será iniciado a partir da análise dos principais fluxos de entrada, ou seja, verificando as consequências que os fluxos elefantes e/ou guepardo e/ou alfa possam gerar em uma rede relativo ao atraso, *jitter*, perda de pacotes e a largura de banda.

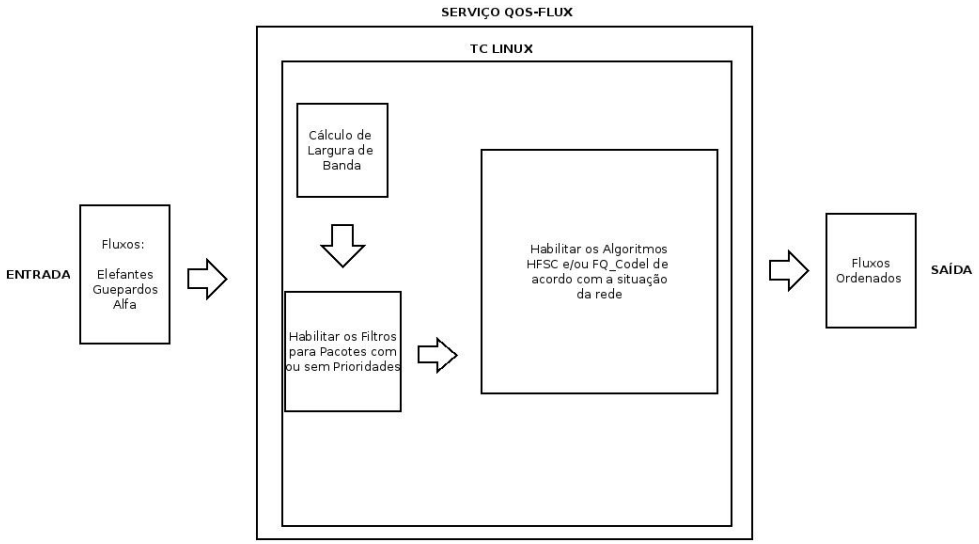

Figura 3.1: Arquitetura Básica de Funcionamento do Serviço QoS-Flux

O serviço QoS-Flux realizará chamadas automáticas no *kernel* do sistema operacional *Linux* através de comandos no módulo *TC Linux* adaptados para *SDN*, sem interação do administrador de rede, explicado na subseção 3.3

No próximo passo será acionado o cálculo do limite de largura de banda na rede, realizado de acordo com a equação 3.1.

$$\sum bandwith^{limit} = \frac{rate^{percent^{(x)}} * rate^{MAX^{(y)}}}{100} \tag{3.1}$$

A equação 3.1 é dada pelo somatório $bandwith^{limit}$ igual a porcentagem da largura de banda disponível na rede no parâmetro $rate^{percent^{(x)}}$ multiplicado pela largura de banda máxima $rate^{MAX^{(y)}}$ divido por 100.

Os parâmetros valor de porcentagem e a largura de banda máxima podem ser alterados de acordo com a velocidade do *link* contratado com o provedor. O limite de largura de banda é necessário ser configurado para não extrapolar o limite máximo que a rede pode suportar. No nosso caso, utilizaremos neste trabalho o valor de porcentagem de 95% e largura de banda máxima de 1 *GB/s*.

No passo seguinte, o módulo *TC Linux* também deve habilitar outros parâmetros do comando *tc*, o qual será acionado em dois estágios. No primeiro estágio, será configurado

Algoritmo 1: Serviço QoS-Flux

Entrada: Fluxos Elefantes e/ou Guepardos e/ou Alfa
Saída: Fluxos ordenados por prioridades

```
1  início
2      Execute algum comando: Start, Stop, Restart, Filter ou Status
3      para Start faça
4          ▷ Ver subseção 3.4;
5          Calcular a largura de banda ▷ Ver equação 3.1;
6          Executar a filtragem de pacotes ▷ Ver subseção 3.3;
7          Habilitar o algoritmo HFSC e entrar no modo de espera ▷ Ver subseção 3.3;
8          Habilitar o algoritmo FQ_Codel e entrar no modo de espera ▷ Ver
             subseção 3.3;
9          Executar os algoritmos somente de acordo com a situação da rede, sendo
             HFSC (alta taxa de transferência) e FQ_Codel (atraso, jitter e perda de
             pacotes);
10         Enviar fluxo para o controlador;
11     fim
12     para Stop faça
13         ▷ Ver subseção 3.4;
14         Parar cálculo de largura de banda
15         Parar a filtragem de pacotes
16         Parar algoritmo HFSC
17         Parar algoritmo FQ_Codel
18     fim
19     para Restart faça
20         Reinício das configurações do módulo, inserindo o comando Stop e iniciando
             o comando Start ▷ Ver subseção 3.4;
21     fim
22     para Filter faça
23         Mostrar a filtragem de pacotes em aplicações com ou sem prioridade ▷ Ver
             subseção 3.4;
24     fim
25     para Status faça
26         Mostrar configurações do comando Start e monitorar algoritmos de QoS ▷
             Ver subseção 3.4;
27     fim
28 fim
```

os filtros de rede verificando se os fluxos possuem prioridade (ex: aplicações HTTP ou videoconferência) ou sem prioridade (ex.: aplicação de backup)(ver subseção 3.3.2). No segundo estágio, será configurado os algoritmos de *QoS* (*Hierarchical Fair-Service Curve – HFSC* e o *Fair Queuing Controlled Delay – FQ_Codel*) de forma automática para aplicar filas dinâmicas de *QoS*(ver subseção 3.3.1.

O algoritmo *HFSC* deve acionar suas configurações utilizando critérios de compartilhamento de *links*. Será distribuído por largura de banda de acordo com valores de porcentagem alocado seguindo os critérios de filtros no QoS-Flux (aplicações com prioridade – valor de 100% e sem prioridade valor de 85%). Este método utiliza o cálculo de curva de serviço em tempo virtual. O *link* com o menor tempo será agendado e aquele com maior tempo será enviado. O algoritmo deve aguardar em modo de espera no serviço QoS-Flux. O *HFSC* será acionado somente se ultrapassar os limites configurados nos *links*.

Com relação ao algoritmo *FQ_Codel* no serviço QoS-Flux, deve ser habilitado suas configurações de forma controlada através do enfileiramento justo no serviço. Além disso, será monitorado o comportamento dos fluxos, aguardando em modo de espera. Este algoritmo somente deve entrar em execução se houver alterações na rede com relação ao atraso e/ou *jitter* e/ou perda de pacotes.

A troca entre os algoritmos devem satisfazer os requisitos acima para serem acionados dinamicamente.

No último passo, o serviço QoS-Flux será enviado para saída, ou seja, encaminhados para o controlador com todos os fluxos ordenados.

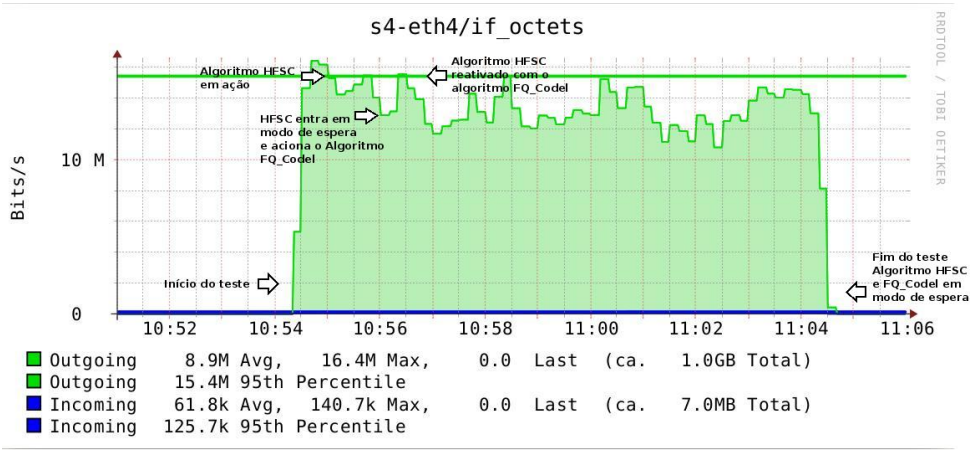

Figura 3.2: Exemplo de funcionamento dos Algoritmos Dinâmicos no Serviço QoS-Flux.

Na figura 3.2 foi analisado como exemplo um teste simples com um fluxo alfa por 10 minutos, o qual foi gerado um gráfico através da ferramenta *RRDtool* configurado no *software collectd-web* [68]. Nesta análise é observado a troca entre os algoritmos (*HFSC* e *FQ_Codel*) no serviço QoS-Flux de forma dinâmica, onde o filtro foi selecionado como tráfego de não prioridade. Quando o fluxo de saída (*Outgoing*) aumenta a taxa de transferência, é acionado o algoritmo *HFSC* para realizar o controle do mesmo. No momento que o fluxo é reduzido, foi gerado atrasos, neste caso será processado o *FQ_Codel*. Quando

foi processado os dois algoritmos ao mesmo tempo, os mesmos realizaram o controle dos fluxos até o término do teste.

Podemos gerenciar o QoS-Flux através de comandos na *CLI*, como observado no Algoritmo 1: *Start*, *Stop*, *Restart*, *Status* e *Filter*, onde serão explicados na subseção 3.4.

Dependendo do comando acionado no serviço QoS-Flux pelo usuário, será gerado uma resposta, onde o mesmo deve enviar a informação para o controlador com destino ao *switch* de borda (agregação), seguindo o exemplo da topologia na Figura 4.1. O *switch* de borda (agregação) deve encaminhar para o destino as informações enviadas pelo serviço QoS-Flux para porta específica do *switch* de acesso.

Todos os fluxos no serviço QoS-Flux são armazenados e podem ser visualizados pelo arquivo *qos-flux.csv*. Porém, é necessário ferramentas externas que consigam analisar grande volume de dados para visualização de todo fluxo na rede.

3.1.2 Implementação do QoS-Flux

O serviço QoS-Flux possui suporte somente para sistemas operacionais Linux, desenvolvido através da linguagem de programação *Shell Script* (Anexo 1). Além disso, esta aplicação trabalha com os componentes do módulo *TC Linux*, apresentado em maiores detalhes na seção 3.3, sendo pré-requisito do QoS-Flux para seu pleno funcionamento.

O QoS-Flux foi elaborado e adaptado para redes *SDN* com base nos projetos existentes para redes tradicionais cabeadas que aplicam *QoS* através de algoritmos *Hierarchical Token Bucket – HTB*, *Stochastic Fairness Queuing – SFQ*, *Fair Queuing Controlled Delay* – FQ_Codel e *Hierarchical Fair-Service Curve* – HFSC, ou seja, aplicações como o *Wonder Shaper* [69] e o *SuperShaper-SOHO* [70].

O *Wonder Shaper* foi desenvolvido na linguagem *Shell Script*, o qual utiliza os algoritmos *HTB* [71] com a função de limitar a largura de banda e o *Stochastic Fairness Queuing – SFQ* [72] para permitir equilibrar os fluxos de tráfego (TCP ou UDP) em um tráfego saturado. O *Wonder Shaper* é acionado na *CLI* pelo usuário configurando as taxas de *download* e *upload* para cada interface de rede individualmente. Esta aplicação trabalha com os comandos do módulo *TC Linux* para seu funcionamento [69].

Com relação ao *SuperShaper-SOHO*, a aplicação é elaborada utilizando a linguagem de programação *Shell Script*, o qual trabalha de forma dinâmica para aplicar filtragens de pacotes e habilitar dois algoritmos para o controle de largura de banda, ou seja, o *HFSC* [20] e o *SFQ* [72]. O algoritmo *FQ_Codel* pode substituir o *SFQ* na aplicação, porém, é opcional. O algoritmo *HFSC* é configurado para aplicar curvas de serviço que limitam a largura de banda e o *SFQ* para habilitar o equilíbrio entre os *links* de tráfego que esteja saturado na rede. O usuário pode acionar comandos para iniciar, parar, reiniciar e

verificar os filtros executados no *SuperShaper-SOHO*. Além disso, emprega o módulo *TC Linux* para seu funcionamento [70].

A Figura 4.1 apresenta a arquitetura de uma rede SDN com o serviço QoS-Flux. Esta topologia é apresentada como exemplo, a qual será utilizada para análise de desempenho do serviço QoS-Flux neste trabalho. Porém, o serviço QoS-Flux pode ser empregado em qualquer outra topologia, bastando ter os elementos mínimos necessários para seu funcionamento. Seguindo a Figura 4.1, no Plano de Aplicação encontramos o serviço QoS-Flux, no plano de controle é identificado o controlador *Ryu* e no plano de dados os *switches Open vSwitch* (*OVS*). No plano de dados, pode ser utilizado outros tipos de *switches* (proprietário ou virtual), sendo necessário habilitá-los no código do serviço QoS-Flux.

Nas próximas seções serão detalhadas todas as tecnologias utilizadas na topologia para o funcionamento do QoS-Flux, explicando as ferramentas que auxiliam, as particularidades dos comandos de gerenciamentos e a operação com o controlador *SDN Ryu*.

3.2 Open vSwitch

O *switch OVS* é considerado como um *software open source* que funciona através da virtualização em multicamadas com objetivo de ser utilizado em produção e licenciado sob a licença *Apache* 2.0 [73]. Os autores também comentam que o *OVS* utiliza seu próprio banco de dados, conhecido como *OVSDB*. Além disso, esse projeto foi desenvolvido para permitir a automação total de rede através de extensões, além de suportar interfaces e protocolos de gerenciamento padrão (ex: *NetFlow, sFlow, IPFIX, RSPAN, CLI, LACP*, 802.1ag e *TC Linux*). O *OVS* foi projetado para oferecer suporte ao protocolo *OpenFlow* em todas as suas versões bem como tendo possibilidade de distribuição em vários servidores físicos parecido com a arquitetura do modelo *switch Nexus* 1000V da *Cisco*.

Este *switch*, além de ser um comutador virtual de alto desempenho, tem uma base flexível para a geração de serviços trabalhando nas camadas 2 e 3 (modelo *TCP/IP*) de forma virtualizada em *datacenters* [74]. Como seu modelo de encaminhamento pode ser baseado em fluxo (através do protocolo *OpenFlow*), isso garante de forma adequada o encaminhamento na camada L2-L3 sem estado, permitindo que ele alcance um alto nível de generalidade sem sacrificar o desempenho.

3.3 Traffic Control Linux

O *TC Linux* é um módulo para *QoS* do pacote *IProute2* para gerenciamento de rede no *kernel* do *Linux* [75]. Esse elemento consegue realizar diversas operações, como: modela-

Tabela 3.1: Componentes de Funcionamento do TC Linux [3]

Elemento	Componente	Função	Algoritmos (exemplos)
Agendador	*Qdisc*	O *qdisc* é o procedimento pelo qual os pacotes são organizados (ou reorganizados) na entrada e na saída de uma fila específica no dispositivo de rede. O agendador padrão e mais simples é o algoritmo *First-In First-Out* (*FIFO*). A *qdisc* pode trabalhar com a modelagem (class) ou sem modelagem (sem class) de tráfego.	qdisc + class: *HTB, HFSC, CBQ, PRIO.* *qdisc* sem *class*: *SFQ, FIFO, RED* e variantes, *FQ_Codel, Codel, TBF.*
Modelagem	*Class*	*Class* é o procedimento pelo qual os pacotes são atrasados, gerando temporização, antes de serem enviados em uma fila de saída para atender a uma largura de banda configurada. A técnica subjacente para modelar geralmente utiliza os mecanismos de *token* e *bucket*.	*HTB, HFSC, CBQ, PRIO.*
Classificação	*Filter*	*Filter* é o procedimento onde os pacotes são separados para diversos tratamentos específicos, podendo ser em diferentes filas de saída. Geralmente é utilizado o classificador *u32*. Neste processo, os tratamentos podem ser configurados pelo tipo de tráfego, ou seja, adicionando prioridades para fluxos com aplicações de tempo real na primeira fila (maior largura de banda) e/ou fluxos sem prioridade na segunda fila (menor largura de banda), por exemplo.	Todos os algoritmos podem trabalhar com filtragem de pacotes.
Policiamento	*Policy*	*Policy* é a ação de policiamento que permite limitar a largura de banda do tráfego correspondente ao filtro (*Filter*) ao qual ela está anexada.	Algoritmo com *token bucket* - utiliza os parâmetros: *rate*, *burst*, *mtu*, *peakrate*, *overhead* e *linklayer*. Algoritmo de amostragem - utiliza o parâmetro: *estimator*.
Descarte	*Drop*	*Drop* é uma ação que descarta os pacotes, caso os mesmos ultrapassem os limites configurados nos parâmetros dos algoritmos. Caso o algoritmo tenha a opção de marcação de pacotes, eles serão notificados através da notificação explícita de congestionamento (*ECN*) e depois quando atingir um limite configurado, os pacotes serão descartados.	*Drop - HTB, FIFO, HFSC, SFQ, CBQ PRIO.* *ECN + Drop - RED* e suas variantes, *Codel* e o *FQ_Codel*
Marcação	*Dsmark*	*Dsmark* é uma ação de enfileiramento que disponibiliza os recursos necessários em *Diffserv* para realizar a marcação de pacotes.	*SCH_DSMARK*

gem, programação de tráfego, policiamento, descarte de pacotes, dentre outros, através do comando *tc*. O *TC Linux* possui alguns componentes fundamentais: *qdiscs*, *class*, *filters*, *policy*, *drop* e *DSmark* [3], como explicado na Tabela 3.1.

Além disso, existem diversos algoritmos desenvolvidos no *TC Linux*, incluindo algoritmos simples como *First In First Out (FIFO)* ou aqueles mais elaborados como os algoritmos *Hierarchical Token Bucket (HTB)*, o *Hierarchical Fair-Service Curve (HFSC)* e o *Stochastic Fairness Queuing (SFQ)*. Também podemos destacar os algoritmos conhecidos como Gerenciamento Ativo de Filas ou *Active Queue Management (AQM)*, como os algoritmos *Random Early Detection (RED)*, *Controlled Delay (Codel)* e o *Fair Queuing Controlled Delay (FQ Codel)* [72].

Neste trabalho utilizaremos na configuração do serviço QoS-Flux os algoritmos *HFSC* e o *FQ_Codel* adaptados para funcionar em redes *SDN*.

3.3.1 Algoritmos TC Linux

Como observado na Tabela 3.1, percebemos que existem algoritmos no campo de agendadores (*Qdisc*) que trabalham com classes de tráfego (*class*), como o *HTB* e o *HFSC*, e aqueles agendadores que não utilizam classes de tráfego, como o *FQ_Codel* [3].

Estes algoritmos foram escolhidos, pois, o algoritmo *HTB* é um dos mais utilizado na literatura para controle de fluxos através de *QoS* em redes *SDN* [15, 56, 65, 66, 76] enquanto que o *HFSC* e o *FQ_Codel* conseguem trabalhar dentro do serviço QoS-Flux de forma dinâmica. Em nossos testes, utilizaremos ambos para avaliação de desempenho, os quais serão explicados a seguir.

Algoritmo *HTB* Este algoritmo permite configurações de filas de controle de largura de banda através do compartilhamento de *links* físicos para simular vários *links* mais lentos, enviando diferentes tipos de tráfego em diversos links simulados. Ele contém elementos de modelagem, com base no algoritmo *TBF* e pode priorizar classes. Utiliza uma disciplina de fila (*classful qdisc*) para gerar o controle de tráfego através de um sistema hierárquico e também uma organização chamada *class*. Esse algoritmo divide suas funções em dois comandos: *tc qdisc* e *tc class* [71].

Algoritmo *HFSC* É um tipo de algoritmo que permite a distribuição proporcional da largura de banda, bem como o controle e a alocação de latências. O *HFSC* tem o objetivo de compartilhar largura de banda de forma precisa (em tempo virtual) e/ou alocação de níveis de atrasos mínimos para todas as classes (em tempo real) através da curva de serviço linear.

