UNIVERSIDADE FEDERAL DE MINAS GERAIS
Instituto de Ciências Exatas
Programa de Pós-Graduação em Ciência da Computação

Gustavo Pantuza Coelho Pinto

Grafos como uma primitiva do plano de controle para análise e gerenciamento de Redes Definidas por Software

Belo Horizonte
2015

Gustavo Pantuza Coelho Pinto

Grafos como uma primitiva do plano de controle para análise e gerenciamento de Redes Definidas por Software

Versão Final

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Minas Gerais, como requisito parcial à obtenção do título de Mestre em Ciência da Computação.

Orientador: Luiz Filipe Menezes Vieira

Belo Horizonte
2015

Pinto, Gustavo Pantuza Coelho.

P659g Grafos como uma primitiva do plano de controle para análise e gerenciamento de redes definidas por software [recurso eletrônico] / Gustavo Pantuza Coelho Pinto. Belo Horizonte – 2015.

1 recurso online (106 f. il., color.) : pdf.

Orientador: Luiz Filipe Menezes Vieira.

Dissertação (Mestrado) - Universidade Federal de Minas Gerais, Instituto de Ciências Exatas, Departamento de Ciência da Computação.

Referências: f. 101-105.

1. Computação – Teses. 2. Software - Arquitetura – Teses. 3. Arquitetura de redes de computador – Teses. 4. Sistemas distribuídos – Teses. 5. Teoria dos grafos – Teses. I. Vieira, Luiz Filipe Menezes. I. Universidade Federal de Minas Gerais Exatas, Departamento de Ciência da Computação. III. Título.

CDU 519.6*32(043)

Ficha catalográfica elaborada pela bibliotecária Irénquer Vismeg Lucas Cruz CRB 6/819 - Universidade Federal de Minas Gerais - ICEx

UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FOLHA DE APROVAÇÃO

Grafos como uma primitiva do plano de controle para análise e gerenciamento de redes definidas por software

GUSTAVO PANTUZA COELHO PINTO

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:

PROF. LUIZ FILIPE MENEZES VIEIRA - Orientador
Departamento de Ciência da Computação - UFMG

PROF. DANIEL FERNANDES MACEDO
Departamento de Ciência da Computação - UFMG

PROF. DORGIVAL OLAVO GUEDES NETO
Departamento de Ciência da Computação - UFMG

PROF. MARCOS AUGUSTO MENEZES VIEIRA
Departamento de Ciência da Computação - UFMG

Belo Horizonte, 06 de julho de 2015.

Este trabalho é dedicado à comunidade científica da Ciência da Computação, no Brasil e no mundo. Em especial, dedico-o aos meus mestres, professores que, de diferentes maneiras, foram mentores fundamentais na minha formação e na construção da minha trajetória acadêmica.

Agradecimentos

Agradeço aos meus pais pelo incentivo constante e por acreditarem no meu esforço ao longo desta jornada.

Agradeço ao Departamento de Ciência da Computação da UFMG pela oportunidade de desenvolver este projeto ao lado de tantas mentes brilhantes. Ao laboratório *WiNet*, agradeço pela infraestrutura e pelo ambiente de pesquisa que tornaram possível a realização deste trabalho, bem como pela valiosa troca de experiências com seus pesquisadores.

Agradeço, em especial, ao colega de pesquisa e amigo Frederico Martins pela dedicação em compartilhar conhecimento e experiência, contribuições fundamentais para minha formação como cientista da computação. Estendo meus agradecimentos aos colegas Bruno Pereira, Erik Britto, Henrique Moura e Rodrigo Caetano pelas pesquisas, discussões e soluções construídas em conjunto.

Agradeço a Fabiana Duffles, minha companheira, pelo apoio incondicional durante todo o desenvolvimento desta pesquisa.

A todos vocês, meu sincero obrigado.

"*Live long and prosper!*"
(Mr. Spock)

Resumo

As redes definidas por software (SDN) representam uma arquitetura emergente dinâmica, flexível, gerenciável e de baixo custo, condizente com a dinamicidade das aplicações modernas. Essa arquitetura desacopla o plano de controle do plano de dados. Redes tipicamente são representadas em forma de grafos. Em função disso, esse trabalho apresenta um modelo de representação da rede em grafos através do plano de controle de um controlador SDN. O protocolo *OpenFlow* é um meio para construção de soluções em SDN. Esse trabalho é baseado no POX, um controlador SDN compatível com dispositivos OpenFlow. Essa abordagem em grafos fornece uma visão global da rede em tempo real e com consistência. Os experimentos demonstram que o grafo proposto representa de maneira fiel o estado da rede assim como suas mudanças em função dos eventos ocorridos ao longo do tempo dentro da rede, facilitando assim, o gerenciamento em Redes Definidas por Software.

Palavras-chave: sistemas distribuídos; redes definidas por software; plano de dados; plano de controle; arquitetura de software.

Abstract

Software Defined Networking (SDN) is an emergent architecture that is dynamic, flexible, manageable and with low cost, consistent with the dynamics of modern applications. This architecture decouples the control plane from the data plane. Tipically, networks are represented as graphs. Based on it, this master thesis presents a network representation model using a graph for the control plane of a SDN controller. Our SDN solution adopts the OpenFlow protocol. This project is based on POX, a SDN controller compatible with OpenFlow devices. The graph aproach provides a global view of the network in real time and with consistency. The experiments show that the graph is a reliable representation of the real network and its events occurring over time in the network, thus easing the management in Software Defined Networking.

Keywords: distributed systems; software defined networking; data plane; control plane; software design.

Lista de Figuras

Lista de Tabelas

Lista de abreviações e siglas

ARP	*Address Resolution Protocol*
BGP	*Border Gateway Protocol*
CCN	*Content-Centric Networking*
CDN	*Content Delivery Network*
CPU	*Central Processing Unit*
DSL	*Domain-Specific Language*
GENI	*Global Environment for Network Innovations*
HTTP	*Hypertext Transfer Protocol*
ICN	*Information-Centric Network*
IP	*Internet Protocol*
LIME	*Live Migration of Ensembles*
MIB	*Management Information Base*
MPLS	*Multi-Protocol Label Switching*
NIB	*Network Information Base*
NOX	*Network Operating System*
P2P	*Peer-to-Peer*
POX	Controlador SDN de código aberto baseado em Python
RON	*Resilient Overlay Network*
SDN	*Software Defined Networking*
SNMP	*Simple Network Management Protocol*
SSL	*Secure Socket Layer*
TCP	*Transmission Control Protocol*
POP	*Point of Presence*
UDP	*User Datagram Protocol*

Sumário

Capítulo 1

Introdução

Este capítulo apresenta uma introdução ao problema solucionado pela presente dissertação de mestrado. Primeiramente as motivações e a definição do problema são apresentados. Em seguida, a proposta de solução é explanada e uma breve discussão sobre o estado da arte é descrita.

1.1 Motivação

Redes definidas por software (SDN) separam o plano de dados do plano de controle [25]. Os ambientes de programação projetados para prover aplicações em SDN são chamados de controladores SDN. Eles são conhecidos também como sistemas operacionais de rede, pois criam uma camada que isola o controle do acesso físico dos elementos de rede através de uma interface padronizada.

Aplicações em rede executam algoritmos em grafos. Em muitos casos essa computação é feita em diferentes nós da rede de maneira repetitiva. Em função da natureza logicamente centralizada com visão global da topologia da rede, o plano de controle possibilita minimizar a quantidade de aplicações computando as mesmas informações ou trocando mensagens na rede.

1.2 Problema

Em sua maioria, as aplicações em Redes Definidas por *Software* necessitam de uma visão topológica da rede. Uma visão global é um dos principais aspectos do paradigma [9]. Grafos são uma modelagem direta para representar de maneira natural e precisa a topologia de uma rede. Em função disso, grafos deveriam ser um recurso básico, uma

premissa em controladores SDN para representar a rede.

A representação em grafos pode ser útil para módulos internos de um controlador ou até para serviços/aplicações externos que dependam de informações topológicas ou do estado da rede.

1.3 Contribuições científicas

O presente trabalho apresenta uma abstração da rede na forma de grafos para o gerenciamento de redes através do plano controle, possibilitando automatizar detecção de falhas e provisionamento para um grafo dinamicamente atualizado. Uma implementação em um sistema utilizando OpenFlow [41] e sua avaliação experimental são apresentados como prova de conceito.

Um módulo dentro do controlador SDN armazena o grafo que representa diretamente a rede. Algoritmos em grafos podem ser computados uma única vez e seus resultados armazenados para consultas posteriores por outros módulos.

A interação de outros módulos com o grafo pode ser encapsulada e semanticamente bem definida em uma interface padronizada. Essa implementação pode ser adaptada para diferentes sistemas, de maneira que o módulo pode utilizar de recursos como memória local, banco de dados remoto, ser distribuído, ter controle de concorrência, paralelismo e outras características relevantes para um determinado cenário de utilização.

1.4 Estado da arte

A ideia de visão topológica da rede está presente em vários controladores SDN. O controlador NOX [24] trabalha a topologia da rede baseando-se em eventos. Um banco de dados distribuído é proposto no controlador Onix [36]. Em [30] um sistema com regras de predição estabelecem essa abstração. Linguagens de domínio específico (DSL) como o Frenetic [20] e o Pyretic [42] permitem a recuperação de informações topológicas da rede. Em [52] uma API em grafos é apresentado dentro do contexto de computação na nuvem. Nossa proposta apresenta a avaliação da abordagem em grafos e sua implementação no controlador POX [51], de código aberto e voltado para pesquisa.

1.5 Organização do texto

O presente projeto de dissertação, primeiramente, apresenta uma fundamentação teórica sobre as Redes Definidas por *Software*, o protocolo OpenFlow e as possíveis arquiteturas de implementação. Um capítulo sobre os trabalhos/pesquisas relacionados comparam e descrevem soluções em relação à abordagem em grafos. Em seguida, todo o projeto de implementação e como a solução foi planejada é apresentado. Decisões de projeto como a adaptação do controlador e como o grafo foi modelado.

O ambiente de experimentos (*testbed*) e simulação é apresentado no capítulo 6. Em seguida, são apresentados os resultados das análises executadas sobre os experimentos. Uma avaliação da solução como um todo.

Ao final são descritos os trabalhos futuros do projeto de dissertação e uma conclusão sobre os resultados obtidos ao longo do trabalho.

Capítulo 2

Fundamentação teórica

Esse capítulo apresenta a fundamentação teórica necessária sobre o paradigma das Redes Definidas por Software, que é o contexto do presente trabalho. São apresentados o protocolo OpenFlow e diversas arquiteturas utilizando esse paradigma.

2.1 Redes definidas por software (SDN)

2.1.1 Definição

SDN é um modelo de implementação de redes em que separa-se o plano de controle (decisões) do plano de encaminhamento (comutação de pacotes). O protocolo OpenFlow [41] é uma proposta para se criar soluções em SDN. Essa separação dos planos de controle e de dados torna o funcionamento da rede mais flexível.

As motivações das redes definidas por software vem desde o surgimento das redes *overlay* [12], às quais as redes são definidas sobre outras redes, como camadas isoladas. O paradigma de Redes definidas por software abre a possibilidade de se desenvolver novas aplicações que controlem os elementos de comutação de uma rede física de maneiras impensadas no passado [25].

As aplicações modernas são muito dinâmicas. Quando se trata de um ambiente distribuído, por exemplo em um datacenter, essas aplicações demandam que o funcionamento de sua rede seja também dinâmico e flexível às necessidades das aplicações. SDN permite satisfazer essas condições de funcionamento.

Novos protocolos, servições e redes podem ser facilmente desenvolvidos através do paradigma das Redes definidas por software. Ao tornar os elementos de comutação de uma rede programáveis, a experimentação torna-se mais direta e independente dos fabricantes de *hardware* e equipamentos de rede. Isso permite que pesquisadores façam

experimentações e gerem inovações na área de Redes de computadores [41].

A Internet é uma rede de redes de computadores madura. No entanto possui muitas deficiências. Para resolver o problema da evolução da Internet, tornar os equipamentos programáveis é uma boa abordagem. Muitas redes *overlay* como *PlanetLab* [50] e *GENI* [7] tentaram resolver o problema mas a necessidade de alteração nos equipamentos de rede fizeram com que tivessem baixa aceitação. Foi nesse ponto que a solução OpenFlow em SDN acertou.

2.1.2 Plano de dados

O plano de dados é responsável pelo encaminhamento de pacotes. Esse encaminhamento pode ser implementado através de *hardware* comum em roteadores e comutadores. O encaminhamento de pacotes consiste em executar algumas operações como alterar cabeçalhos dos pacotes, descartá-los e encaminhar para alguma porta específica do equipamento.

2.1.3 Plano de controle

O plano de controle consiste em tomar as decisões de como as operações do plano de dados serão executadas. Como uma entidade separada do plano de dados, o plano de controle, para tomar as decisões, precisa ter uma visão topológica e global da rede. A visão global pode levar a entender uma entidade centralizada. No entanto o plano de controle pode ser distribuído. Decisões como roteamento, *firewall*, priorização de pacotes são responsabilidade do plano de controle. O plano de controle tem uma natureza de sistemas de tempo real.

2.1.4 Características

As Redes definidas por software tornam a rede programável. Essa característica dá flexibilidade na administração da rede. O plano de controle isolado, cria uma entidade

logicamente centralizada. A evolução das redes em relação às aplicações é simplificada. Novos experimentos em redes podem ser criados da mesma forma como se cria novos algoritmos e aplicações. Isso torna essa evolução menos custosa, tanto tecnicamente quanto financeiramente. SDN é apenas um modelo, um novo paradigma em redes de computadores.

2.2 OpenFlow

2.2.1 Definição

Em 2008 o protocolo OpenFlow foi publicado. Ele permitiu que pesquisadores pudessem criar experimentos com novos protocolos em redes convencionais [41]. O OpenFlow foi criado como um padrão aberto, o que permite que todos os fabricantes de equipamentos de redes possam habilitar seus produtos a esse padrão.

O protocolo consiste em uma interface de programação para o comutador. Assim, um programador pode, através de um programa, controlar a forma como um comutador executa seu encaminhamento de pacotes. De uma maneira bem clara, o protocolo OpenFlow separa o plano de dados do plano de controle, fazendo com que soluções SDN possam ser criadas e experimentadas. Por ser uma solução de baixo custo, o OpenFlow obteve boa aceitação na academia e no mercado, dado o volume de empresas e pesquisas relacionadas ou que utilizam o protocolo.

2.2.2 Componentes

Na arquitetura estabelecida pelo protocolo OpenFlow existem dois papéis principais. O controlador e o comutador OpenFlow. Uma separação baseada no modelo das Redes definidas por software (SDN) conforme pode ser visto na figura 2.1.

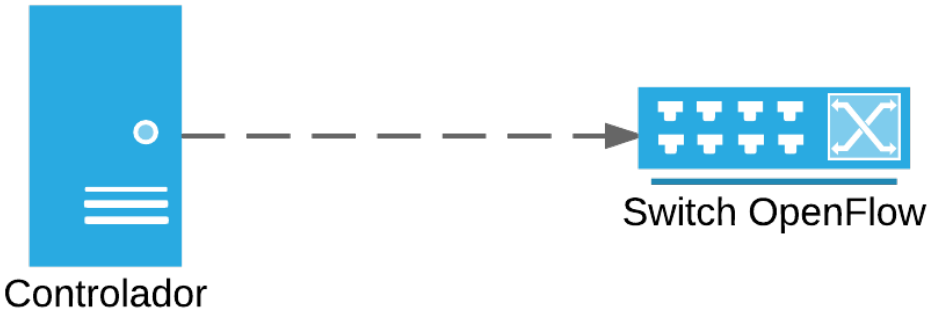

Figura 2.1: Divisão da arquitetura OpenFlow

2.2.3 Arquitetura do Comutador

O comutador OpenFlow pode ser dividido em três principais partes. A primeira é o canal de comunicação seguro com o controlador. A segunda é a interface do protocolo OpenFlow que permite ao controlador controlar o comutador. A terceira é sua tabela de fluxos.

O canal de comunicação seguro tem a garantia de confiabilidade na troca de mensagens entre o controlador e o comutador através do protocolo SSL (Secure Socket Layer). Isso adiciona proteção à rede em relação a ataques de elementos mal intencionados [56].

A interface do protocolo OpenFlow é a padronização das mensagens enviadas pelo controlador ao comutador de modo a definir o comportamento do encaminhamento de pacotes. Ou seja, é o conjunto de possíveis instruções para se controlar de maneira genérica o plano de dados de qualquer *hardware* de redes habilitado ao padrão OpenFlow.

A tabela de fluxos é composta por regras. Cada regra consiste em ações associadas à fluxos. Através dessa tabela o comutador executa o encaminhamento de pacotes. As entradas dessa tabela são atualizadas pelo controlador.

2.2.4 Fluxos

Um fluxo é a representação de um ou mais pacotes em função de suas características. Essas características variam de acordo com os campos que definem um fluxo. Os fluxos são genéricos. Um fluxo pode caracterizar pacotes que ainda não passaram pelo comutador mas que possuem as mesmas características que compõem aquele fluxo. Sendo assim, a ação associada ao fluxo é aplicada a esses novos pacotes.

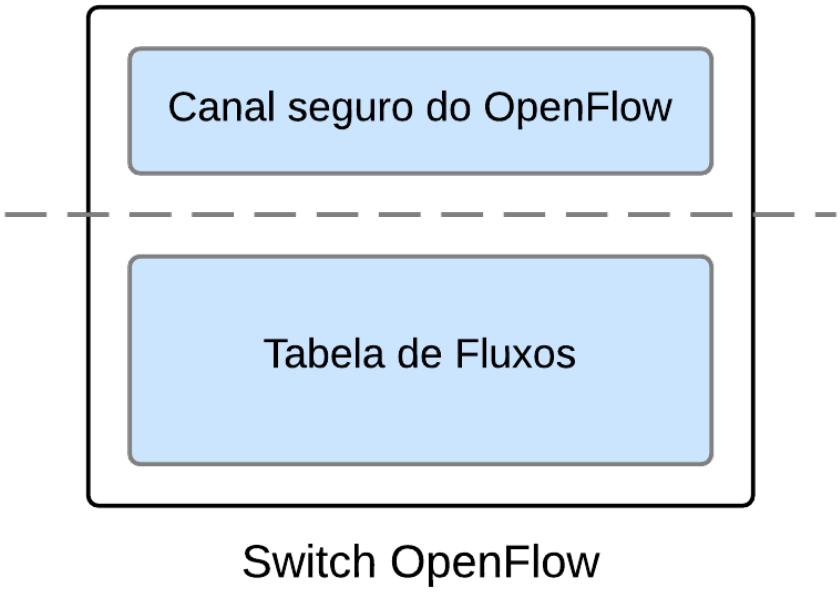

Figura 2.2: Arquitetura do comutador OpenFlow

Cabeçalho	**Contadores**	**Ações**	**Prioridade**
porta de ingresso=5	55635 bytes	Encaminhar porta=8	100
Endereço ip=192.168.1.42 porta=80	4032 bytes	Rescrita ip=192.168.1.100	500
Protocolo IP=UDP	100 bytes	Drop	700

Tabela 2.1: Tabela de fluxos simplificada

A tabela de fluxos dentro do comutador OpenFlow identifica os fluxos para que o plano de dados execute ações sobre os pacotes que são pertencentes àquele fluxo. A tabela 2.1 apresenta de maneira simplificada a tabela de fluxos dentro do comutador OpenFlow.

2.2.5 Campos para definição de fluxos

Os campos que definem fluxos se estendem da camada 1 até a camada 4 da pilha TCP/IP. A figura 2.3 apresenta os campos e suas respectivas camadas.

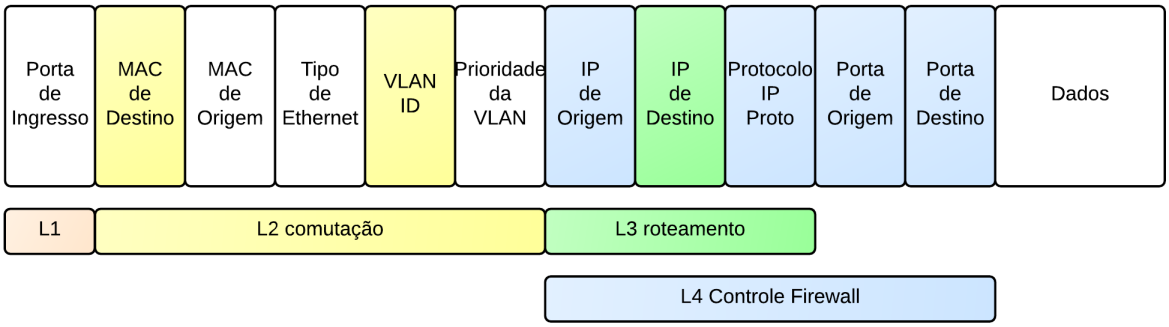

Figura 2.3: Campos para definição de fluxos OpenFlow

Quando um novo pacote ingressa no comutador OpenFlow esses campos são preenchido e encaminhado para o controlador. Através da análise dessas informações do fluxo o controlador envia uma mensagem ao comutador instalando/atualizando regras na tabela de fluxos.

Esse mecanismo de rotulagem e identificação de tráfego e pacotes (fluxos) é inspirado no MPLS (*Multi-protocol Label Switching*) [14].

2.2.6 Ações

As ações (*actions*) são aplicadas aos fluxos especificados na tabela de fluxos do comutador. Ações como encaminhamento, atualização de cabeçalho e rejeição podem ser aplicadas aos pacotes pertencentes àquele fluxo.

A especificação do protocolo OpenFlow [45] descreve todas as possíveis ações que podem ser aplicadas. A lista abaixo apresenta algumas ações:

- Forwarding
- Drop
- Set
- Strip
- Copy-in
- Copy-out
- Push
- Pop
- Dec

Essas ações são aplicadas a cada novo pacote pertencente ao algum determinado fluxo pertencente à tabela de fluxos.

2.2.7 Controlador

O controlador é a entidade na rede que representa o plano de controle dentro da arquitetura OpenFlow. Apesar de ser um entidade logicamente centralizada, ele pode ser distribuído. O controlador é um software que se conecta de maneira segura ao comutador OpenFlow. Através da interface de programação do protocolo o controlador manipula a tabela de fluxos do comutador.

Em função dessa arquitetura, é possível ter visão e controle de estado global da rede. Como o protocolo OpenFlow especifica que o comutador armazena alguns dados estatísticos, por exemplo de portas e fluxos, o controlador pode tomar decisões ou executar análises sobre a rede baseando-se nessas estatísticas.

O controlador exerce um papel importante para os serviços em execução dentro da rede. Outros servidores rodando aplicações podem fazer requisições e troca de mensagens com o controlador da rede, de modo a obter informações topológicas ou de estado da rede.

Com o plano de controle isolado em uma aplicação logicamente centralizada, um programador tem de lidar com problemas típicos de desenvolvimento de software e sistemas distribuídos como:

- Tolerância a falhas
- Persistência
- Eficiência
- Design de implementação
- Debugging
- Testes

Existem vários softwares controladores *open source* no mercado que podem ser utilizados em produção em para desenvolvimento de pesquisas:

- Biblioteca Libfluid para criação de aplicações/controladores em SDN [38]
- Nox Controller [43]
- Beacon [5]
- Pox Controller [51]
- Ryu [57]

2.3 Arquiteturas

Esse capítulo apresenta diversas arquiteturas possíveis através do protocolo OpenFlow. O protocolo foi desenvolvido de uma maneira bem aberta para que pudesse compreender diversas topologias de rede.

2.3.1 Topologia simples

Uma topologia simples pode ser vista na figura 2.4 onde um controlador manipula o plano de dados através de um comutador OpenFlow que possui 3 computadores (*hosts*).

Nesse cenário, até mesmo resolução ARP *(Address Resolution Protocol)* ficam a cargo do controlador. A tabela ARP não é resolvida no comutador, mesmo todos as máquinas *(hosts)* estando na mesmo sub-rede. Todo as decisões são feitas no plano de controle, logo o controlador é quem manipula e atualiza a tabela de fluxos com as regras para implementar a resolução de pacotes ARP.

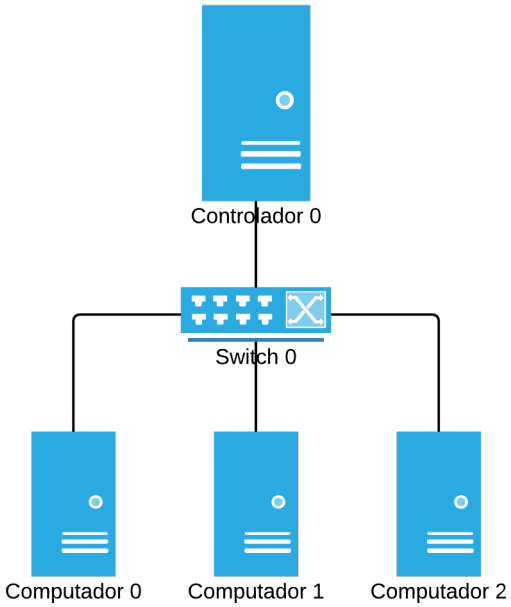

Figura 2.4: Topologia simples de redes de computadores

2.3.2 Um controlador para n comutadores

O mesmo controlador pode gerenciar vários comutadores OpenFlow. Essa abordagem permite que o controlador tenha uma visão global da topologia da rede. Ele pode coletar informações estatísticas e ter controle sobre o estado da rede em tempo real. A figura 2.5 apresenta essa topologia.

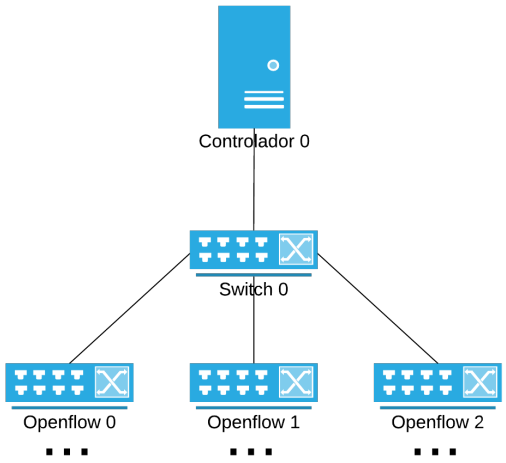

Figura 2.5: Topologia com um controlador e n comutadores

É importante notar que os comutadores podem estar conectados uns aos outros. A figura 2.6 mostra um exemplo em que dois comutadores OpenFlow estão conectados entre si.

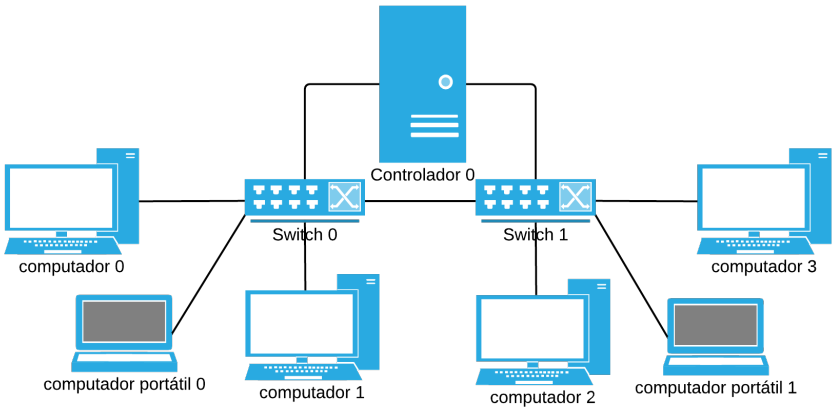

Figura 2.6: Topologia em que os comutadores OpenFlow estão diretamente conectados

2.3.3 *n* Controladores para *n* comutadores

Diversas sub-redes podem ter seus próprios controladores. Essa topologia configura um cenário em que não há visão global da rede por apenas um controlador. Para que esse controle global existisse seria necessário que esses controladores trocassem mensagens sobre a topologia da rede. A figura 2.7 apresenta um exemplo dessa topologia.