A curva de serviço funciona como uma função não decrescente, retornando a quantidade de serviço de uma aplicação (uma quantidade alocada de largura de banda ou atraso mínimo) em algum ponto específico no tempo (configurado pelo usuário ou pelo próprio algoritmo).

Algoritmo *FQ_Codel* É um algoritmo *AQM* que determina o enfileiramento justo contribuindo para redução de congestionamento na rede através do esquema adotado no algoritmo *CoDel*. O *FQ_Codel* usa um modelo estocástico que ordena os pacotes de entrada em múltiplos fluxos, contribuindo para uma largura de banda justa, redução de atraso e perda de pacotes nos fluxos através de filas. Cada um desses fluxos é gerenciado pela disciplina de enfileiramento *CoDel*. A reordenação de pacotes dentro de algum fluxo é bloqueada, pois, o algoritmo *Codel* utiliza uma fila *FIFO*.

3.3.2 Filtros TC Linux

Os filtros são usados para classificar pacotes em suas propriedades. Estas propriedades podem ser definidas, como por exemplo, usando o *byte TOS* (tipo de serviço) no cabeçalho *IP*, os endereços *IP*, os números de portas e etc [76].

A possibilidade de configuração dos filtros no módulo *TC Linux* é bastante ampla. Por isso, o controle dos filtros são divididos em três categorias: filtros simples, filtros complexos e filtros baseados em múltiplos critérios [77].

1. Filtros Simples: São filtros que permitem configurações comuns de portas e protocolos para somente uma interface de rede especificada. Além disso, isso implica que o cabeçalho *IP* é considerado de tamanho constante (20 *bytes*) e, portanto, não deve

incluir nenhuma opção. A exclusão de filtros simples somente pode ser feita para uma faixa de prioridade completa.

2. Filtros Complexos: Trabalham com números de identificação configurados pelo usuário e pelas tabelas de *hash*. A tabela de *hash* possui *slots*, os quais abrangem regras de filtragem, agilizando a classificação exata de uma regra de filtro. A tabela de *hash* armazena qual protocolo será utilizado (ex: *IPv4*). Isso possibilita a análise do campo *ihl* (comprimento do cabeçalho da internet) para alcançar o ponto de início correto do protocolo da camada superior.

3. Filtros Baseados em Múltiplos Critérios: Neste tipo de filtro, podemos utilizar duas possibilidades para combinar diferentes regras de filtragem, sendo eles o *AND* lógico e o *OR* lógico.

    a) *AND* lógico - É utilizado tipos de critérios de filtragem que podem ser concatenados para possibilitar uma filtragem mais exclusiva. Para limitar pacotes em mais de um campo, um filtro com vários componentes pode ser utilizado.

    b) *OR* lógico - Em uma classe pode ter filtros diferentes. Por isso, todos os pacotes que se encontram nos filtros serão processados pela respectiva classe. Os filtros são acionados de acordo com cada prioridade e na estrutura em que foram gerados.

Neste trabalho, selecionamos para o funcionamento do serviço QoS-Flux os filtros simples e os baseados em múltiplos critérios (Anexo 1). Os filtros simples foram habilitados no serviço para os seguintes protocolos: *VoIP*/videoconferência/*skype*, *SMTP*, *IMAP*, *POP3*, *HTTP*, *HTTPS*, *FTP* e o *SFTP*. Já nos filtros complexos foram configurados para o protocolo *DNS* e no Comprimento total do pacote *IP* (valores menores a 256 *bytes*).

As aplicações e protocolos como *VoIP*/videoconferência/*skype*, *SMTP*, *IMAP*, *POP3*, *HTTP*, *DNS*, HTTPS bem como o comprimento total do pacote *IP* deverão ter prioridade no serviço QoS-Flux no envio e recebimento de pacotes. Os demais protocolos e aplicações como *FTP, SFTP, Torrent* e aqueles que não estejam configurados nos filtros, serão classificados como sem prioridade no serviço.

3.4 Gerenciamento de QoS no Serviço QoS-Flux

No módulo QoS-Flux, existem 5 comandos (ver Algoritmo 1) que o usuário pode utilizar para realizar o gerenciamento em uma rede SDN:

Start Este comando selecionado pelo usuário no QoS-Flux, consegue adicionar as configurações do *TC Linux* automaticamente em cada porta dos *switches OVS*, ou seja, habilita os algoritmos através da *Qdisc* (*FQ_Codel*) e Classe (*HFSC*), cálculo de largura de banda e filtros para aplicações que demandam prioridades, explicado na subseção 3.3.2.

Stop Este comando no QoS-Flux, tem como finalidade parar a execução do módulo QoS-Flux, removendo as configurações habilitadas nas portas dos *switches OVS* pelo comando *Start*.

Restart O comando *Restart* possui a função de reiniciar o módulo QoS-Flux. Este comando é bastante útil quando precisamos resolver algum problema em algum dos componentes do módulo ou somente alguma manutenção preventiva.

Status O comando *Status* tem como propósito mostrar todas as configurações executadas pelo comando *Start*, exceto aquelas relacionadas aos filtros de prioridade. Além disso, conseguimos monitorar a rede e os algoritmos de QoS aplicados.

Filter Este comando tem como objetivo mostrar as configurações de filtragem de aplicações de *Internet* por prioridade (*DNS*, *VoIP*, videoconferência, *Skype*, *IMAP*, *POP3*, *SMTP*, *HTTP* e etc.) e sem prioridade (*HTTPS* e aqueles sem classificação por protocolo ou porta). As prioridades são executadas seguindo as diretrizes do filtro padrão *u32* disponibilizado pelo *TC Linux* e explicado em maiores detalhes na seção 3.3.2.

3.5 Controlador SDN Ryu no QoS-Flux

O controlador *SDN Ryu* é baseado em componentes, ou seja, tem um conjunto de elementos pré-configurados. Esses componentes podem ser alterados, ampliados e compostos para gerar um aplicativo de controlador personalizado, tanto interno como externo, como o QoS-Flux. Qualquer linguagem de programação com suporte a linguagem *Python* pode ser utilizada para produzir um novo elemento neste controlador [78].

O QoS-Flux consegue trabalhar como um serviço externo do controlador *SDN Ryu*, realizando o gerenciamento de *QoS* dinâmico em uma rede *SDN*. Por isso, ele deve ser instalado e configurado na mesma máquina do controlador *SDN*.

A escolha desse controlador se deve ao fato do mesmo possuir boa documentação, sendo *open source* e funcional em qualquer sistemas operacional *Linux*, alinhando-se com as primitivas do serviço QoS-Flux.

3.6 Considerações Finais

Neste capítulo foi explicado sobre um serviço que consegue prover *QoS* dinâmico para diversos tipos de fluxos em uma rede *SDN*, chamado QoS-Flux. Esclarecemos sobre uma visão global do serviço QoS-Flux, seu funcionamento em uma rede *SDN*, meios de implementação, o trabalho realizado junto do controlador *SDN* e as ferramentas que são mecanismos para execução do QoS-Flux:

a) *TC Linux* - Utilizado através do comando *tc* no código do QoS-Flux, o qual consegue habilitar filtros de pacotes, troca entre os algoritmos *HFSC* e o *FQ_Codel* de forma automática.

b) *Switches OVS* - São dispositivos virtuais utilizado para demonstrar a configuração do QoS-Flux em funcionamento. Entretanto, podemos utilizar outros tipos de switches na topologia (ex: switches proprietários), sendo necessário habilitá-los no código do serviço QoS-Flux.

Além disso, foi explicado sobre os comandos de gerenciamento do serviço QoS-Flux em uma rede *SDN*, ou seja, instruções para iniciar, parar, reiniciar, monitorar e verificar os filtros habilitados.

No capítulo 4 será apresentado os experimentos e análises dos testes comparando algoritmos configurados estaticamente pela ferramenta *TC Linux* e o serviço QoS-Flux configurado dinamicamente em uma rede *SDN* virtualizada.

Capítulo 4

Experimentos

Este capítulo discute as ferramentas utilizadas nos experimentos, as configurações adotadas, as métricas analisadas e apresenta uma série de experimentos com diversas aplicações, de acordo com o tipo de fluxo (tamanho – elefante, velocidade – guepardo ou rajada – alfa). Os resultados obtidos nos experimentos serão discutidos e analisados.

4.1 Ferramentas Utilizadas

O ambiente de testes foi emulado através da ferramenta *Mininet*, enquanto que a ferramenta *Distributed Internet Traffic Generator (D-ITG)* foi utilizada para geração de tráfegos distintos. Estas ferramentas são detalhadas a seguir.

Mininet O *Mininet* é um *software* que permite agilizar a prototipação dentro de um computador tanto de redes pequenas quanto de grandes redes (*datacenters*) [79]. O *Mininet* consegue gerar redes *SDN* através de técnicas de virtualização utilizando processos e *namespaces* em rede. Além disso, esse sistema consegue ter flexibilidade, aplicabilidade, interatividade, escalabilidade, realismo e o projeto desenvolvido pode ser compartilhado com outros colaboradores.

Além disso, o *Mininet* tem a capacidade de emular diversos elementos de rede (ex.: *hosts*, *switches* de camada 2 – *OVS*, roteadores de camada 3 e links) [80] e pode funcionar em um computador através do *kernel Linux*, tendo o propósito de emular uma rede completa através do comando *mn* e seus parâmetros.

D-ITG O *software Distributed Internet Traffic Generator* (*D-ITG*) é uma ferramenta que gera tráfegos *IPv4* e *IPv6*, conseguindo reproduzir fielmente cargas de trabalho dos aplicativos conhecidos pela *Internet*. Além disso, o *D-ITG* consegue mensurar métricas de desempenho através de relatórios (ex.: taxa de transferência, atraso, *jitter* e perda

de pacotes) em nível de pacote. Essa ferramenta tem a capacidade de moldar modelos estocásticos de tamanho de pacote (*PS*) e tempo de partida (*IDT*) que reproduzem o desempenho de protocolo em nível de aplicação (ex.: *Telnet*, *VoIP - G.711, G.723, G.729, RTP, DNS* e alguns modelos de jogos online). Na camada de transporte, o *D-ITG* suporta tanto o protocolo *TCP* quanto o *UDP*. Além disso, essa ferramenta também consegue simular tráfegos do tipo *Internet Control Message Protocol* (*ICMP*).

De acordo com [81], na arquitetura do *D-ITG* os recursos são gerados pelo comando *ITGSend* e pelo *ITGRecv*. O *ITGSend* se comunica reproduzindo tráfegos para o *ITGRecv* realizando uma comunicação do tipo cliente para servidor. O *ITGSend* pode encaminhar diversos fluxos de tráfego paralelo para diversas instâncias do *ITGRecv* dependendo da configuração do usuário para geração do tráfego. O comando *ITGLog* é utilizado para coleta de informações dos testes de desempenho e o *ITGDec* para analisar esses dados coletados.

4.2 Ambiente Experimental

O QoS-Flux foi implementado em um ambiente simulado utilizando um servidor *Dell Optiplex* 7050 *Intel Core i7* com 8 cores, 16 *GB RAM* com sistema operacional Microsoft Windows 10 instalado. Todo ambiente simulado em *SDN* utiliza um computador virtualizado (*VM*) no *software VirtualBox* com configurações de *hardware* utilizando 4 Cores, 8 *GB RAM*, uma *interface* de rede *Gigabit* e um sistema operacional *Linux Ubuntu* 16.04.

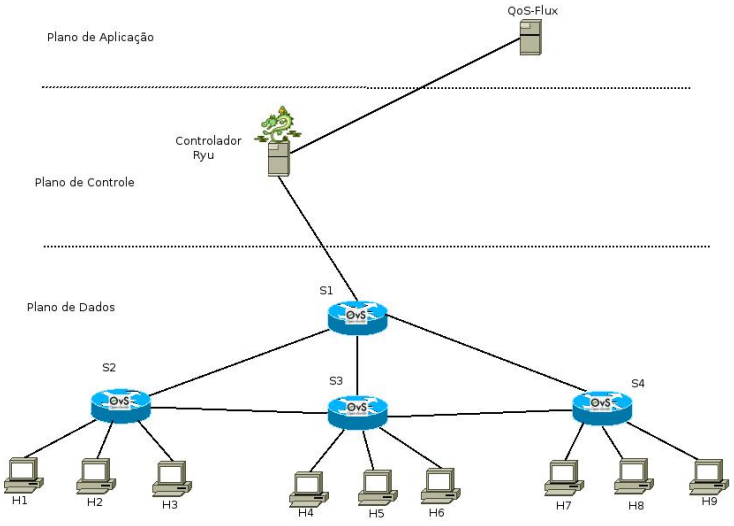

Figura 4.1: Topologia da Rede Utilizada para o Serviço QoS-Flux.

O controlador *SDN Ryu* (plano de controle) e o serviço QoS-Flux (plano de aplicação) serão virtualizados na mesma VM, porém, ficam separados no *Kernel* por processos distintos. Além disso, a ferramenta *Mininet* deve gerar uma topologia de rede através de

namespaces (plano de dados) em outro processo distinto no sistema operacional virtualizado.

A topologia que foi gerada pela ferramenta *Mininet* no nosso trabalho, é o modelo *Topo Tree* (conforme a Figura 4.1), contendo 9 *hosts* e 4 *switches OVS* com a largura de banda configurada em $1 GBit/s$.

No ambiente experimental, 3 *hosts* foram distribuídos para cada *switch OVS*. Com relação aos *switches* (veja Figura 4.1), o S1 será o *switch* de borda (agregação) enquanto que o S2, S3 e S4 são *switches* de acesso. A porta 4 de cada *switch* de acesso será usada para a conexão *uplink* para o *switch* de borda (agregação).

Para que as funções do *switch* funcionem no controlador *Ryu*, é necessário acionar o módulo *simple_switch_13.py* no próprio controlador [57]. Devemos também determinar a versão do *OpenFlow* a ser usada em cada *switch OVS* para a versão 1.3 e configurar a porta 6632 para que o banco de dados dos *switches OVS* (*OVSDB*) possa se comunicar com o controlador.

O módulo *TC Linux* já estava instalado no sistema operacional *Linux* quando foi realizado a preparação para o ambiente de testes. Neste caso, escolhemos 3 algoritmos para serem testados e comparados com configurações estáticas no nosso trabalho, ou seja, os algoritmos *HTB*, *HFSC* e o *FQ_Codel*. Os parâmetros de configuração padrão de cada algoritmo estático neste trabalho foi baseado nos exemplos utilizados para redes cabeadas [72, 77, 82], como observado nas explicações a seguir.

1. Algoritmo HTB: Utilizamos as configurações básicas para todas as portas dos *switches OVS* no *Mininet*, com exceção do *switch* S1 de borda (agregação). Foram geradas duas classes, a *Basic*, que são fluxos que necessitam de muita largura de banda e a *Default* que não precisa de tanta largura de banda. Para as duas classes foi adicionada um valor limite máximo na taxa de transferência de dados na classe pai (parent 1:1) de 1000 Mbit/s. Um resumo das configurações podem ser visualizadas na tabela 4.1 e os parâmetros utilizados no final do trabalho (Anexo 4).

2. Algoritmo HFSC: Utilizamos as configurações básicas para todas as portas dos *switches OVS* no *Mininet*. Foram geradas duas classes, a *Basic*, que são fluxos que necessitam de muita largura de banda e a *Default* que não precisa de tanta largura de banda. Para as duas classes foi adicionada a limitação máxima na taxa de transferência de dados com curva de serviço (*sc*) na classe pai (*parent 1:1*) de 1000 *Mbit/s*. Um resumo das configurações podem ser visualizadas na tabela 4.1 e os parâmetros utilizados no final do trabalho (Anexo 4).

3. Algoritmo FQ_Codel: Utilizamos as configurações básicas (default) para todas as portas dos *switches OVS*, gerado pelo *Mininet* a partir do módulo *TC Linux*. Todos

os valores são configurados automaticamente, gerando todos os números padrões. Um resumo das configurações podem ser visualizadas na tabela 4.1 e os parâmetros utilizados no final do trabalho (Anexo 4).

Tabela 4.1: Configurações dos Algoritmos de QoS Estáticos

Algoritmo	Característica	Configurações	Observação
HTB	Controle de largura de banda por compartilhamento de links simulando em um link físico vários links com taxas menores.	Classe Basic (link com taxa de transferência de 700 Mb/s mínimo e 800 Mb/s máximo); Classe Default (link com taxa de transferência de 900 Mb/s mínimo e 950 Mb/s máximo);	A classe Basic possui prioridade 1 (um) e a classe Default prioridade 0 (zero), portanto, os pacotes devem ser enviados para classe Basic. A classe Default será acionada quando estiver sobrando largura de banda na classe Basic.
HFSC	Compartilhar largura de banda em tempo virtual e/ou alocação de níveis de atrasos mínimos para todas as classes em tempo real através do cálculo da curva de serviço linear.	Classe Basic_0 (link com taxa de transferência de 700 Mb/s mínimo e 800 Mb/s máximo); Classe Basic_1 (link com taxa de transferência de 900 Mb/s mínimo e 950 Mb/s máximo); Nas classes foram configuradas curva de serviço com 1500 bytes, 53 milissegundos (Basic) e 30 milissegundos (Default).	Os pacotes são enviados para classe Basic_0 e quando estiver com sobra de largura de banda é enviado para classe Basic_1. A classe Basic_0 sempre deve ter prioridade sobre a classe Basic_1.
FQ_Codel	Modelo estocástico que ordena os pacotes de entrada em múltiplos fluxos, controlando o atraso jitter e a perda de pacotes.	Foi definido o parâmetro limit com o valor de 10240 pacotes e na métrica flows o valor de 1024. Além disso, foram inseridos os valores de target de 5 milissegundos, interval de 100 milissegundos, quantum de 1514 bytes e habilitado com notificação ecn.	Não possui prioridade.

Vale destacar novamente que nos experimentos cada um destes algoritmos foram analisados separadamente de forma estática e também o QoS-Flux que utiliza os algoritmos *HFSC* e *FQ_Codel* dinamicamente.

4.3 Métricas

Escolhemos para cada tipo de fluxo (elefante, guepardo ou alfa) um tipo de aplicação distinta no protocolo *TCP*, seguindo o padrão de estudo das principais aplicações de internet [1]. Para fluxos elefantes será uma carga de trabalho simulando um armazenamento em nuvem (Dropbox) [83]; nos fluxos guepardos será uma carga de trabalho simulando uma aplicação de *backup* de dados [84]; e nos fluxos alfas uma aplicação de *streaming* em alta definição gerando rajadas [85]. Para maiores detalhes, observar as explicações detalhadas nas cargas de trabalho na seção 4.4.

Para efeitos de comparação, a análise de desempenho será efetuada com duas baterias de testes. Na primeira e na segunda bateria de teste utilizamos períodos contínuos na

ferramenta *D-ITG*, sendo efetuado uma pausa a cada 1 minuto, 15 minutos e 30 minutos para coleta das informações e análise dos dados em cada período individualmente.