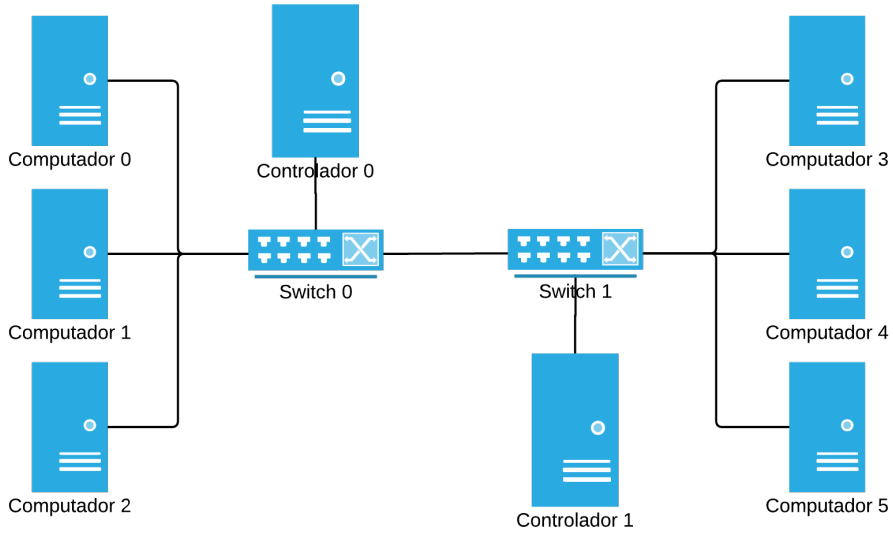

Figura 2.7: Topologia com n controladores e n comutadores

2.3.4 Controlador distribuído

Um dos fatores mais importantes do protocolo OpenFlow foi permitir que uma arquitetura distribuída pudesse ser aplicada ao controlador da rede. Sem essa possibilidade, o controlador se torna um gargalo na rede, pois à medida que o número de pacotes aumenta, o volume de trabalho do controlador cresce. Através de uma arquitetura distribuída, o controlador pode dividir a carga de trabalho com outros controladores. É importante ressaltar que mesmo distribuído, a entidade controlador é tratada como logicamente centralizada.

Conforme apresentado na figura 2.8, em uma arquitetura distribuída, os diversos servidores que compõem o controlador devem estabelecer um canal de comunicação para que possam trocar mensagens e tomar decisões de controle da rede de maneira global baseando-se nas informações contidas em cada um dos servidores.

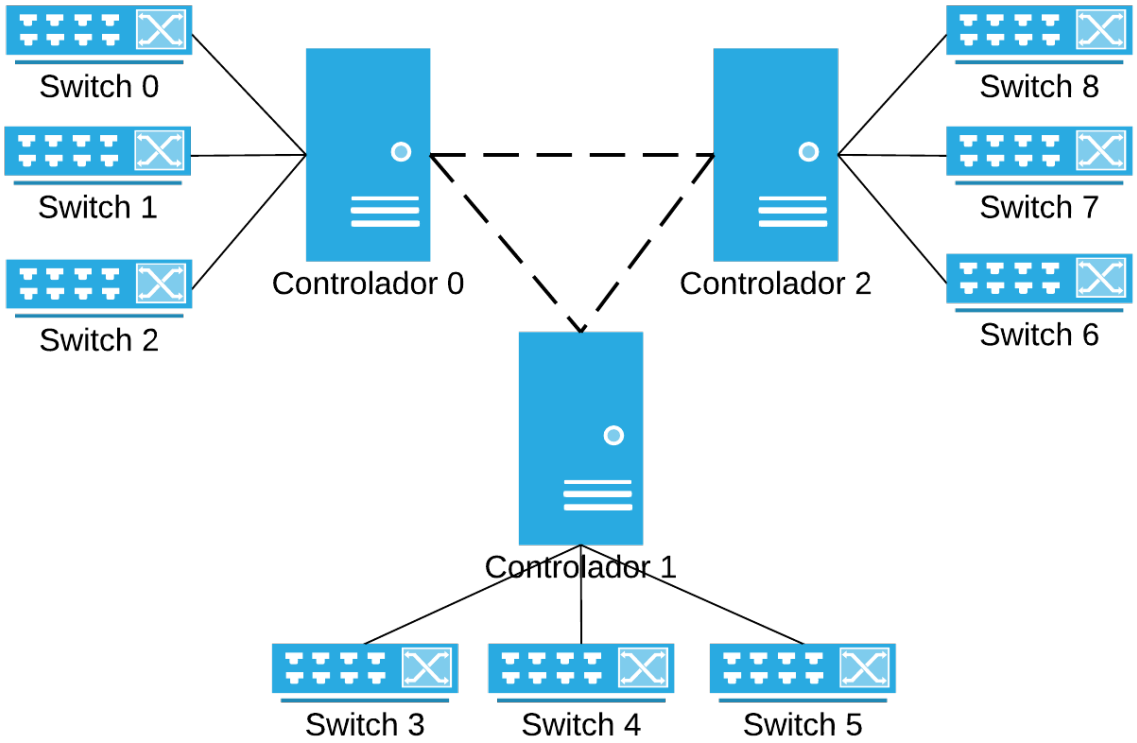

Figura 2.8: Topologia de um controlador distribuído

O protocolo OpenFlow, a partir de sua versão 1.3 permite que o comutador seja configurado para trabalhar com controladores distribuídos. Quando operam no modo réplica, os controladores recebem as mesmas mensagens encaminhadas pelo comutador. Distribuir o controlador torna a rede mais confiável, dado que se um controlador cair os comutadores continuam operando em modo OpenFlow.

Os controladores podem também operar no modo mestre/escravo (*Master/slave*). Nesse cenário, um dos controladores é eleito mestre. Ao configurar o controlador mestre, o comutador OpenFlow altera todos os demais controladores para o modo escravo. Em qualquer instante, outro controlador pode solicitar alteração para mestre, recomeçando assim, o processo de eleição.

2.3.5 Arquitetura para a Internet

Uma arquitetura foi proposta por [4] em que SDN, através do OpenFlow, fosse aplicada a arquitetura da Internet. A figura 2.9 apresenta uma arquitetura de domínios ao longo da Internet. Nesse cenário, os comutadores internos aos domínios de rede cui-

dam do encaminhamento de pacotes no núcleo da rede. Já os comutadores de borda, são responsáveis pela comunicação entre domínios *(Interdomain communication)*. Essa comunicação de borda pode ocorrer entre serviços, ou seja, na camada de aplicação.

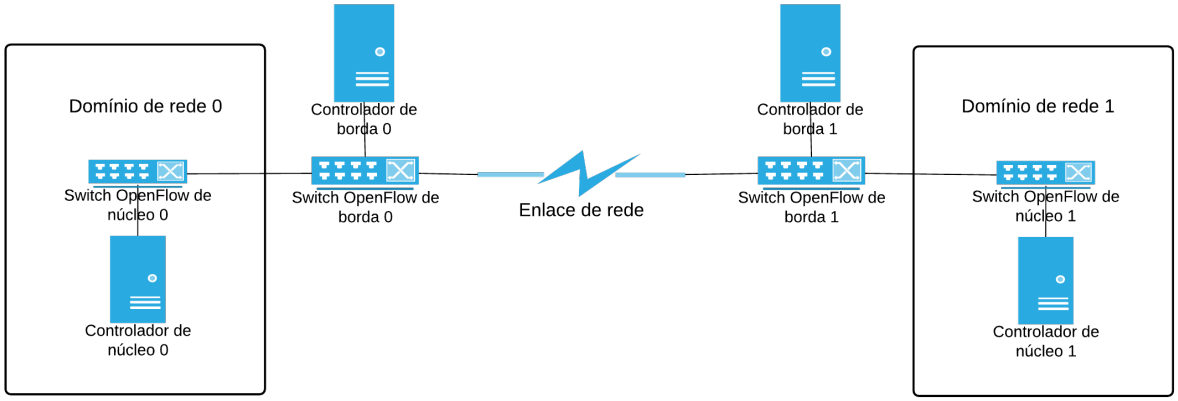

Figura 2.9: Arquitetura para Internet através de SDN e OpenFlow

Capítulo 3

Trabalhos relacionados

Essa seção apresenta os trabalhos relacionados ao presente projeto de dissertação e discute suas características. A seção segue uma linha cronológica com temas e artigos que levam a formulação que é a base do presente trabalho.

3.1 As redes em camadas (*Overlay*)

Um entendimento das implicações de redes *overlays* para a arquitetura da Internet, para o mercado e para a política é apresentado em [12]. De modo geral o artigo descreve, principalmente, as redes de CDN (Redes de entrega de conteúdo), segurança e roteamento em camadas. O artigo posiciona as *overlays* como uma uma camada intermediária, acima dos protocolos básicos IP e abaixo da camada de aplicação. Segundo essa visão, *overlays* são para a internet básica (ponto-a-ponto) usuários finais em que, por exemplo, um roteador apenas encaminha pacotes sem se importar com seu conteúdo ou finalidade. Por outro lado, para a aplicação ela se comporta como sua infraestrutura. Segundo o autor, as *overlays* se tornaram o principal meio de evolução da arquitetura da Internet.

Utilizar *overlay* para tratar as deficiências da rede é custoso, no entanto atualizar a infraestrutura da Internet básica seria ainda mais. As redes *overlays* irão, de maneira disruptiva, representar o modo de inovar dentro da Internet criando novos jogadores, um novo cenário econômico e novas regras.

A arquitetura RON (*Resilient Overlay Network*) [13] é capaz de tornar a entrega de pacotes na Internet mais confiável através de detecção e recuperação de interrupções e falhas no roteamento. A Internet foi criada como uma rede *overlay* que funcionava sobre a rede de telefonia. Ou seja, o conceito de Redes *overlay* não é uma ideia nova. No entanto, poucas dessas redes foram estruturadas para se recuperar e tolerar falhas de forma eficiente. Nesse contexto é que as RONs demonstram ser confiáveis.

As RONs possuem três Metas/objetivos. O primeiro objetivo e principal é permitir que um grupo de nós possam se comunicar independente de haver uma falha na rota

entre eles. Isso torna o roteamento confiável. O segundo objetivo é tornar o roteamento e a seleção de próximo host com aplicações distribuídas mais forte, mais eficiente do que tradicionalmente é feito com outras arquiteturas e protocolos. O terceiro e último objetivo é fornecer um framework para implementação de políticas de roteamento que governam a escolha de rotas dentro da rede.

As implicações de desempenho entre a rede *overlay* Gnutella e a infraestrutura da Internet é descrito em [55]. Redes P2P agregam vários computadores que entram e saem da rede todo o tempo. Esses Peers (computadores) podem não ter um endereço IP permanente. As redes P2P são entidades independentes e auto organizadas. Esse trabalho descreve um período de avaliação de sete meses do crescimento das redes Gnutella assim como suas implicações de desempenho. A rede é composta por serventes (computadores) que são os nós da rede virtual na camada de aplicação em que seus enlaces são formados por conexões TCPs abertas. Ao analisar a topologia utilizada pelo Gnutella o artigo explicita que apesar de essas redes serem eficientes em lidar com falhas aleatórias em computadores, elas estão vulneráveis a ataques bem planejados.

3.2 Redes centradas em conteúdo

Content-Centric Networking (CCN, redes centradas no conteúdo) trata o conteúdo como uma primitiva[60]. Esse conteúdo é requisitado através de um nome. Em analogia ao IP, a CCN substitui o 'onde', utilizado no IP, pelo 'o que'. A comunicação dentro da CCN é definida por 'consumidores de dados'. Existem dois tipos de pacotes: *Interest* (interesse) e *Data* (dado).

O documento descreve o funcionamento completo do roteamento dos pacotes baseados em conteúdo, assim como seria seu comportamento dentro de intra-domínios e inter-domínios. A segurança acontece no nível do dado ao invés de apenas uma propriedade da conexão pela qual o dado trafega. O projeto de CCN's a protege de várias classes de ataques de rede. O artigo apresenta comparações e avaliações de CCN's em relação ao TCP/IP. São relacionados temas como eficiência na transferência de dados, eficiência na distribuição de conteúdo, a estratégia de camadas da rede e voz sobre CCN. A CCN foi projetada para substituir o IP, mas pode ser distribuída como um overlay.

O artigo [35] apresenta como o CDN (*Content Delivery Network*) da Akamai's lidou com gargalos e falhas através de servidores de borda. Esses servidores são responsáveis por aliviar o serviço de estressar um único servidor. Servir conteúdo na *Web* com apenas um servidor são sérios problemas para escalabilidade, confiança e performance de um sítio/serviço.

Ao estabelecer servidores de borda que possuem tarefas específicas, é necessário tratar suas falhas individualmente. Em função dessas características, o artigo apresenta soluções com servidores de borda através da utilização de aproximação (proxy), provisionamento (caches), replicação, sincronia, balanceamento de carga, tolerância a falhas, autenticação, segurança, recuperação de erros e monitoramento de serviços. O artigo propõe que o grande desafio para lidar com esses circunstâncias é identificar os design patterns (padrões de desenho/implementação) que são soluções de custo efetivo e úteis ao sistema de maneira global.

Uma experiência relevante foi a utilização do CoralCDN durante seus cinco anos de deploy (publicação). Com o enorme crescimento em utilizar virtualização e deploy em cloud, os serviços na Internet estão cada vez mais descentralizando sua infraestrutura. O CoralCDN foi desenvolvido de maneira escalável e automática para lidar com picos repentinos no tráfego de conteúdo [21]. O artigo demonstra a sequência de acontecimentos que ocorrem dentro da rede de aproximadores (proxys), servidores de índice, DNS e origem quando um cliente requisita uma URL ao CDN.

O CoralCDN, objetiva poupar ao máximo os servidores origem. A utilização de aproximadores (proxys) e provisionamento (cache) garantem que esse objetivo seja cumprido. O funcionamento interno do CDN estabelece políticas de acesso aos conteúdos. Diferenciando políticas para recuperar conteúdos antigos, conteúdos não populares, excessos para conteúdos populares e de muito acesso. O CDN disponibiliza também, uma API aberta para permitir elasticidade na distribuição de conteúdo.

O protocolo Chord de pesquisa de itens em nós dentro de uma rede Peer-to-peer é apresentado em [32]. O Chord executa apenas uma operação em que, dada uma chave, ele mapeia a chave a um nó (host). A tabela de roteamento Chord é distribuída e utiliza hashing para atribuir uma chave a um nó. O artigo prova que a complexidade algorítmica dessa busca é O(log N). O Chord simplifica o modelo P2P, lidando apenas com balanceamento de carga, descentralização, escalabilidade, disponibilidade e nomeação flexível. É provado que as buscas recursivas são mais eficientes em função do tempo de execução do que as pesquisas iterativas. Diferentemente de P2Ps como Napster e Gnutella, o protocolo Chord melhora a escalabilidade da tarefa de embaralhamento (hashing), pois ele evita que todos os nós tenham que saber sobre todos os demais nós.

ICN (Information Centric Network) [8] é uma abordagem parecida com a CCN (Content centric Network). O artigo mostra uma arquitetura bem parecida com SDNs onde o plano de controle está separado do plano de dados. Mostra também que esses planos são unidades lógicas dentro da rede. Ou seja, o controlador pode ser composto por diversos equipamentos (entidades) na rede.

O artigo, abstrai o funcionamento da busca de dados dentro da rede. Ele parte do pressuposto que a busca funcione como um DNS, dado que a busca é feita por pacotes que carregam os interesses de quem iniciou a requisição. Os comutadores ao longo da rede

fazem cache dos pacotes de dado, podendo responder a requisição sozinhos.

3.3 O protocolo OpenFlow e as inovações em rede

A evolução da Internet é uma discussão polêmica. O artigo [34] apresenta dois pontos de vista. O primeiro defende que a Internet necessita de um novo projeto. O segundo apresenta um ponto de vista contrário argumentando que a rede deve evoluir e não ser reconstruída.

O primeiro ponto de vista levanta questionamentos como se deve-se continuar fazendo melhorias na rede ou se construir uma nova arquitetura seria melhor, dado que elas seriam irrestritas pelo atual modelo. É argumentado que o sucesso da atual Internet não significa que ela está amadurecida. É descrito que a rede não está preparada para os dispositivos e pequenos sensores, atualmente utilizados dentro da rede, que podem revolucionar a sociedade contemporânea.

Um contraponto é apresentado dizendo que a economia industrial moderna não está habilitada a receber essas tecnologias emergentes e que isso envolveria custos inconcebíveis em mudar operação e tecnologia envolvida. É descrito também que os estudos de novas arquiteturas para a rede, para que seja aceitável, deve substituir por completo o atual modelo. Do contrário, seria apenas mais exercício intelectual para a construção da nova rede.

O projeto do protocolo OpenFlow foi apresentado em [41]. Ele é uma forma de pesquisadores experimentarem novos protocolos nas redes utilizadas no dia-a-dia. Ele é baseado nos comutadores ethernet com uma tabela de fluxos e uma interface bem definida de comunicação com computadores externos.

Os pesquisadores podem controlar seus próprios fluxos, escolhendo uma rota alternativa para os pacotes ou executando algum processamento no pacote. Eles podem testar modelos de segurança, schemas de endereçamento e até alternativas ao protocolo IP. É possível fazer estudos em redes isoladas, em administração de redes, em controle de acesso, autenticação e processamento de pacotes. O OpenFlow possibilita experimentações de maneira uniforme em comutadores completamente heterogêneos.

Um projeto (*design*) de rede é apresentado através abordagem híbrida entre MPLS e SDN com o objetivo de simplificar o hardware e flexibilizar o controle [39].

O artigo apresenta uma divisão lógica entre núcleo (*core*) e borda (*edge*) da rede. O núcleo simplifica a rede. Ele se preocupa apenas com o encaminhamento de pacotes dentro do núcleo através dos rotulagens (*labels*) MPLS. A borda mantém um controlador SDN que permite adicionar os rótulos *labels* MPLS para repassar ao núcleo. O serviços con-

versam apenas com a borda. Uma máquina (*host*) ao enviar um pacote a outra máquina em outro domínio, primeiramente passa pelo controlador na borda. Esse pacote chega pelo controlador de entrada (borda) e é repassado ao controlador do núcleo com o rótulo colocado pelo controlador da borda. O núcleo por sua vez faz o devido encaminhamento à máquina de destino. Essa abordagem simplifica e desacopla as redes. Elas podem ter no núcleo algoritmos e protocolos próprios sem interferir nas redes adjacentes pois há um isolamento bem estruturada através do controlador na borda das redes.

Baseando-se em SDN, o artigo [4] propõe uma arquitetura na linha evolucionária para Internet. Esse trabalho é uma generalização para a Internet do [39]. Como a arquitetura da Internet possui deficiências e possui diversos fatores que impedem que ela seja substituída, o artigo traz uma abordagem que dissocia a infraestrutura da arquitetura da Internet.

A arquitetura proposta envolve fundamentos trazidos de SDN, MPLS e encaminhamento via programas (*software forwarding*). Nessa arquitetura, as redes seriam domínios. Cada domínio possui um controlador do núcleo (*core*) e um de borda (*edge*). Os controladores de borda controlam o tráfego externo. Os de núcleo controlam o interior dos domínios. Isso cria uma camada de isolamento em que pode-se alterar protocolos e algoritmos em ambos os ambientes, núcleo e borda, sem que um interfira no outro. Além disso foram mostrados modelos de serviços que podem ser estruturados sobre essa arquitetura. O artigo apresenta três exemplos de sistemas de redes que poderiam funcionar perfeitamente sobre essa arquitetura.

3.4 Computação em arquiteturas na nuvem

O artigo [3] apresenta o NOX (*Networking Operanting System*). O NOX traz um controle lógico centralizado de alto nível de abstrações de rede como usuários, topologia, serviços e controle da rede. Através do OpenFlow ele adiciona entradas de fluxo (*flow entries*) na tabela de encaminhamento dos comutadores.

O projeto permite implementações em C++ e Python. A abordagem do artigo é voltada para datacenters. São demonstradas interações com PortLand e LV2. O grande objetivo desse sistema de gerenciamento é prover um controle da rede, flexível suficiente, para atender uma ampla gama de necessidades de rede em datacenters. Pelo fato de artigo trazer uma abordagem mais comercial (*datacenters*), são demonstradas as necessidades e problemas enfrentados por *datacenters* e como o NOX demonstra-se uma ferramenta que flexibiliza e facilita o design dessas aplicações comerciais.

Visando solucionar o problema de migração de máquinas virtuais, o artigo [18]

apresenta LIME. Ele é uma solução baseada em SDN que isola a aplicação da topologia da rede e de como ela é controlada. A proposta visa reduzir o tempo com que a migração e sincronia acontecem.

Ao iniciar uma migração de máquina virtual, os comutadores que estavam no local de origem e os do local de destino são agrupados em um único comutador virtual. Todo o fluxo de dados é repassado ao controlador SDN que controla todo o tráfego durante o processo de migração. Concluída a migração o LIME reprograma os comutadores e descarta a necessidade do controlador SDN interferir no trafego.

Os dispositivos intermediários (*middleboxes*) são parte crucial das grandes redes corporativas, centros de dados (*datacenters*) e computação na nuvem (*clouds*). Seu gerenciamento é complexo dinâmico. O artigo [2] apresenta a ideia de um arcabouço (*framework*) utilizando SDN para gerenciar os dispositivos *middleboxes*.

O trabalho apresenta um arcabouço (*framework*) que possibilita novas aplicações. Com a separação do plano de dados do plano de controle através da SDN, tem-se maior flexibilidade para posicionar os dispositivos (*middleboxes*) dentro da rede, assim como simplificar seu gerenciamento. São apresentados também abstrações e interfaces para lidar com os estados dos dispositivos intermediários (*middleboxes*). Essa tarefa não é nada trivial, pois esses dispositivos (*middleboxes*) podem ser muito diferentes uns dos outros.

A configuração em tempo real (*run-time*) de redes para grandes volumes de dados (*big data*) com o objetivo de otimizar a aplicação juntamente com o desempenho e utilização da rede é apresentado em [26]. O trabalho se baseia na utilização de *switches* óticos como premissa para aumentar a performance. A abordagem dos autores envolve um controlador SDN que é uma interface para as aplicações dentro do datacenter. O artigo foca bastante na topologia física e no roteamento de aplicações em *big data*. Ao utilizar um controlador SDN, as aplicações em *big data* tornam-se mais próximas à rede que está abaixo da aplicação. O artigo utiliza o *Hadoop* como exemplo de testes. São discutidos aspectos de integração de rede, agendamento de tarefas, topologia e configuração de rotas para processos *Hadoop*.

O Floodlight [19] é um controlador SDN baseado em *cloud*. Foi construído pensando em performance. Ele apresenta uma abordagem *multithread* e integrada ao gerenciador de *cloud* OpenStack. Internamente ele utiliza programa para habilitar o *OpenFlow* em *switches* físicos e virtualizados. O Floodlight é extensível e foi usado como base para o controlador ONOS [6].

3.5 Recuperação de informação topológica

A necessidade de se buscar e monitorar dados específicos, principalmente em redes complexas, atrai o desenvolvimento de linguagens DSL (linguagens de domínio especifico) que simplifique, organize, generalize e garanta eficiência na manipulação desses dados. Esse é o exemplo do Frenetic [20] e do Pyretic [42].

O Pyretic introduz novas abstrações para criação de aplicações de vários módulos independentes, que em juntos administram o tráfego e abstraem a topologia da rede. O sistema utiliza uma linguagem chamada Pyretic [42]. O sistema funciona em cima do POX. Ele converte regras de plano de controle programadas em Pyretc para o plano de dados dos dispositivos de rede dentro de um SDN. O artigo apresenta o conjunto de regras de composição como regras de ação, regras de predicado, regras de consulta, etc. O Pyretic é uma linguagem que permite programadores a criarem grandes e sofisticadas aplicações SDN com pequenos módulos independentes.

Enquanto a abordagem da DSL é baseada em atuação, execução, conjunta com o controlador, o NIB é baseada em um banco de dados distribuído, com uma abordagem mais ampla e atacando problemas mais gerais como persistência, concorrência, redundância, escalabilidade, etc.

3.6 A abordagem em grafos

A abordagem de representar a rede na forma de um grafo foi mencionada por Casado *et al.* em um dos primeiros artigos sobre SDN [9]. No entanto, nenhum detalhe de implementação é apresentado. Em um trabalho futuro, uma solução SDN foi desenvolvida através de diferentes topologias de rede dentro do contexto de *datacenter* em que a abstração em grafos não foi adotada [29].

Raghavendra *et al.* apresenta um módulo em grafos com capacidade de atualização dinâmica com uma API para algoritmos em grafos [52]. Esse trabalho não possui nenhuma integração com algum controlador SDN, que é a base da avaliação do presente trabalho.

O controlador *Onix* [36] foi projetado em torno do conceito NIB (*Network Information Base*), que é uma base de informações da rede. Essa base mantém uma visão global da rede de maneira similar à MIB (*Management Information Base*) implementada sobre o protocolo SNMP. Essa representação baseada em grafos é alcançada indexando cada entrada de elemento em relação a seus vizinhos.

Outra abordagem distribuída é apresentada pelo controlador *ONOS* [6]. É demostrado uma arquitetura distribuída em que se utiliza um banco de dados de chave/valor em memória, o RamCloud [47] em que o grafo é implementado. É apresentado um modelo de controlador com vários computadores sincronizados e redundantes. São discutidos pontos de performance e a disponibilização do projeto como código aberto.

Capítulo 4

Uma avaliação do OpenFlow

Esse capítulo avalia o protocolo OpenFlow através de um sistema de balanceamento de carga. Através dos recursos fornecidos pelo protocolo é possível maximizar a justiça no balanceamento de carga em um serviço HTTP. Detalhes sobre a implementação e os resultados desse experimentos são apresentados nesse capítulo.

4.1 Proposta

Sistemas de balanceamento de carga são baseados em políticas de balanceamento. Essas políticas podem se aproveitar positivamente da separação do plano de dados e do plano de controle e sua flexibilidade. O presente trabalho apresenta um sistema de balanceamento de carga inteligente baseado em Redes Definidas por *Software* (SDN). As políticas de balanceamento são baseadas no trafego da rede, na carga dos servidores e no estado corrente do serviço. Os experimentos mostram que a abordagem em SDN permite um balanceamento de carga que reduz a carga dos servidores, aumenta a disponibilidade do serviço e otimiza a instalação de fluxos no plano de dados.

4.2 Introdução

Os serviços modernos devem ser escaláveis para atender a milhões ou milhares de clientes. Para realizar essa tarefa, os serviços devem ser distribuídos em vários servidores. Como consequência, para garantir a experiência do usuário/cliente, o volume de consultas em processamento por cada servidor deve ser compatível com sua capacidade.

Balanceamento de carga é um requisito para sistemas distribuídos que se escalam através de vários servidores. Conforme apresentado em [59], balanceamento de carga em

Redes de computadores consiste em uma técnica usada para distribuir a carga de trabalho entre vários enlaces ou computadores. Um balanceamento de carga deve ser transparente para o usuário final e permitir que a aplicação seja escalável e flexível.

As soluções atuais são muito caras. Normalmente, dispositivos intermediários (*middleboxes*) são utilizados, no entanto não são customizáveis [61]. Balanceadores de carga são caros e, em muitos casos, tornam-se um ponto de congestionamento da rede [27]. Os dispositivos intermediários para balanceamento de carga não levam em conta a largura de banda ou a latência da rede. Contudo, alguns balanceadores não conseguem agrupar as requisições por similaridade [61], nem avaliar diretamente a topologia da rede como em [28].

Uma solução viável seria executar a tarefa de balanceamento de carga em comutadores comerciais. O protocolo OpenFlow permite que, políticas de balanceamento de carga, sejam aplicadas a rede com baixo custo e que podem se adaptar à aplicação ou serviço.

A solução proposta nessa avaliação, é aproveitar da flexibilidade do plano de controle separado do plano de dados. Como uma entidade logicamente centralizada, o controlador possui uma visão topológica global da rede [41]. Os experimentos mostram que a abordagem em SDN a perda de pacotes do serviço e seu atraso. A disponibilidade e a vazão da aplicação é maximizada. As políticas de balanceamento de carga propostos não distribuem igualmente as requisições entre os servidores. Ao invés disso, elas garantem que a carga de trabalho ao longo dos servidores da aplicação sejam mais justos e homogênea.

Um sistema de balanceamento de carga corporativo utilizando OpenFlow é apresentado em [28]. Uma generalização do fluxo de pacotes através de *wildcards* (fluxos curinga) foi feita em [61]. Uma medição da latência e da largura de banda é descrita em [59]. Um modelo de protocolo para balanceamento de carga foi proposto por [10]. Esse trabalho apresenta uma proposta que verifica a carga de servidores de aplicação, a fila de requisições (HTTP), a carga prevista e o número de requisições respondidads. Esses parâmetros são a basa do balanceamento de carga. Os fluxos de ida e de volta (fluxo reverso) são instalados nos comutadores a fim de reduzir o número de pacotes enviados ao controlador.

Primeiramente apresentamos o projeto do sistema de balanceamento de carga. Em seguida, são discutidas as políticas de balanceamento de carga. Os experimentos e a avaliação dos resultados são apresentados. Ao final, são discutidos trabalhos futuros e uma conclusão.