Nas duas baterias de testes, será observado a evolução de cada métrica de *QoS* comparando os algoritmos configurados estaticamente e o QoS-Flux configurado dinamicamente. Isto possibilita um comparativo entre todas as situações com base nas métricas padrão para análise de desempenho em *QoS*: atraso, *jitter*, largura de banda e perda de pacotes.

a) Atraso (*Delay*): também conhecido como latência, é a quantidade de tempo gasta para um pacote ir da origem até o destino, passando por *switches*, roteadores, enlaces ou meios físicos. São estabelecidos três tipos de atraso em redes: atraso de propagação, atraso de serialização e atraso no manuseio. Normalmente, a rede *IP* divide recursos de rede através de tecnologias de multiplexação estatística, onde o atraso pode ser relativo às circunstâncias de carga da rede. Por isso, devemos reduzir o tempo de atraso para evitar problemas de lentidão nas diversas aplicações de *Internet*.

b) Largura de banda (*Bandwidth*): a rede *IP* original não consegue atingir uma largura de banda apropriada para taxas de transmissão, principalmente em aplicações de tempo real (ex: *VoIP*). Porém, aplicações pesadas que funcionam na rede junto com o *VoIP* podem provocar lentidão ou atrasos na transmissão do mesmo. Além disso, para aplicações que dependem de alta largura de banda, como é o caso de aplicações não interativas (ex.: banco de dados e *backup*), baixas taxas de transferências podem gerar travamentos ou até reinicio dessas aplicações.

c) *Jitter*: é definido como a variação do tempo de atraso. Caso o valor de *jitter* seja muito alto, alguns pacotes poderão ser rejeitados por atraso, gerando interrupções na conexão, por exemplo. Quando esse problema ocorre em aplicações sensíveis na rede (ex: aplicações de *streaming*), pode ser necessário um valor grande de *buffer* de dados para compensar.

d) Perda de pacotes (*Packet Loss*): esta métrica pode ser considerada comum e esperada em redes de computadores. Diversos protocolos e aplicações utilizam a perda de pacotes para analisar a condição da rede e diminuir o número de pacotes que estão sendo encaminhados. A perda de pacotes em aplicações diversas podem gerar problemas de congestionamento na rede.

Tabela 4.2: Requisitos Mínimos de QoS para Aplicações de Streaming [4, 5]

Metricas	Streaming on-demand de Alta Definição
Atraso	<=100 ms
Largura de Banda	>=3 MBit/s
Jitter	<=50 ms
Perda de Pacotes	<=0.05%

Tabela 4.3: Requisitos Mínimos de QoS para Transferência de Dados [4, 5]

Métricas	Backup e Dropbox
Atraso	<=500 ms
Largura de Banda	>=3 MBit/s
Jitter	<=500 ms
Perda de Pacotes	0%

Tabela 4.4: Requisitos Mínimos de QoS para Todas as Aplicações [4, 5]

Métricas	Transferência de dados e Streaming de Alta Definição
Atraso	>=100 ms
Largura de Banda	>=3 MBit/s
Jitter	>=50 ms
Perda de Pacotes	0%

Nesse trabalho foi utilizado como padrão as métricas definidas pela *ITU-T G. Rec. 1010* [4] e *ITU-T G. Rec. 1050* [5] para *QoS*, tanto para *streaming* de alta definição (fluxos alfa) na Tabela 4.2 bem como para transferência de dados (fluxos elefantes e guepardos) na Tabela 4.3.

Na Tabela 4.4 é possível observar os valores adotados como parâmetros de *QoS* para os testes utilizando todas aplicações ao mesmo tempo realizadas para este trabalho. Os valores definidos nesta tabela foram obtidos elegendo-se os menores valores das tabelas anteriores, baseado no fato de que se tais parâmetros forem alcançados, será possível estar de acordo com ambas as métricas definidas pela *ITU-T* para *streaming* e transferência de dados.

4.4 Cargas de Trabalho

O *software D-ITG* será responsável pela geração de carga de trabalho para os testes a partir de scripts de execução (Anexo 2). A análise de desempenho deve ser realizada utilizando o conceito de cliente e servidor.

Será divido em 2 baterias de testes, sendo a primeira mostrado na Tabela 4.5, com fluxos individuais, ou seja, somente elefante, guepardo ou alfa. Além disso, o *host* h1 será o servidor e os demais os hosts clientes. Na segunda bateria, mostrado na Tabela 4.6, será realizado com todos os fluxos funcionando ao mesmo tempo, sendo o *host* h1 o servidor e cada grupo de *hosts* clientes serão separados por tipo de fluxo: h2, h3, h4 (fluxo elefante – aplicação *Dropbox*); h5, h6, h7 (fluxo guepardo – aplicação de *backup* de dados); h8 e h9 (fluxo alfa – *streaming* de alta definição).

Tabela 4.5: Carga de Trabalho na Bateria de Teste 1

Características	Dropbox	Backup	Streaming HD
Clientes	h2 até h9	h2 até h9	h2 até h9
Servidores	h1	h1	h1
Largura de Banda	1 Gbit/s	1 Gbit/s	1 Gbit/s
Tempo	1, 15, 30 minutos	1, 15, 30 minutos	1, 15, 30 minutos
Ferramenta	D-ITG	D-ITG	D-ITG
Porta	17500	82	443
Protocolo	TCP	TCP	TCP

Tabela 4.6: Carga de Trabalho na Bateria de Teste 2

Características	Dropbox	Backup	Streaming HD
Clientes	h2, h3, h4	h5, h6, h7	h8, h9
Servidores	h1	h1	h1
Largura de Banda	1 Gbit/s	1 Gbit/s	1 Gbit/s
Tempo	1, 15, 30, 60 minutos	1, 15, 30, 60 minutos	1, 15, 30, 60 minutos
Ferramenta	D-ITG	D-ITG	D-ITG
Porta	17500	82	443
Protocolo	TCP	TCP	TCP

Tabela 4.7: Total de Dados Transmitidos (bytes)

Fluxos	HTB	HFSC	FQ_Codel	Serviço QoS-Flux
Elefante	4.2 GB	8.4 GB	9 GB	12.4 GB
Guepardo	24 GB	26 GB	25.5 GB	42.5 GB
Alfa	4.1 GB	14.3 GB	12 GB	17 GB
Todos os Fluxos	33.1 GB	48 GB	47 GB	73 GB

a) Elefantes - Nesse trabalho foi considerado os testes de aplicações do tipo *Dropbox* na porta 17500 utilizando o protocolo *TCP*. Escolhemos esse tipo de tráfego com simulações de *upload* de arquivos, ou seja, realizando transferências de dados considerando um dia inteiro atípico com 3 usuários. Esta aplicação pode ser considerada como característico de fluxos elefantes.

b) Guepardo - No trabalho em questão foi avaliado para os testes aplícações do tipo *backup* na porta 82 utilizando o protocolo *TCP*. Para realizar o processo executado por aplicações de *backup* é necessário alta largura de banda para transferência de grande massa de dados em curto período de tempo, característico de fluxos guepardos.

c) Alfa - Nesse trabalho foi selecionado para os testes aplicações do tipo *streaming* em *High Definition* (*HD*) na porta 443 utilizando o protocolo *TCP*. Esta opção de aplicação foi escolhida por esse tráfego ser bastante utilizado hoje em dia, sendo um tipo de aplicação com altas incidências de rajadas de tráfego (fluxos intermitentes).

Na Tabela 4.7, observamos o total de dados transmitidos para todos os cenários de testes nos fluxos elefante (*Dropbox*), guepardo (*backup*), alfa (*streaming HD*) e todos os fluxos, respectivamente, conforme cada algoritmo de *QoS* habilitado estaticamente e no serviço QoS-Flux habilitado dinamicamente.

4.5 Resultados e Análises

Esta seção apresenta alguns resultados preliminares obtidos em simulações realizadas com os algoritmos estudados e o serviço QoS-Flux. Como os testes de desempenho foram realizados para contemplar toda rede, foi utilizado a média aritmética e o desvio padrão para calcular todos os resultados de acordo com cada parâmetro de *QoS*. Além disso, o nível de confiança calculado foi de 95% para todos os cenários de teste.

$$\bar{x} = \frac{1}{N} \sum_{i=1}^{N} x_i \tag{4.1}$$

A média aritmética (4.1) é definido como igual ao somatório dos valores numéricos por $\sum_{i=1}^{N} x_i$ multiplicado pelo valor 1 divido pelo número total de análises N, gerando o resultado de $\bar{x}$.

$$\sigma = \sqrt{\frac{\sum_{i=1}^{n} (x_i - \bar{x})^2}{n-1}} \tag{4.2}$$

O desvio padrão (4.2) pode ser calculado pelo valor da raiz quadrada do $\sum_{i=1}^{n}$, o qual o valor de x_i é considerado os números analisados nos elementos da amostra equivalente a x_1, $x_2 ... x_n$. No $\bar{x}$ será analisado o resultado da média aritmética e n é o número total de observações na amostra, gerando o valor final de σ.

4.5.1 Atraso

Na Figura 4.2 observamos o tráfego gerado para aplicação de *Dropbox* na métrica de atraso em milissegundos (ms) gerando fluxos elefantes. Nos testes com fluxos elefantes utilizando o algoritmo *HTB*, foi visualizado o valor médio de 880 ms (1 minuto) e 730 ms para 15 minutos e 30 minutos, respectivamente; com relação aos testes com algoritmo *HFSC*, foram gerados os valores médios de 450 ms (1 minuto), 420 ms (15 minutos) e 840 ms (30 minutos); na avaliação com algoritmo *FQ_Codel*, gerou os valores médios de 380 ms (1 minuto), 460 (15 minutos) e 350 ms (30 minutos). Quando habilitado o serviço QoS-Flux, os valores médios gerados foram de 220 ms (1 minuto), 270 ms (15 minutos) e novamente 220 ms (30 minutos).

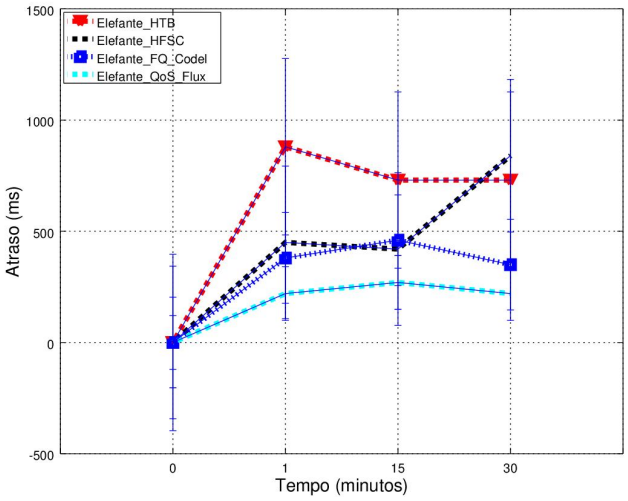

Figura 4.2: Atraso em Fluxos Elefante.

Quando foi utilizado o algoritmo *HTB*, com relação a todos intervalos de tempo (1, 15 e 30 minutos), o sistema não consegue alcançar o mínimo exigido para métricas de atraso em *QoS* para transferência de dados de 500 ms (ver Tabela 4.3). No algoritmo *HFSC*, quando chega no período a partir de 30 minutos, o requisito mínimo não é atingido. O algoritmo *FQ_Codel* e o serviço QoS-Flux, com relação a todos os períodos de tempo, conseguiram seguir os requisitos mínimos exigidos para métrica de atraso durante todo o período.

O desvio padrão para métrica de atraso em fluxos elefantes gerou o valor de 396 ms para os testes realizados utilizando o algoritmo *HTB*, 343 ms no algoritmo *HFSC*, 203 ms no algoritmo *FQ_Codel* e 120 ms no serviço QoS-Flux.

Na Figura 4.3, foi verificado o gráfico para aplicações de *backup* na métrica de atraso definido em milissegundos (ms) gerando fluxos guepardos. Quando utilizado o algoritmo *HTB*, foram gerados os valores médios de 490 ms (1 minuto), 1650 ms para 15 minutos e permanecendo neste valor no período de 30 minutos. No algoritmo *HFSC*, foi percebido os valores médios de 734 ms para 1 minuto, 700 ms para 15 minutos e 1000 ms no período de 30 minutos. Quando é empregado o algoritmo *FQ_Codel*, foram gerados os valores médios de 400 ms (1 minuto), 420 ms (15 minutos) e 370 ms (30 minutos). No momento que é habilitado o serviço QoS-Flux, foi constatado os valores médios de 420 ms para 1 minuto, 400 ms para 15 minutos e 410 ms para 30 minutos.

Foi observado que quando habilitamos o algoritmo *HTB*, com relação ao intervalo entre 1 até 30 minutos, o mesmo não consegue alcançar o mínimo exigido para métricas de atraso em *QoS* com relação a transferência de dados (ver Tabela 4.3). No algoritmo *HFSC*, com referência a todos os intervalos de tempo, o requisito mínimo de *QoS* não

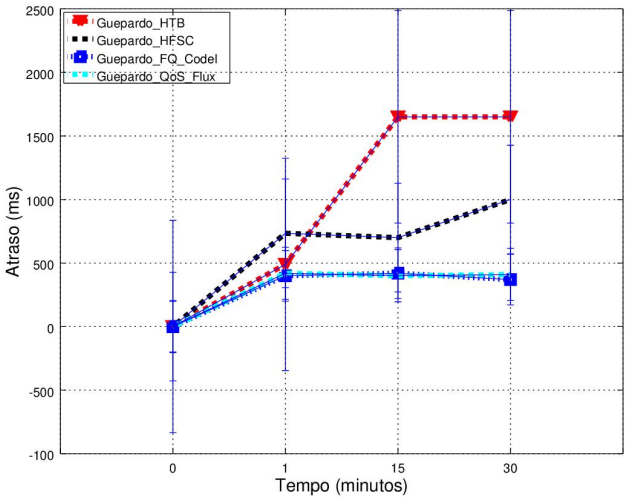

Figura 4.3: Atraso em Fluxos do Tipo Guepardo.

é atingido. Somente o algoritmo *FQ_Codel* e o serviço QoS-Flux em todos os períodos, conseguiram seguir os requisitos mínimos exigidos para métrica de atraso em transferência de dados.

O desvio padrão para métrica de atraso em fluxos guepardos gerou o valor de 835 ms para os testes realizados utilizando o algoritmo *HTB*; 427 ms no algoritmo *HFSC*; 199 ms no algoritmo *FQ_Codel* e 205 ms no serviço QoS-Flux.

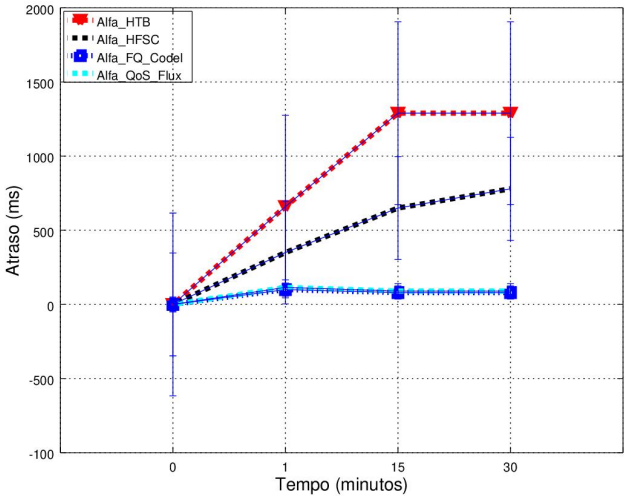

Figura 4.4: Atraso em Fluxos do Tipo Alfa.

Na Figura 4.4 é analisado a métrica de atraso em aplicações de *streaming em HD* determinado por milissegundos (ms) gerando fluxos do tipo alfa. No momento em que utilizamos o algoritmo *HTB*, foi observado os valores médios de 660 ms em 1 minuto, 1290

ms para 15 minutos e permanecendo quando atinge 30 minutos. No algoritmo *HFSC*, percebe-se que os valores médios foram de 350 ms (1 minuto), 650 ms (15 minutos) e 780 ms (30 minutos). Quando é utilizado o algoritmo *FQ_Codel*, foi descoberto os valores médios de 100 ms para 1 minuto e 80 ms tanto no período de 15 minutos como também em 30 minutos. Quando habilita o serviço QoS-Flux, percebe-se os valores médios de 115 ms para 1 minuto e 90 ms para 15 minutos permanecendo até o período de 30 minutos com este valor.

Constata-se que quando são habilitados os algoritmos *HTB* e o *HFSC*, relacionando os três períodos de tempo, não conseguem alcançar os requisitos mínimos exigidos para *QoS* em métricas de atraso para *streaming HD* (ver Tabela 4.2). Com relação aos algoritmos *FQ_Codel* e o módulo QoS-Flux, percebe-se que os dois conseguem atingir os requisitos mínimos de *QoS* para *streaming HD* na métrica de atraso.

Os valores de desvio padrão para aplicações de *streaming HD* na métrica de atraso utilizando o algoritmo *HTB*, foram de 616 ms. No algoritmo *HFSC* gerou valor de 347 ms, com relação ao algoritmo *FQ_Codel* gerou o valor de 44 ms e quando habilitado com o serviço QoS-Flux gerou o valor de 50 ms.

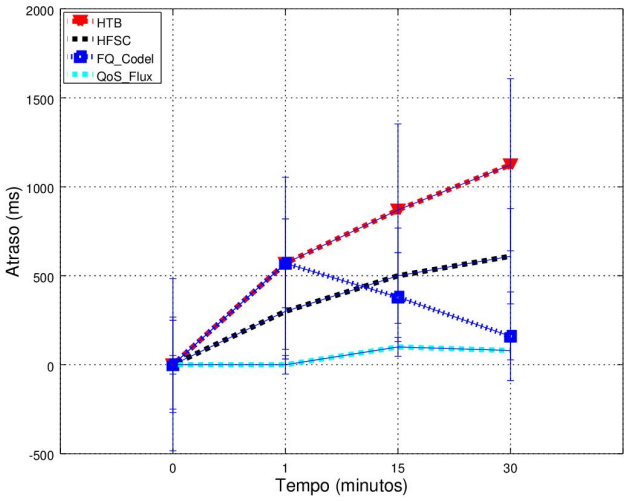

Figura 4.5: Atraso Utilizando Todos os Fluxos.

Na Figura 4.5 é observado a métrica de atraso definido em milissegundos (ms) quando utilizamos todos os fluxos ao mesmo tempo. No momento em que usamos o algoritmo *HTB* na rede, observamos o valor médio de 570 ms (1 minuto), 870 ms (15 minutos) e 1124 ms (30 minutos). No algoritmo *HFSC*, percebemos os valores de 300 ms (1 minuto), 500 ms (15 minutos), 610 ms (30 minutos). Com o algoritmo *FQ_Codel*, conseguimos verificar os valores de 570 ms para 1 minuto, 380 ms para 15 minutos, 160 ms para 30

minutos. Quando habilitamos o serviço QoS-Flux, constatamos os valores de 0 ms para 1 minuto, 100 ms para 15 minutos e 80 ms para 30 minutos.

Percebe-se que nos testes efetuados com os algoritmos *HTB*, *HFSC* e o FQ_Codel não conseguiram alcançar os requisitos mínimos de *QoS* utilizando todas as aplicações na rede com relação a métrica de atraso (ver Tabela 4.4). Diferentemente quando acionamos o serviço QoS-Flux, o qual obteve valores médios em todos os períodos de tempo que satisfazem os requisitos mínimos de *QoS* para todas as aplicações na rede.

No desvio padrão calculado para todas aplicações na rede, quando o algoritmo *HTB* está funcionando, obteve valores de 483 ms. No algoritmo *HFSC*, obtemos os valores médio de 267 ms, com relação ao algoritmo *FQ_Codel* temos os valores de 249 ms e no serviço QoS-Flux os valores de 5 ms.

4.5.2 *Jitter*

Na Figura 4.6 é analisado o parâmetro *jitter* determinado por milissegundos (ms) para fluxos do tipo elefante. Nos testes utilizando o algoritmo *HTB*, foram gerados valores médios de 50 ms (1 minuto), 20 ms (15 minutos) e permanecendo no período de 30 minutos. Quando usamos os algoritmos *HFSC* ou *FQ_Codel*, percebe-se os valores médios de 10 ms para 1 minuto, mantendo-se nos períodos de 15 e 30 minutos, respectivamente. Quando é habilitado o serviço QoS-Flux, foi constatado o valor médio de 0 ms em todos os períodos de tempo.