4.3 Trabalhos relacionados

Balancemento de carga é importante para serviços que exigem robustês ao dividir a carga de entrega de pacotes de modo agregado. O OpenFlow permite criar soluções de baixo custo. Considerar o balanceamento de carga uma primitiva em Redes de computadores é apresentado em [27].

Medições de latência e largura de banda foram feitos através do controlador NOX em [59]. Abordagens utilizando regras genéricas (*wildcards*) para evitar instalar separadamente fluxos é apresentado em [61]. Esses fluxos reduzem o volume de pacotes enviados para o controlador.

Um balanceamento de carga dinâmico para *cluster* de computadores em ambientes virtualizados utilizando OpenFlow é mostrado em [11]. Um algoritmo de balanceamento de carga chamado *Server-based* (SBLB) é proposto. Dinamicamente ele recupera dados da carga nos servidores para a tomada de decisão do balanceamento de carga.

A proposta do presente trabalho é uma extensão do módulo de balanceamento de carga do controlador POX [51]. São reduzidos a perda de pacotes e o atraso na resposta através da instalação de fluxos reversos. São aumentados a disponibilidade e a largura de banda da rede. Além disso, a distribuição de carga foi aplicada de maneira mais justa entre os servidores de aplicação.

4.4 Projeto de implementação

A solução resume-se em um balanceador de carga TCP. A implementação não lida com DNS (*domain name service*). O balanceador de carga toma decisões avaliando a carga dos servidores (*load avarage*), o número de pendências na fila de requisições do servidor HTTP e o número requisições já respondidas. Os dados monitorados pelas políticas de balanceamento de carga foram coletados através de um comutador *OpenFlow* e através do sistema de arquivos distribuído NFS (*Network File Syste*).

O controlador POX [51] foi adotado como base da solução proposta. A arquitetura baseada em eventos permite que módulos atuem como produtores e consumidores de eventos. O módulo *core* encapsula o controle de eventos. Novos eventos podem ser registrados por outros módulos.

Carga média de CPU	Fila de pendências do servidor HTTP	Número de requisições respondidas

Tabela 4.1: Dados monitorados de cada servidor HTTP

4.4.1 Balanceador de carga

O módulo *load_balancer* foi implementado como uma classe. Esse módulo se registra no *core* no momento em que o controlador é executado. Para inicialiar o módulo é necessário informar uma lista de endereços IPs que representam a lista de servidores a serem balanceados. Um IP lógico deve ser informado ao módulo ao qual todas as requisições durante os experimentos serão direcionadas. Cada requisição direcionada ao IP lógico é direcionada a um dos servidores cuja carga, no momento, seja a menor.

4.4.2 Fluxo de trabalho do controlador

Novos pacotes na rede geram pacotes de entrada *PacketIn* no controlador. O balancedor de carga avalia o novo fluxo e instala uma regra no comutador baseado em uma política de balanceamento de carga. Os demais pacotes similares a esse fluxo são encaminhados para o mesmo servidor. Após algum tempo, os fluxos expiram no comutador, permitindo maior dinamismo na rede. A decisão de qual servidor escolher para encaminhar os pacotes depende diretamente da política de balancemanto de carga. Através do sistema de arquivos remoto a carga dos servidores foi monitorada. A tabela 4.1 apresenta os valores coletados de cada servidor.

O fluxo da execução das requisições feitas ao serviço é mostrado na figura 4.1 Cinco políticas de balanceamento de carga foram utilizadas pelo módulo para distribuir a carga de trabalho entre os servidores: *Round-robin*, Aleatória, Carga, Fila, Mistura.

Round Robin é uma política que divide de maneira idêntica o volume de requisições entre os servidores. A política de balanceamento Aleatória escolhe a cada nova requisição, aleatoriamente, o servidor a responder pela requisição. A política de balanceamento Carga é baseada em avaliar a carga de CPU de cada servidor e encaminhar a requisição para o servidor com a menor carga do momento. Fila é uma política de balanceamento que avalia a quantidade de requisições pendentes na fila de requisições do servidor HTTP. A política de balanceamento Mistura combina as políticas Carga e Fila e escolhe o servidor com a menor valor da combinação.

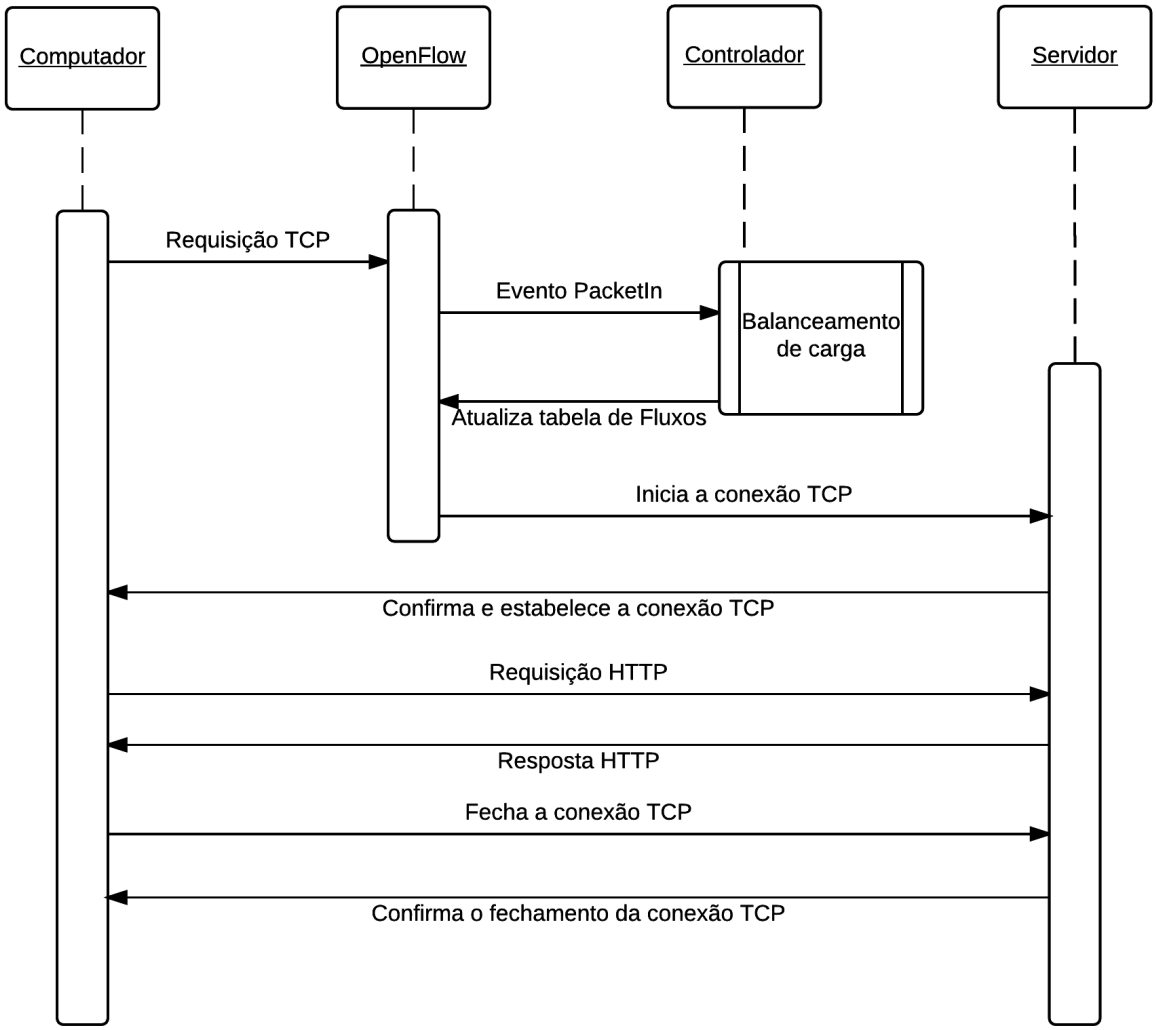

Figura 4.1: Fluxo de execução das requisições ao serviço de balanceamento de carga

4.4.3 Servidor HTTP

O servidor responsável por responder as requisições HTTP foi o *Tornado* [22]. O *Tornado* é um arcabouço e uma biblioteca HTTP assíncrona. Nos experimentos, o *Tornado* escreve, periodicamente, em um arquivo no sistema de arquivos os dados da sua fila de requisições pendentes, a carga do servidor e o número de requsições já respondidas. O balanceador de carga lê remotamente esses dados para compor os dados avaliados pelas políticas de balancemento de carga.

4.4.4 Ambiente de simulação

O ambiente de simulação é composto por 6 dispositivos. Um controlador *OpenFlow*, um *switch OpenFlow*, um cliente e quatro servidores. Cada computador está conectado diretamente a uma porta do *switch*.

Os servidores são heterogêneos. Cada computador executa um sistema operacional *Linux* diferente. Eles possuem arquiteturas, adaptadores de rede, mémórias primárias e discos rígidos diferentes. Essa característica é importante para avaliar o quão justo o balanceador de carga é um ambiente heterogêneo.

4.5 Experimentos

Essa seção apresenta os experimentos executados com requisições TCP e HTTP dentro da rede. Foram medidos largura de banda e latência. Foram avaliados também, o tempo de resposta e a disponibilidade do serviço.

4.5.1 Ambiente e testes

Todas as requisições dos testes foram direcionadas ao IP lógico do serviço. No cenário dos experimentos o 4 servidores possuíam IP do seguinte faixa de endereços: 192.168.1.100 até 192.168.1.103. O endereço IP do serviço é 192.168.1.111. Toda requisição para esse endereço IP deve ser balanceado. O computador cliente possui o endereço IP 192.168.1.50.

O experimento TCP foi feito através do utilitário de linha de comandos *iperf* através do computador cliente. Cada conexão com 30 segundos de duração com medições feitas a cada 5 segundos.

O experimento com requisições HTTP possui duas abordagens. Uma sequencial e outra paralela. A abordagem sequencial utiliza a feramenta de linha de comandos *curl*. A em paralelo utiliza o utilitário *httperf*. Para cada caso de experimento, foram criadas 500 novas conexões. Cada conexão com 50 requisições ao endereço IP do serviço. No caso do experimento em paralelo, foram executadas 4 *threads* para enviar as requisições. Todas

as requisiões foram feitas para a mesma URI (*Uniform Resource Identifier*).

A figura 4.2 mostra o ambiente dos experimentos. O controlador lê os dados dos servidores monitorados a cada segundo. Baseado nos dados lidos, as políticas de balanceamento de carga são aplicadas.

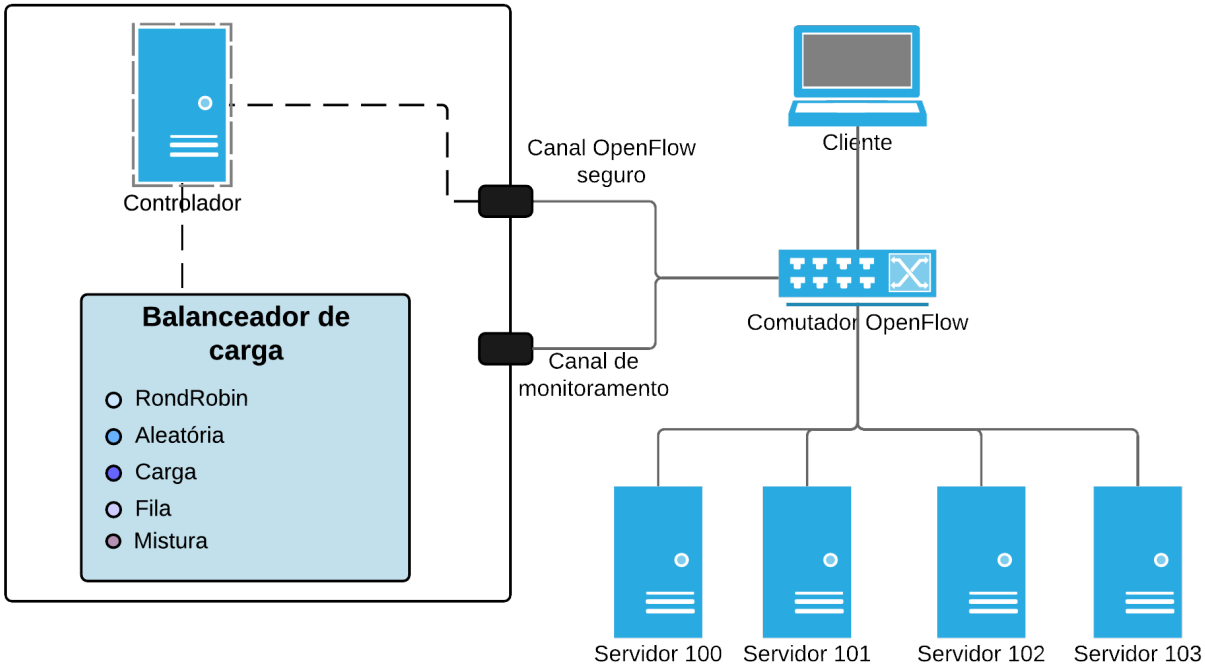

Figura 4.2: Ambientes dos experimentos do balanceador de carga

Todos os servidores rodam o servidor web *Tornado*[22]. Esse servidor grava o tamanho da fila de requisições pendentes e o número de requisições já respondidas.

4.5.2 Distribuição de carga

Através de políticas como *RoundRobin* e Aleatória, o balanceamento de carga distribui praticamente o mesmo número de requisições entre os servidores. Em políticas como Carga, Fila e Mistura a distribuição de carga é mais justa em relação à capacidade em responder requisições por parte dos servidores. Em um cenário heterogênio, como o presente, isso garante que a carga interna de cada servidor seja parecida. Por outro lado, a quantidade de requisições respondidas é diferente. Ou seja, a distribuição é justa com

o esforço nos servidores. A figura 4.3 apresenta uma comparação entre as políticas de balanceamento de carga em um cenário com 200 conexões.

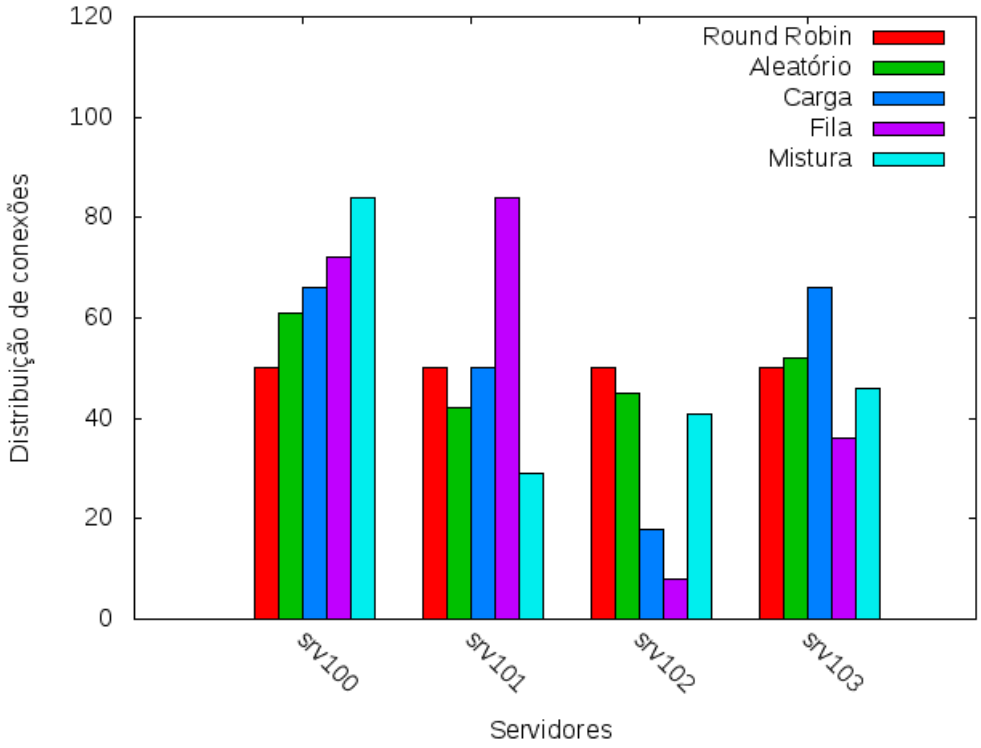

Figura 4.3: Distribuição de requisições (200 conexões com 20 requisições cada)

4.5.3 Justiça em relação à carga dos servidores

A figura 4.4 apresenta o crescimento da carga nos servidores utilizando a política de balanceamento de carga *Round Robin*. O servidor *srv102* apresenta a maior carga individual, pois respondeu a mesma quantidade de requisições que os demais servidores. Como pode ser visto, em um cenário onde os servidores não são iguais, existe uma diferença considerável de carga entre os servidores.

A política *RoundRobin* não é justa. Se um servidor se sobrecarregar, ele continuará recebendo o mesmo volume de requisições que os demais servidores. Esse comportamento pode gerar um aumento da taxa de perda de pacotes e uma redução na disponibilidade do serviço.

A política Carga reduz a diferença entre os servidores. Em cada servidor a carga tende a ser mesma. Se um servidor começar a sobrecarregar ele não será balanceado até que a carga reduza. Essa abordagem torna a distribuição de carga mais justa. Se um

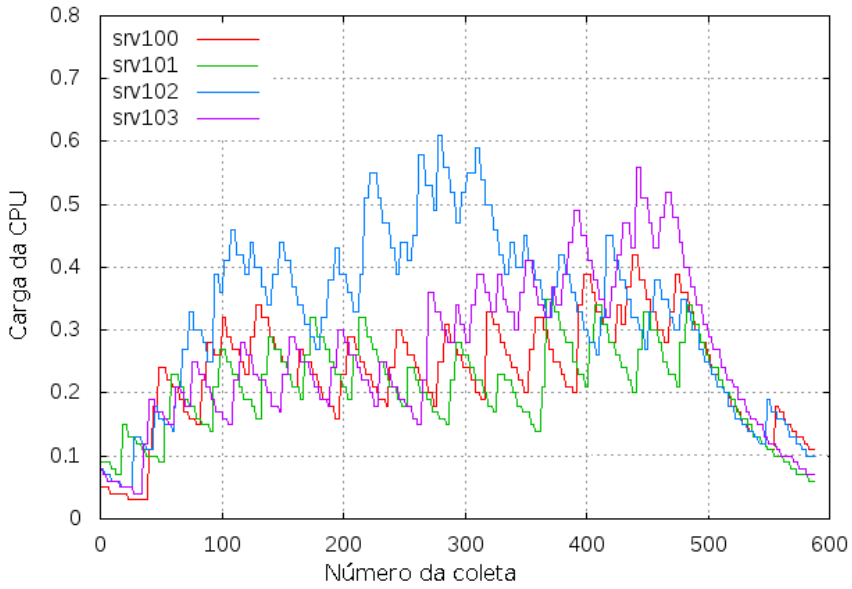

Figura 4.4: Crescimento da carga dos servidores com o balanceador de carga utilizando a política *RoundRobin*. O experimento gerou 500 conexões com 50 requisições cada

servidor consegue responder mais requisições, ele deve receber mais requisições. A figura 4.5 mostra a distribuição mais justa através da política Carga. Esse comportamento se manteve quando os experimentos foram executados em paralelo utilizando *httperf*.

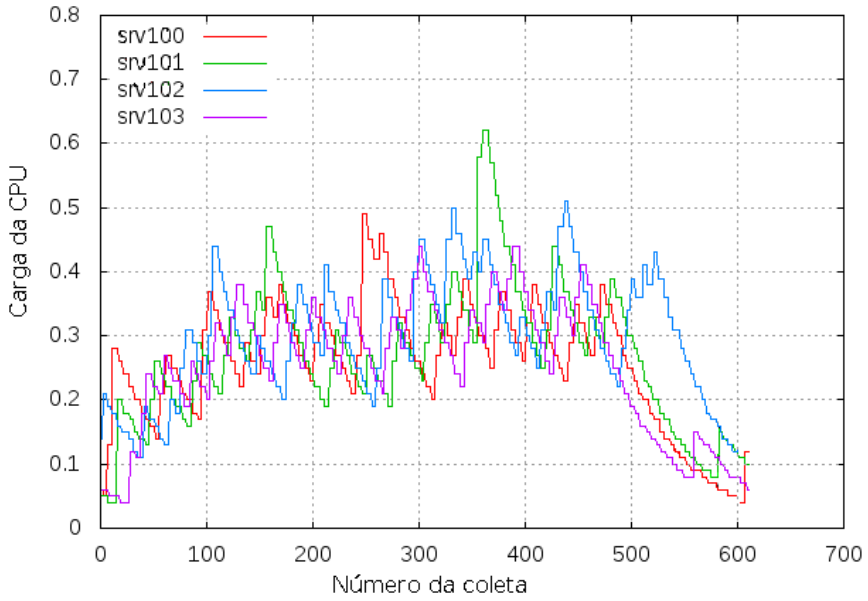

Figura 4.5: Crescimento da carga dos servidores com o balanceador de carga utilizando a política *Carga*. O experimento gerou 500 conexões com 50 requisições cada

As figuras 4.6 e 4.7 apresentam o resultado desses experimentos. A política Carga distribui a carga de maneira mais justa entre os servidores. Isso é importante em um cenário onde os servidores são heterogênios. Toda a carga de trabalho deve ser disseminada com justiça entre os servidores.

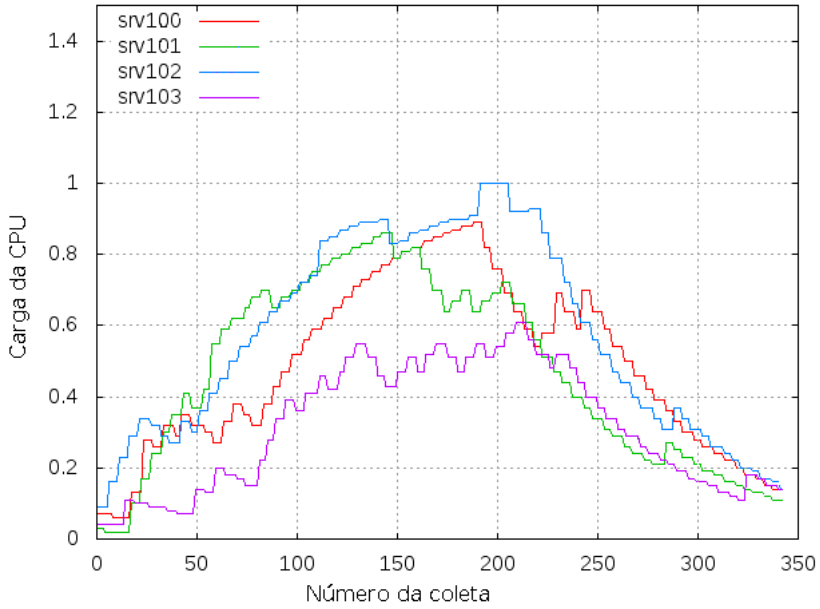

Figura 4.6: Crescimento da carga dos servidores utilizando a política *RoundRobin* com as requisições feitas em paralelo. O experimento gerou 500 conexões com 50 requisições em paralelo cada.

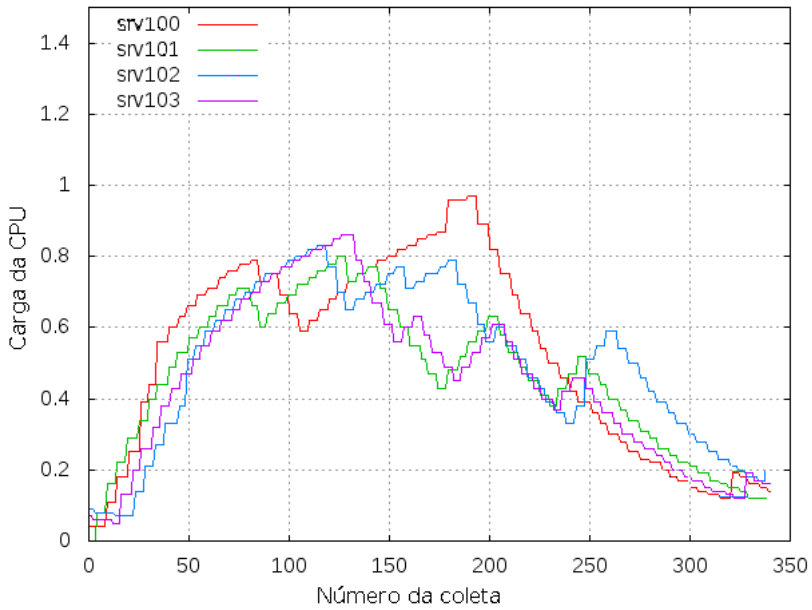

Figura 4.7: Crescimento da carga dos servidores utilizando a política Carga com as requisições feitas em paralelo. Mesmo cenário do experimento serial

A medição de justiça é utilizada em engenharia de redes para mostrar como recursos são divididos. *Jain* definiu em [33] um índice de justiça em função da largura de banda. Foi adaptado o conceito do índice de justiça para considerar a carga dos servidores e mostrar o quão justo seria o balanceamento de carga. A adaptação foi definida como:

$$j\left(x_1, x_2, ..., x_n\right) = \frac{\left(\sum_{i=1}^{n} x_i\right)^2}{n \cdot \sum_{i=1}^{n} x_i^2}$$

Na avaliação de equidade/justiça de um conjunto de cargas, existem n servidores e x_i é a carga da amostragem. O resultado varia de $\frac{1}{n}$ (pior caso) até 1 (melhor caso). O valor é máximo quando o todos os servidores recebem o mesmo número de alocações. O índice é $\frac{k}{n}$ quando k servidores tiveram o mesmo esforço e os demais $n - k$ servidores não foram escolhidos nenhuma vez.

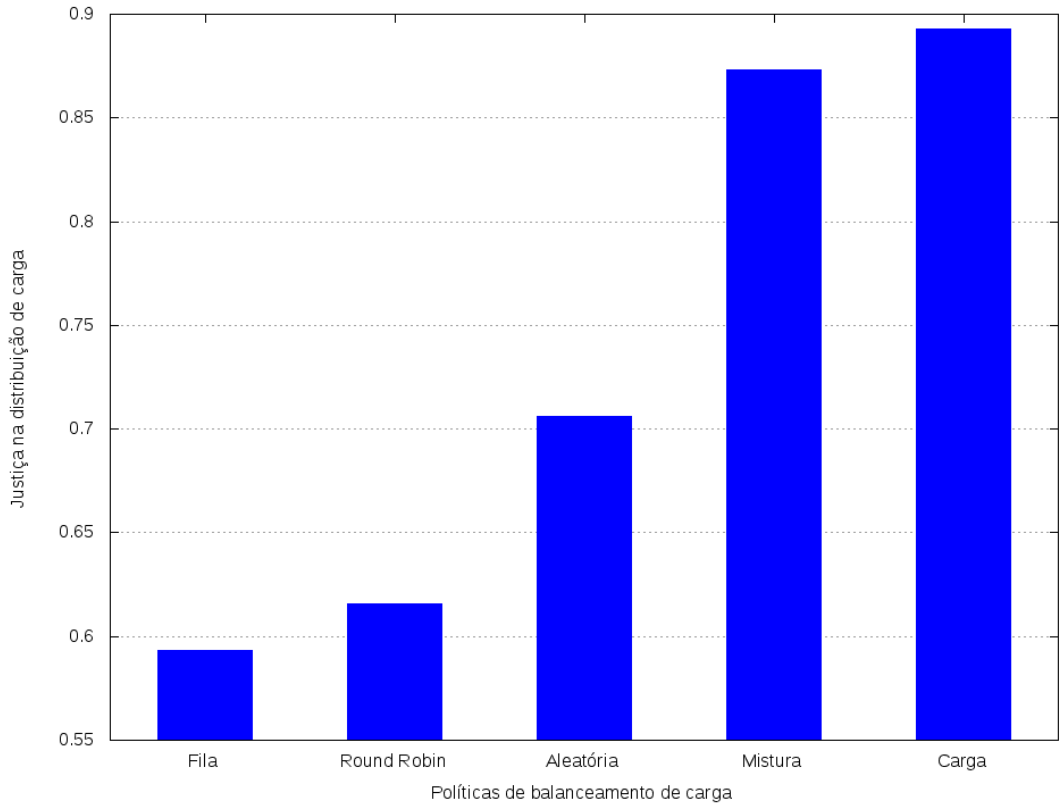

Figura 4.8: Comparação da justiça calculada para cada política de balanceamento de carga

4.5.4 Experimento com TCP

A largura de banda foi medida executando 15 conexões TCP de 20 segundos cada. Todas as requisições passaram pelo balanceador de carga no controlador *OpenFlow*. A largura de banda foi medida a cada 5 segundos de execução do experimento. Conforme apresentado na figura 4.9, nota-se o impacto do controlador na largura de banda da rede. A política Carga reduz a largura de banda global da rede, pois sua implementação demora mais para ser executada do que um simples algorítmo de *RoundRobin*.

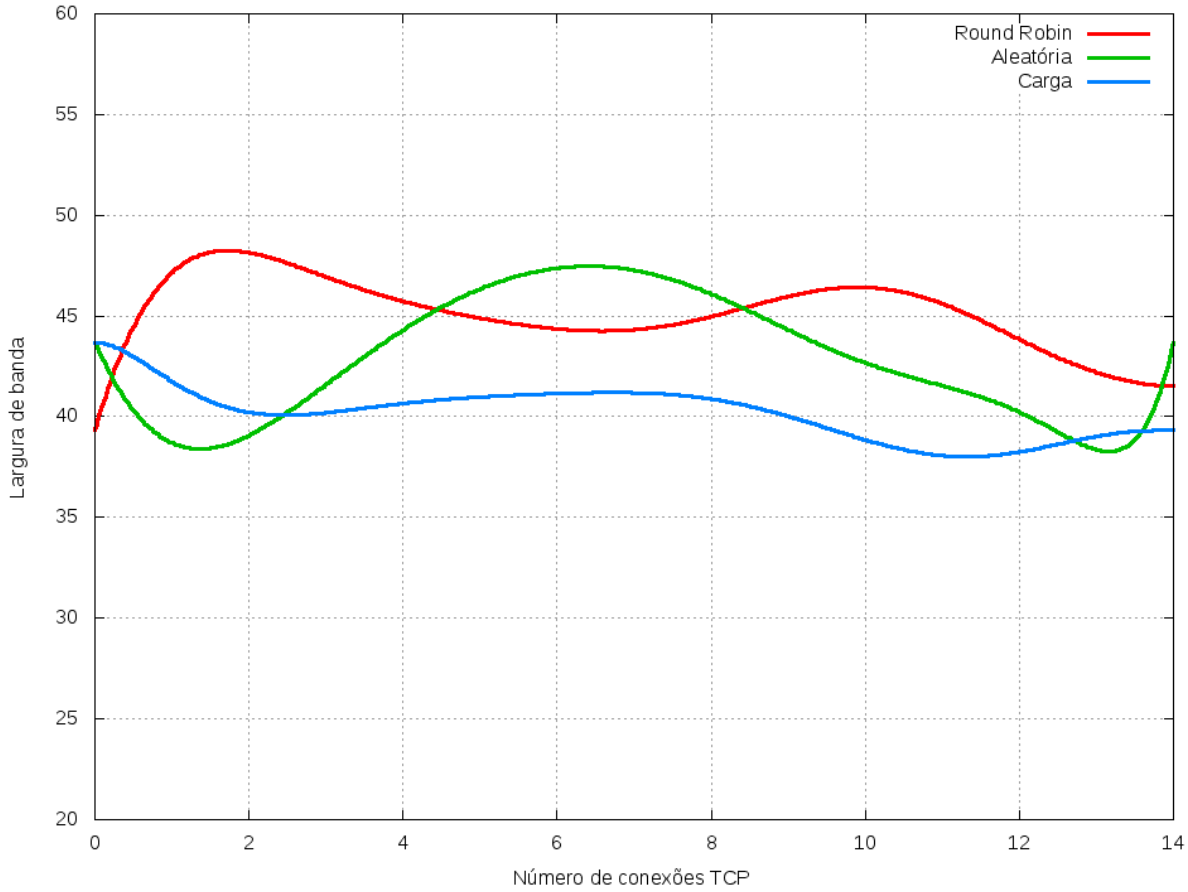

Figura 4.9: Largura de banda TCP medida em 50 conexões utilizando o *iperf*

4.5.5 Experimento HTTP

A figura 4.10 mostra o resultado do experimento com 5000 conexões. O primeiro pacote de cada conexão é encaminhado para o controlador. O controlador instala, no

comutador *OpenFlow*, as regras para todas as requisições dentro de cada conexão.

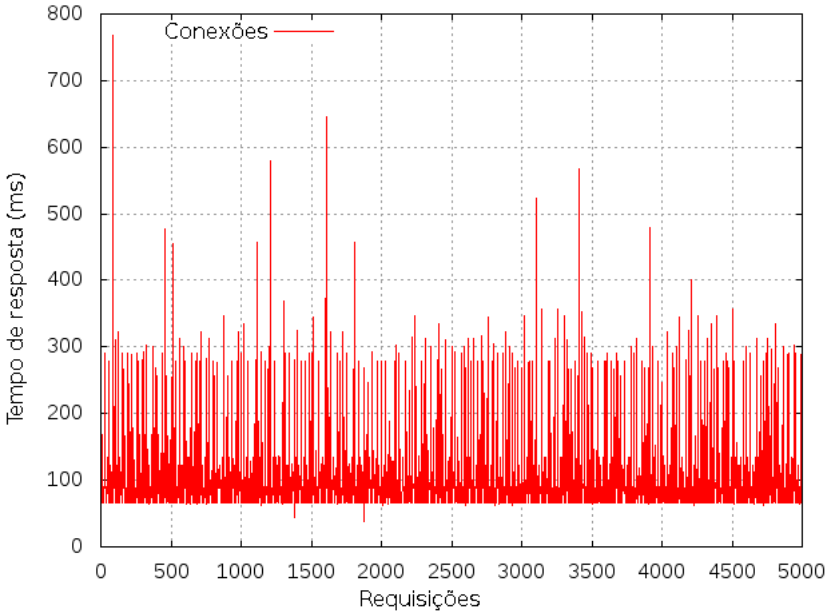

Figura 4.10: Experimento com 5000 conexões com 50 requisições cada

Os valores mais altos representam o atraso decorrido em função do controlador processar os primeiros pacotes de cada conexão. Os fluxos, para esse experimento, possuem tempo de vida (*timeout*). É importante manter a tabela de fluxos pequena e o balanceamento de carga flexível e dinâmico.

Para compreender melhor os primeiros pacotes de cada conexão enviados para o controlador, a figura 4.11 mostra o experimento HTTP da requisição 0 até a 150.

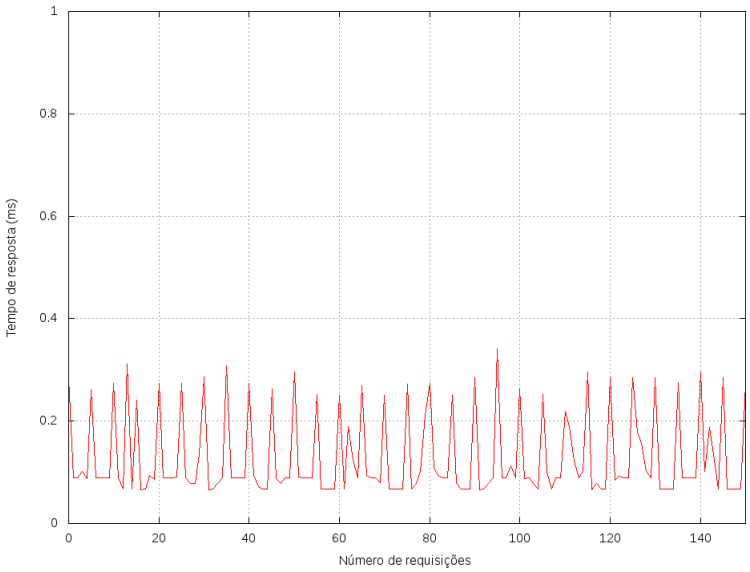

Figura 4.11: Tempo de resposta das requisições 0 até 150

O primeiro pacote de cada conexão é encaminhado para o controlador. O controlador instala um fluxo para os pacotes daquela conexão. No controlador, para todos novoso fluxos, é instalado um fluxo reverso. Esse fluxo representa o caminho de volta da requisição, que vai do servidor escalonado para o cliente requisitante. Instalar esse fluxo no comutador é importante para reduzir o número de pacotes entrantes (*PacketIn*) que o controlador precisa lidar.

A figura 4.12 apresenta o tempo de resposta em segundos de cada política de balanceamento de carga. São apresentados os valores mínimos, máximos, médio e o desvio padrão do tempo de resposta das políticas.

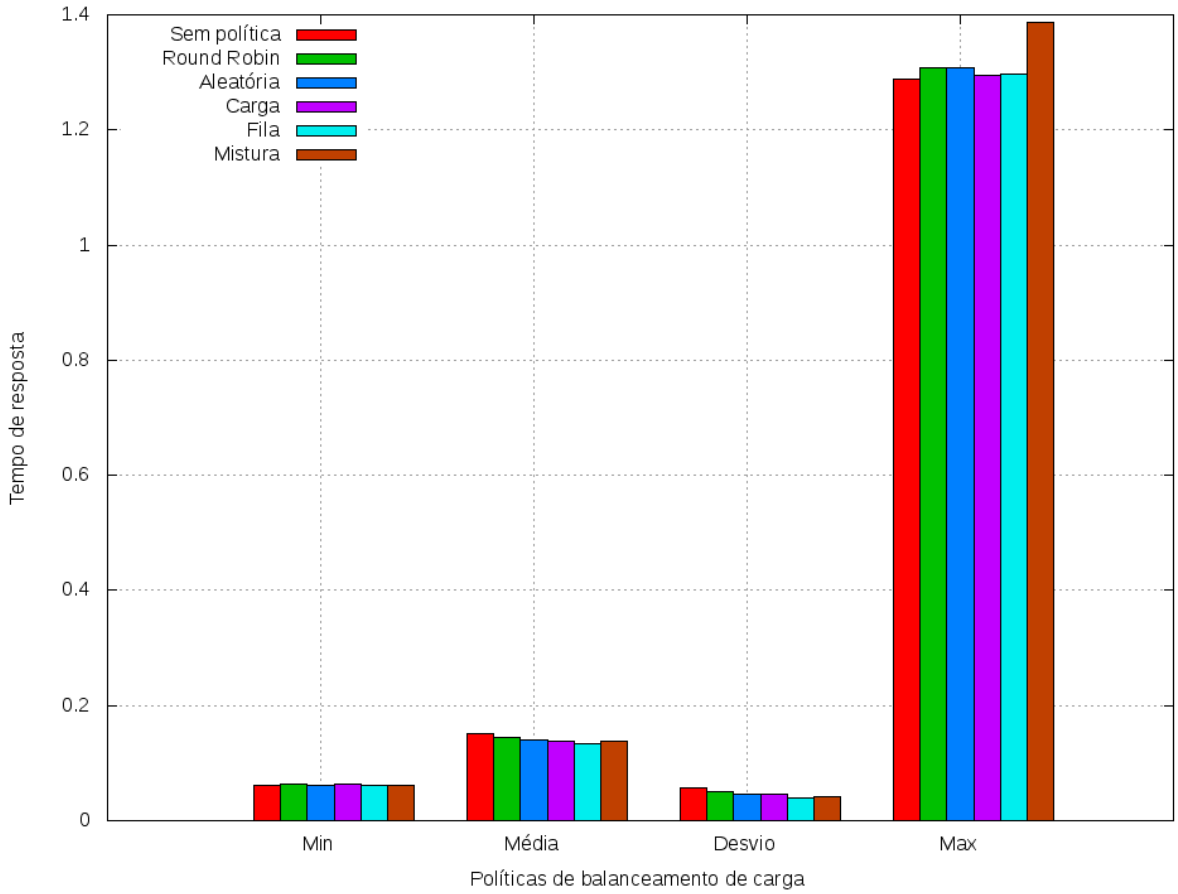

Figura 4.12: Tempo de resposta de cada política de balanceamento de carga

A política Sem Política representa requisições feitas diretamente ao servidor, sem passar pelo balanceador de carga. Como pode ser visto, a média e o desvio padrão mostram que o tempo de resposta em políticas com monitoramento é menor do que o tempo de resposta das demais políticas. Apesar de o controlador reduzir a largura de banda da rede, o balanceador de carga reduz o tempo de resposta do serviço.

4.5.6 Avaliação

O balanceador de carga não distribui o mesmo número de requisições para os servidores. Fazer isso, não torna a carga bem distribuída para os servidores em um cenário heterogêneo. Considerando o esforço de cada servidor como o parâmetro de justiça, a solução de balanceamento de carga proposto distribui de maneira mais justa a carga entre os servidores de aplicação. A carga dos servidores cresce de maneira homogênea.

Como pode ser visto na figura 4.9, pacotes encaminhados para o controlador impactam a largura de banda da rede. No entanto, a solução de balanceamento de carga reduz o tempo de resposta do serviço. Os fluxos reversos são uma estratégia inteligente que instala fluxos no comutador antes de eles saírem do servidor.

Esse capítulo apresentou uma breve avaliação do *OpenFlow* aplicado a um sistema de balanceamento de carga. Os próximos capítulos irão abordar a solução e os resultados obtidos para a proposta de um módulo em grafos para o controlador.

Capítulo 5

Proposta de Solução em grafos

A solução proposta por este trabalho foi implementada em cima do controlador POX [51]. Uma abstração da rede em forma de grafo foi construída como um módulo do controlador da rede. Esse grafo, em tempo real, representando a rede é a base para toda computação executada sobre a rede. A solução objetiva simplificar o gerenciamento em redes reduzindo o volume de computações ao longo dos nós da rede aproveitando a separação do plano de dados e do plano de controle.

5.1 A abstração em grafos

O grafo proposto é representado por $G = (V, A)$, em que V e A são conjuntos finitos de vértices e arestas, respectivamente. Cada vértice $v \in V$ representa um computador (*host*) ou comutador (*switch*) dentro da rede. Cada aresta $u \rightarrow v \in A$ representa um enlace (*link*) entre dois vértices. O peso das arestas $g(u, v)$ descreve a quantidade de *bytes* trafegados na aresta entre os dois vértices.

5.2 Controlador

O controlador POX é um arcabouço para elaboração de módulos/programas em Redes Definidas por *Software*. Totalmente voltado para pesquisa, o POX é um controlador simples, escrito na linguagem de programação *Python*.

Ao ser carregado, o POX, executa o seu núcleo (*core*). Esse módulo principal, é responsável por carregar os demais módulo e garantir a comunicação entre eles. Ele exporta uma interface baseada em eventos. Eventos que descrevem ou representam ações ocorridas na rede. Um módulo implementado dentro do controlador POX pode ser um

produtor/consumidor de eventos.

O POX contém alguns módulos de descoberta que publicam eventos relacionados à topologia da rede:

- *topology*:

  O módulo *topology*, é responsável por mapear a topologia da rede. Utilizando os recursos do *core*, ele atua como um canal de eventos relacionados à topologia e que os demais módulos podem fazer escuta (*listening*).

- *host_tracker*:

  Seu objetivo é buscar e rastrear informações sobre os computadores (*hosts*) dentro da rede.

- *openflow.topology*:

  Um controle de topologia específico para comutadores OpenFlow. Ele é integrado com o módulo *topology* do POX.

- *openflow.discovery*:

  Identifica enlaces entre comutadores OpenFlow por meio do protocolo LLDP.

- *misc.dhcpd*:

  Servidor de DHCP para configuração dinâmica da rede IP.

Esses módulos são os módulos relevantes para o presente trabalho. Existem diversos outros módulos dentro do controlador POX, no entanto eles não serão abordados nesse trabalho.

5.3 Projeto de implementação

A proposta do módulo em grafos é integrada aos módulos citados na seção do 5.2 (Controlador). A implementação do módulo é composta por cinco classes:

- Graph:

  Classe responsável por gerenciar o grafo. Todo o tratamento de eventos é feito por essa classe. O controle de criação, remoção e atualização de vértices e arestas é feito por essa classe.

- GraphEntity:

  Classe abstrata que implementa arestas e vértices. Essas entidades, vértices e arestas, herdam da classe *GraphEntity*

- Vertex:

  Classe que representa os vértices do grafo. Vértices podem ser máquinas (*hosts*) ou comutadores (*switches*)

- Edges:

  Classe que representa as arestas do grafo. Arestas são enlaces (*links*) entre dois vértices. O peso das arestas é um atributo dos objetos dessa classe.

- GraphManager:

  Classe responsável por executar computações no grafo. Toda implementação de algoritmos em grafos é feita dentro dessa classe.

5.4 Vértices

O módulo *topology* dispara os eventos *HostJoin*, *HostLeave*, *SwitchJoin*, *SwitchLeave*. Ao ser carregado, o módulo *graph* passa a escutar esses eventos. Quando eles são disparados pelo *topology* o grafo é atualizado. Assim, os vértices são criados no grafo em tempo real, à medida que os eventos ocorrem na rede.

Todo vértice tem um identificador único. No caso de vértices do tipo computador (*Host*), o identificador é o endereço MAC (*Media Access Control*) da interface de rede. Já os vértices do tipo comutador (*Switch*) são identificados pelo DPID (*Datapath ID*) que é associado ao comutador no momento de sua conexão com o controlador.

5.5 Arestas

As arestas podem ser enlaces (*links*) entre computadores e comutadores ou entre comutadores. Para a identificação das arestas entre os comutadores, o módulo *graph* escuta o evento *LinkJoin* que é disparado pelo módulo *openflow.discovery*. Os enlaces

entre computadores e comutadores é identificado quando, ao adicionar um vértice do tipo *Host*, esse vértice possuir a informação de qual comutador ele está diretamente conectado.

O peso das arestas é obtido através da leitura dos contadores de fluxos presentes nos comutadores OpenFlow. Para o módulo *graph* são lidos os contadores de fluxo da porta no switch. A lista de contadores é apresentada na Tabela 5.1 e em [45]. Através da contagem desses fluxos é possível mensurar o tráfego decorrente na aresta.

Contador	**Bits**
Pacotes Recebidos	64
Pacotes Transmitidos	64
Bytes Recebidos	64
Bytes Transmitidos	64
Drops Recebidos	64
Drops Transmitidos	64
Erros Recebidos	64
Erros Transmitidos	64
Erros de Frames Recebidos	64
Erros de Transbordo Transmitidos	64
Erros de CRC Recebidos	64
Colisões	64

Tabela 5.1: Contadores de fluxos por Porta

5.6 O módulo *host_tracker*

Além da criação do módulo composto pelas classes acima descritas, alterações no módulo *host_tracker* foram feitas. O *host_tracker* identifica os pacotes de DHCP (*Dinamic Host Configuration Protocol*) e descobre novos computadores (*hosts*) na rede. Ao construir um novo objeto da classe *Host* o *topology* dispara um evento de *HostJoin* ao qual o módulo *graph* irá atualizar o grafo da rede.

Assim como *openflow.discovery* identifica os comutadores, o *host_tracker* o faz para máquinas (*hosts*). Isso é feito através de um interceptador ARP (*Address Resolution Protocol*). Todos os comutadores (*switches*) OpenFlow são programados para encaminhar os pacotes de consulta *ARPQuery* para o controlador. Se essa consulta for destinada a um computador conhecido pelo *host_tracker*, um pacote de resposta *ARPReply* é construído e encaminhado para a máquina que originou a consulta.

Periodicamente, o *host_tracker* envia mensagens *ARPQuery* falsas para todos os computadores conhecidos a fim de verificar se eles estão ativos na rede. Através dos

contadores dos comutadores OpenFlow, o *host_tracker* monitora a atividade nos enlaces podendo inferir a presença de atividade dos computadores sem ter que enviar mensagens ARP.

5.7 Interface de programação

O módulo *Graph* exporta uma interface de programação (API). Através dessa interface, outros módulos do controlador podem requisitar informações da topologia, do tráfego e do estado da rede através do grafo. Essas informações podem ser utilizadas para implementar serviços em diversas camadas de rede que necessitam dessa informação. Os métodos permitidos por esse módulo são:

- *get_vertex(id)*

  Retorna um vértice do grafo (máquinas ou comutadores). O parâmetro *id* pode ser um endereço MAC ou o DPID.

- *get_adjacents(id)*

  Dado um vértice, retorna sua lista de adjacência.

- *snapshot()*

  Retorna o estado instantâneo da aplicação na forma de duas coleções, uma de vértices e outra de arestas.

5.8 Integração

A integração do módulo *graph* com os demais módulos do controlador POX é apresentado na figura 5.1. A forma como o controlador POX implementa seu controle de eventos é inteiramente dependente do seu núcleo (*core*). O núcleo é responsável por notificar os módulos inscritos (que fazem escuta) nos eventos a serem disparados. Toda troca de mensagens passa primeiramente por ele. A implementação do módulo *graph* está disponível em https://github.com/pantuza/pox.

Os módulos *openflow.discovery* e *openflow.topology* identificam comutadores (*switches*) e notificam o módulo *topology* através do *core*. O *topology* por sua vez, cria o evento que causa a atualização no grafo através do módulo *graph*.

As ações referentes a computadores são de responsabilidade dos módulos *misc.dhcp* e *host_tracker*. Nesses casos, o *topology* também é comunicado através do *core* e dispara eventos que atualizam o grafo através do módulo *graph*.

Caso algum módulo queira recuperar informações topológicas ou do estado da rede, ele o faz através de chamadas à interface (API) do módulo *graph*.

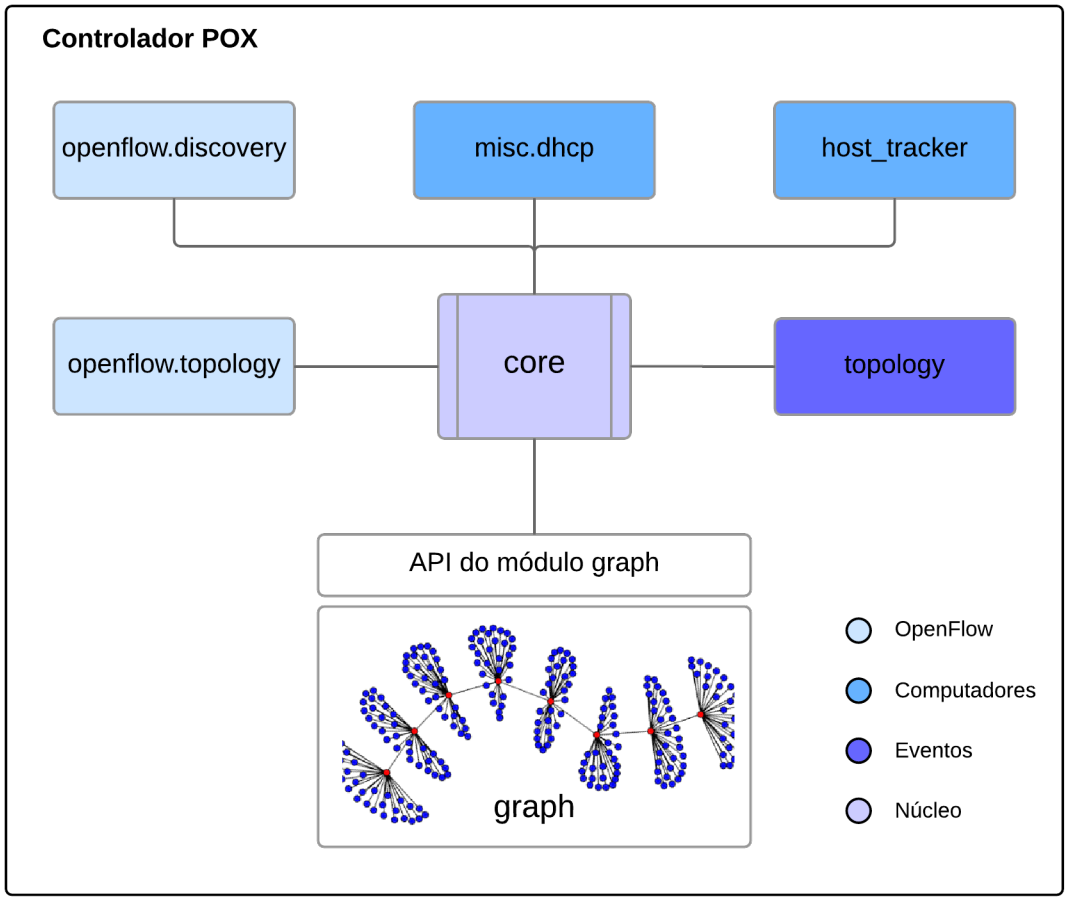

Figura 5.1: Integração do módulo *graph* no controlador POX

Capítulo 6

Experimentação e resultados

6.1 Ambiente de simulação

Uma simulação da rede nacional de pesquisa, a rede Ipê [54] foi criada em laboratório. Um comutador (*Switch*) OpenFlow e quatro servidores foram utilizados para executar a simulação no laboratório WiNet [15] no departamento de Ciência da Computação (DCC) da Universidade Federal de Minas Gerais (UFMG).

6.1.1 A rede Ipê

Operada pela Rede Nacional de Pesquisa (RNP), a rede Ipê é uma infraestrutura de rede Internet dedicada à comunidade de pesquisa brasileira. Inaugurada em 2005, foi a primeira rede óptica nacional acadêmica a entrar em operação na América Latina.

A rede IPÊ implementa 27 Pontos de Presença (*POPs*), um para cada unidade da federação. Em cada unidade existem diversos clientes que, para se comunicar com outras máquinas em outros Pontos de Presença, transmitem seus pacotes através do POP ao qual atuam como clientes. Essas ramificações constituem mais de 800 instituições de ensino, saúde e pesquisa em todo o país, beneficiando mais de 3,5 milhões de usuários [54].

Através da rede RedCLARA [53], a rede Ipê se conecta à 2,5 Gb/s com, atualmente, 15 países da América Latina e à 5 Gb/s com a rede europeia Géant [23]. Além disso, por meio de quatro conexões de 10 Gb/s, duas pelo Oceano Atlântico e duas pelo Oceano Pacífico, operadas em parceria com a ANSP, totalizando 40 Gb/s, a rede Ipê se conecta às redes acadêmicas norte-americanas, em especial, a Internet2 [31], a outras redes acadêmicas internacionais e à internet comercial mundial.

Os Pontos de presença estão interconectados conforme mostrado na figura 6.1.

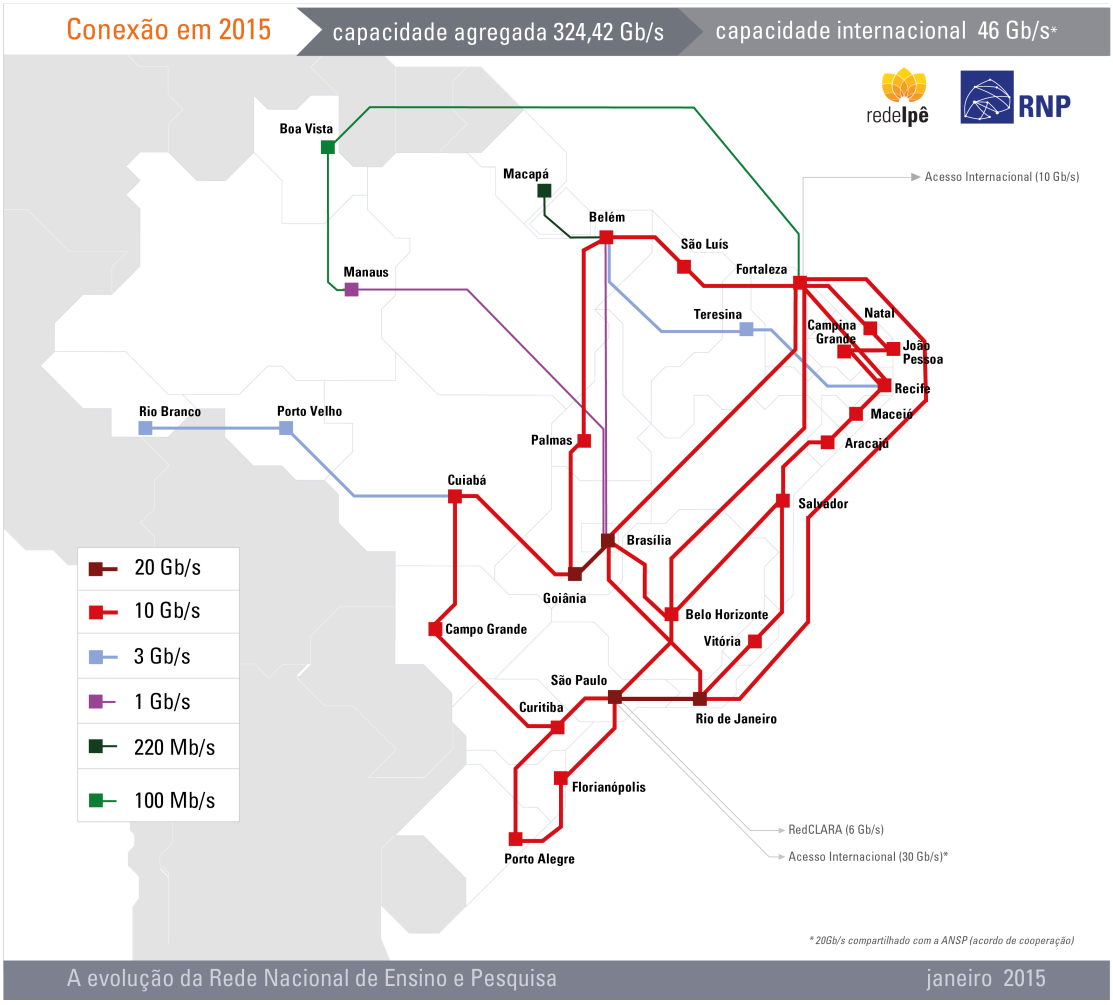

Figura 6.1: Rede Nacional de Pesquisa IPÊ [1]

[1] Imagem retirada de http://www.rnp.br/servicos/conectividade/rede-ipe

6.1.2 Arquitetura da simulação

O ambiente de simulação é composto por um comutador HP com OpenFlow habilitado. Quatro servidores são responsáveis por gerenciar máquinas virtuais que representam as unidades federativas e suas instituições clientes. Conforme pode ser visto na figura 6.2, a rede física do experimento é separada da rede virtual.