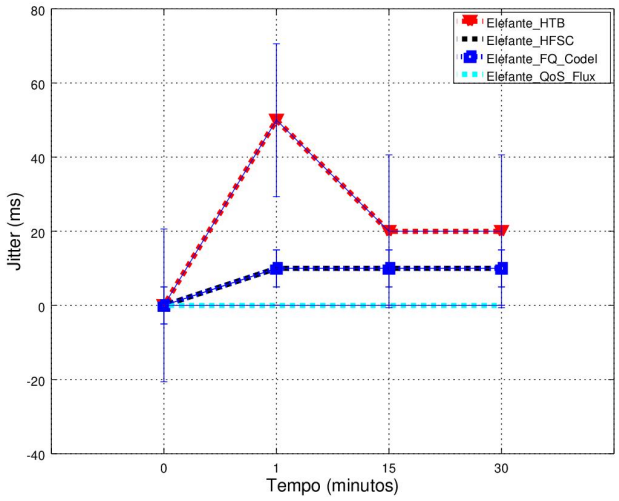

Figura 4.6: Jitter em Fluxos do Tipo Elefante.

No comparativo entre os algoritmos e o serviço QoS-Flux, podemos averiguar que todos conseguiram seguir os requisitos mínimos para *QoS* na métrica de *jitter* para transferência

de dados (ver Tabela 4.3). Porém, percebe-se que o serviço QoS-Flux conseguiu atingir melhores valores, comparado com os demais para fluxos elefantes.

Os valores de desvio padrão na métrica *jitter* para aplicações que geram fluxos elefantes, obtemos para o algoritmo *HTB* o valor de 21 ms. No algoritmo *HFSC* e *FQ_Codel* o valor de 10 ms e no serviço QoS-Flux, foi alcançado o valor de 0 ms.

Na figura 4.7 é observado o parâmetro *jitter* definido em milissegundos (ms) para fluxos do tipo guepardo. Com relação aos testes efetuados com o algoritmo *HTB*, foi observado os valores médios de 10 ms para 1 minuto e 20 ms para os demais períodos de tempo (15 e 30 minutos). Quando é utilizado o algoritmo *HFSC*, foi constatado os valores de 0 ms para 1 minuto, aumentando para 10 ms nos períodos de tempo de 15 minutos e permanecendo nos 30 minutos. No algoritmo *FQ_Codel* é observado os valores médios de 0 ms no período de 1 minuto e 15 minutos, subindo para 10 ms quando alcança os 30 minutos. Quando utiliza-se o serviço QoS-Flux, os valores gerados em todos os períodos de tempo, sempre ficaram em torno de 0 ms.

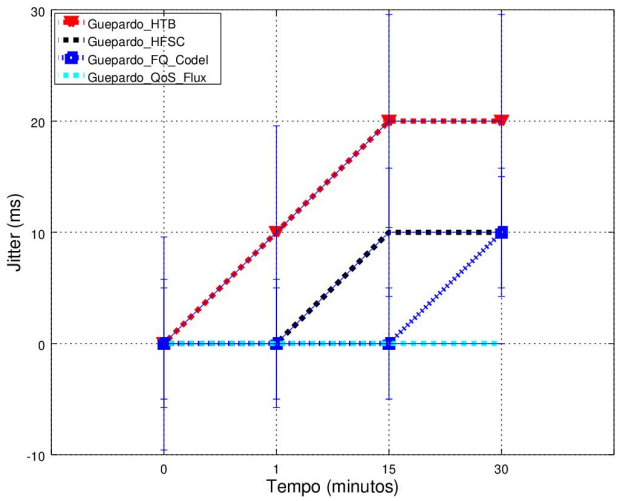

Figura 4.7: Jitter em Fluxos do Tipo Guepardo.

Foi observado que os resultados gerados pelos algoritmos e o serviço QoS-Flux ficaram no padrão mínimo exigido para *QoS* na métrica de *jitter* em transferência de dados (ver Tabela 4.3). Além disso, percebe-se que o serviço QoS-Flux, novamente, consegue valores médios melhores em comparação com os algoritmos.

Com relação ao desvio padrão na métrica *jitter* para aplicações que geram fluxos guepardos, obtemos para o algoritmo *HTB* o valor de 22 ms, no algoritmo *HFSC* e *FQ_Codel* foi gerado o valor de 15 ms. No serviço QoS-Flux foi gerado o valor de 0 ms.

Na figura 4.8 é examinado o parâmetro *jitter* definido em milissegundos (ms) para fluxos do tipo alfa. Analisando os testes efetuados com o algoritmo *HTB*, verifica-se os

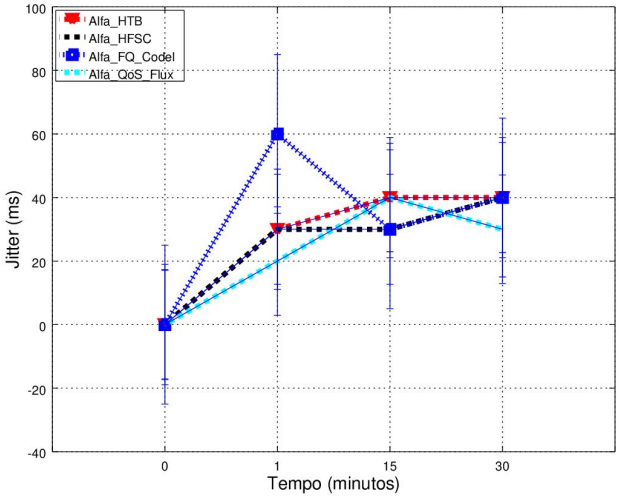

Figura 4.8: Jitter em Fluxos do Tipo Alfa.

valores médios de 30 ms para 1 minuto e 40 ms para os demais períodos de tempo (15 e 30 minutos). Quando é utilizado o algoritmo *HFSC*, foi constatado os valores de 30 ms em 1 minuto e nos 15 minutos, aumentando para 40 ms no período de tempo de 30 minutos. No algoritmo *FQ_Codel*, é observado os valores médios de 60 ms no período de 1 minuto, reduzindo para 30 ms no período de 15 minutos e aumentando para 40 ms em 30 minutos. Quando habilitamos o serviço QoS-Flux, o valor gerado foi de 20 ms para 1 minuto, 40 ms para 15 minutos e reduzindo para o valor de 30 ms quando alcança 30 minutos.

Nos resultados gerados, o algoritmo FQ_Codel com relação a *QoS* na métrica de *jitter* em *streaming HD* (ver Tabela 4.2), conseguiu alcançar o mínimo exigido somente a partir de 15 minutos. Diferentemente dos demais algoritmos e o serviço QoS-Flux que conseguiu atingir os requisitos mínimos em todos os períodos de tempo nos testes realizados.

Entretanto, apesar do serviço QoS-Flux alcançar o mínimo exigido para *QoS*, ocorreu algumas variações nos valores com relação aos outros resultados com fluxos elefantes e guepardos na métrica de *jitter*. O algoritmo *FQ_Codel*, responsável por reduzir estes efeitos no tráfego de fluxos, não conseguiu amenizar totalmente estas variações. Nesta situação, caso fosse uma aplicação em tempo real (ex: VoIP), poderia ser afetado. Porém, os filtros do serviço QoS-Flux entraria em ação, dando prioridade para esta aplicação com relação aos demais, reduzindo assim os problemas de *jitter* na transmissão entre o remetente e o destinatário.

Com relação ao desvio padrão na métrica *jitter* para aplicações que geram fluxos alfa, foi obtido para o algoritmo *HTB* o valor de 18 ms, *HFSC* o valor de 17 ms, no algoritmo *FQ_Codel* foi gerado o valor de 25 ms. No serviço QoS-Flux foi gerado o valor de 17 ms.

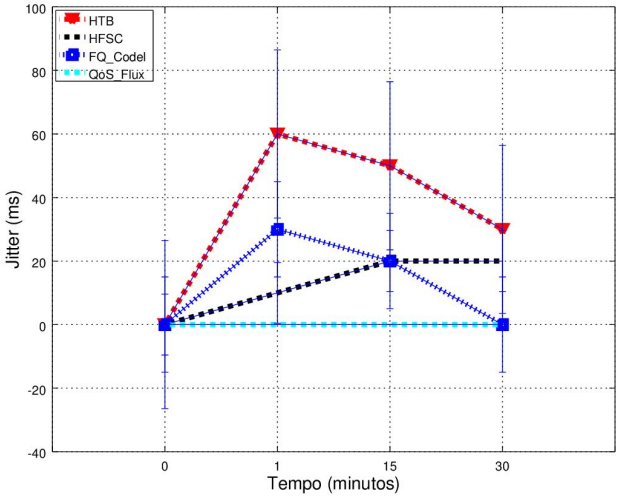

Figura 4.9: Jitter Utilizando Todos os Fluxos.

Na Figura 4.9 é examinado o parâmetro *jitter* definido em milissegundos (ms) para o teste utilizando todos os fluxos. Examinando os testes realizados com o algoritmo *HTB*, foi analisado os valores médios de 60 ms para 1 minuto, 50 ms para 15 minutos e 30 ms para 30 minutos. Quando utiliza-se o algoritmo *HFSC*, foi observado os valores de 10 ms para 1 minuto, aumentando para 20 ms nos períodos de tempo de 15 minutos e permanecendo nos 30 minutos finais. No algoritmo *FQ_Codel*, é observado os valores médios de 30 ms no período de 1 minuto, reduzindo para 20 ms no período de 15 minutos, reduzindo para 0 ms no período de 30 minutos. Quando é utilizado o serviço QoS-Flux, o valor médio gerado foi de 0 ms para todos os períodos de tempo.

Quando é verificado os resultados gerados na métrica de *jitter* para o algoritmo *HTB* com todos os fluxos (ver Tabela 4.4), conseguiu alcançar o mínimo exigido somente a partir de 15 minutos. No comparativo com os demais algoritmos e o serviço QoS-Flux, todos conseguiram obter os requisitos mínimos em todos os períodos de tempo nos testes realizados. Além disso, percebe-se que o serviço QoS-Flux consegue se sobressair em comparação com os algoritmos, ou seja, atingiu valores de 0 ms em todos os períodos de tempo.

Com relação ao desvio padrão na métrica jitter para todas as aplicações gerando vários tipos de fluxos, obtemos para o algoritmo *HTB* o valor de 18 ms, no algoritmo *HFSC* foi gerado o valor de 5 ms e no algoritmo *FQ_Codel* temos o valor gerado de 9 ms. No serviço QoS-Flux foi gerado o valor de 0 ms.

4.5.3 Largura de Banda

Na figura 4.10 é verificado o parâmetro de largura de banda definido em *Megabit* por segundo (MB/s) para o teste com fluxos do tipo elefante. Pela análise nos testes realizados com o algoritmo *HTB*, foi verificado os valores médios de 25 MB/s para 1 minuto, 53 MB/s para 15 minutos e permanecendo com mesmo valor para 30 minutos. Quando foi utilizado o algoritmo *HFSC*, percebe-se os valores médios de 117 MB/s em 1 minuto, 125 MB/s no período de tempo de 15 minutos e reduzindo este valor para 88 MB/s em 30 minutos. No algoritmo *FQ_Codel*, é observado os valores médios de 163 MB/s no período de 1 minuto, 158 MB/s no período de 15 minutos e 87 MB/s no período de 30 minutos. Quando é habilitado o serviço QoS-Flux, o valor médio gerado foi de 119 MB/s para 1 minuto, 113 MB/s para 15 minutos e 112 MB/s no período de tempo de 30 minutos.

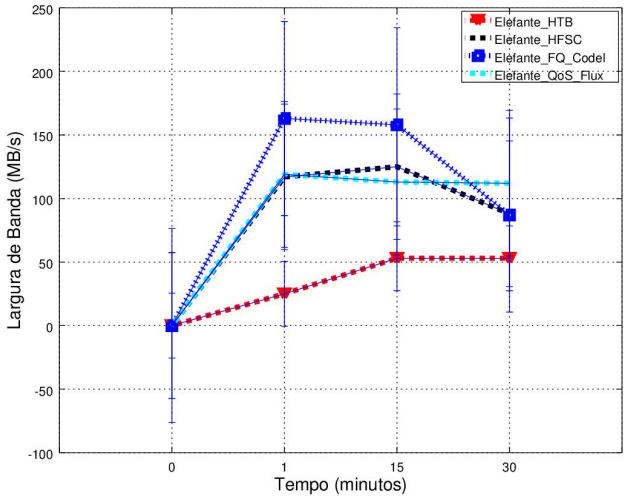

Figura 4.10: Largura de Banda em Fluxos do Tipo Elefante.

Os resultados gerados utilizando os algoritmos e o serviço QoS-Flux relacionado a métrica de largura de banda em fluxos elefantes (ver Tabela 4.3), todos conseguiram alcançar o mínimo exigido em *QoS* para esta métrica nos períodos de tempo. Porém, percebe-se que o algoritmo *HTB* gerou valores médios de largura de banda muito inferiores comparado com os demais algoritmos e o serviço QoS-Flux. Diferentemente do algoritmo *FQ_Codel* que obteve valores médios bastante expressivos, principalmente no período entre 0 até 15 minutos. Com relação ao serviço QoS-Flux, podemos notar que no período partir de 30 minutos consegue se destacar dos demais, alcançando valor médio de 112 MB/s.

Com relação ao desvio padrão na métrica de largura de banda para aplicações gerando fluxos elefantes, obtemos para o algoritmo *HTB* o valor de 25 MB/s, no algoritmo *HFSC*

foi gerado o valor de 57 MB/s e no algoritmo *FQ_Codel* temos o valor gerado de 76 MB/s. No serviço QoS-Flux foi gerado o valor de 57 MB/s.

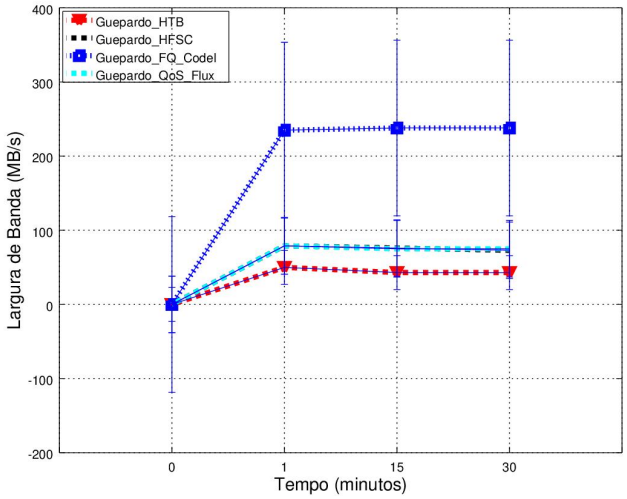

Figura 4.11: Largura de Banda em Fluxos do Tipo Guepardo.

Na Figura 4.11 é analisado o parâmetro de largura de banda definido em *Megabit* por segundo (MB/s) para o teste com fluxos do tipo guepardo. Considerando os testes realizados com o algoritmo *HTB*, percebe-se que foram gerados valores médios de 50 MB/s para 1 minuto, 46 MB/s para 15 minutos não alterando este valor para 30 minutos. Quando habilitado o algoritmo *HFSC*, foi constatado os valores médios de 79 MB/s em 1 minuto, 76 MB/s no período de tempo de 15 minutos e reduzindo este valor para 73 MB/s em 30 minutos. No algoritmo *FQ_Codel*, é observado os valores médios de 235 MB/s no período de 1 minuto, aumentando para 238 MB/s no período de 15 minutos e permanecendo no período de 30 minutos. Quando é habilitado o serviço QoS-Flux, o valor médio gerado foi de 79 MB/s para 1 minuto, 78 MB/s para 15 minutos e continuando com este valor no período de tempo de 30 minutos.

Os resultados gerados utilizando os algoritmos e o serviço QoS-Flux relacionado a métrica de largura de banda em fluxos elefantes (ver Tabela 4.3), todos conseguiram alcançar o mínimo exigido em *QoS* para esta métrica nos períodos de tempo. Porém, percebemos que o algoritmo *HTB*, novamente, gerou valores médios de largura de banda bem abaixo comparado com os outros algoritmos e o serviço QoS-Flux. Além disso, mais uma vez, foi constatado que o algoritmo *FQ_Codel* obteve valores médios bastante alto no comparativo.

No desvio padrão para métrica de largura de banda em aplicações que geram fluxos elefantes, foi obtido para o algoritmo *HTB* o valor de 22 MB/s, no algoritmo *HFSC* foi

gerado o valor de 38 MB/s e no algoritmo *FQ_Codel* temos o valor gerado de 118 MB/s. No serviço QoS-Flux foi gerado o valor de 38 MB/s.

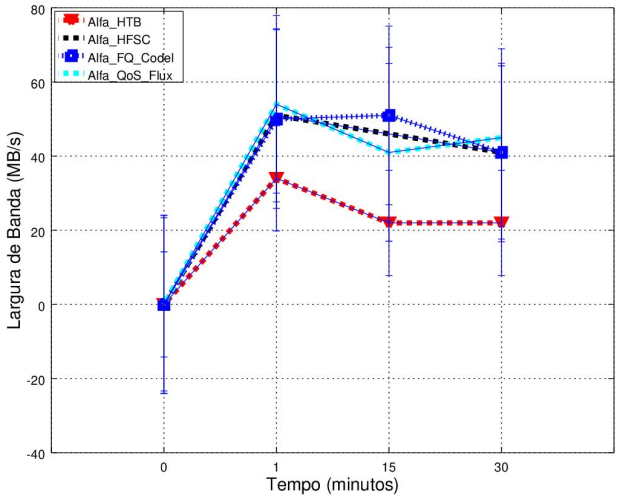

Figura 4.12: Largura de Banda em Fluxos do Tipo Alfa.

Na Figura 4.12 é pesquisado o parâmetro de largura de banda definido em *Megabit* por segundo (MB/s) para o teste com fluxos do tipo alfa. Quando é utilizado o algoritmo *HTB*, foi constatado que foram produzidos valores médios de 34 MB/s para 1 minuto, 22 MB/s para 15 minutos não alterando este valor para 30 minutos. Quando é habilitado o algoritmo *HFSC*, foi verificado os valores médios de 51 MB/s em 1 minuto, 46 MB/s no período de tempo de 15 minutos e reduzindo este valor para 41 MB/s em 30 minutos. No algoritmo *FQ_Codel*, percebe-se os valores médios de 50 MB/s no período de 1 minuto, aumentando para 51 MB/s no período de 15 minutos e 41 MB/s no período de 30 minutos. Quando é utilizado o serviço QoS-Flux, o valor médio gerado foi de 54 MB/s para 1 minuto, 41 MB/s para 15 minutos e 45 MB/s no período de tempo de 30 minutos.

Com relação aos resultados gerados com os algoritmos e o serviço QoS-Flux na métrica de largura de banda em fluxos alfa (ver Tabela 4.2), todos conseguiram alcançar o mínimo exigido em *QoS* para esta métrica nos períodos de tempo. No entanto, foi descoberto que o algoritmo *HTB*, mais uma vez, gerou valores médios de largura de banda bem menor comparado com os outros algoritmos e o serviço QoS-Flux. Aliás, percebe-se que os algoritmos *HFSC* e o *FQ_Codel* obtiveram valores próximos comparado com o serviço QoS-Flux.

Para o desvio padrão na métrica de largura de banda em aplicações que geram fluxos alfa, foi obtido no algoritmo *HTB* o valor de 14 MB/s, no algoritmo *HFSC* foi de 23 MB/s e no *FQ_Codel* atingiu o valor de 24 MB/s. No serviço QoS-Flux foi gerado o valor de 23 MB/s.