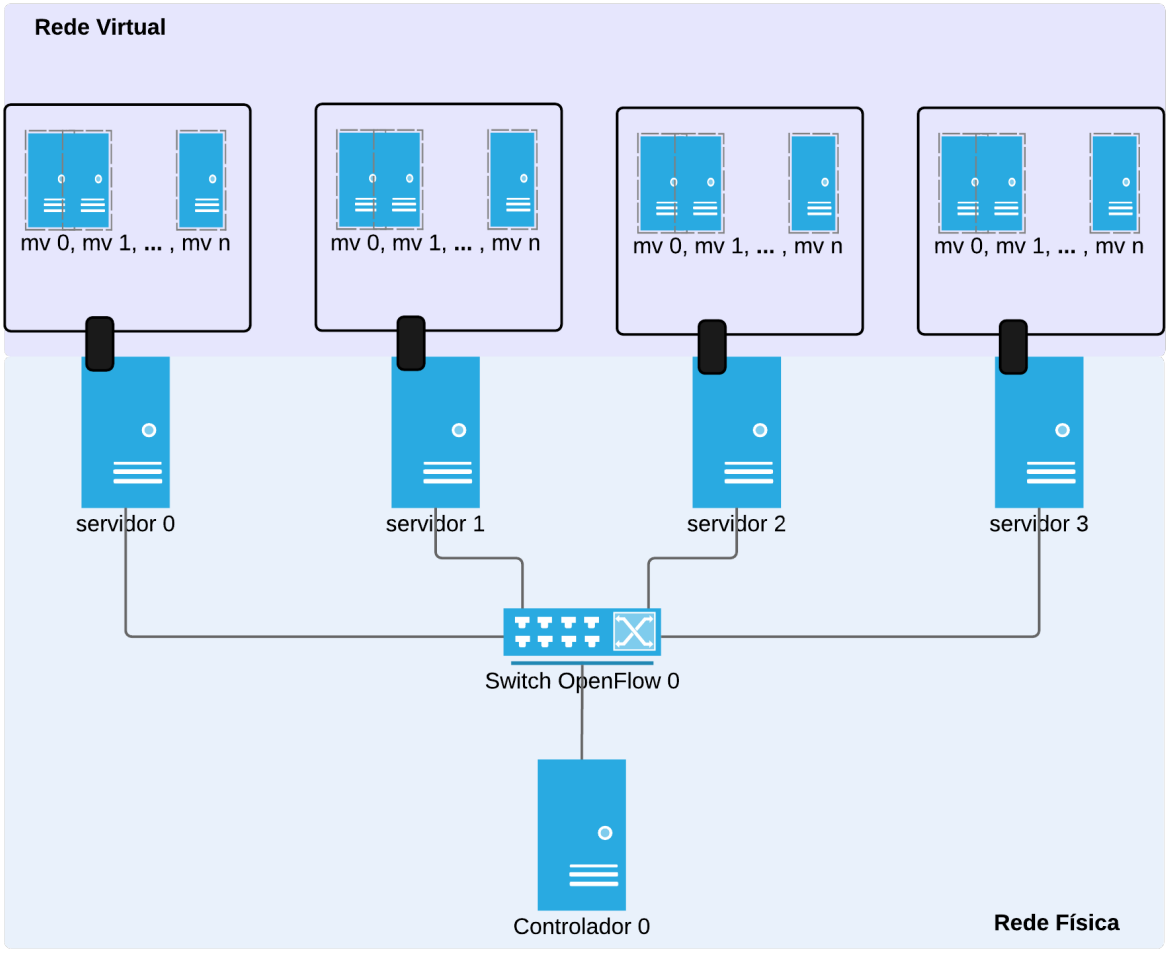

Figura 6.2: Arquitetura do ambiente de simulação

Uma rede de administração, com acesso aos servidores e a uma instância do controlador, na porta 6633. A segunda rede é a rede virtual à qual todos os POPs simulados estão conectados. Para implementar essas duas redes, cada servidor e o controlador possuem duas interfaces de rede que isolam seus funcionamentos. Assim, a rede física/administrativa e a rede virtual/Ipê estão separadas em nosso experimento.

Assim, como pode ser visto na figura 6.2, cada máquina virtual representada por *vm 0*, *vm 1*, *vm n*, é um POP. Em cada máquina virtual existe um *OpenVSwitch* que representa a borda da rede interna àquele POP. Como temos 27 unidades federativas

(POPs), temos 27 *switches OpenFlow*. Todos controlados pelo *controlador 0*. A relação dos *switches* reais da rede Ipê com os *switches OpenFlow* do ambiente simulado é baseada no gráfico de conectividade 6.3 disponível em [17].

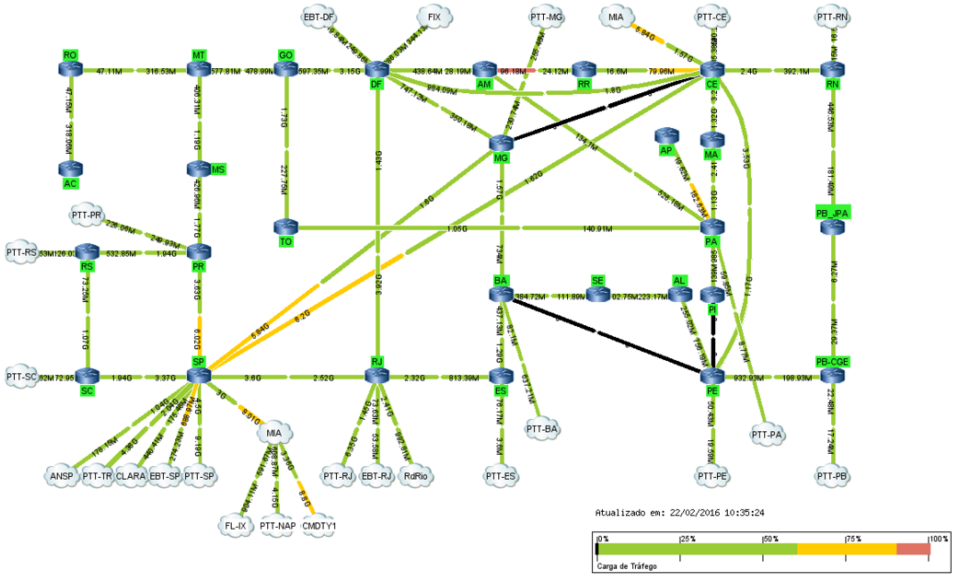

Figura 6.3: Conectividade da Rede Ipê

Para o ambiente virtualizado, o controlador está associado a porta 6653. Cada POP é um comutador (*switch*) OpenFlow simulado via OpenVSwitch [46]. As instituições clientes de cada POP são simuladas através do *Mininet* [37]. O número de instituições clientes foi coletado do sítio de cada Ponto de Presença.

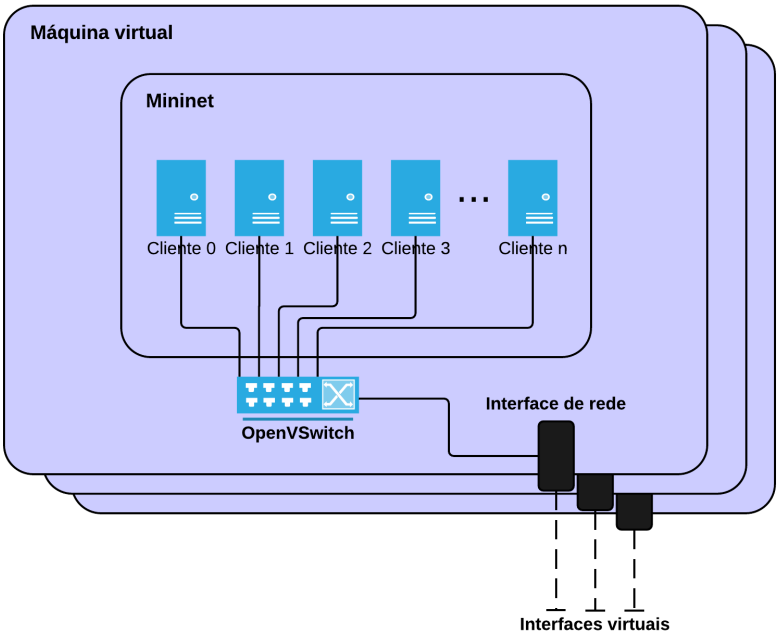

Figura 6.4: Arquitetura de cada máquina virtual que representa um POP

Cada máquina virtual representa um POP. Cada POP possui uma sub-rede à qual uma instância do *Mininet* simula seus clientes. Essa sub rede está conectada através de um *switch* virtual *OpenVSwitch* que é controlado pelo controlador da rede virtualizada. Assim, para cada POP, temos as decisões e atualizações da tabela de fluxos feita pelo controlador. Toda comunicação de saída das sub-redes utilizando *mininet* passam pelo *OpenVSwitch* que funciona em modo *bridge* (ponte) com a interface de rede virtual conforme pode ser visto na figura 6.4.

A tabela 6.1 apresenta as sub-redes de cada unidade federativa (POP), qual servidor físico gerencia a máquina virtual referente ao POP e o número de clientes a ele associado.

Associação de Servidores à simulação da rede IPÊ					
UF	**N clientes**	**Sub-rede local**	**Servidor**	**IP local**	**IP Gateway**
AC	10	10.10.1.0	Shiva	10.10.42.51	
AL	13	10.10.2.0	Shiva	10.10.42.52	
AM	20	10.10.3.0	Shiva	10.10.42.53	
AP	7	10.10.4.0	Shiva	10.10.42.54	10.10.42.50
BA	25	10.10.5.0	Shiva	10.10.42.55	
CE	50	10.10.6.0	Shiva	10.10.42.56	
DF	16	10.10.7.0	Shiva	10.10.42.57	
ES	26	10.10.8.0	Eden	10.10.42.101	
GO	26	10.10.9.0	Eden	10.10.42.102	
MA	7	10.10.10.0	Eden	10.10.42.103	
MG	32	10.10.11.0	Eden	10.10.42.104	10.10.42.100
MS	10	10.10.12.0	Eden	10.10.42.105	
MT	8	10.10.13.0	Eden	10.10.42.106	
PA	13	10.10.14.0	Eden	10.10.42.107	
PB	14	10.10.15.0	Diablos	10.10.42.151	
PE	41	10.10.16.0	Diablos	10.10.42.152	
PI	25	10.10.17.0	Diablos	10.10.42.153	
PR	71	10.10.18.0	Diablos	10.10.42.154	10.10.42.150
RJ	27	10.10.19.0	Diablos	10.10.42.155	
RN	11	10.10.20.0	Diablos	10.10.42.156	
RO	10	10.10.21.0	Diablos	10.10.42.157	
RR	7	10.10.22.0	Leviathan	10.10.42.201	
RS	10	10.10.23.0	Leviathan	10.10.42.202	
SC	47	10.10.24.0	Leviathan	10.10.42.203	10.10.42.200
SE	13	10.10.25.0	Leviathan	10.10.42.204	
SP	25	10.10.26.0	Leviathan	10.10.42.205	
TO	11	10.10.27.0	Leviathan	10.10.42.206	

Tabela 6.1: Mapeamento de endereços IP internos e externos de cada POP

6.2 Detecção de entidades

Ao ser carregado, o módulo *graph* inicia o grafo da rede vazio. Os *switches* são os primeiros a serem identificados. Como o controlador está ligado diretamente a eles pela interface *OpenFlow*, um evento de *ConnectionUp* é disparado pelo núcleo do controlador. Esse evento dispara o evento *SwitchJoin* através do módulo *topology* que faz com que o grafo adicione vértices.

```
1 INFO:topology.graph:SwitchJoin id: 2
2 INFO:topology.graph:SwitchJoin id: 1
3 INFO:topology.graph:1, 2
4 DEBUG:openflow.discovery:Dropping LLDP packet 275
5 INFO:topology.graph:LinkEvent fired
6 INFO:host_tracker:Learned 1 1 7e:e6:9b:89:39:2e got IP 10.0.0.1
7 INFO:topology.graph:HostJoin id: 7e:e6:9b:89:39:2e
8 INFO:host_tracker:Learned 2 1 62:77:44:24:13:49 got IP 10.0.0.2
9 INFO:topology.graph:HostJoin id: 62:77:44:24:13:49
```

Figura 6.5: Detecção de entidades

Nas duas primeiras linhas do *log* mostrado na figura 6.5, o módulo *graph* foi notificado da descoberta de dois *switches* na rede. Na linha 5, nota-se a descoberta de um enlace (*link*) entre dois comutadores (*switches*). O módulo *openflow.discovery*, através do protocolo LLDP identificou o link entre *switches*. O grafo foi notificado e estabeleceu uma aresta entre os vértices (*switches*).

As linhas 7 e 9 mostram a descoberta de dois hosts. Esses *hosts* foram descobertos pelo módulo *host_tracker* via escuta dos eventos de DHCP. É importante ressaltar que a descoberta de *hosts* também ocorre independente do DHCP. Um novo pacote que passa por um *switch* e não possui regra instalada na tabela de fluxos é encaminhado ao controlador que dispara um evento de *PacketIn*. Para tal, o *host_tracker* se encarrega de escutar esse evento e notificar o grafo através do evento *HostJoin* disparado pelo módulo *topology* ao criar um novo *Host*. O grafo, ao ser notificado, cria vértices para esses *hosts* associando uma aresta entre eles e o *switch* ao qual eles estão conectados. Dessa forma as entidades da rede são identificadas e computadas no grafo.

6.3 Remoção de entidades

6.3.1 Remoção de computadores

Diversos computadores foram desligados de maneira aleatória. O *host_tracker*, após um tempo fixo (*timerInterval*) verifica via ARPPing se os *hosts* conhecidos estão ativos. Para esse cenário da remoção de um computador (*host*), o *host_tracker* identifica a inatividade do *host* e remove o *Host*. O evento de *HostLeave* é disparado pelo *topology*, atualizando assim o grafo.

Nesse cenário de experimento, cinco computadores de cada sub-rede foi desligado. Para cada computador foi coletado o tempo em que ele foi desligado. Quando o controlador, através do *host_tracker*, identifica a saída desse computador o tempo é novamente coletado. Assim, para cada computador desligado, foi computado o tempo decorrido até que o controlador o removesse do grafo da rede.

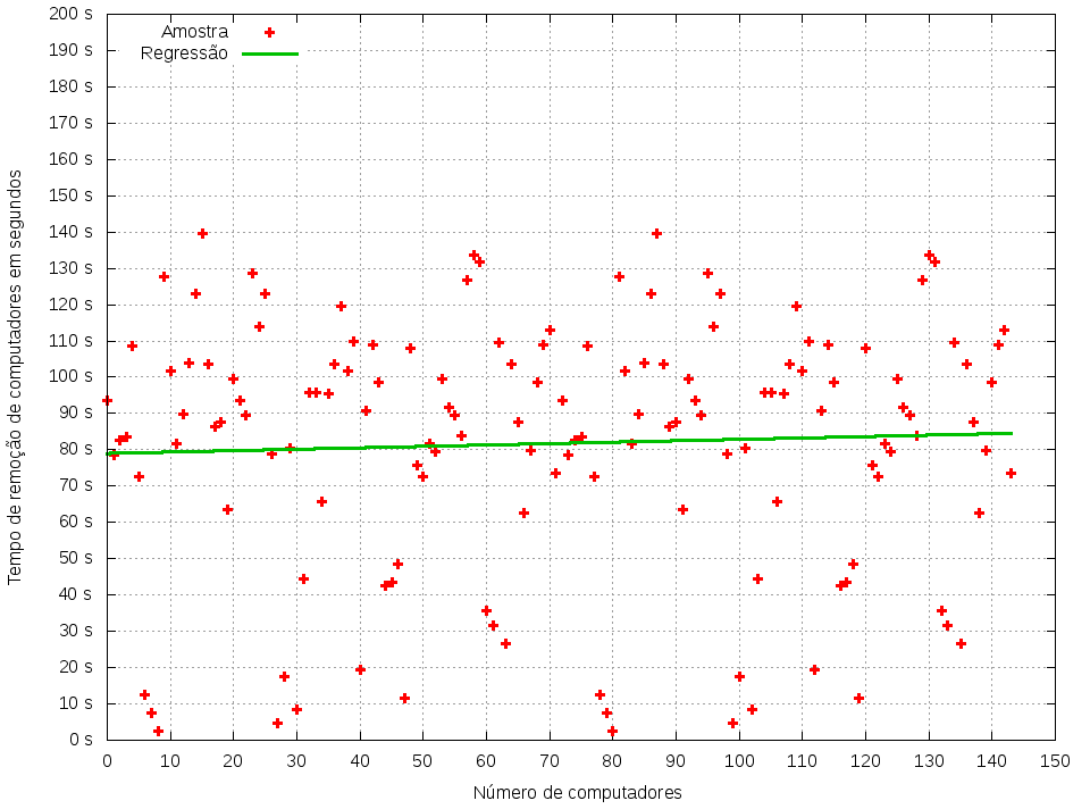

Figura 6.6: Tempo decorrido entre o desligamento de um computador e a remoção do mesmo no grafo

Os resultados da amostra coletada pode ser visto na figura 6.6. Em média, o

tempo de remoção de um computador do grafo demorou 80 segundos. A sondagem foi configurada para ser executada com uma periodicidade de 100 segundos. A regressão linear computada mostra que esse valor médio é praticamente constante ao longo dos valores amostrados. Computadores cuja remoção ocorreram em um intervalo de tempo muito reduzido tiveram seu tempo de sondagem expirado muito próximos ao momento em que o *host_tracker* executava sua sondagem.

6.3.2 Remoção de comutadores

Para o caso de comutadores (*switch*), foram desligados os *switches* e medido o tempo de atualização do grafo. O controlador está ligado diretamente aos *switch*. Em função disso os comutadores são removidos mais rapidamente do grafo. A figura 6.7 apresenta os resultados da computação da amostra de *switches* removidos. Assim, ao ser desligado, o *core* do POX dispara um evento de *SwitchLeave* ao qual o módulo *graph* está inscrito. Logo, o grafo é atualizado removendo o vértice do *comutador*.

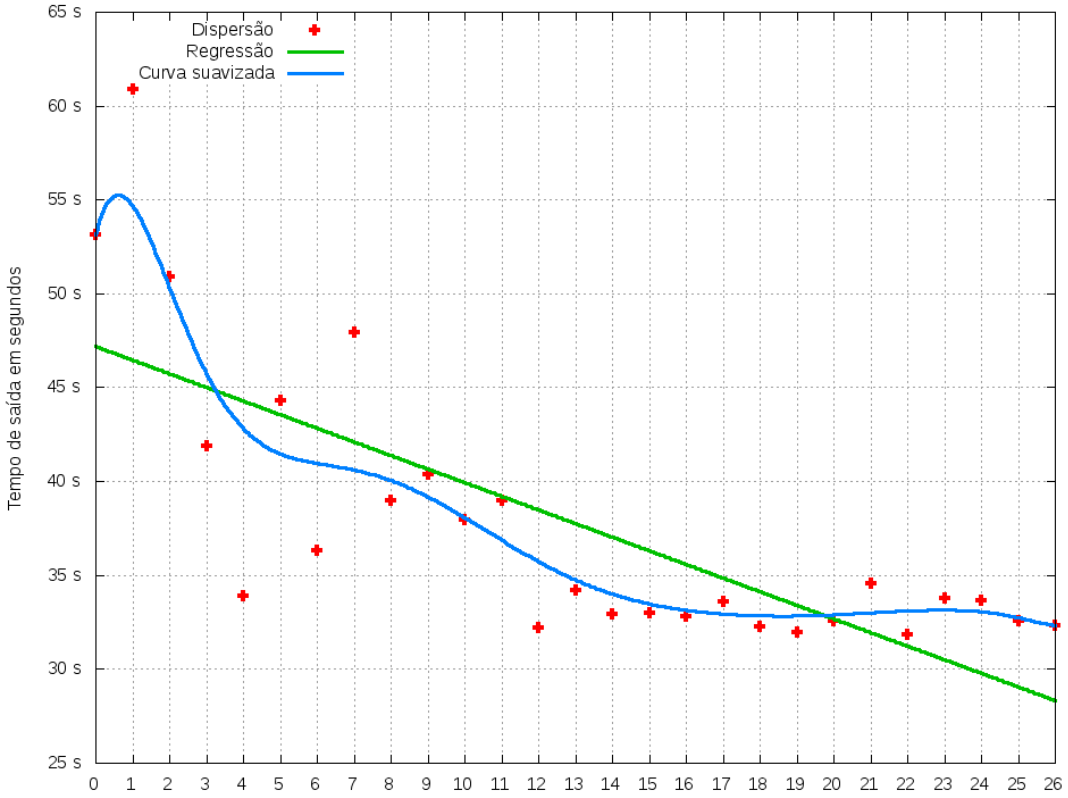

Figura 6.7: Tempo decorrido entre o desligamento de um comutador (*switch*) e a remoção do mesmo no grafo

Uma curva suavizada da interpolação dos pontos mostra que apesar dos valores de tempo mais elevados nos primeiros valores da amostra, ao final esses valores são praticamente constantes, em um valor de 30 a 35 segundos para cada remoção.

Após algum tempo o *host_tracker* identifica se os computadores associados ao *comutador* removido estão ativos por outra rota. Caso negativo, os computadores são removidos do grafo. Considerando o experimento da remoção dos comutadores (*switches*), foi computado o tempo decorrido entre a remoção do *comutador* e a remoção dos computadores na rede interna ao *comutador*.

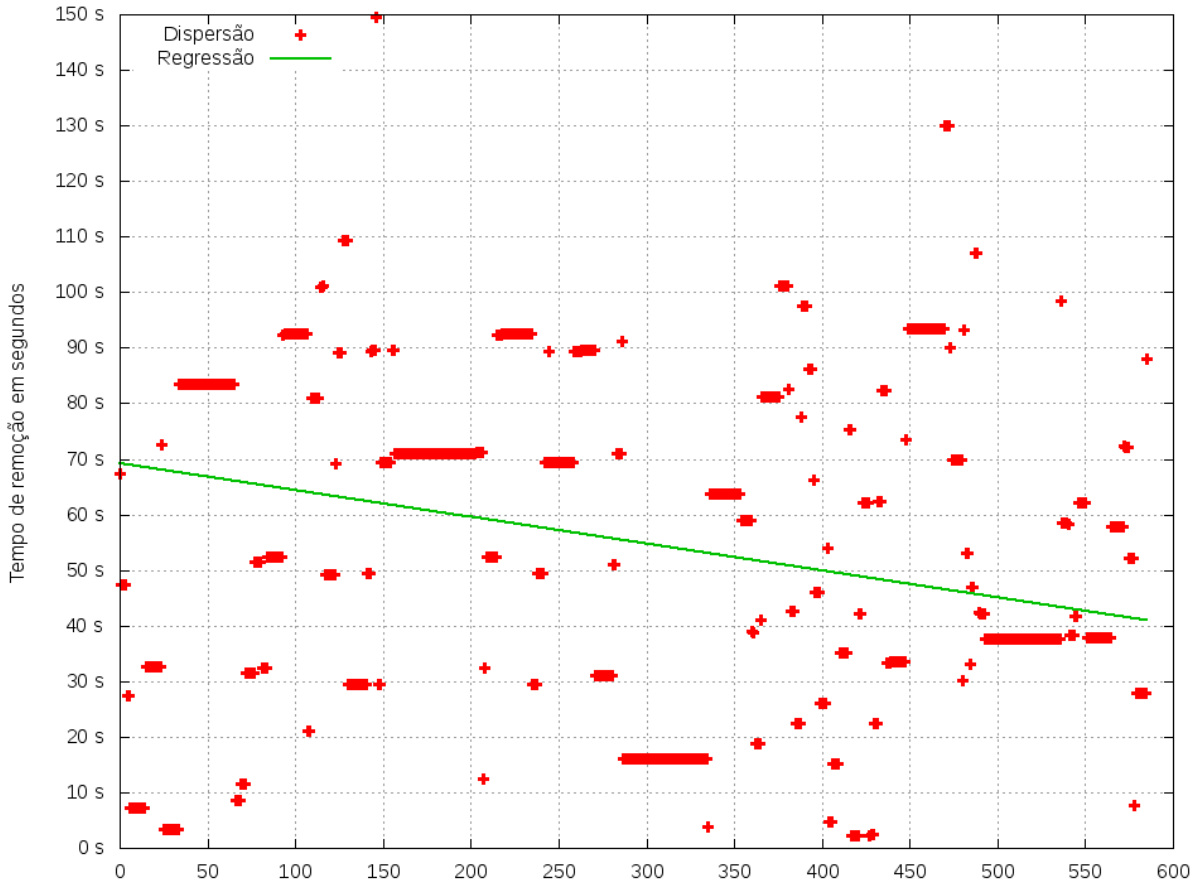

Figura 6.8: Tempo decorrido entre a remoção dos comutadores (*switches*) e a remoção dos computadores na rede interna ao *(*switch)

A figura 6.8 mostra o resultado do experimento para o tempo de remoção de computadores em função da remoção do *comutador*. Os resultados mostram uma média que varia entre 40 e 70 segundos para a remoção dos computadores. No geral nota-se que computadores de uma mesma sub-rede foram removidos em instantes próximos. Isso pode ser notado pelos pequenos grupos de pontos muito próximos na figura.

6.4 Visualização em tempo real

Uma imagem representando o grafo da rede é gerada pelo módulo *graph*. Essa imagem consiste na representação da topologia da rede no instante em que foi gerada. Periodicamente uma nova imagem é criada. Um exemplo de grafo é apresentado na figura 6.9. Essa figura representa uma rede simulada, dentro do ambiente do *Mininet*, com 8 comutadores (*switches*), cada um com 30 computadores (*hosts*) conectados.

Essa topologia totaliza 248 entidades na rede. Na figura, os vértices vermelhos representam os comutadores. Os vértices azuis, os computadores. Cada aresta é um enlace entre dois vértices.

Atualizações na topologia como, remover e adicionar entidades na rede, foram executados. A visualização da rede atualiza junto com as alterações no grafo.

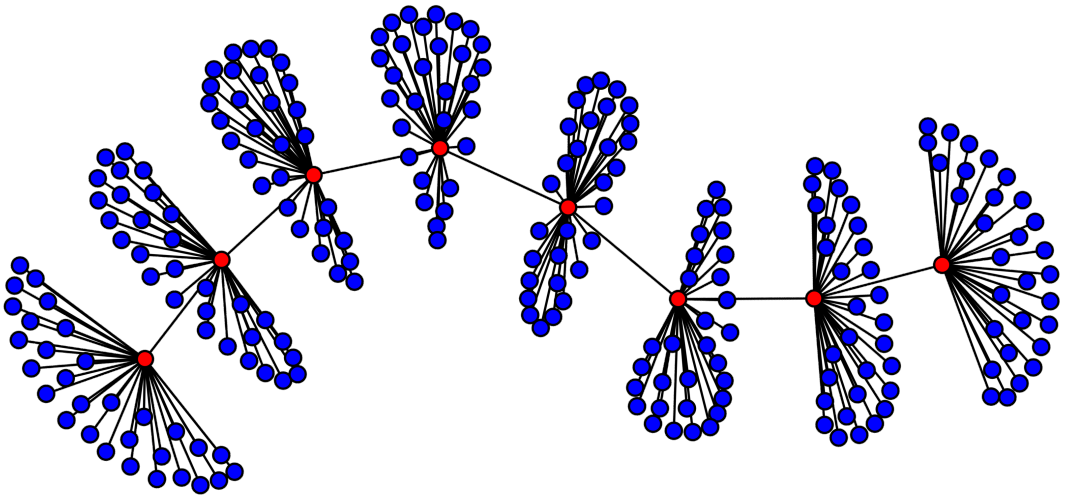

Figura 6.9: Grafo representando uma rede com 248 entidades

Um experimento foi executado na rede Ipê para mostrar as alterações na representação do grafo.