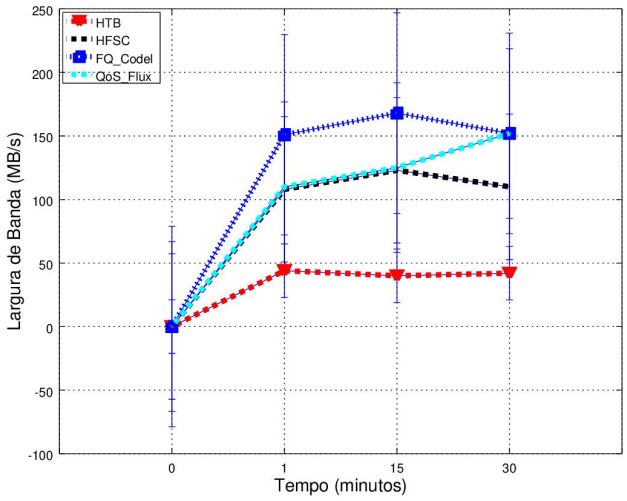

Figura 4.13: Largura de Banda Utilizando Todos os Fluxos.

Na Figura 4.13 é pesquisado o parâmetro de largura de banda definido em *Megabit* por segundo (MB/s) para o teste com todos os fluxos na rede. A partir da análise dos testes gerados com o algoritmo *HTB*, percebe-se que foram produzidos valores médios de 44 MB/s para 1 minuto, 40 MB/s para 15 minutos e 42 MB/s para 30 minutos. Quando é utilizado o algoritmo *HFSC*, foi constatado os valores médios de 108 MB/s em 1 minuto, 123 MB/s no período de tempo de 15 minutos e 111 MB/s para 30 minutos. No algoritmo *FQ_Codel*, podemos perceber os valores médios de 151 MB/s no período de 1 minuto, aumentando para 168 MB/s no período de 15 minutos, reduzindo para 152 MB/s no período de 30 minutos. Quando é habilitado o serviço QoS-Flux, o valor médio gerado foi de 110 MB/s para 1 minuto, 125 MB/s para 15 minutos e 152 MB/s no período de tempo de 30 minutos.

Com relação aos resultados gerados utilizando os algoritmos e o serviço QoS-Flux na métrica de largura de banda para todos os fluxos (ver Tabela 4.4), os mesmos conseguiram alcançar o mínimo exigido em *QoS* para esta métrica nos períodos de tempo. Entretanto, foi identificado que o algoritmo *HTB*, novamente, gerou valores médios de largura de banda inferiores comparado com os demais algoritmos e o serviço QoS-Flux.

Para o desvio padrão na métrica de largura de banda para todos os fluxos, foi obtido no algoritmo *HTB* o valor de 21 MB/s, no algoritmo *HFSC* atingiu o valor de 57 MB/s e no algoritmo *FQ_Codel* alcançou o valor de 78 MB/s. No serviço QoS-Flux foi gerado o valor de 66 MB/s.

4.5.4 Perda de Pacotes

Na Figura 4.14 é analisado o parâmetro de perda de pacotes definido em porcentagem (%) para o teste com fluxos elefantes. Considerando os testes realizados com o algoritmo HTB, percebe-se os valores médios de 0.05% para 1 minuto, 0.02% para 15 minutos e permanecendo aos 30 minutos. Nos algoritmos *HFSC* e *FQ_Codel*, observamos o valor médio de 0.01% permanecendo em todos os períodos de tempo. Com relação ao serviço QoS-Flux, o valor médio manteve-se em 0%.

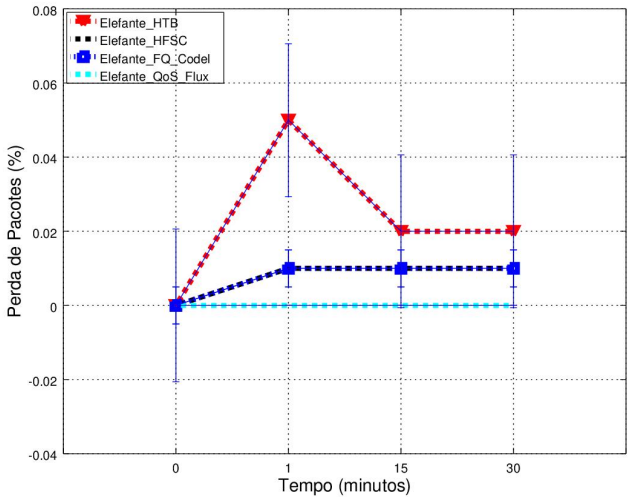

Figura 4.14: Perda de Pacotes em Fluxos Elefantes.

Foi identificado que os algoritmos *HTB, HFSC* e o *FQ_Codel* não atendem os requisitos mínimos de *QoS* para perda de pacotes (ver Tabela 4.3). Diferentemente ocorre no serviço QoS-Flux, o qual consegue atingir valores iguais a 0% em perda de pacotes em todos os períodos de tempo.

No desvio padrão na métrica de perda de pacotes em fluxos elefantes, o algoritmo *HTB* atingiu o valor de 0.04%, nos algoritmos *HFSC* e *FQ_Codel* foram obtidos os valores de 0.005% e o serviço QoS-Flux de 0%.

Na Figura 4.15 é analisado o parâmetro de perda de pacotes definido em porcentagem (%) para o teste com fluxos guepardos. Com relação ao algoritmo *HTB*, o mesmo conseguiu alcançar valores médios de 0,02% (1 minuto), 0.03% (15 minutos) e 0.035% (30 minutos). No algoritmo *HFSC*, percebe-se o valor médio de 0% no intervalo de 1 e 15 minutos, sendo o intervalo de 30 minutos o valor médio gerado de 0.01%. O oposto ocorre no algoritmo *FQ_Codel* e o serviço QoS-Flux, os quais conseguem atingir valores médios de 0% em todos os tempos até o final do teste.

A partir da análise efetuada, constatamos que os algoritmos *HTB* e o *HFSC* não conseguiram alcançar os requisitos mínimos de *QoS* para perda de pacotes em todos os

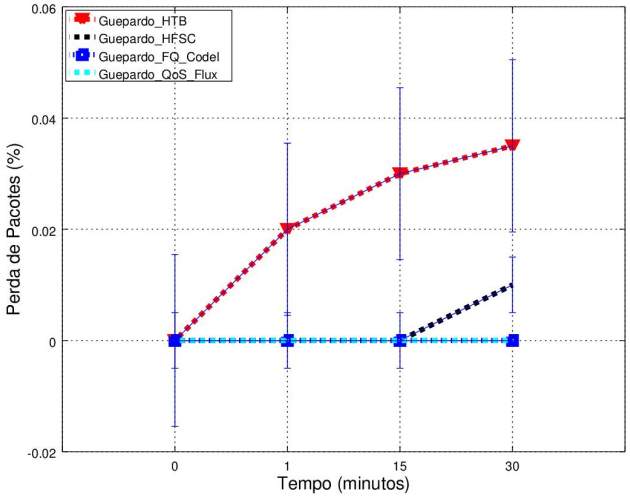

Figura 4.15: Perda de Pacotes em Fluxos Guepardo.

intervalos de tempo (ver Tabela 4.3). Somente o *FQ_Codel* e o serviço QoS-Flux seguiu as recomendações mínimas em todos os períodos de tempo.

Com relação ao desvio padrão para métrica de perda de pacotes em fluxos guepardos, o algoritmo *HTB* produziu o valor de 0.03%, no algoritmo *HFSC* foi gerado o valor de 0.001%, no algoritmo *FQ_Codel* e o serviço QoS-Flux, foi fornecido o valor de 0%.

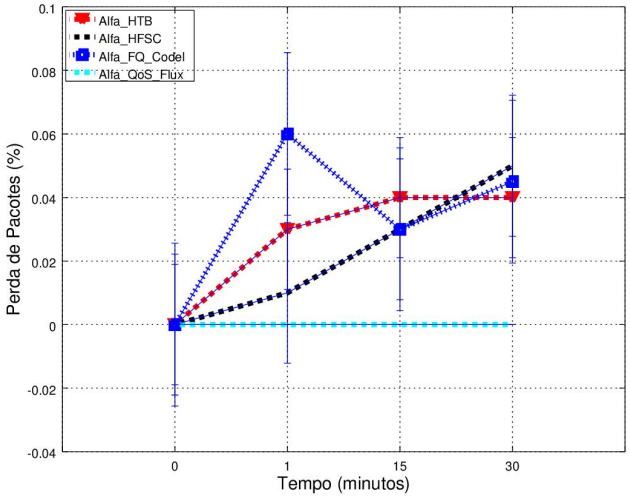

Figura 4.16: Perda de Pacotes em Fluxos Alfa.

Na Figura 4.16 é analisado o parâmetro de perda de pacotes definido em porcentagem (%) para o teste com fluxos alfa. A partir dos testes realizados, foi verificado os valores médios do algoritmo *HTB*, os quais foram gerados 0.03% (1 minuto) e 0.04% para 15 e 30 minutos, respectivamente. No algoritmo *HFSC*, percebe-se os valores médios de 0.01 (1

minuto), 0.03% (15 minutos) e 0.05% (30 minutos). Com relação ao algoritmo *FQ_Codel*, foram gerados valores médios de 0.06% para 1 minuto, 0.03% para 15 minutos e 0.045% para 30 minutos. Quando acionado o serviço QoS-Flux, é gerado o valor médio de 0% em todos os períodos de tempo.

Os resultados gerados utilizando fluxos alfa na métrica de *QoS* para perda de pacotes, é identificado que somente o algoritmo *FQ_Codel* não consegue atingir os valores mínimos recomendados para *QoS* no comparativo, observando o intervalo de tempo entre 0 a 1 minuto (ver Tabela 4.2). Com relação aos demais algoritmos e o serviço QoS-Flux, todos alcançaram o requisito mínimo para métrica de perda de pacotes, sendo o serviço QoS-Flux atingindo o menor valor.

Com relação ao desvio padrão para métrica de perda de pacotes em fluxos alfa, o algoritmo *HTB* produziu o valor de 0.01%, no algoritmo *HFSC* e *FQ_Codel* foram gerados os valores de 0.02%, com relação ao serviço QoS-Flux foi gerado o valor de 0%.

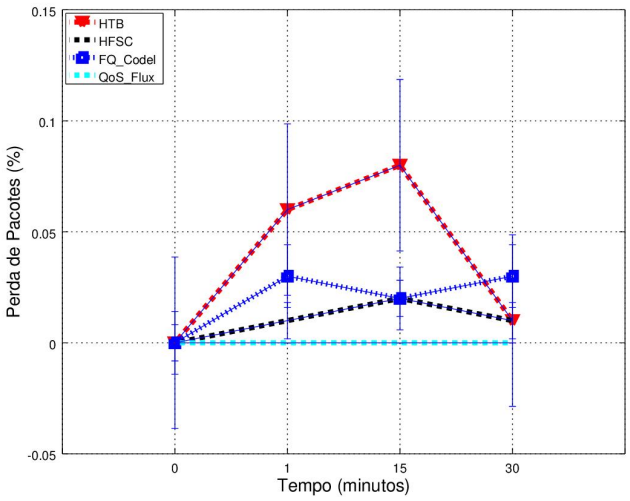

Figura 4.17: Perda de Pacotes Utilizando Todos os Fluxos.

Na Figura 4.17 é analisado o parâmetro de perda de pacotes definido em porcentagem (%) para o teste utilizando todos os fluxos simultaneamente. Analisando os testes realizados, percebe-se os valores médios para o algoritmo *HTB* de 0.06% para 1 minuto, 0.08% para 15 minutos e 0.01% para 30 minutos. No algoritmo *HFSC*, temos os valores médios de 0.01% para 1 minuto, 0.02% para 15 minutos e reduzindo para 0.01% para 30 minutos. No algoritmo *FQ_Codel*, é observado os valores médios de 0.03% (1 minuto), 0.02% (15 minutos) e aumentando para 0.03% (30 minutos). Quando habilitado o serviço QoS-Flux, o valor médio gerado permanece em 0% em todos os períodos de tempo.

Com relação aos resultados gerados na avaliação da métrica de perda de pacotes para todos fluxos, foi constatado que todos os algoritmos (*HTB*, *HFSC* e *FQ_Codel*) não

conseguiram alcançar os requisitos mínimos exigidos para *QoS* (ver Tabela 4.4). Quando é acionado o serviço QoS-Flux, percebe-se que no comparativo é o único que atinge os requisitos mínimos para perda de pacotes, ou seja, mantendo-se no valor médio de 0% em todos os períodos de tempo.

No desvio padrão para métrica de perda de pacotes quando utilizamos todos os fluxos, o algoritmo *HTB* gerou o valor de 0.003%, no algoritmo *HFSC* foi gerado o valor de 0.003%, no algoritmo *FQ_Codel* foi fornecido o valor de 0.005% e o serviço QoS-Flux, foi gerado o valor de 0%.

4.6 Discussões sobre os Resultados

Nos resultados obtidos, quando avaliamos as métricas de *QoS* para atraso, *jitter* e perda de pacotes em todos os cenários, percebe-se que o serviço QoS-Flux consegue aplicar nos fluxos elefantes, guepardos, alfas bem como quando utilizamos todos os fluxos ao mesmo tempo o algoritmo *FQ_Codel* quando necessário e os filtros *tc* (padrão *u32*. O filtro aplicado nos cenários, geralmente, foram aqueles com pacotes mínimos de 256 *bytes* com opção de prioridade em situações com pouco tráfego. Já o algoritmo *FQ_Codel*, foi acionado para minimizar os efeitos de latência, variações do *jitter* e quando ocorre incidência de perda de pacote na rede. Mesmo com algumas intermitências no resultado da métrica *jitter* quando foi utilizado o fluxo alfa, o serviço QoS-Flux conseguiu manter-se nos valores mínimos recomendados para *QoS*, ou seja, abaixo de 50 ms (milissegundos).

Com relação aos testes realizados para métrica de largura de banda, percebe-se que o serviço QoS-Flux em todos os cenários, conseguiu acionar o algoritmo *HFSC* quando necessário alcançando o mínimo recomendado para *QoS*. Este algoritmo consegue regular as altas taxas de transferências que possam ocorrer na análise de desempenho. Apesar disso, o QoS-Flux gerou valores abaixo do algoritmo *FQ_Codel* nos fluxos elefantes, guepardos e quando foi utilizado todos os fluxos ao mesmo tempo no intervalo entre 1 a 15 minutos. Entretanto, nos cenários com avaliação nos fluxos elefantes e alfas, o serviço QoS-Flux conseguiu os melhores resultados em comparação com os demais algoritmos nos 30 minutos finais. Além disso, o QoS-Flux também conseguiu gerar resultados semelhantes ao algoritmo *FQ_Codel* com todos os fluxos ao mesmo tempo, como observado nos 30 minutos finais.

4.7 Considerações Finais

Neste capítulo foi apresentado as ferramentas utilizadas, explicações sobre a montagem do ambiente experimental, as métricas de *QoS* utilizadas como padrão para este trabalho

e as cargas de trabalho adotadas nesta pesquisa. O capítulo também aborda os softwares utilizados para o bom desenvolvimento e análise dos experimentos, como o *VirtualBox* (virtualização das máquinas virtuais), o *Mininet* (simulação e configuração da topologia de rede) e o *D-ITG* (geração das cargas de trabalho para os fluxos elefantes, guepardos e alfas).

Além disso, o capítulo também mostra uma análise de desempenho entre os algoritmos de *QoS* configurados pela módulo *TC Linux* estaticamente e o serviço QoS-Flux configurado dinamicamente, utilizando aplicações que simulam os principais fluxos em uma rede *SDN*. Em cada cenário de teste será avaliado as principais métricas de *QoS*.

Por fim, foi realizado no Capítulo uma breve discussão sobre os resultados obtidos do serviço QoS-Flux diante do comparativo entre os algoritmos estáticos. Será explicado como o serviço QoS-Flux conseguiu se destacar utilizando os seus componentes para o controle dos fluxos apresentando os resultados recomendados para *QoS* nas métricas de atraso, *jitter*, largura de banda e perda de pacotes.

No Capítulo 5 será apresentado a conclusão e trabalhos futuros deste trabalho. Este capítulo está divido em algumas subseções para melhor entendimento de todo trabalho realizado: visão geral do trabalho, revisão dos objetivos e contribuições e, por fim, uma breve explicação dos trabalhos futuros.

Capítulo 5

Conclusões e Trabalhos Futuros

Este capítulo conclui esta dissertação. Primeiramente, uma visão geral sobre o trabalho é apresentada. Após isso, os objetivos desta dissertação são relembrados e, por fim, possibilidades de trabalhos futuros são discutidas.

5.1 Visão Geral do Trabalho

Neste trabalho foi desenvolvido uma implementação de *QoS* utilizando engenharia de tráfego através de algoritmos dinâmicos para redes *SDN*, chamada de QoS-Flux. Neste contexto, foi realizado um comparativo entre os algoritmos *HTB*, *HFSC* e o *FQ_Codel* habilitados estaticamente e o serviço QoS-Flux configurado dinamicamente para avaliação de desempenho. Todos foram analisados com relação aos requisitos mínimos de *QoS* recomendados para as métricas de atraso, *jitter*, largura de banda e perda de pacotes. Utilizamos dois cenários de testes de desempenho, sendo o primeiro com fluxos característicos do tipo elefante, guepardo e alfa, separadamente; e o segundo cenário com todos os fluxos simultaneamente.

Pelos resultados gerados nas simulações, quando é avaliado a métrica de atraso, o algoritmo *HTB* gera muitas oscilações nos 30 minutos finais, produzindo valores acima das recomendações para *QoS* em praticamente quase todos os cenários. No momento em que consideramos a métrica *jitter*, o algoritmo *HTB* (avaliação com todos os fluxos) e o algoritmo *FQ_Codel* (avaliação com fluxos alfa) não conseguiram atingir o mínimo recomendado para *QoS*. Com relação a métrica de largura de banda, percebemos que todos os algoritmos e o serviço QoS-Flux atingiram os valores mínimos recomendados. No entanto, o algoritmo *HTB*, gerou valores médios de largura de banda muito inferior em comparação com os demais. Na métrica perda de pacotes identificamos que nenhum algoritmo estático no comparativo atingiu os valores mínimos recomendados para *QoS*, somente o serviço QoS-Flux.

Por fim, percebemos nos testes realizados que o serviço QoS-Flux, quando habilitado, atinge as recomendações mínimas para atraso, *jitter*, largura de banda e perda de pacotes em *QoS* para todos os cenários no comparativo entre algoritmos configurados estaticamente. Além disso, diferentemente como ocorre nos algoritmos estáticos, percebe-se que uma configuração híbrida (como o serviço QoS-Flux), consegue aplicar *QoS* nos principais fluxos em uma rede *SDN* com trocas dinâmicas em tempo real entre algoritmos (*HFSC* e o *FQ_Codel*) sem precisar do administrador de rede para realizar alguma configuração.

5.2 Revisão dos Objetivos e Contribuições

Os objetivos desta dissertação são aqui relembrados, onde é indicado qual parte deste trabalho apresenta o estudo que visa cumprir cada objetivo.

1. **Estudar os conceitos e finalidade de algoritmos que tenham propósito de aplicar QoS em uma rede *SDN*.**

   No Capítulo 2 é realizada uma revisão dos conceitos fundamentais envolvendo este trabalho bem como soluções de *QoS* em redes *SDN* encontradas na literatura. Já no Capítulo 3, foram selecionados três algoritmos de *QoS* para análise de desempenho, ou seja, os algoritmos (*HTB*, *HFSC* e o *FQ_Codel*). Foram apresentados as definições e características de cada algoritmo de *QoS* em uma rede *SDN*.

   O algoritmo *HTB* foi escolhido por ser o mais citado e utilizado na literatura para realizar o controle e gerenciamento de fluxos em redes *SDN*. Os algoritmos *HFSC* e o *FQ_Codel* foram selecionados por trabalharem no mecanismo de aplicação de filas de *QoS* no serviço QoS-Flux.

2. **Desenvolvimento e avaliação do serviço QoS-Flux utilizando somente *softwares* sem custo.**

   O Capítulo 3 apresenta o serviço QoS-Flux, demonstrando as suas particularidades de funcionamento, configurações e o modo de funcionamento de acordo com o tipo de fluxo que acessa a rede, ou seja, fluxo elefante e/ou guepardo e/ou alfa.

   O serviço QoS-Flux necessita de algumas *ferramentas* e sistemas para suas operações e análise de desempenho. Por este motivo, estas ferramentas são explicadas nos capítulos 3 e 4:

   - Sistema operacional *Linux* - Utilizado para execução de todo serviço QoS-Flux;
   - Ferramenta *VirtualBox* - Empregado para realizar a virtualização de todo sistema *SDN*;

- Emulador *Mininet* - Aplicado para gerar e executar a topologia da rede *SDN* virtualizada;
- Módulo *TC Linux* - Configurado para habilitar no *kernel* do sistema *Linux* os filtros e os algoritmos configurados de forma estática (*HTB*, *HFSC* e o *FQ_Codel*) ou dinâmica (QoS-Flux) aplicados para rede *SDN*;
- *Switches* virtuais *OVS* - Utilizado para executar os fluxos no plano de dados em *SDN*;
- Controlador *SDN Ryu* - Empregado para gerenciar a topologia *SDN* e enviar os comandos do serviço QoS-Flux para o plano de dados.

3. **Analisar o desempenho dos algoritmos para *QoS* e o serviço QoS-Flux em redes *SDN*.**

   No Capítulo 4 é mostrado uma análise de desempenho em redes *SDN* utilizando um comparativo entre algoritmos de *QoS* configurados estaticamente (*HTB*, *HFSC* e o *FQ_Codel*) e o serviço de QoS-Flux configurado com acionamento dinâmico. Para os cenários de testes realizados, foram selecionados aplicações com características semelhantes aos fluxos elefantes, guepardos e alfas.

   Os algoritmos de *QoS* e o serviço QoS-Flux foram avaliados de acordo com as principais métricas de *QoS* (atraso, *jitter*, largura de banda e perda de pacotes). Além disso, adotamos os requisitos mínimos baseado nas normas *ITU-T G.1010* e a *ITU-T G.1050* para *QoS* na avaliação em todos os cenários de testes.

Por fim, como contribuição, enviamos um artigo científico baseado na qualificação com o tema *"Supporting and Evaluating Quality of Service Solutions when Applied to Different Types of Flows in Software Defined Networks"* para revista *CLEI Electronic Journal*, o qual estamos aguardando resposta da comissão de avaliação.

5.3 Trabalhos Futuros

O estado da arte da área de *SDN* e aplicação de *QoS* apresentam diversos desafios e encontram-se ainda abertos, os quais foram constatados durante a revisão bibliográfica e a análise de desempenho entre os algoritmos e o serviço QoS-Flux. Uma vez que a proposta desenvolvida nesta dissertação tem escopo e restrições definidas, não foram acrescentados algumas formas de utilização, como funcionalidades para funções mais específicas. Assim, as possíveis evoluções futuras do serviço QoS-Flux proposta nesta trabalho incluem:

- Utilizar outras topologias (ex: *fat tree* ou *mesh* para redes *datacenter*) para avaliar melhor o desempenho do serviço QoS-Flux em redes *SDN*;

- Empregar o uso de configurações diferentes de fluxos executados ao mesmo tempo em diversas partes da rede *SDN* com o objetivo de analisar os limites do serviço QoS-Flux;
- Analisar o serviço QoS-Flux comparando com outros algoritmos que aplicam *QoS* em uma rede *SDN*;
- Avaliar os requisitos mínimos de *QoS* e o tempo de resposta para controle do tráfego comparando o serviço QoS-Flux com outras propostas para redes *SDN*, como o *framework* para o controlador *OpenDaylight* [64], o *Queue Pusher* para o controlador *Floodlight* [55] e o *HiQoS* [42].

Referências

[1] I. Sandvine, "The global internet phenomena report," https://www.sandvine.com/hubfs/downloads/phenomena/2018-phenomena-report.pdf, 2018, accessed: 18/01/2019. xi, 6, 7, 34

[2] B. A. A. Nunes, M. Mendonca, X.-N. Nguyen, K. Obraczka, and T. Turletti, "A survey of software-defined networking: Past, present, and future of programmable networks," *IEEE Communications Surveys & Tutorials*, vol. 16, no. 3, pp. 1617–1634, 2014. xi, 15, 16

[3] M. A. Brown, "Traffic control howto," http://tldp.org/HOWTO/Traffic-Control-HOWTO/index.html, 2006, accessed: 16/05/2018. xii, 26

[4] I. T. UNION, "Itu-t g. rec. 1010: End-user multimedia qos categories," *Technical Report, ITU*, 2001. xii, 35, 36

[5] ——, "Itu-t g. rec. 1050: Multimedia quality of service and performance – generic and user-related aspects," *Technical Report, ITU*, 2016. xii, 35, 36

[6] J. L. Shah and J. Parvez, "Evaluation of queuing algorithms on qos sensitive applications in ipv6 network," in *Advances in Computing, Communications and Informatics (ICACCI, 2014 International Conference on.* IEEE, 2014, pp. 106–111. 1, 11

[7] J. Domżał, "Flow-aware networking as an architecture for the ipv6 qos parallel internet," in *2013 Australasian Telecommunication Networks and Applications Conference (ATNAC).* IEEE, 2013, pp. 30–35. 1

[8] D. M. Divakaran, "A spike-detecting aqm to deal with elephants," *Computer Networks*, vol. 56, no. 13, pp. 3087–3098, 2012. 1, 8

[9] Z. Yan, C. Tracy, M. Veeraraghavan, T. Jin, and Z. Liu, "A network management system for handling scientific data flows," *Journal of Network and Systems Management*, vol. 24, no. 1, pp. 1–33, 2016. 1, 7

[10] S. Maji, M. Veeraraghavan, M. Buchanan, F. Alali, J. Ros-Giral, and A. Commike, "A high-speed cheetah flow identification network function (cfinf)," in *Network Function Virtualization and Software Defined Networks (NFV-SDN), 2017 IEEE Conference on.* IEEE, 2017, pp. 1–7. 1, 18

[11] K.-c. Lan and J. Heidemann, "A measurement study of correlations of internet flow characteristics," *Computer Networks*, vol. 50, no. 1, pp. 46–62, 2006. 1, 7, 8, 9

[12] H.-W. Tin, S.-W. Leu, and S.-H. Chang, "Measurement of flow burstiness by fractal technique," in *Computer Symposium (ICS), 2010 International*. IEEE, 2010, pp. 722–727. 1

[13] S. Maji, X. Wang, M. Veeraraghavan, J. Ros-Giralt, and A. Commike, "A pragmatic approach of determining heavy-hitter traffic thresholds," in *2018 European Conference on Networks and Communications (EuCNC)*. IEEE, 2018, pp. 1–9. 1, 2, 10, 18

[14] J. Ros-Giralt, A. Commike, S. Maji, and M. Veeraraghavan, "A mathematical framework for the detection of elephant flows," *arXiv preprint arXiv:1701.01683*, 2017. 1

[15] A. O. Adedayo and B. Twala, "Qos functionality in software defined network," in *Information and Communication Technology Convergence (ICTC), 2017 International Conference on*. IEEE, 2017, pp. 693–699. 2, 12, 18, 26

[16] S. Tariq and M. Bassiouni, "Qamo-sdn: Qos aware multipath tcp for software defined optical networks," in *Consumer Communications and Networking Conference (CCNC), 2015 12th Annual IEEE*. IEEE, 2015, pp. 485–491. 2, 18

[17] A. M. Abdelmoniem and B. Bensaou, "Incast-aware switch-assisted tcp congestion control for data centers," in *Global Communications Conference (GLOBECOM), 2015 IEEE*. IEEE, 2015, pp. 1–6. 2, 18

[18] M. Cello, M. Marchese, and M. Mongelli, "On the qos estimation in an openflow network: The packet loss case," *IEEE Communications Letters*, vol. 20, no. 3, pp. 554–557, 2016. 2, 18

[19] H. T. Hong, Q. B. Xuan, D. D. Van, N. P. Ngoc, and T. N. Huu, "Hardware-efficient implementation of wfq algorithm on netfpga-based openflow switch," in *Advanced Technologies for Communications (ATC), 2016 International Conference on*. IEEE, 2016, pp. 431–436. 2, 18

[20] I. Stoica, H. Zhang, and T. E. Ng, "A hierarchical fair service curve algorithm for link-sharing, real-time, and priority services," *IEEE/ACM transactions on Networking*, vol. 8, no. 2, pp. 185–199, 2000. 2, 20, 24

[21] T. Hoeiland-Joergensen, P. McKenney, D. Taht, J. Gettys, and E. Dumazet, "The flow queue codel packet scheduler and active queue management algorithm," *RFC 8290*, 2016. 2, 20

[22] H. E. Egilmez, S. Civanlar, and A. M. Tekalp, "An optimization framework for qos-enabled adaptive video streaming over openflow networks," *IEEE Transactions on Multimedia*, vol. 15, no. 3, pp. 710–715, 2013. 3

[23] O. N. Fundation, "Software-defined networking: The new norm for networks," *ONF White Paper*, vol. 2, pp. 2–6, 2012. 3, 14

[24] C. Caba and J. Soler, "Apis for qos configuration in software defined networks," in *Network Softwarization (NetSoft), 2015 1st IEEE Conference on*. IEEE, 2015, pp. 1–5. 3

[25] C. Networking, "Forecast and methodology, white paper," *San Jose, CA, USA*, vol. 1, 2016. 6

[26] A. Callado, C. Kamienski, G. Szabó, B. P. Gero, J. Kelner, S. Fernandes, and D. Sadok, "A survey on internet traffic identification," *IEEE communications surveys & tutorials*, vol. 11, no. 3, 2009. 6, 7

[27] P. Giacomazzi, "Analisi e identificazione del traffico internet," *Mondo Digitale*, vol. 2, no. 2, pp. 13–28, 2010. 7

[28] P. Megyes, "Traffic measurements, characterization and emulation in emerging networks," Master's thesis, Budapest University of Technology and Economics, 2017, master Thesis. 7

[29] T. T. Nguyen and G. Armitage, "A survey of techniques for internet traffic classification using machine learning," *IEEE Communications Surveys & Tutorials*, vol. 10, no. 4, pp. 56–76, 2008. 7

[30] S. Shirali-Shahreza and Y. Ganjali, "Delayed installation and expedited eviction: An alternative approach to reduce flow table occupancy in sdn switches," *IEEE/ACM Transactions on Networking*, vol. 26, no. 4, pp. 1547–1561, 2018. 8

[31] B. Wang, J. Su, L. Chen, J. Deng, and L. Zheng, "Effieye: Application-aware large flow detection in data center," in *Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. IEEE Press, 2017, pp. 794–796. 8

[32] R. d. O. Schmidt, R. Sadre, N. Melnikov, J. Schonwalder, and A. Pras, "Linking network usage patterns to traffic gaussianity fit," in *Networking Conference, 2014 IFIP*. IEEE, 2014, pp. 1–9. 8

[33] M. Noormohammadpour and C. S. Raghavendra, "Datacenter traffic control: Understanding techniques and tradeoffs," *IEEE Communications Surveys & Tutorials*, vol. 20, no. 2, pp. 1492–1525, 2017. 9

[34] MacMichael, Duncan, "Interrupt moderation," https://docs.microsoft.com/en-us/windows-hardware/drivers/network/interrupt-moderation, 2017, accessed: 05/01/2019. 9

[35] L. Fan, H. Cruickshank, and Z. Sun, "Ip networking over next-generation satellite systems," in *International Workshop, Budapest*. Springer, 2007. 9

[36] R. Kapoor, A. C. Snoeren, G. M. Voelker, and G. Porter, "Bullet trains: a study of nic burst behavior at microsecond timescales," in *Proceedings of the ninth ACM conference on Emerging networking experiments and technologies*. ACM, 2013, pp. 133–138. 9

[37] S. Molnar and Z. Moczar, "Three-dimensional characterization of internet flows," in *Communications (ICC), 2011 IEEE International Conference on.* IEEE, 2011, pp. 1–6. 10

[38] J. Gettys and K. Nichols, "Bufferbloat: Dark buffers in the internet," *Queue*, vol. 9, no. 11, p. 40, 2011. 10

[39] A. Bechtolsheim, L. Dale, H. Holbrook, and A. Li, "Why big data needs big buffer switches," *Arista White Paper*, 2016. 10

[40] M. Karakus and A. Durresi, "Quality of service (qos) in software defined networking (sdn): A survey," *Journal of Network and Computer Applications*, vol. 80, pp. 200–218, 2017. 10, 16

[41] F. Pana and F. Put, "A survey on the evolution of rsvp," *IEEE Communications Surveys & Tutorials*, vol. 15, no. 4, pp. 1859–1887, 2013. 11

[42] Y. Jinyao, Z. Hailong, S. Qianjun, L. Bo, and G. Xiao, "Hiqos: An sdn-based multipath qos solution," *China Communications*, vol. 12, no. 5, pp. 123–133, 2015. 11, 18, 58

[43] T. Szigeti, C. Hattingh, R. Barton, and K. Briley Jr, *End-to-End QoS Network Design: Quality of Service for Rich-Media & Cloud Networks.* Cisco Press, 2013. 12

[44] S. Przylucki and D. Czerwinski, "Priority-aware packet pre-marking for diffserv architecture based on h. 264/svc video stream structure," *Wireless Personal Communications*, vol. 96, no. 4, pp. 5391–5408, 2017. 12

[45] U. M. Mir, A. H. Mir, A. Bashir, and M. A. Chishti, "Diffserv-aware multi protocol label switching based quality of service in next generation networks," in *Advance Computing Conference (IACC), 2014 IEEE International.* IEEE, 2014, pp. 233–238. 12

[46] A.-B. R. Sulaiman and O. K. S. Alhafidh, "Performance analysis of multimedia traffic over mpls communication networks with traffic engineering," *International Journal of Computer Networks and Communications Security*, vol. 2, no. 3, pp. 93–101, 2014. 12

[47] W. Johnston, C. Guok, and E. Chaniotakis, "Motivation, design, deployment and evolution of a guaranteed bandwidth network service," in *Proceedings of the TERENA Networking Conference*, 2011. 12

[48] M. F. Al-Naday, A. Bontozoglou, V. G. Vassilakis, and M. J. Reed, "Quality of service in an information-centric network," in *2014 IEEE Global Communications Conference.* IEEE, 2014, pp. 1861–1866. 13

[49] A. Gertsiy and S. Rudyk, "Analysis of quality of service parameters in ip-networks," in *Problems of Infocommunications Science and Technology (PIC S&T), 2016 Third International Scientific-Practical Conference.* IEEE, 2016, pp. 75–77. 13

[50] P. S. Tomar, P. Mishra, and D. Das, "Study on selection of an efficient routing protocol for tactical wireless communication network," in *Convergence in Technology (I2CT), 2017 2nd International Conference for*. IEEE, 2017, pp. 51–54. 13

[51] S. Hasija, R. Mijumbi, S. Davy, A. Davy, B. Jennings, and K. Griffin, "Domain federation via mpls and sdn for dynamic, real-time end-to-end qos support," in *2018 4th IEEE Conference on Network Softwarization and Workshops (NetSoft)*. IEEE, 2018, pp. 177–181. 13

[52] R. Amin, M. Reisslein, and N. Shah, "Hybrid sdn networks: A survey of existing approaches," *IEEE Communications Surveys & Tutorials*, 2018. 14

[53] D. Kreutz, F. M. Ramos, P. E. Verissimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-defined networking: A comprehensive survey," *Proceedings of the IEEE*, vol. 103, no. 1, pp. 14–76, 2015. 14

[54] Y. Zhang, L. Cui, W. Wang, and Y. Zhang, "A survey on software defined networking with multiple controllers," *Journal of Network and Computer Applications*, 2017. 16

[55] D. Palma, J. Gonçalves, B. Sousa, L. Cordeiro, P. Simoes, S. Sharma, and D. Staessens, "The queuepusher: Enabling queue management in openflow," in *Software Defined Networks (EWSDN), 2014 Third European Workshop on*. IEEE, 2014, pp. 125–126. 17, 18, 58

[56] S. U. Baek, C. H. Park, E. Kim, and D.-R. Shin, "Implementation and verification of qos priority over software defined networking," in *Proceedings on the International Conference on Internet Computing (ICOMP)*. The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp), 2016, p. 104. 17, 26

[57] Project, Ryu, "Qos — ryubook 1.0 documentation," https://osrg.github.io/ryu-book/en/html/, 2014, accessed: 12/05/2018. 17, 33

[58] M. Afaq, S. U. Rehman, and W.-C. Song, "A framework for classification and visualization of elephant flows in sdn-based networks," *Procedia Computer Science*, vol. 65, pp. 672–681, 2015. 17

[59] L. A. D. Knob, R. P. Esteves, L. Z. Granville, and L. M. R. Tarouco, "Sdefix—identifying elephant flows in sdn-based ixp networks," in *NOMS 2016-2016 IEEE/IFIP Network Operations and Management Symposium*. IEEE, 2016, pp. 19–26. 17

[60] R. Trestian, G.-M. Muntean, and K. Katrinis, "Micetrap: Scalable traffic engineering of datacenter mice flows using openflow," in *2013 IFIP/IEEE International Symposium on Integrated Network Management (IM 2013)*. IEEE, 2013, pp. 904–907. 17

[61] R. Trestian, K. Katrinis, and G.-M. Muntean, "Ofload: An openflow-based dynamic load balancing strategy for datacenter networks," *IEEE Transactions on Network and Service Management*, 2017. 17

[62] J. Rasley, B. Stephens, C. Dixon, E. Rozner, W. Felter, K. Agarwal, J. Carter, and R. Fonseca, "Planck: Millisecond-scale monitoring and control for commodity networks," in *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4. ACM, 2014, pp. 407–418. 18

[63] L. Yao, P. Hong, W. Zhang, J. Li, and D. Ni, "Controller placement and flow based dynamic management problem towards sdn," in *2015 IEEE International Conference on Communication Workshop (ICCW)*. IEEE, 2015, pp. 363–368. 18

[64] H. Ghalwash and C.-H. Huang, "A qos framework for sdn-based networks," in *2018 IEEE 4th International Conference on Collaboration and Internet Computing (CIC)*. IEEE, 2018, pp. 98–105. 18, 58

[65] S. Ren, Q. Feng, and W. Dou, "An end-to-end qos routing on software defined network based on hierarchical token bucket queuing discipline," in *Proceedings of the 2017 International Conference on Data Mining, Communications and Information Technology*. ACM, 2017, p. 31. 19, 26

[66] A. Ishimori, F. Farias, E. Cerqueira, and A. Abelém, "Control of multiple packet schedulers for improving qos on openflow/sdn networking," in *Software Defined Networks (EWSDN), 2013 Second European Workshop on*. IEEE, 2013, pp. 81–86. 19, 26

[67] Lima, Alessandro C., "Qos-flux: Dynamic control traffic for sdn in different flows," https://github.com/alessandro-lima/Qos-flux, 2018, accessed: 30/01/2019. 20

[68] K. Belitzky, "Collectd-web," https://collectd.org/wiki/index.php/Collectd-web, 2012, accessed: 13/05/2018. 23

[69] Hubert, Bert and Geul, Jacco and Séhier, Simon, "Wonder shaper: A tool to limit network bandwidth in linux," https://github.com/magnific0/wondershaper, 2002, accessed: 20/09/2018. 24

[70] Smidsrod, Robin, "Supershaper-soho: A tool traffic shaping for dsl connections which prioritizes voip and interactive traffic," https://github.com/robinsmidsrod/SuperShaper-SOHO, 2018, accessed: 20/09/2018. 24, 25

[71] M. A. Mortol and J. Moreira, "Qos em um cenário voip-utilizando o agendador htb," *Revista TIS*, vol. 1, no. 2, 2012. 24, 27

[72] B. Hubert, "Ubuntu manpage: tc - show / manipulate traffic control settings," http://manpages.ubuntu.com/manpages/xenial/man8/tc.8.html, 2016, accessed: 12/05/2018. 24, 26, 33