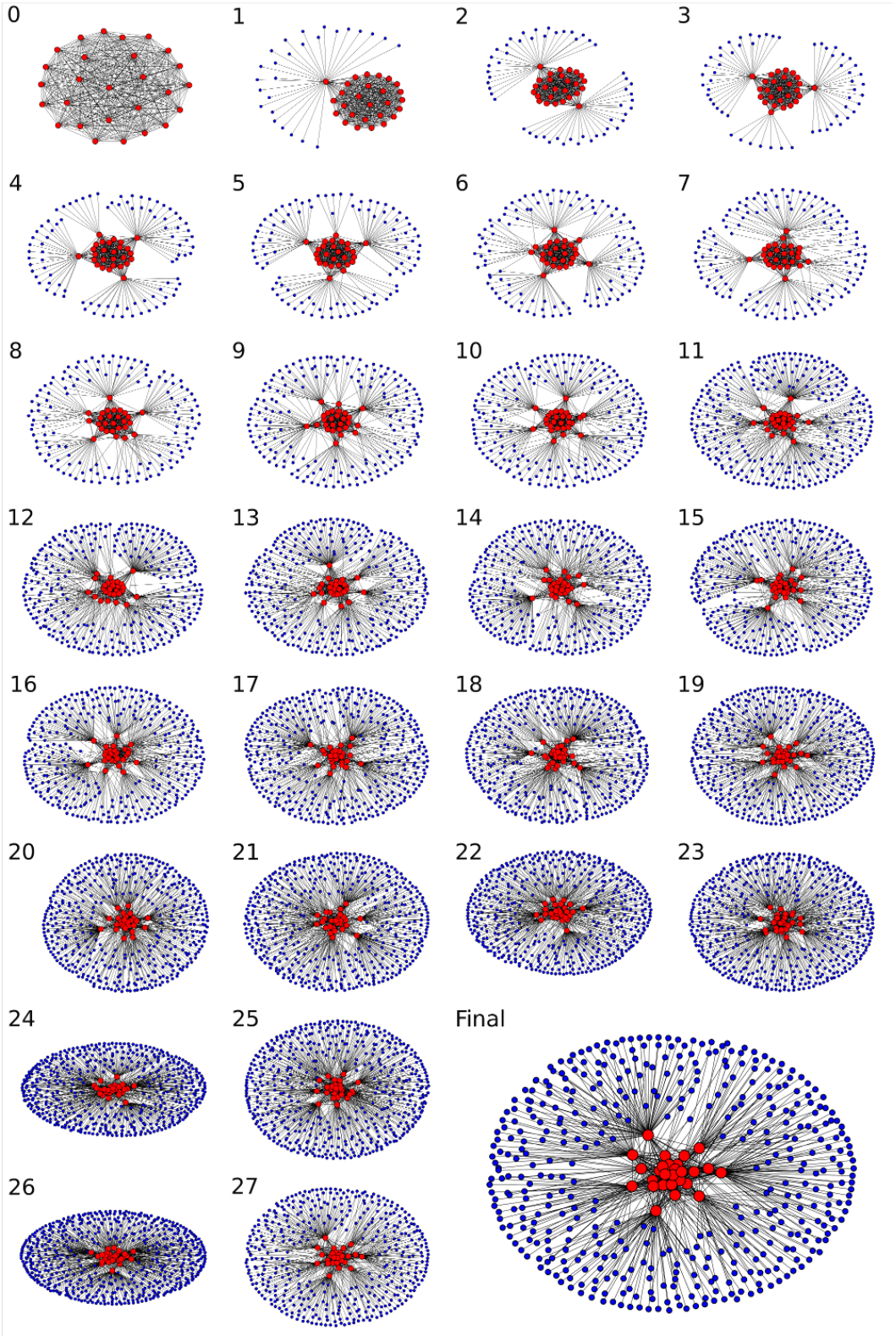

Figura 6.10: Representação da rede Ipê à medida que os computadores de cada sub-rede são identificados

No instante inicial, o controlador é iniciado. Cada comutador estabelece uma conexão segura com o controlador. Nesse momento, todos os vértices comutadores são adicionados ao grafo. No segundo instante, para apenas uma sub-rede, representando uma unidade federativa dentro da rede Ipê, foi simulado tráfego entre seus computadores. Para cada computador, uma sondagem de todos os outros computadores dessa sub-rede foi feita através do utilitário *ping*. Assim, todas as máquinas dessa sub-rede foram identificadas pelo controlador e adicionadas ao grafo representando a rede. No terceiro instante, outra sub-rede foi escolhida e a mesma simulação de tráfego foi executada. O mesmo procedimento foi repetido até que todas as 27 sub-redes da rede Ipê fossem computadas pelo grafo.

A figura 6.10 apresenta a execução desse experimento cronologicamente. Cada número representa a quantidade de redes computadas pelo grafo no instante da execução do experimento. Uma rede com 563 computadores e 27 comutadores (*switches*) é mostrada na imagem Final da figura 6.10 representando toda a rede Ipê. No total o grafo computou 590 vértices.

6.5 Identificação de tráfego

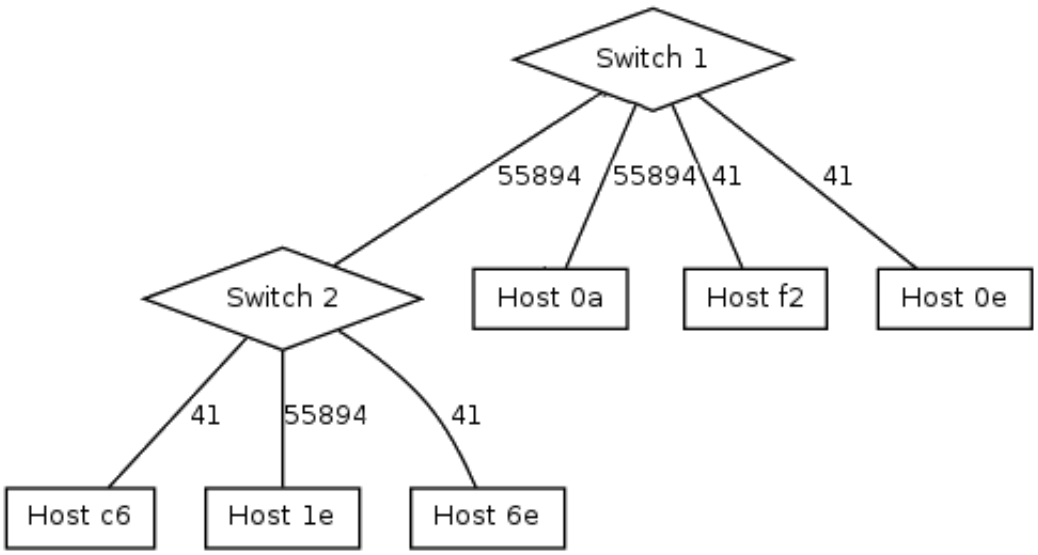

Figura 6.11: Tráfego TCP (em bytes) entre os *hosts* 'Host 1e' e 'Host 0a'

Para computar o tráfego (TCP) em uma rede simples foi executado o programa *iperf* como servidor no *host* 'Host 0a'. A topologia dessa rede é composta por dois comutadores interligados. Cada comutador possui três computadores em cada sub-rede. O

host 'Host 1e' conecta-se como cliente. Conformo pode ser notado na figura 6.11, o tráfego em bytes na arestas desses *hosts* é superior ao demais *hosts*. No momento em que foram lidos os contadores *OpenFlow* e computados os pesos das arestas, obteve-se um tráfego de 55894 bytes através do caminho entre os dois *hosts* citados. Os valores apresentados para os demais *hosts* (41 bytes) são referentes ao tráfego ARP PING disseminado pelo módulo *host_tracker*

6.6 Avaliação do Controlador

O consumo da unidade central de processamento (CPU) no processo do controlador, no sistema operacional, assim como a utilização de memória e taxa de escrita em disco foram medidos através da biblioteca utilitária *sysstat* [58]. Um experimento variando o número de computadores na rede permitiu coletar as métricas citadas acima. Coletou-se durante 30 segundos a utilização da CPU do processo do controlador, do sistema operacional, a utilização de memória e a taxa de escrita em disco em cada interação da execução do experimento. A cada iteração do experimento, 50 novos computadores foram adicionados a rede. Ou seja, de 50 até 550.

6.6.1 Consumo de processador no processo do controlador

Conforme pode ser visto na figura 6.12, considerando a variação das medições feitas no experimento, o crescimento da curva de utilização de CPU mostra-se logarítmica. Apesar desse comportamento para o processo do controlador, nota-se que o consumo, analisando todo o sistema, é linear. Esses resultados são apresentados na próxima seção. O comportamento logaritmo mostra que, apesar do aumento do número de computadores na rede, o processo do controlador não gerou exaustão na CPU alocada para o processo ao longo do experimento. À medida que a rede cresce, a utilização da CPU escalonada ao processo do controlador, cresce na mesma proporção.

Uma regressão linear foi computada em cima da amostra coletada da utilização de CPU pelo processo do controlador. A figura 6.13 apresenta os resultados dessa computação. É possível notar o crescimento da utilização de CPU à medida que a rede cresce. A dispersão dos valores no início da amostragem estão mais próximos de 0. Ou seja, a rede com poucos computadores demandou baixo processamento. Já nos instantes finais,

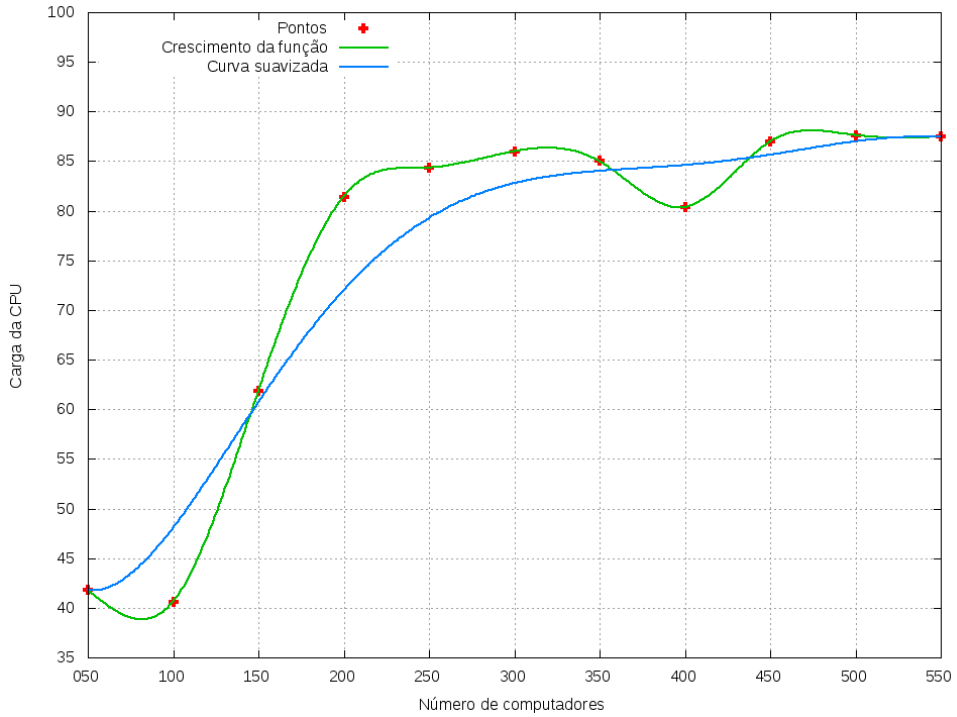

Figura 6.12: Variação do consumo médio de CPU no processo do controlador

há mais pontos amostrados próximos a 100% da utilização de CPU, o que mostra o estresse do processador com uma rede maior. A reta representada pela regressão mostra o quão crescente é a utilização de CPU pelo processo à medida que a rede cresce.

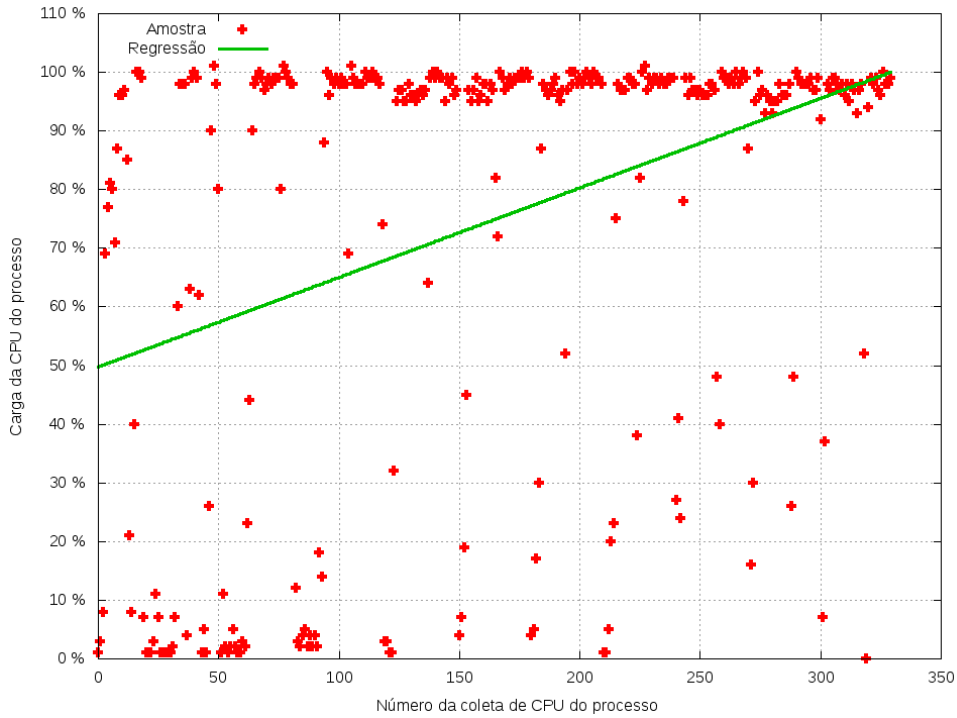

Figura 6.13: Regressão linear do % de CPU do controlador à medida que a rede crescia

6.6.2 Consumo do controlador em relação ao sistema operacional

Para computar um rede com 550 computadores o controlador utilizou 10% da CPU do sistema operacional. Como pode ser visto na figura 6.14 o crescimento da função de utilização de CPU do sistema é praticamente linear. À medida que a rede cresce, os processadores do computador do controlador aumentam sua carga.

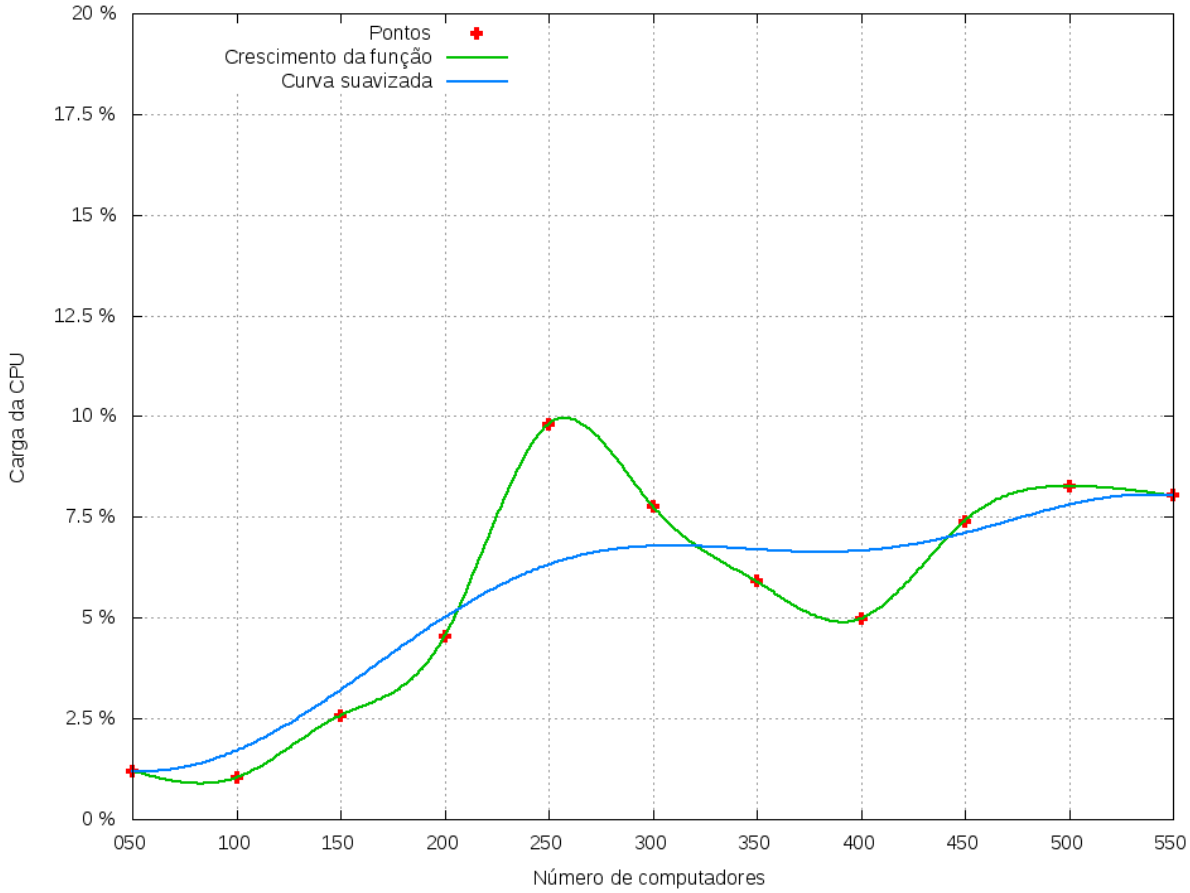

Figura 6.14: Crescimento da função de utilização da CPU do sistema operacional em função do crescimento de computadores na rede

O controlador POX é escrito na linguagem de programação *Python*. Cada nova entidade (computador ou comutador) ao ser detectado pelo controlador recebe um identificador único. Essas entidades são armazenadas em um dicionário *Python*. O dicionário é uma estrutura de dados primitiva da linguagem que implementa uma tabela *hash* [40]. Em função disso, para a computação do experimento temos que a complexidade de tempo é na ordem de $O(n)$ conforme mostrado a seguir.

As operações de adição, remoção e busca em tabelas *hash* tem, no caso médio, custo constante na ordem de $O(1)$. Assim, considerando-se n o número de vértices, temos

que para cada vértice do grafo o custo para inseri-lo durante o experimento é $O(1)$. A complexidade de tempo do experimento, à medida que a rede cresce, é na ordem de $O(n)$.

Uma regressão linear foi computada da amostra coletada da carga de *CPU* do sistema operacional durante o experimento. A figura 6.15 apresenta o resultado da regressão.

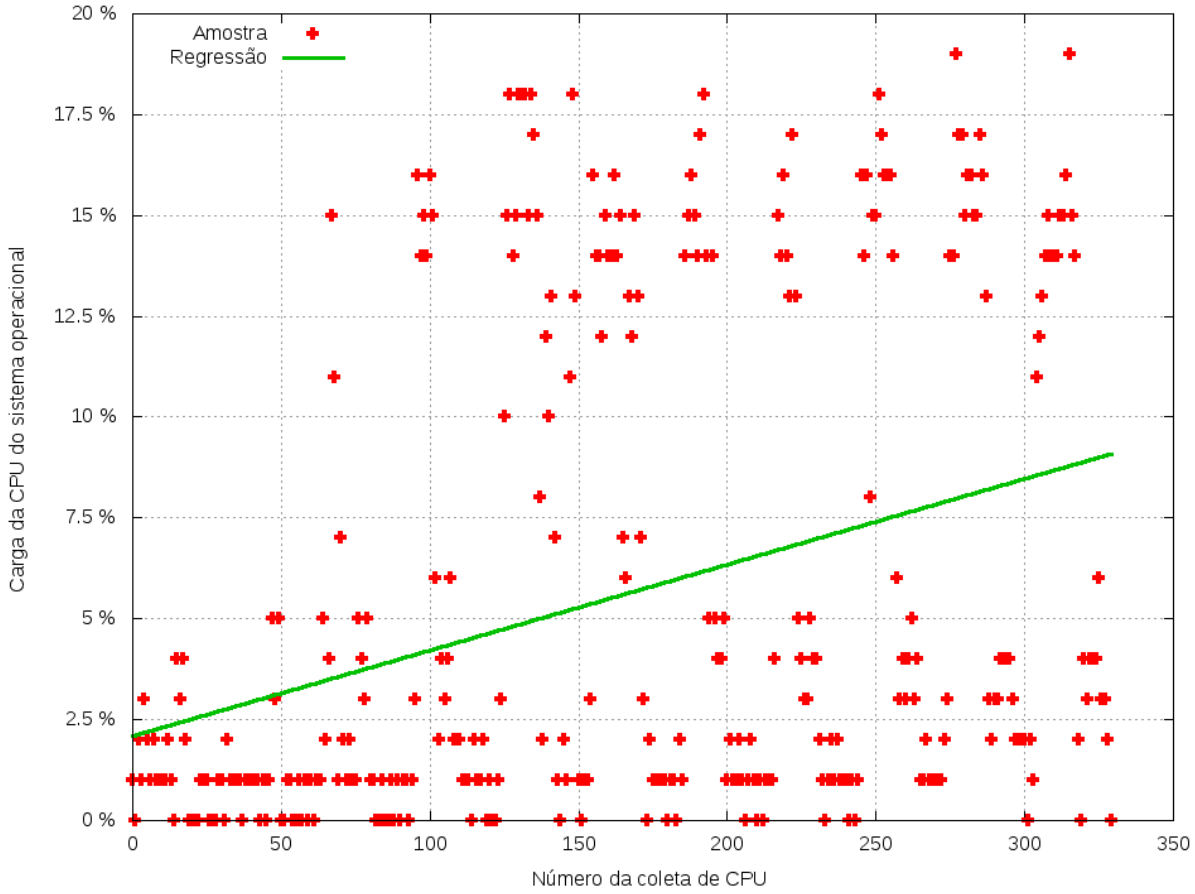

Figura 6.15: Regressão linear sobre a carga de CPU do sistema operacional

6.6.3 Consumo de memória

A figura 6.16 apresenta o crescimento da utilização de memória à medida que a rede cresce. O computador do controlador possui 8 *Gigabytes* de memória RAN (*random access memory*). O consumo de memória pelo controlador durante o experimento variou de 2% a 4% da memória global do sistema operacional.

O módulo *Graph* adiciona vértices para cada computador e comutador identificados na rede. Cada vértice é um objeto da classe *Vertex*. Esse objeto contém a referência

para objetos das classes *Host* e *Switch* da entidade (computador ou comutador) que é representada pelo vértice. Assim, à medida que vértices são adicionados à rede, novos objetos são criados. As arestas são objetos da classe *Edge* e guardam referências para os objetos dos vértices que compõem a aresta.

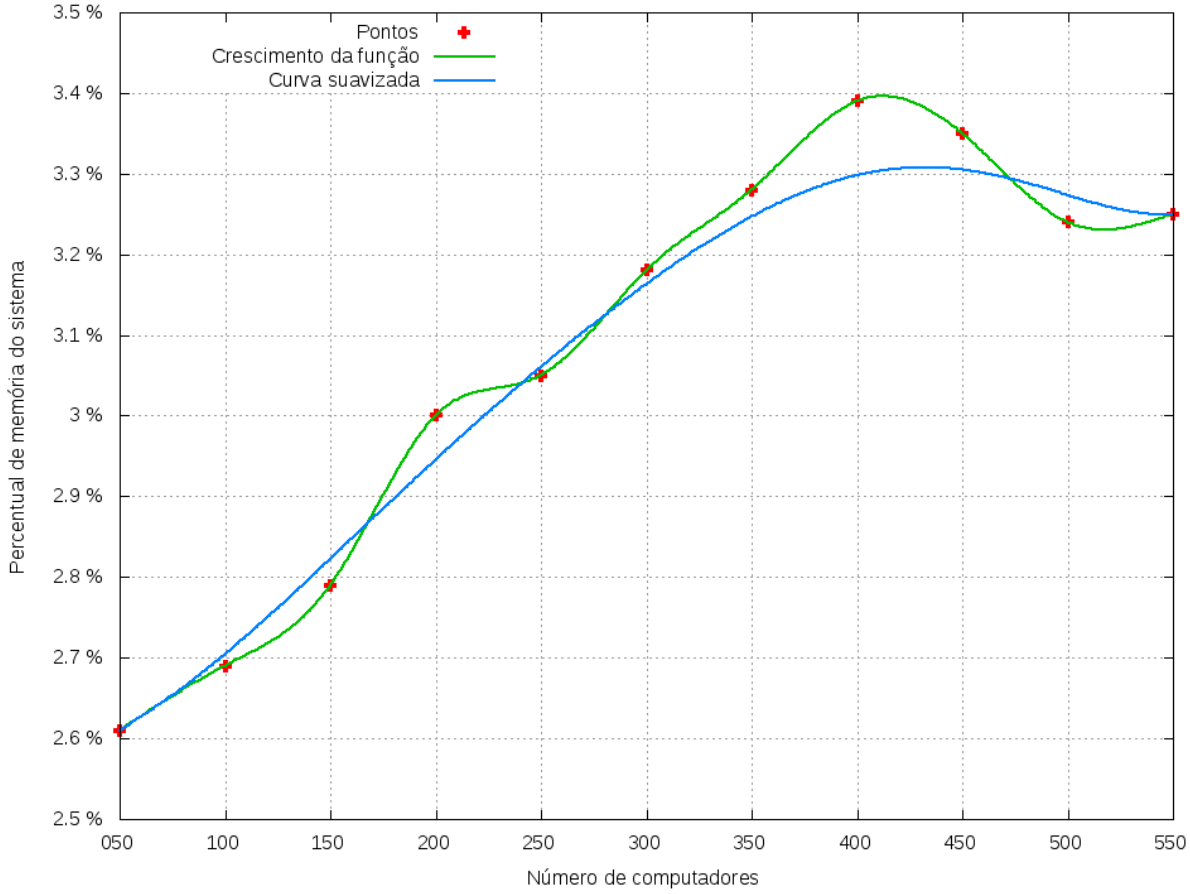

Figura 6.16: Crescimento do percentual de utilização de memória pelo controlador à medida que o número de computadores na rede aumenta

Em função disso, assumiu-se que: Um objeto *Vertex* ocupa na memória um espaço de endereçamento de tamanho x. E um objeto *Edge* ocupa y. Considerando-se n o número de vértices e m o número de arestas do grafo, temos que a função que representa a utilização de memória (complexidade de espaço) é $nx + my$. Ou seja, o número de vértices no grafo multiplicado pelo tamanho de um objeto de vértice na memória somado ao número de arestas do grafo multiplicado pelo tamanho de um objeto de aresta na memória.

A função de complexidade de espaço $nx + my$ mostra que o crescimento de utilização de memória do computador é dada em função do aumento de vértices e arestas na rede. No experimento executado, o número de vértices (coeficiente n) cresceu linearmente. A regressão linear computada mostra o crescimento na utilização de memória pelo controlador à medida que mais computadores foram inseridos na rede ao longo da

amostragem. A figura 6.17 apresenta o resultado dessa computação.