[73] Linux, Foundation, "Production quality, multilayer open virtual switch," https://www.openvswitch.org/, 2018, accessed: 31/03/2018. 25

[74] E. J. Jackson, M. Walls, A. Panda, J. Pettit, B. Pfaff, J. Rajahalme, T. Koponen, and S. Shenker, "Softflow: A middlebox architecture for open vswitch," in *2016 USENIX, Annual Technical Conference (USENIX ATC 16)*. Denver, CO: USENIX Association, 2016, pp. 15–28. 25

[75] P. Imputato and S. Avallone, "Design and implementation of the traffic control module in ns-3," in *Proceedings of the Workshop on ns-3*. ACM, 2016, pp. 1–8. 25

[76] C.-H. Lee and Y.-T. Kim, "Qos-aware hierarchical token bucket (qhtb) queuing disciplines for qos-guaranteed diffserv provisioning with optimized bandwidth utilization and priority-based preemption," in *Information Networking (ICOIN), 2013 International Conference on*. IEEE, 2013, pp. 351–358. 26, 27

[77] A. Keller, "Manual tc packet filtering and netem," *ETH Zurich, July*, vol. 20, 2006. 27, 33

[78] R. Khondoker, A. Zaalouk, R. Marx, and K. Bayarou, "Feature-based comparison and selection of software defined networking (sdn) controllers," in *2014 World Congress on Computer Applications and Information Systems (WCCAIS)*. IEEE, 2014, pp. 1–7. 29

[79] R. L. S. De Oliveira, A. A. Shinoda, C. M. Schweitzer, and L. R. Prete, "Using mininet for emulation and prototyping software-defined networks," in *Communications and Computing (COLCOM), 2014 IEEE Colombian Conference on*. IEEE, 2014, pp. 1–6. 31

[80] F. Keti and S. Askar, "Emulation of software defined networks using mininet in different simulation environments," in *Intelligent Systems, Modelling and Simulation (ISMS), 2015 6th International Conference on*. IEEE, 2015, pp. 205–210. 31

[81] Manual, D-ITG, "D-itg-2.8.1-manual.pdf," http://traffic.comics.unina.it/software/ITG/manual/D-ITG-2.8.1-manual.pdf, 2006, accessed: 06/04/2018. 32

[82] M. A. Brown, K. Rechert, and P. McHardy, "Hfsc scheduling with linux," http://linux-ip.net/articles/hfsc.en/, 2006, accessed: 16/05/2018. 33

[83] G. Gonçalves, I. Drago, A. P. C. Da Silva, A. B. Vieira, and J. M. Almeida, "Modeling the dropbox client behavior," in *2014 IEEE International Conference on Communications (ICC)*. IEEE, 2014, pp. 1332–1337. 34

[84] G. Wallace, F. Douglis, H. Qian, P. Shilane, S. Smaldone, M. Chamness, and W. Hsu, "Characteristics of backup workloads in production systems." in *FAST*, vol. 12, 2012. 34

[85] A. Rao, A. Legout, Y.-s. Lim, D. Towsley, C. Barakat, and W. Dabbous, "Network characteristics of video streaming traffic," in *Proceedings of the Seventh COnference on emerging Networking EXperiments and Technologies*. ACM, 2011, p. 25. 34

Anexo I

QoS-Flux

```
#!/bin/bash

### BEGIN INIT INFO
# Provides:          QoS-Flux
# Short-Description: Management of dynamic traffic control for SDN
# Description:       Management of dynamic traffic control for SDN
### END INIT INFO

#################################################################################
#
# QoS-Flux 1.0
#
# QoS-Flux is a management of dynamic traffic control algorithms
# to provide quality of service in SDN in different flows (elephants vs. mice,
# alpha vs. beta and Cheetah vs. snails) on one or more switches in an SDN network.
#
# Copyright (C) 2018 Alessandro Lima <alessandrolima1987@gmail.com>
#
# License details and more available at:
# https://github.com/alessandro-lima/Qos-flux
#
#

#The script was developed based on the SuperShaper-SOHO 2.0 and Wonder Shaper 1.4.
```

```
#This script has the perspective of improving on SDN (Software Defined Network)
#networks the loss of packets, delay, jitter and bandwidth in different streams
#for applications that work using TCP.
#To run this script, you will need iproute2 (tc) and the netfilter of the Linux
#kernel installed.
#The howto of the TC filter can be found here:
#http://lartc.org/howto/lartc.qdisc.filters.html
#Man page for u32 classifier can be found here:
#http://man7.org/linux/man-pages/man8/tc-u32.8.html
#
################################################################################

#Its output interface to SDN switches or routers

# Your outbound interface Open Vswitch (OVS)
IF_1=s2-eth1
IF_2=s2-eth2
IF_3=s2-eth3
IF_4=s3-eth1
IF_5=s3-eth2
IF_6=s3-eth3
IF_7=s4-eth1
IF_8=s4-eth2
IF_9=s4-eth3

# Definitions meter and percent Bandwidth

BANDWIDTH_MAX=1000000

BANDWIDTH_PERCENT=95

BANDWIDTH=$((BANDWIDTH_PERCENT*BANDWIDTH_MAX/100))

# Set full path to TC command, unless it's in PATH
TC=tc

# Monitoring of Flows
```

```
# It will store the SDN network flows, being
# accessible to other external tools generate charts.
FLOWS_FILENAME="/root/qos-flux.csv"

# Which qdisc to use for HFSC leaves
QDISC_FLOW="fq_codel ecn"

#############################################################################
######## FILTER PRIORITY #########
#############################################################################

######## FUNCTION FILTER FOR FLOW #########

function filter_for_flow {
    flowid="$1"; shift
    $TC filter add dev $IF_1 parent 1: protocol ip prio
    $((filter_prio++)) u32 "$@" flowid "1:$flowid"
    $TC filter add dev $IF_2 parent 1: protocol ip prio
    $((filter_prio++)) u32 "$@" flowid "1:$flowid"
    $TC filter add dev $IF_3 parent 1: protocol ip prio
    $((filter_prio++)) u32 "$@" flowid "1:$flowid"
    $TC filter add dev $IF_4 parent 1: protocol ip prio
    $((filter_prio++)) u32 "$@" flowid "1:$flowid"
    $TC filter add dev $IF_5 parent 1: protocol ip prio
    $((filter_prio++)) u32 "$@" flowid "1:$flowid"
    $TC filter add dev $IF_6 parent 1: protocol ip prio
    $((filter_prio++)) u32 "$@" flowid "1:$flowid"
    $TC filter add dev $IF_7 parent 1: protocol ip prio
    $((filter_prio++)) u32 "$@" flowid "1:$flowid"
    $TC filter add dev $IF_8 parent 1: protocol ip prio
    $((filter_prio++)) u32 "$@" flowid "1:$flowid"
    $TC filter add dev $IF_9 parent 1: protocol ip prio
    $((filter_prio++)) u32 "$@" flowid "1:$flowid"
}

#############################################################################
TC class for algorithm HFSC enabling LINK SHARING (ls)
```

```
and UPPER LIMIT (ul) for HFSC
#############################################################################
#############################################################################
                  TC qdisc for algorithm chosen in "QDISC_FLOW"
#############################################################################

######## FUNCTION LS FLOW #########

used_bw_percent_0=0
function define_ls_flow_0 {
     flowid="$1"; min_bw_percent="$2"; label="$3"; shift 3
    min_bw=$((min_bw_percent*BANDWIDTH/100))
    $TC class add dev $IF_1 parent 1:1 classid "1:$flowid"
    hfsc ls m2 "${min_bw}Kbit"
    $TC qdisc add dev $IF_1 parent "1:$flowid" handle "$flowid:" $QDISC_FLOW
    used_bw_percent_0=$((used_bw_percent_0+min_bw_percent))
}



used_bw_percent_1=0
function define_ls_flow_1 {
    flowid="$1"; min_bw_percent="$2"; label="$3"; shift 3
    min_bw=$((min_bw_percent*BANDWIDTH/100))
    $TC class add dev $IF_2 parent 1:1 classid "1:$flowid"
    hfsc ls m2 "${min_bw}Kbit"
    $TC qdisc add dev $IF_2 parent "1:$flowid" handle "$flowid:" $QDISC_FLOW
    used_bw_percent_1=$((used_bw_percent_1+min_bw_percent))
}

used_bw_percent_2=0
function define_ls_flow_2 {
    flowid="$1"; min_bw_percent="$2"; label="$3"; shift 3
    min_bw=$((min_bw_percent*BANDWIDTH/100))
    $TC class add dev $IF_3 parent 1:1 classid "1:$flowid"
    hfsc ls m2 "${min_bw}Kbit"
    $TC qdisc add dev $IF_3 parent "1:$flowid" handle "$flowid:" $QDISC_FLOW
    used_bw_percent_2=$((used_bw_percent_2+min_bw_percent))
```

```
}

used_bw_percent_3=0
function define_ls_flow_3 {
    flowid="$1"; min_bw_percent="$2"; label="$3"; shift 3
    min_bw=$((min_bw_percent*BANDWIDTH/100))
    $TC class add dev $IF_4 parent 1:1 classid "1:$flowid"
    hfsc ls m2 "${min_bw}Kbit"
    $TC qdisc add dev $IF_4 parent "1:$flowid" handle "$flowid:" $QDISC_FLOW
    used_bw_percent_3=$((used_bw_percent_3+min_bw_percent))

}

used_bw_percent_4=0
function define_ls_flow_4 {
    flowid="$1"; min_bw_percent="$2"; label="$3"; shift 3
    min_bw=$((min_bw_percent*BANDWIDTH/100))
    $TC class add dev $IF_5 parent 1:1 classid "1:$flowid"
    hfsc ls m2 "${min_bw}Kbit"
    $TC qdisc add dev $IF_5 parent "1:$flowid" handle "$flowid:" $QDISC_FLOW
    used_bw_percent_4=$((used_bw_percent_4+min_bw_percent))

}

used_bw_percent_5=0
function define_ls_flow_5 {
    flowid="$1"; min_bw_percent="$2"; label="$3"; shift 3
    min_bw=$((min_bw_percent*BANDWIDTH/100))
    $TC class add dev $IF_6 parent 1:1 classid "1:$flowid"
    hfsc ls m2 "${min_bw}Kbit"
    $TC qdisc add dev $IF_6 parent "1:$flowid" handle "$flowid:" $QDISC_FLOW
    used_bw_percent_3=$((used_bw_percent_5+min_bw_percent))

}

used_bw_percent_6=0
```

```
function define_ls_flow_6 {
    flowid="$1"; min_bw_percent="$2"; label="$3"; shift 3
    min_bw=$((min_bw_percent*BANDWIDTH/100))
    $TC class add dev $IF_7 parent 1:1 classid "1:$flowid"
    hfsc ls m2 "${min_bw}Kbit"
    $TC qdisc add dev $IF_7 parent "1:$flowid" handle "$flowid:" $QDISC_FLOW
    used_bw_percent_6=$((used_bw_percent_6+min_bw_percent))
}

used_bw_percent_7=0
function define_ls_flow_7 {
    flowid="$1"; min_bw_percent="$2"; label="$3"; shift 3
    min_bw=$((min_bw_percent*BANDWIDTH/100))
    $TC class add dev $IF_8 parent 1:1 classid "1:$flowid"
    hfsc ls m2 "${min_bw}Kbit"
    $TC qdisc add dev $IF_8 parent "1:$flowid" handle "$flowid:" $QDISC_FLOW
    used_bw_percent_7=$((used_bw_percent_7+min_bw_percent))

}

used_bw_percent_8=0
function define_ls_flow_8 {
    flowid="$1"; min_bw_percent="$2"; label="$3"; shift 3
    min_bw=$((min_bw_percent*BANDWIDTH/100))
    $TC class add dev $IF_9 parent 1:1 classid "1:$flowid"
    hfsc ls m2 "${min_bw}Kbit"
    $TC qdisc add dev $IF_9 parent "1:$flowid" handle "$flowid:" $QDISC_FLOW
    used_bw_percent_8=$((used_bw_percent_8+min_bw_percent))

}

######## FUNCTION INTERFACE START #########

function start_me {
    stop_me quiet

    printf "Qos-flux Shaping on $IF_1\n"
```

```
printf "Qos-flux Shaping on $IF_2\n"
printf "Qos-flux Shaping on $IF_3\n"
printf "Qos-flux Shaping on $IF_4\n"
printf "Qos-flux Shaping on $IF_5\n"
printf "Qos-flux Shaping on $IF_6\n"
printf "Qos-flux Shaping on $IF_7\n"
printf "Qos-flux Shaping on $IF_8\n"
printf "Qos-flux Shaping on $IF_9\n"


# Add root qdisc
$TC qdisc add dev $IF_1 root handle 1: hfsc default 50
$TC qdisc add dev $IF_2 root handle 1: hfsc default 50
$TC qdisc add dev $IF_3 root handle 1: hfsc default 50
$TC qdisc add dev $IF_4 root handle 1: hfsc default 50
$TC qdisc add dev $IF_5 root handle 1: hfsc default 50
$TC qdisc add dev $IF_6 root handle 1: hfsc default 50
$TC qdisc add dev $IF_7 root handle 1: hfsc default 50
$TC qdisc add dev $IF_8 root handle 1: hfsc default 50
$TC qdisc add dev $IF_9 root handle 1: hfsc default 50

# Add main class, setting interface BANDWIDTH limit

$TC class add dev $IF_1 parent 1: classid 1:1 hfsc ls m2
"${BANDWIDTH}Kbit" ul m2 "${BANDWIDTH}Kbit"

$TC class add dev $IF_2 parent 1: classid 1:1 hfsc ls m2
"${BANDWIDTH}Kbit" ul m2 "${BANDWIDTH}Kbit"

$TC class add dev $IF_3 parent 1: classid 1:1 hfsc ls m2
"${BANDWIDTH}Kbit" ul m2 "${BANDWIDTH}Kbit"

$TC class add dev $IF_4 parent 1: classid 1:1 hfsc ls m2
"${BANDWIDTH}Kbit" ul m2 "${BANDWIDTH}Kbit"

$TC class add dev $IF_5 parent 1: classid 1:1 hfsc ls m2
"${BANDWIDTH}Kbit" ul m2 "${BANDWIDTH}Kbit"
```

```
    $TC class add dev $IF_6 parent 1: classid 1:1 hfsc ls m2
    "${BANDWIDTH}Kbit" ul m2 "${BANDWIDTH}Kbit"

    $TC class add dev $IF_7 parent 1: classid 1:1 hfsc ls m2
    "${BANDWIDTH}Kbit" ul m2 "${BANDWIDTH}Kbit"

    $TC class add dev $IF_8 parent 1: classid 1:1 hfsc ls m2
    "${BANDWIDTH}Kbit" ul m2 "${BANDWIDTH}Kbit"

    $TC class add dev $IF_9 parent 1: classid 1:1 hfsc ls m2
    "${BANDWIDTH}Kbit" ul m2 "${BANDWIDTH}Kbit"


################ Define_ls_flow ########################


# Define_ls_flow <flow id> <min_bw_percent> <label>.
# min_bw_percent is how much bandwidth (in percent) to use for each flow.
#The numbers should add up to exactly 100.

    define_ls_flow_0 11  100 #"ICMP"
    define_ls_flow_0 12  100 #"DNS"
    define_ls_flow_0 20  100 #"Traffic VoIP and Videoconferece"
    define_ls_flow_0 40  100 #"SMTP"
    define_ls_flow_0 41  100 #"IMAP and POP3"
    define_ls_flow_0 45  100 #"HTTP"
    define_ls_flow_0 48  85 #"FTP"
    define_ls_flow_0 49  85 #"SFTP"
    define_ls_flow_0 50  85 #"Default and Torrent" # unclassified traffic
    define_ls_flow_0 54  85 #"HTTPS"



    define_ls_flow_1 11  100 #"ICMP"
    define_ls_flow_1 12  100 #"DNS"
    define_ls_flow_1 20  100 #"Traffic VoIP and Videoconferece"
```

```
define_ls_flow_1 40  100 #"SMTP"
define_ls_flow_1 41  100 #"IMAP and POP3"
define_ls_flow_1 45  100 #"HTTP"
define_ls_flow_1 48  85 #"FTP"
define_ls_flow_0 49  85 #"SFTP"
define_ls_flow_1 50  85 #"Default and Torrent" # unclassified traffic
define_ls_flow_1 54  85 #"HTTPS"



define_ls_flow_2 11  100 #"ICMP"
define_ls_flow_2 12  100 #"DNS"
define_ls_flow_2 20  100 #"Traffic VoIP and Videoconferece"
define_ls_flow_2 40  100 #"SMTP"
define_ls_flow_2 41  100 #"IMAP and POP3"
define_ls_flow_2 45  100 #"HTTP"
define_ls_flow_2 48  85 #"FTP"
define_ls_flow_0 49  85 #"SFTP"
define_ls_flow_2 50  85 #"Default and Torrent" # unclassified traffic
define_ls_flow_2 54  85 #"HTTPS"


define_ls_flow_3 11  100 #"ICMP"
define_ls_flow_3 12  100 #"DNS"
define_ls_flow_3 20  100 #"Traffic VoIP and Videoconferece"
define_ls_flow_3 40  100 #"SMTP"
define_ls_flow_3 41  100 #"IMAP and POP3"
define_ls_flow_3 45  100 #"HTTP"
define_ls_flow_3 48  85 #"FTP"
define_ls_flow_0 49  85 #"SFTP"
define_ls_flow_3 50  85 #"Default and Torrent" # unclassified traffic
define_ls_flow_3 54  85 #"HTTPS"



define_ls_flow_4 11  100 #"ICMP"
```

```
define_ls_flow_4 12  100 #"DNS"
define_ls_flow_4 20  100 #"Traffic VoIP and Videoconferece"
define_ls_flow_4 40  100 #"SMTP"
define_ls_flow_4 41  100 #"IMAP and POP3"
define_ls_flow_4 45  100 #"HTTP"
define_ls_flow_4 48  85 #"FTP"
define_ls_flow_0 49  85 #"SFTP"
define_ls_flow_4 50  85 #"Default and Torrent" # unclassified traffic
define_ls_flow_4 54  85 #"HTTPS"



define_ls_flow_5 11  100 #"ICMP"
define_ls_flow_5 12  100 #"DNS"
define_ls_flow_5 20  100 #"Traffic VoIP and Videoconferece"
define_ls_flow_5 40  100 #"SMTP"
define_ls_flow_5 41  100 #"IMAP and POP3"
define_ls_flow_5 45  100 #"HTTP"
define_ls_flow_5 48  85 #"FTP"
define_ls_flow_0 49  85 #"SFTP"
define_ls_flow_5 50  85 #"Default and Torrent" # unclassified traffic
define_ls_flow_5 54  85 #"HTTPS"



define_ls_flow_6 11  100 #"ICMP"
define_ls_flow_6 12  100 #"DNS"
define_ls_flow_6 20  100 #"Traffic VoIP and Videoconferece"
define_ls_flow_6 40  100 #"SMTP"
define_ls_flow_6 41  100 #"IMAP and POP3"
define_ls_flow_6 45  100 #"HTTP"
define_ls_flow_6 48  85 #"FTP"
define_ls_flow_0 49  85 #"SFTP"
define_ls_flow_6 50  85 #"Default and Torrent" # unclassified traffic
define_ls_flow_6 54  85 #"HTTPS"
```

```
define_ls_flow_7 11  100 #"ICMP"
define_ls_flow_7 12  100 #"DNS"
define_ls_flow_7 20  100 #"Traffic VoIP and Videoconferece"
define_ls_flow_7 40  100 #"SMTP"
define_ls_flow_7 41  100 #"IMAP and POP3"
define_ls_flow_7 45  100 #"HTTP"
define_ls_flow_7 48  85 #"FTP"
define_ls_flow_0 49  85 #"SFTP"
define_ls_flow_7 50  85 #"Default and Torrent" # unclassified traffic
define_ls_flow_7 54  85 #"HTTPS"



define_ls_flow_8 11  100 #"ICMP"
define_ls_flow_8 12  100 #"DNS"
define_ls_flow_8 20  100 #"Traffic VoIP and Videoconferece"
define_ls_flow_8 40  100 #"SMTP"
define_ls_flow_8 41  100 #"IMAP and POP3"
define_ls_flow_8 45  100 #"HTTP"
define_ls_flow_8 48  85 #"FTP"
define_ls_flow_0 49  85 #"SFTP"
define_ls_flow_8 50  85 #"Default and Torrent # unclassified traffic
define_ls_flow_8 54  85 #"HTTPS"



################### FILTERS ###################

# Remember that filters for flows are defined in order of how they
# should match IP packet data.  It's crucial that you match on very
# narrow terms first and leave the broad matches for last.


# IP packet total lenght < 128
filter_for_flow 11 \
    match ip protocol 1 0xff \
    match u16 0x0000 0xff80 at 2
```

```
# DNS (small packets)
# IP dst port == 53
# IP packet total lenth < 128
filter_for_flow 12 \
    match ip dport 53 0xffff \
    match u16 0x0000 0xff80 at 2

# VoIP / Videoconference
filter_for_flow 21 match ip sport 16384 0xffff
filter_for_flow 21 match ip dport 16384 0xffff
filter_for_flow 20 match ip sport 5060 0xffff
filter_for_flow 20 match ip dport 5060 0xffff
filter_for_flow 20 match ip sport 5061 0xffff # TLS
filter_for_flow 20 match ip dport 5061 0xffff # TLS

# SMTP (with and without SSL)
filter_for_flow 40 match ip dport 25 0xffff
filter_for_flow 40 match ip dport 465 0xffff
filter_for_flow 40 match ip dport 587 0xffff

# IMAP (with and without SSL)
filter_for_flow 41 match ip dport 143 0xffff
filter_for_flow 41 match ip dport 993 0xffff

# POP3 (with and without SSL)
filter_for_flow 41 match ip dport 110 0xffff
filter_for_flow 41 match ip dport 995 0xffff

# HTTP
filter_for_flow 45 match ip dport 80 0xffff

# FTP (with and without SSL)
filter_for_flow 48 match ip dport 20 0xffff
filter_for_flow 48 match ip dport 21 0xffff
filter_for_flow 48 match ip dport 989 0xffff
filter_for_flow 48 match ip dport 990 0xffff
```

```
    #SFTP
    filter_for_flow 49 match ip dport 115 0xffff
    filter_for_flow 48 match ip dport 26 0xffff

    # HTTPS
    filter_for_flow 51 match ip dport 443 0xffff
    filter_for_flow 51 match ip sport 443 0xffff

    # Small packets < 128 bytes
    # IP packet total length < 128
    filter_for_flow 13 \
        match u16 0x0000 0xff80 at 2

    # Small packets < 256 bytes
    # IP packet total length < 256
    filter_for_flow 14 \
        match u16 0x0000 0xff00 at 2
}


############ FUNCTION INTERFACE STOP ################

function stop_me {

    # Interface removal settings

    active_qdisc=$(tc qdisc show dev $IF_1| head -n 1 | cut -d" " -f 2)
    if [ -n "$active_qdisc" -a "$active_qdisc" = "hfsc" ]; then
        if [ -z "$1" -o "$1" != "quiet" ]; then
            printf "Qos-flux off shaping on $IF_1\n"
        fi
        $TC qdisc del root dev $IF_1 2>&1 >/dev/null
    fi


    active_qdisc=$(tc qdisc show dev $IF_2| head -n 1 | cut -d" " -f 2)
```

```
if [ -n "$active_qdisc" -a "$active_qdisc" = "hfsc" ]; then
    if [ -z "$1" -o "$1" != "quiet" ]; then
        printf "Qos-flux off shaping on $IF_2\n"
    fi
    $TC qdisc del root dev $IF_2 2>&1 >/dev/null
fi

active_qdisc=$(tc qdisc show dev $IF_3| head -n 1 | cut -d" " -f 2)
if [ -n "$active_qdisc" -a "$active_qdisc" = "hfsc" ]; then
    if [ -z "$1" -o "$1" != "quiet" ]; then
        printf "Qos-flux off shaping on $IF_3\n"
    fi
    $TC qdisc del root dev $IF_3 2>&1 >/dev/null
fi

active_qdisc=$(tc qdisc show dev $IF_4| head -n 1 | cut -d" " -f 2)
if [ -n "$active_qdisc" -a "$active_qdisc" = "hfsc" ]; then
    if [ -z "$1" -o "$1" != "quiet" ]; then
        printf "Qos-flux off shaping on $IF_4\n"
    fi
    $TC qdisc del root dev $IF_4 2>&1 >/dev/null
fi

active_qdisc=$(tc qdisc show dev $IF_5| head -n 1 | cut -d" " -f 2)
if [ -n "$active_qdisc" -a "$active_qdisc" = "hfsc" ]; then
    if [ -z "$1" -o "$1" != "quiet" ]; then
        printf "Qos-flux off shaping on $IF_5\n"
    fi
    $TC qdisc del root dev $IF_5 2>&1 >/dev/null
fi



active_qdisc=$(tc qdisc show dev $IF_6| head -n 1 | cut -d" " -f 2)
if [ -n "$active_qdisc" -a "$active_qdisc" = "hfsc" ]; then
    if [ -z "$1" -o "$1" != "quiet" ]; then
        printf "Qos-flux off shaping on $IF_6\n"
    fi
```

```
        $TC qdisc del root dev $IF_6 2>&1 >/dev/null
    fi

    active_qdisc=$(tc qdisc show dev $IF_7| head -n 1 | cut -d" " -f 2)
    if [ -n "$active_qdisc" -a "$active_qdisc" = "hfsc" ]; then
        if [ -z "$1" -o "$1" != "quiet" ]; then
            printf "Qos-flux off shaping on $IF_7\n"
        fi
        $TC qdisc del root dev $IF_7 2>&1 >/dev/null
    fi


    active_qdisc=$(tc qdisc show dev $IF_8| head -n 1 | cut -d" " -f 2)
    if [ -n "$active_qdisc" -a "$active_qdisc" = "hfsc" ]; then
        if [ -z "$1" -o "$1" != "quiet" ]; then
            printf "Qos-flux off shaping on $IF_8\n"
        fi
        $TC qdisc del root dev $IF_8 2>&1 >/dev/null
    fi


    active_qdisc=$(tc qdisc show dev $IF_9| head -n 1 | cut -d" " -f 2)
    if [ -n "$active_qdisc" -a "$active_qdisc" = "hfsc" ]; then
        if [ -z "$1" -o "$1" != "quiet" ]; then
            printf "Qos-flux off shaping on $IF_9\n"
        fi
        $TC qdisc del root dev $IF_9 2>&1 >/dev/null
    fi
    # Remove existing flows file if it exists
     [ -e "$FLOWS_FILENAME" ] && rm -f "$FLOWS_FILENAME"
}


############ FUNCTION INTERFACE STATUS ################

function status_me {
```

```
 # Interface status settings
 printf "************************ QDISC ******************************\n"
 $TC qdisc show dev $IF_1
 printf "************************ CLASS ******************************\n"
 $TC -s class show dev $IF_1 | grep -v 'tokens:' | grep -v 'lended:'

 # Interface status settings
 printf "************************ QDISC ******************************\n"
 $TC qdisc show dev $IF_2
 printf "************************ CLASS ******************************\n"
 $TC -s class show dev $IF_2 | grep -v 'tokens:' | grep -v 'lended:'

 # Interface status settings
 printf "************************ QDISC ******************************\n"
 $TC qdisc show dev $IF_3
 printf "************************ CLASS ******************************\n"
 $TC -s class show dev $IF_3 | grep -v 'tokens:' | grep -v 'lended:'

# Interface status settings
 printf "************************ QDISC ******************************\n"
 $TC qdisc show dev $IF_4
 printf "************************ CLASS ******************************\n"
 $TC -s class show dev $IF_4 | grep -v 'tokens:' | grep -v 'lended:'

# Interface status settings
 printf "************************ QDISC ******************************\n"
 $TC qdisc show dev $IF_5
 printf "************************ CLASS ******************************\n"
 $TC -s class show dev $IF_5 | grep -v 'tokens:' | grep -v 'lended:'

 # Interface status settings
 printf "************************ QDISC ******************************\n"
 $TC qdisc show dev $IF_6
 printf "************************ CLASS ******************************\n"
 $TC -s class show dev $IF_6 | grep -v 'tokens:' | grep -v 'lended:'

 # Interface status settings
```

```
    printf "************************ QDISC ******************************\n"
    $TC qdisc show dev $IF_7
    printf "************************ CLASS ******************************\n"
    $TC -s class show dev $IF_7 | grep -v 'tokens:' | grep -v 'lended:'

   # Interface status settings
    printf "************************ QDISC ******************************\n"
    $TC qdisc show dev $IF_8
    printf "************************ CLASS ******************************\n"
    $TC -s class show dev $IF_8 | grep -v 'tokens:' | grep -v 'lended:'

    # Interface status settings
    printf "************************ QDISC ******************************\n"
    $TC qdisc show dev $IF_9
    printf "************************ CLASS ******************************\n"
    $TC -s class show dev $IF_9 | grep -v 'tokens:' | grep -v 'lended:'
}

############ FUNCTION INTERFACE FILTERING ################

function filter_me {

    # Interface filtering settings
    printf "************************ FILTER *****************************\n"
    $TC -p filter show dev $IF_1

    # Interface filtering settings
    printf "************************ FILTER *****************************\n"
    $TC -p filter show dev $IF_2

    # Interface filtering settings
    printf "************************ FILTER *****************************\n"
    $TC -p filter show dev $IF_3

    # Interface filtering settings
    printf "************************ FILTER *****************************\n"
    $TC -p filter show dev $IF_4
```

```
    # Interface filtering settings
    printf "************************ FILTER ****************************\n"
    $TC -p filter show dev $IF_5

    # Interface filtering settings
    printf "************************ FILTER ****************************\n"
    $TC -p filter show dev $IF_6

    # Interface filtering settings
    printf "************************ FILTER ****************************\n"
    $TC -p filter show dev $IF_7

    # Interface filtering settings
    printf "************************ FILTER ****************************\n"
    $TC -p filter show dev $IF_8

    # Interface filtering settings
    printf "************************ FILTER ****************************\n"
    $TC -p filter show dev $IF_9
}


case "$1" in
    start)
        start_me
        ;;
    stop)
        stop_me
        ;;
    restart)
        stop_me
        start_me
        ;;
    status)
        status_me
        ;;
```

```
    filter)
        filter_me
        ;;
    *)
        printf "Usage: $0 {start|stop|restart|status|filter}\n" >&2
        exit 1
esac

exit 0
```