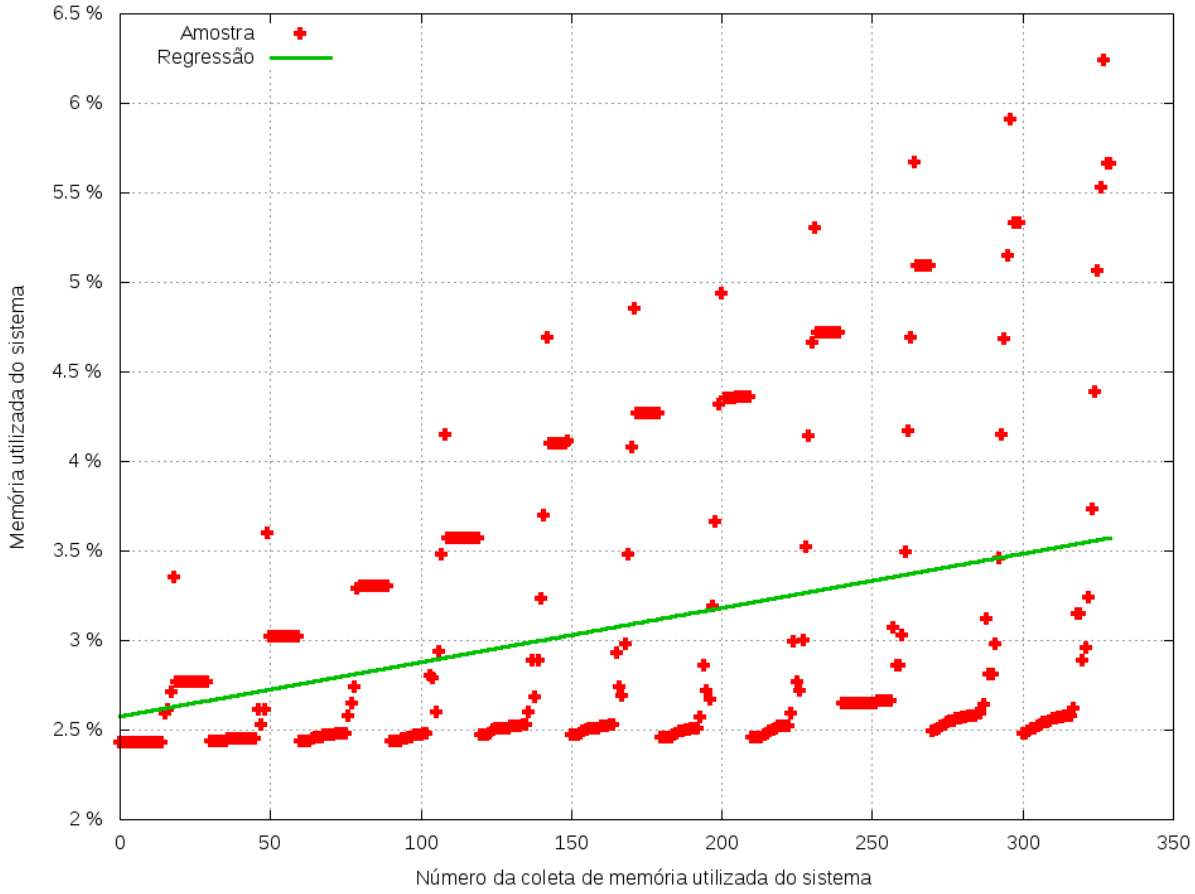

Figura 6.17: Regressão linear da amostra do consumo de memória do sistema à medida que rede cresce

6.6.4 Taxa de escrita em memória secundária

O módulo *Graph* periodicamente salva uma imagem em memória secundária com a visualização do grafo. Durante o experimento de avaliação do controlador foi coletada a taxa de escrita em disco à medida que a rede crescia.

Em média, a taxa de escrita foi de 2 *Megabytes* ao longo de cada iteração do experimento. Para o experimento, uma imagem era gravada a cada 25 segundos. Em cada iteração, mais vértices haviam no grafo, logo, mais objetos a serem mapeados na imagem.

Na figura 6.18 é possível ver o crescimento na taxa de escrita em memória secundária à medida que a rede crescia. Os pontos em vermelho representam os valores máximos de escrita coletados em cada iteração do experimento. Uma interpolação passando por todos os pontos é dada pela curva de coloração verde. Uma suavização da curva de interpolação é traçada na cor azul mostrando o crescimento da taxa de escrita à medida que novos vértices vão sendo criados durante as iterações do experimento.

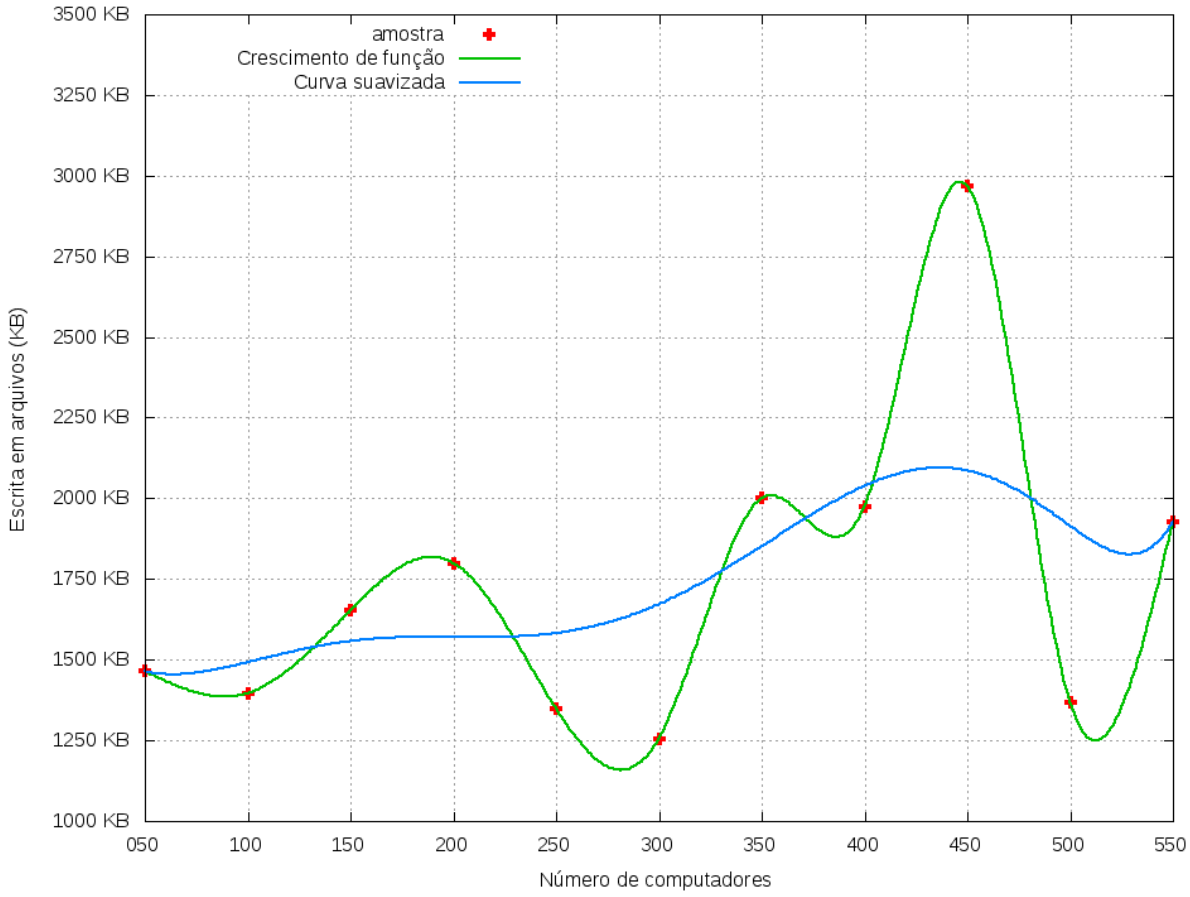

Figura 6.18: Taxa de escrita em memória secundária à medida que a rede cresce

Uma regressão linear foi computada na amostra coletada das taxas de escrita em memória secundária ao longo do experimento. A figura 6.19 apresenta o resultado dessa regressão mostrando o crescimento e a correlação entre os valores amostrados.

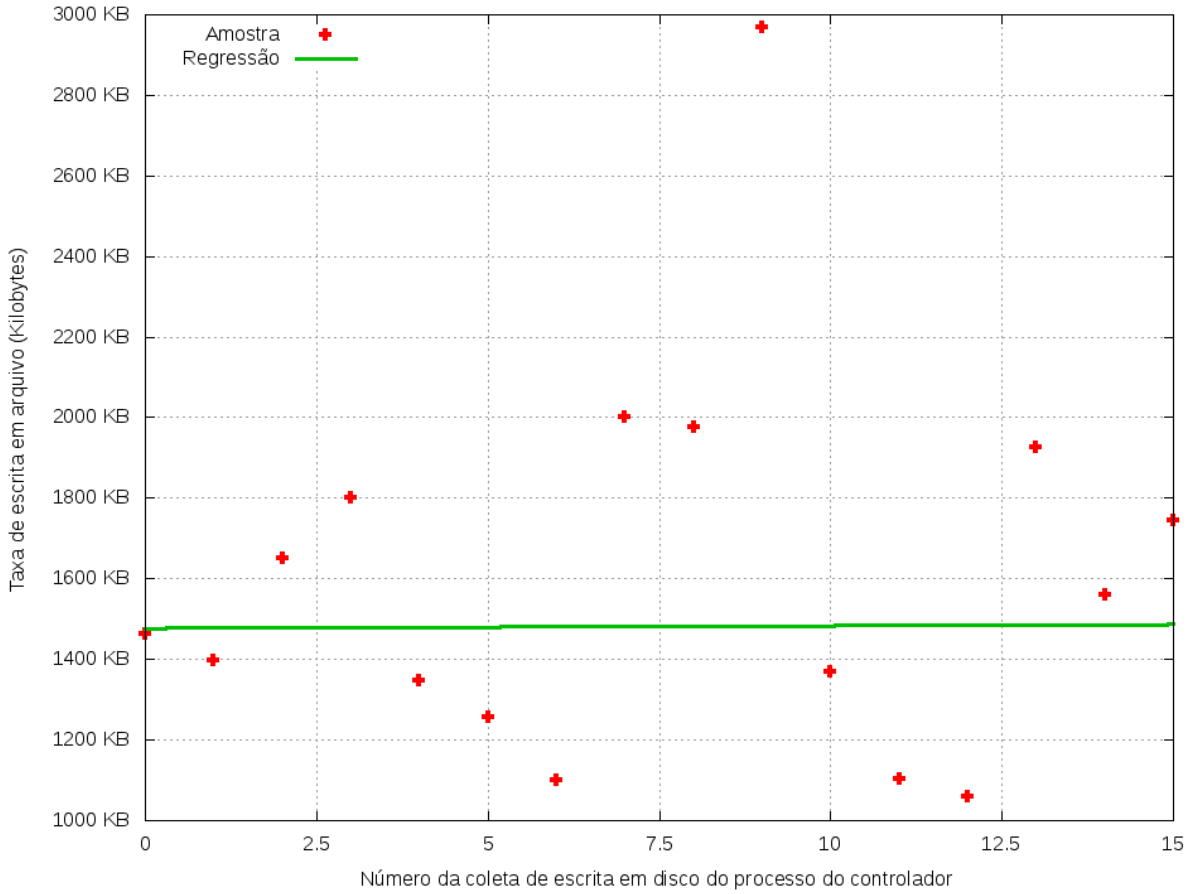

Figura 6.19: Regressão linear dos valores coletados da taxa de escrita em memória secundária

6.7 Avaliação do módulo *host_tracker*

O módulo *host_tracker* periodicamente envia pacotes *APPPings* de sondagem aos computadores da rede. Esta seção avalia o impacto e comportamento desse módulo em relação a rede e a solução.

Primeiramente é avaliada a largura de banda da rede e o impacto da sondagem de computadores nessa propriedade. Em seguida a proporção de pacotes manipulados pelo controlador é comparada em relação a quantidade de pacotes de sondagem.

6.7.1 Avaliação de largura de banda

Para medir a largura de banda foi utilizada a ferramenta de linha comando *iperf*. A rede Ipê foi simulada com o conjunto completo de computadores. 6 pares de clientes e servidores *iperf* foram executados durante 300 segundos. Cada par de cliente e servidor foi instanciado em máquinas virtuais diferentes e fisicamente separadas.

Teste	**pingLim**	**entryMove**	**arpReply**
Teste 0	5	50	5
Teste 1	4	40	4
Teste 2	3	30	3
Teste 3	2	20	2
Teste 4	1	10	1

Tabela 6.2: Tabela com a relação de parâmetros por teste

Foram executados 5 baterias de testes. Cada teste variou 3 parâmetro do módulo *host_tracker*. O parâmetro *pingLim* representa a quantidade de pacotes de sondagem necessários para considerar que um computador está inativo. O parâmetro *entryMove* é o tempo até que uma entrada do dicionário de endereços MAC seja alterado. E o parâmetro *arpReply* é o tempo a ser esperado pela resposta ARP até que uma nova tentativa de sondagem seja enviada. A tabela 6.2 mostra de maneira resumida os experimentos.

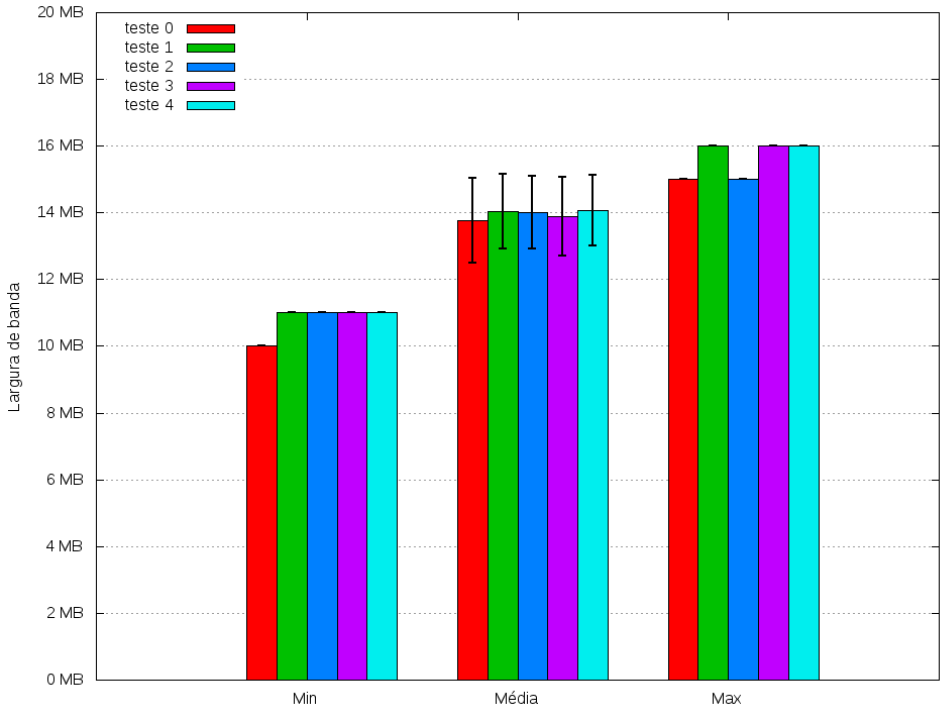

Figura 6.20: Larguras de banda mínima, média e máxima observadas

Para cada teste, durante 300 segundos, a cada 5 segundos a largura de banda foi coletada. A figura 6.20 mostra os valores mínimos, médio com intervalo de confiança e máximo da largura de banda coletada nos testes.

Como existiam 6 pares de clientes e servidores, a largura de banda é dividida entre esses computadores. A média foi computada do somatório da largura de banda computada por cada par de cliente e servidor.

Conforme visto na tabela 6.2 do Teste 0 ao Teste 4 os parâmetros avaliados foram decrementados. Em função disso, à medida que os testes foram executados nessa ordem, o intervalo de sondagem, o tempo de expiração e o tempo de aguardo por resposta reduziram. Essa redução faz com que a cada iteração de testes mais pacotes de sondagem fossem disseminados na rede. A figura 6.20 mostra que, apesar do aumento de pacotes de sondagem, a largura de banda média da rede não sofreu impacto considerável. A figura 6.21 apresenta a curva de interpolação entre os pontos representando a largura de banda média em cada teste do experimento.

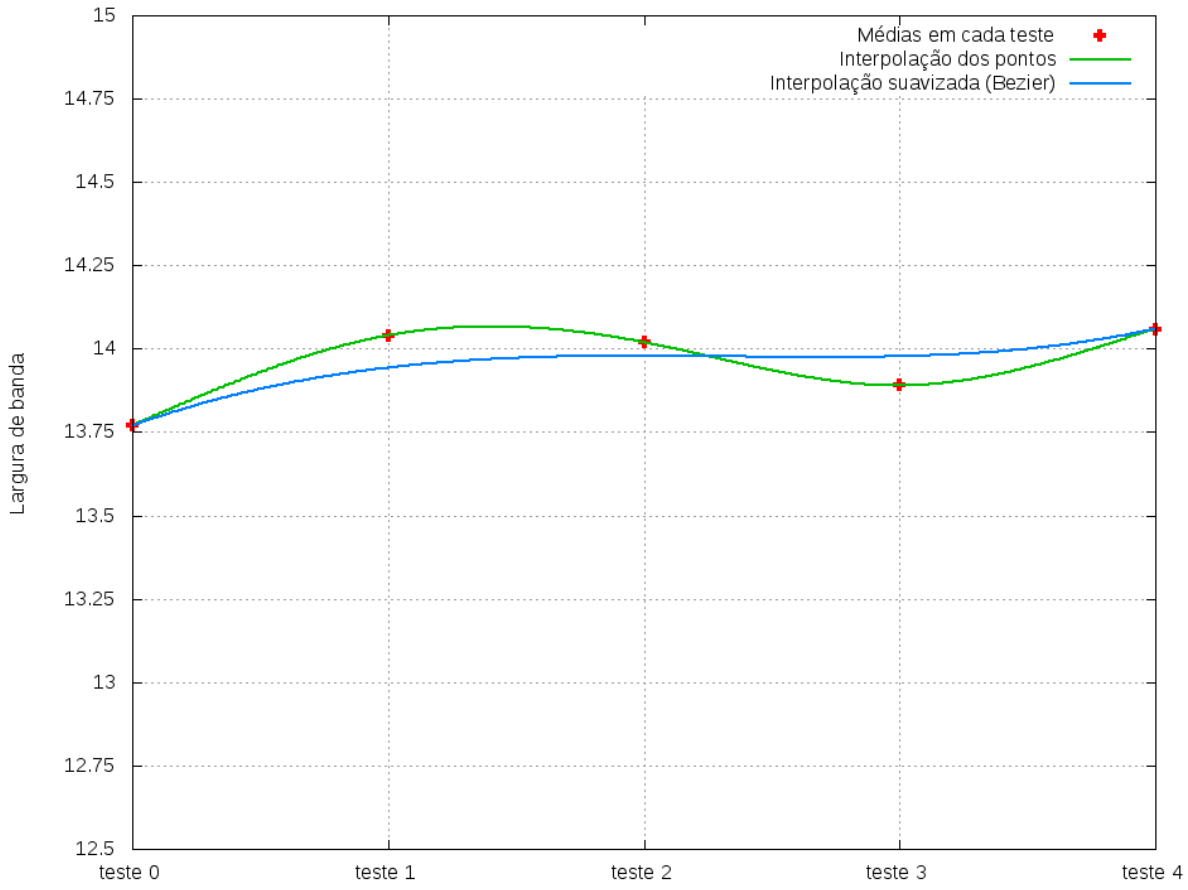

Figura 6.21: Curva de interpolação das médias das larguras de banda medidas em teste do experimento

6.7.2 Avaliação do número de pacotes

Essa subseção avalia a quantidade de pacotes de sondagem disparadas pelo módulo *host_tracker* à medida que mais computadores estão presentes na rede. São avaliados também, a quantidade de pacotes de entrada (*PacketIn*) que o controlador lida à medida que a rede cresce.

O experimento começou com 50 computadores na rede. Em cada iteração, novos 50 computadores foram adicionados à rede. Ao final, a rede Ipê simulada, possuía 550 computadores. Para computar o número de pacotes de sondagem foi adicionado um contador na função que envia esses pacotes. A figura 6.22 mostra o gráfico resultante dos valores computados dos número de pacotes de sondagem enviados à medida que mais computadores estavam presentes na rede.

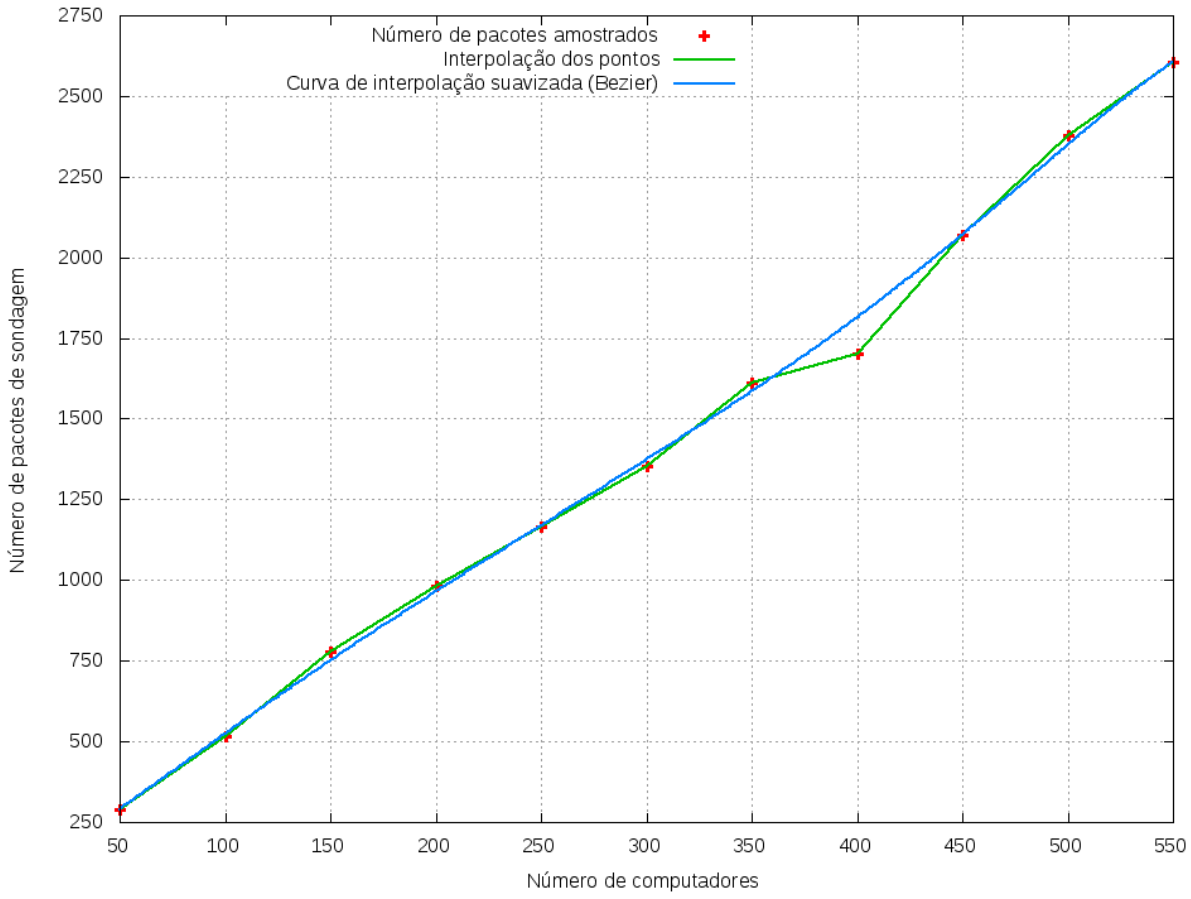

Figura 6.22: Crescimento do número de pacotes de sondagens encaminhados à medida que a rede crescia

O aumento do número de pacotes apresentou um crescimento linear. Ou seja, à medida que mais computadores entram na rede, o número de pacotes cresce na mesma proporção. Para esse experimento, foi calculada a diferença da quantidade de pacotes

da iteração posterior com a iteração anterior. Assim, é possível saber quantos pacotes variaram a cada iteração do experimento. Esse resultado é apresentado na figura 6.23.

Em média, 200 a 250 pacotes a mais foram disparados a cada iteração. O experimento durou, para cada iteração, dois minutos. Considerando esse intervalo de tempo e a média de pacotes enviados, pode-se dizer que aproximadamente 5 pacotes foram enviados para cada computador identificado pelo grafo. Levando em conta a média de 250 pacotes por iteração e que cada iteração adiciona 50 novos computadores, temos que a razão desses valores mostra que, para cada computador, durante dois minutos foram enviados 5 pacotes de sondagem.

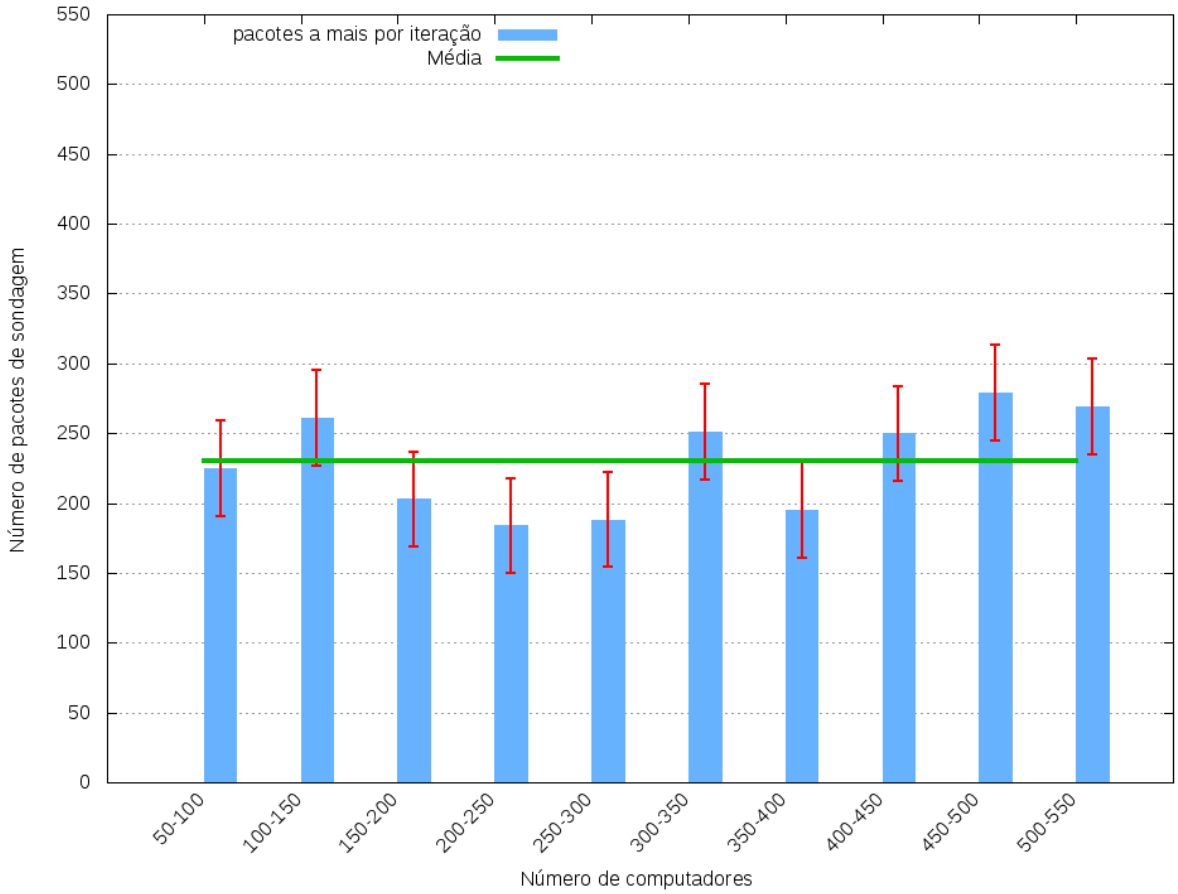

Figura 6.23: Diferença da quantidade de pacotes de sondagem enviados entre cada par de iterações do experimento

O número de pacotes de entrada foi contabilizado contando as chamadas do evento *PacketIn* no controlador. Foram medidos quantos pacotes de entrada o controlador manipulava a cada iteração do experimento.

A figura 6.24 mostra o crescimento no número de pacotes ao longo do experimento. Assim como o número de pacotes de sondagem, o número de pacotes (*PacketIn*) cresce linearmente. Ou seja, à medida que mais computares estão na rede, o volume de pacotes de entrada cresce proporcionalmente.