Anexo II

Scripts D-ITG

Scripts utilizados na ferramenta D-ITG para os testes executados com os comandos ./ITGSend (host cliente) e ITGRecv (host servidor) de acordo com cada tipo de fluxo.

A) Fluxo Elefante

- Script para gerar fluxos que simulem aplicações de Dropbox:

```
cat > script\_2 <<END

> -a 10.0.0.X -t 600000 -rp 17500 -T TCP -v 0.504 0.016
> -a 10.0.0.X -t 600000 -rp 17500 -T TCP -v 0.438 0.014
> -a 10.0.0.X -t 600000 -rp 17500 -T TCP -v 0.426 0.010
> END
```

- Foi executado em cada host cliente:

```
./ITGSend script\_2 -x receiver(número do host).log
```

B) Fluxo Guepardo

- Script para gerar fluxos que simulem aplicações de Backup:

```
cat > script\_10 <<END
```

```
> -a 10.0.0.X -t 600000 -rp 82 -T TCP -C 8333 -u 4000 16000

> END
```

- Foi executado em cada host cliente:

```
./ITGSend script\_10 -x receiver(número do host).log
```

C) Fluxo Alfa

- Script para gerar fluxos que simulem aplicações de Streaming HD intermitente:

```
cat > script\_3 <<END

> -a 10.0.0.X -t 600000 -rp 443 -T TCP -C 166 -w 13.795 17.370

>END
```

- Foi executado em cada host cliente:

```
./ITGSend script\_3 -x receiver(número do host).log
```

Anexo III

Comandos do TC Linux para Habilitar Algoritmos de QoS

Todas as configurações e as explicações sobre os parâmetros foram retirados do site: http://man7.org/linux/man-pages/man8/tc.8.html.

1) Algoritmo HTB:

```
tc qdisc add dev DEV ( parent classid | root) [ handle major: ]
htb [default minor-id ]
```

```
tc class add dev DEV parent major:[minor] [ classid major:minor ]
htb rate RATE [ ceil RATE ] burst BYTES [ cburst BYTES ] [ prio PRIORITY ]
```

a) dev - Neste parâmetro é adicionado a interface de rede.

b) tc qdisc - Temos o parâmetro parent major: minor | root usado para aplicação na raiz de uma interface ou dentro de uma classe existente; handle major usado para herdar um identificador no HTB, sendo opcional, mas útil se as classes forem criadas dentro da qdisc; default minor-id todo tráfego não rastreado é enviado para este ID.

c) tc class - Temos o parâmetro parent major:minor o qual é definida a classe pai; o comando classid major:minor configura a classe filha com sua determinada identificação (ID), onde podemos gerar diversas classes filhas.

d) prio - É designado para classes com o campo de prioridade executando do menor para o maior.

e) rate - Tem o objetivo de definir a largura de banda máxima da classe e de seus filhos.

f) ceil - Esse parâmetro define a largura de banda máxima que uma classe pode encaminhar, porém, somente caso seu pai tenha largura de banda sobrando.

g) burst - É o parâmetro de rajadas definido em bytes, será acionado quando o ceil ultrapassar o limite bem como o parâmetro rate estiver em excesso.

h) cburst - Tem o objetivo de estabelecer a quantidade de bytes máximo do tipo de fluxo em rajadas de acordo com a largura de banda que a interface de rede pode suportar.

2) Algoritmo HFSC:

```
tc qdisc add dev hfsc [ default CLASSID ]

tc class add dev hfsc [ [ rt SC ] [ ls SC ] | [ sc SC ] ] [ ul SC ]
```

a) dev - Neste parâmetro definimos a interface de rede.

b) rt - É definido como configuração de aplicações de tempo real (real time) no HFSC. Este componente contribui para garantir classes rígidas sobre o atraso máximo até que um pacote seja enviado.

c) ls - Conhecido como compartilhamento de link (link sharing) no HFSC. Se o link da largura de banda estiver saturado, o compartilhamento de

link deve avisar ao HFSC qual deve ser o valor de banda disponível para cada classe.

d) ul - Parâmetro conhecido como limite superior no HFSC. Este componente configura a largura de banda máxima que o compartilhamento de link pode enviar. Sempre que configuramos o ls, será necessário habilitar o parâmetro ul.

g) sc - Esse parâmetro tem a função de habilitar os componentes rt e o ls configurados ao mesmo tempo no TC Linux quando utilizamos o algoritmo HFSC.

h) SC - parâmetro conhecido como curva de serviço, o qual pode ser configurado em qualquer um dos parâmetros rt, ls, sc, ul. O parâmetro SC deve ser configurado planejando o atraso máximo para determinada quantidade de trabalho (fluxos na rede), ou como uma largura de banda atribuída por determinado período de tempo.

3) Algoritmo FQ_Codel:

tc qdisc add dev fq_codel [limit PACKETS] [flows NUMBER] [target TIME] [interval TIME] [quantum BYTES] [ecn | noecn]}

a) dev - Neste parâmetro configuramos a interface de rede.

b) limit - Este parâmetro configura o limite de pacotes no tamanho da fila. Caso esse limite seja ultrapassado, os pacotes serão descartados.

c) flows - É conhecido como o número de fluxos que os pacotes de entrada podem ser classificados. Como este algoritmo é estocástico, vários fluxos podem ser separados no mesmo slot. Aqueles fluxos mais novos têm prioridade sobre os mais antigos.

d) target - Este componente configura o atraso mínimo da fila. Esse

atraso mínimo é detectado pelo atraso da fila mínima que os pacotes possam experimentar.

e) interval - Este parâmetro tem como primitiva de garantir que o atraso mínimo calculado não se torne excessivo, podendo gerar valores altos de jitter. Ele deve ser acionado no pior caso de RTT (Round Trip Time) através de situações de congestionamento ou gargalo na rede, pois, é necessário dar tempo suficiente para reação do algoritmo FQ_Codel tentar amenizar esses problemas.

f) quantum - Este componente é considerado como o número de bytes utilizados como déficit para o algoritmo Codel. O padrão é determinado como 1514 bytes, o que corresponde à Ethernet MTU somado ao comprimento do cabeçalho de hardware de 14 bytes.

g) ecn | noecn - Quando configurado, será utilizado para marcar pacotes em vez de descartá-los (gerando perda de pacotes). Se o componente ecn for configurado, noecn pode ser usado para desligá-lo e vice-versa. Ao contrário do algoritmo Codel, o componente ecn, no algoritmo FQ_Codel, é ativado por padrão.

Anexo IV

Exemplo de Configurações Estáticas Utilizadas neste Trabalho

1) Algoritmo HTB:

a) Basic - Na classe filho (1:10) foi configurado o limite de 700000 Kbit/s (700 MBit/s). Nas configurações de ceil e prio, foram definidas 800000 kbit/s (800 MBit/s) e o valor 0 (zero), respectivamente.

b) Default: Na classe filho (1:20) foi configurado o limite de 900000 kbit/s (900 MBit/s). Nas configurações ceil e prio, foram determinadas 950000 kbit/s (950 MBit/s) e o valor 1 (um), respectivamente. Abaixo temos um exemplo de configuração adotada:

```
tc qdisc add dev s2-eth1 root handle 1:0 htb default 20

tc class add dev s2-eth1 parent 1:0 classid 1:1 htb rate 1000mbit

tc class add dev s2-eth1 parent 1:1 classid 1:10 htb rate 700000kbit
ceil 800000kbit  prio 0   \# Basic}

tc class add dev s2-eth1 parent 1:1 classid 1:20 htb rate 900000kbit
ceil 950000kbit  prio 1   \# Default}
```

2) Algoritmo HFSC:

a) Basic - Na classe filho (1:10) foi configurado a curva de serviço (sc) umax com 1500 bytes, dmax 53 ms, largura de banda mínima de 900000 Kbit/s (900 MBit/s) e limite superior de 950000 Kbit/s (950 MBit/s), respectivamente.

b) Default - Na classe filho (1:20) foi configurado a curva de serviço (sc) umax com 1500 bytes, dmax 30 ms, largura de banda mínima de 700000 Kbit/s (700 MBit/s) e limite superior de 800000 Kbit/s (950 MBit/s). Abaixo temos um exemplo de configuração adotada:

```
tc qdisc add dev s4-eth3 root handle 1: hfsc default 10

tc class add dev s4-eth3 parent 1: classid 1:1 hfsc sc rate 1000000kbit ul
rate 1000000kbit

tc class add dev s4-eth3 parent 1:1 classid 1:10 hfsc sc umax 1500b dmax 53ms
rate 900000kbit ul rate 950000kbit \# Basic_0

tc class add dev s4-eth3 parent 1:1 classid 1:20 hfsc sc umax 1500b dmax 30ms
rate 700000kbit ul rate 800000kbit  \# Basic_1
```

3) Algoritmo FQ_Codel:

a) Default: Nesta configuração é adotado no parâmetro limit o valor de 10240 pacotes e na métrica flows o valor de 1024. Além disso, foram inseridos os valores de target de 5 milissegundos, interval de 100 milissegundos, quantum de 1514 bytes e habilitado com notificação ecn. Abaixo podemos visualizar um exemplo de como foi adotado as configurações:

```
tc qdisc add dev s2-eth1 root fq\_codel
```