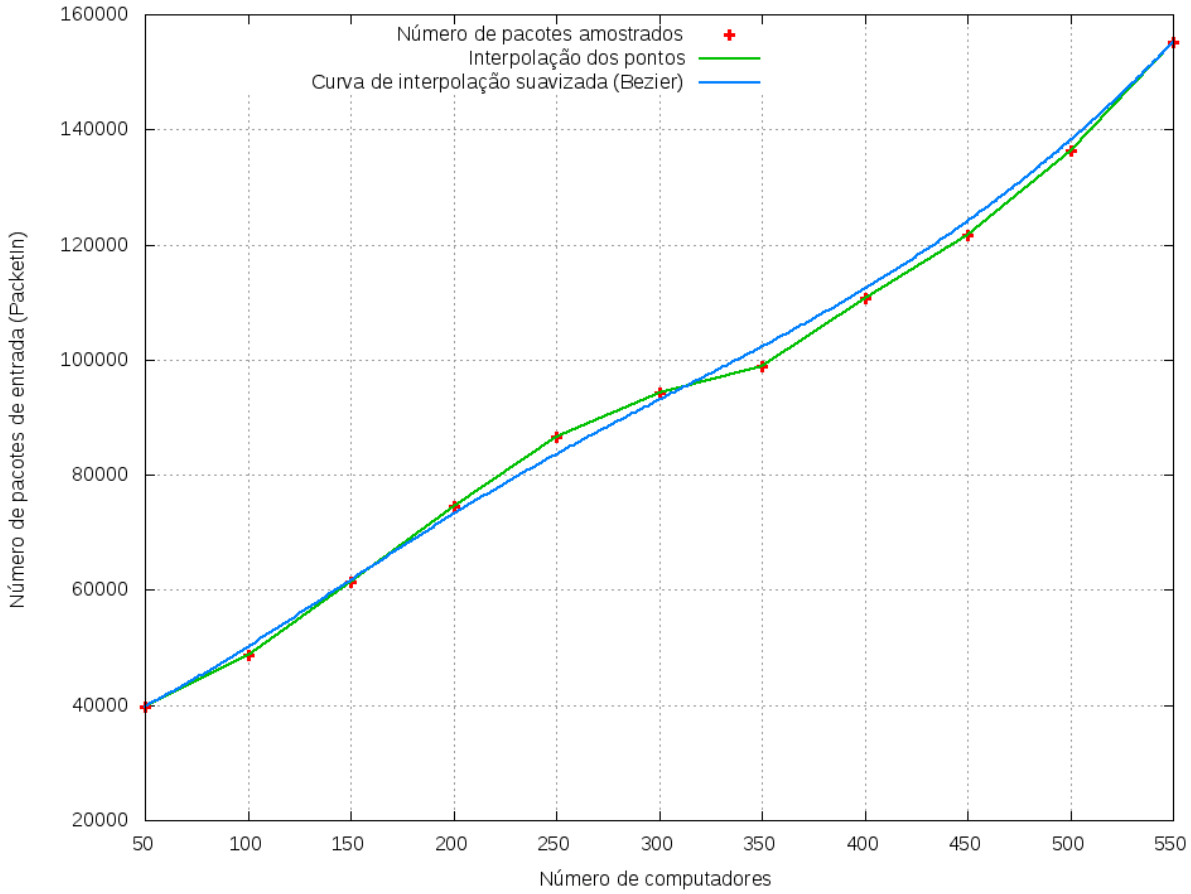

Figura 6.24: Número de pacotes de entrada (*PacketIn*) manipuladas pelo controlador em cada iteração do experimento

Para o experimento da contagem do número de pacotes de entrada no controlador foram computadas as diferenças entre os experimentos posteriores e os anteriores. Em média doze mil pacotes de entrada foram manipulados a mais em cada iteração do experimento.

A figura 6.25 mostra os resultados desse experimento. A linhas verticais vermelhas apresentam o intervalo de confiança dos valores amostrados das diferenças. O erro padrão baixo prova o comportamento mostrado na figura 6.24 sobre o crescimento linear no número de pacotes de entrada.

Considerando o intervalo de tempo de dois minutos de execução do experimento e que, a cada iteração cinquenta novos computadores foram adicionados, tem-se que a razão da média pelo número de computadores por iteração é igual a 250 pacotes. O controlador lida com pacotes que nem sempre são originados de computadores. Assim, esse valor calculado é apenas uma aproximação do número de pacotes que o controlador manipula em função do número de computadores.

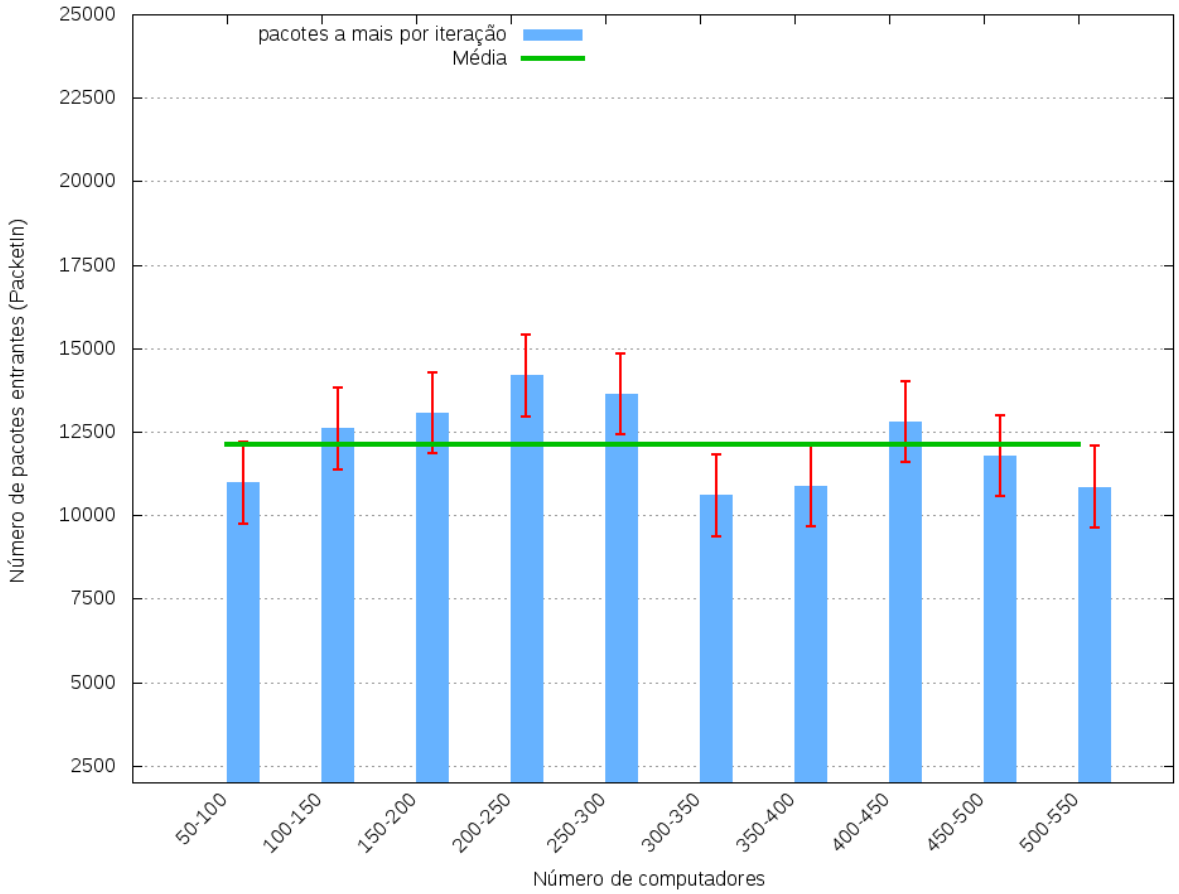

Figura 6.25: Diferença da quantidade de pacotes de entrada (*PacketIn*) manipuladas no controlador entre cada par de iterações do experimento

6.7.3 Comparação dos números de pacotes

Essa subseção compara o número de pacotes de sondagem com o número de pacotes de entrada que o controlador lida. A comparação é feita em quantidade e em percentual. A seguir serão apresentados os resultados dessa avaliação.

A comparação do número de pacotes é mostrada na figura 6.26. Os valores nas coordenadas y (ordenadas) estão em escala exponencial, o que facilita a visualização da diferença entre o número de pacotes de entrada (*PacketIn*) e pacotes de sondagem.

Com a rede Ipê simulada completa foram enviados, no máximo, quase 3 mil pacotes de sondagem e quase 150.000 pacotes de entrada manipulados pelo controlador.

O total de pacotes manipulados pelo controlador é representado pela soma do número total de pacotes de sondagem com o número total de pacotes de entrada (*PacketIn*). Em função disso a figura 6.27 apresenta as proporções em percentuais dos dois tipos de pacotes.

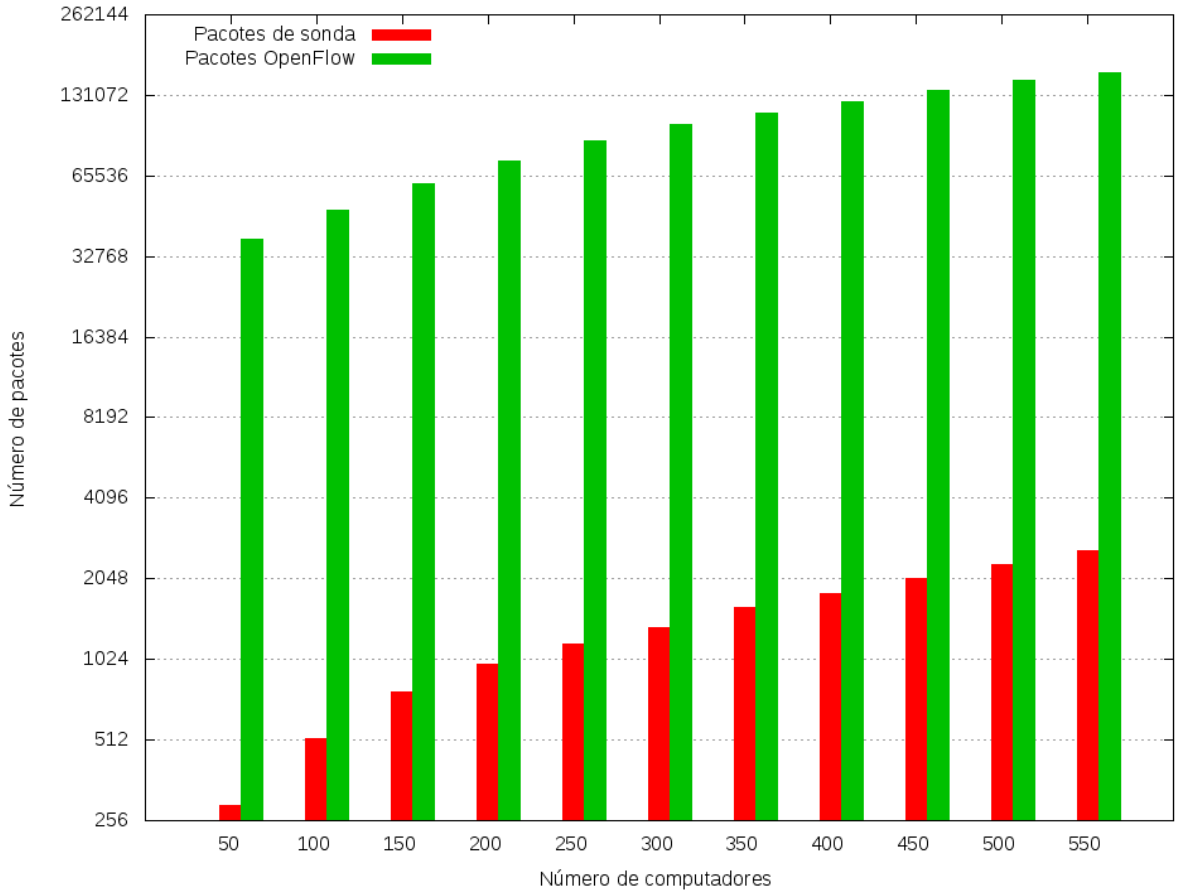

Figura 6.26: Comparação do número de pacotes de sondagem e de entrada manipulados pelo controlador

Nas abscissas tem-se o número de computadores na rede a cada iteração do experimento. Nas ordenadas tem-se os percentuais do número total de pacotes manipulados pelo controlador. É possível notar que os pacotes de sondagem representam um percentual muito baixo em relação ao total de pacotes.

Como apresentado na subseção de avaliação da largura de banda em função do módulo *host_tracker*, os pacotes de sondagem não causam muito impacto no funcionamento da rede e não representam um grande volume de pacotes. Para o presente experimento esses pacotes, em seu pior caso, representaram 2% do total de pacotes manipulados pelo controlador.

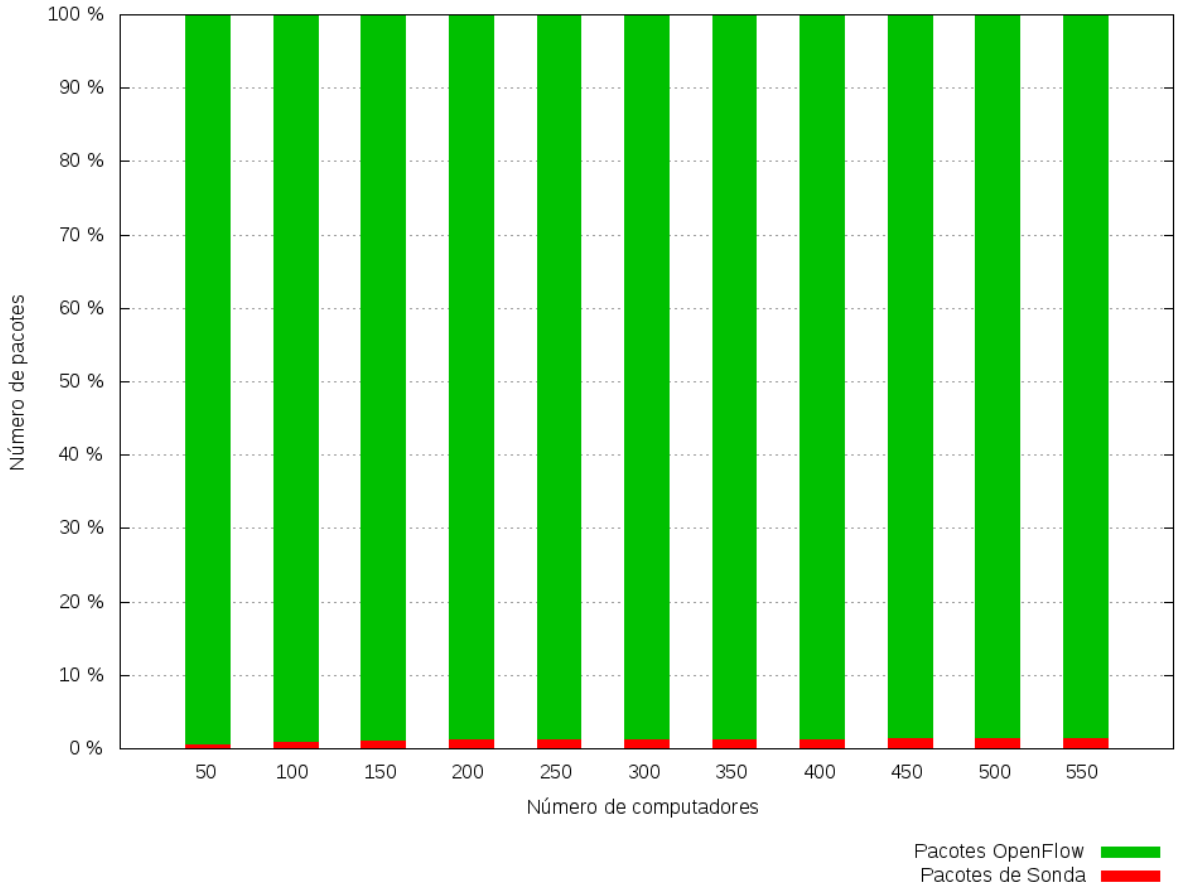

Figura 6.27: Valores percentuais do número de pacotes de sondagem e de entrada manipulados pelo controlador

6.8 Avaliação da rede

Nesta seção são avaliados latência, largura de banda e *jitter* da rede Ipê simulada. Para cada cenário avaliado foram feitas comparações com e sem a presença do controlador com a solução em grafos. É avaliado o impacto da solução em grafos na rede.

6.8.1 Avaliação de latência

Em função do ambiente virtualizado, o experimento de latência foi dividido em duas partes. A primeira avaliou a latência das redes locais. Ou seja, a latência entre computadores de uma mesma sub-rede dentro de uma máquina virtual. A segunda parte, avaliou a latência da rede global. Ou seja, entre servidor físicos diferentes, fazendo com

que os pacotes passem por toda a infraestrutura física do ambiente de simulação. Para cada caso, foram medidos a latência com e sem o controlador em grafos.

Com a rede Ipê simulada completa foram executados, simultaneamente, 6 pares de computadores enviando pacotes de ICMP PING. As medições da latência foram coletadas a cada segundo da execução do experimento. A latência em cada instante é a média de todos os 6 pares de computadores a cada iteração de coleta.

6.8.1.1 Latência da rede local

A figura 6.28 mostra a latência medida ao longo do experimento para as redes locais. É possível notar que a latência da solução com controlador, no início do experimento, está em torno de 4 milissegundos. Esse valor ocorre em função do encaminhamento, pelo comutador, do primeiro pacote para o controlador. Os valores subsequentes são próximos ao valor da latência medido na solução sem controlador.

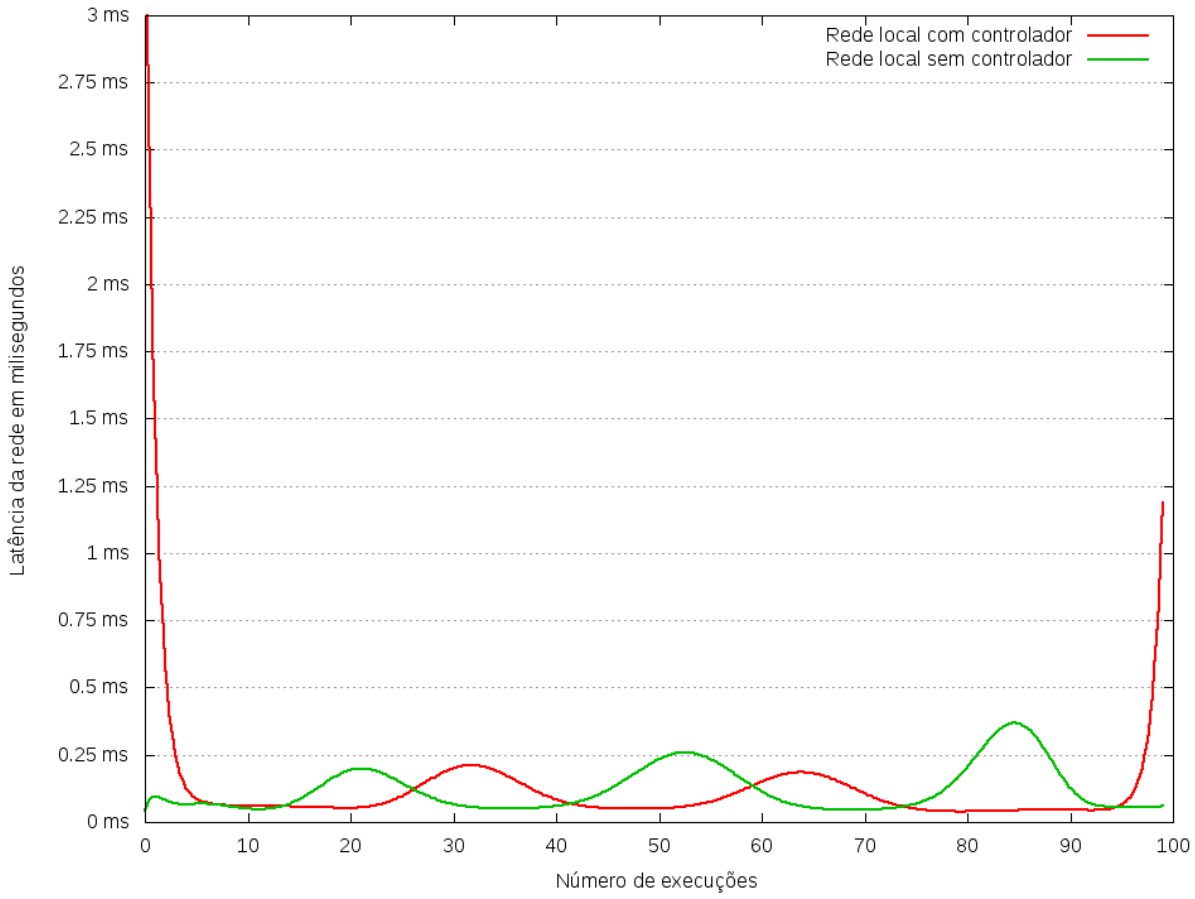

Figura 6.28: Latência média das redes locais com e sem o controlador com a solução em grafos

Uma avaliação dos dados coletados da largura de banda no ambiente das redes locais foi realizada. Os valores de latência mínima, máxima e média com intervalo de confiança são apresentados na figura 6.29. No geral, as latências da rede com a solução do controlador em grafos e sem controlador se mantiveram parecidas. Em média, a latência no experimento das redes locais foi de $0,13$ milissegundos. É possível notar através do intervalo de confiança que a latência na rede com o controlador se manteve mais constante ao longo do experimento. Ou seja, mais confiável, com baixa variância.

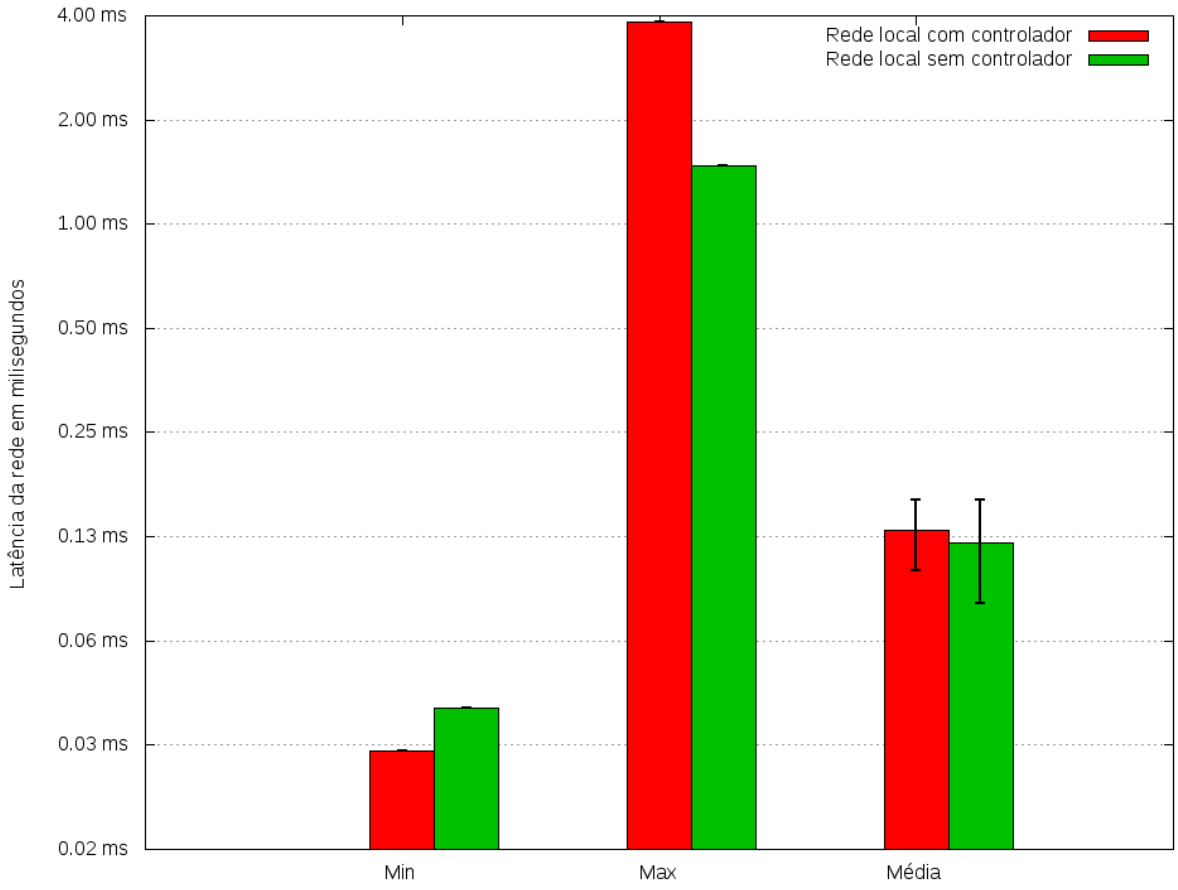

Figura 6.29: Latência mínima, máxima e média das redes locais

6.8.1.2 Latência da rede global Ipê

O mesmo perfil de experimento foi executado para o cenário da rede global. 6 pares de computadores executaram PINGs coletando os valores da latência a cada segundo. Para esse experimento, foram separados os pares em computadores diferentes para garantir que cada pacote trafegasse por toda infraestrutura física da rede simulada.

A figura 6.30 apresenta os resultados dessa avaliação. Como pode ser visto, a latência inicial coletada na rede com a solução do controlador com o módulo grafo chegou a 400 milissegundos. Esse valor ocorre em função do primeiro pacotes que é encaminhado para o controlador. No geral, a latência da rede sem controlador sofreu uma variação mais alta. Em contraponto a curva da latência com o controlador demonstra-se mais constante.

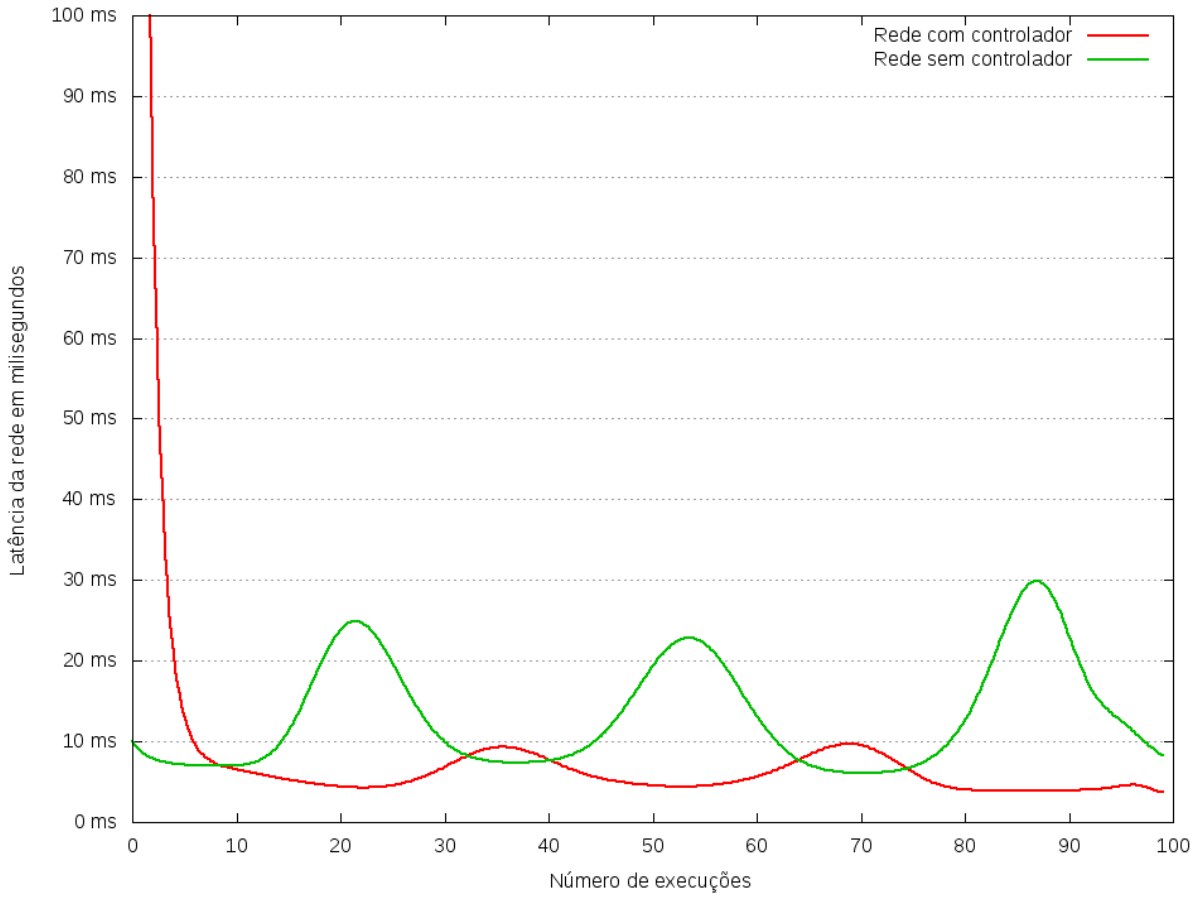

Figura 6.30: Latência da rede Ipê com e sem o controlador com a solução em grafos

Em média, a latência da rede Ipê simulada foi 13 milissegundos. Na rede global, o experimento sem controlador teve oscilações bem altas que chegaram a valores de 100 milissegundos de latência. É possível notar novamente a estabilidade da rede com controlador observando o desvio padrão mais baixo. A figura 6.31 apresenta os resultados da avaliação dos valores mínimos, máximos e médios com desvio padrão da rede Ipê simulada.

Através dos experimento de latência realizados tanto nas redes locais quanto na rede Ipê global, pode-se notar que o controlador com a solução do módulo em grafos não gerou grande impacto na rede. Com exceção dos primeiros pacotes, com uma latência alta, a rede com o controlador se mostrou mais estável do que sem.

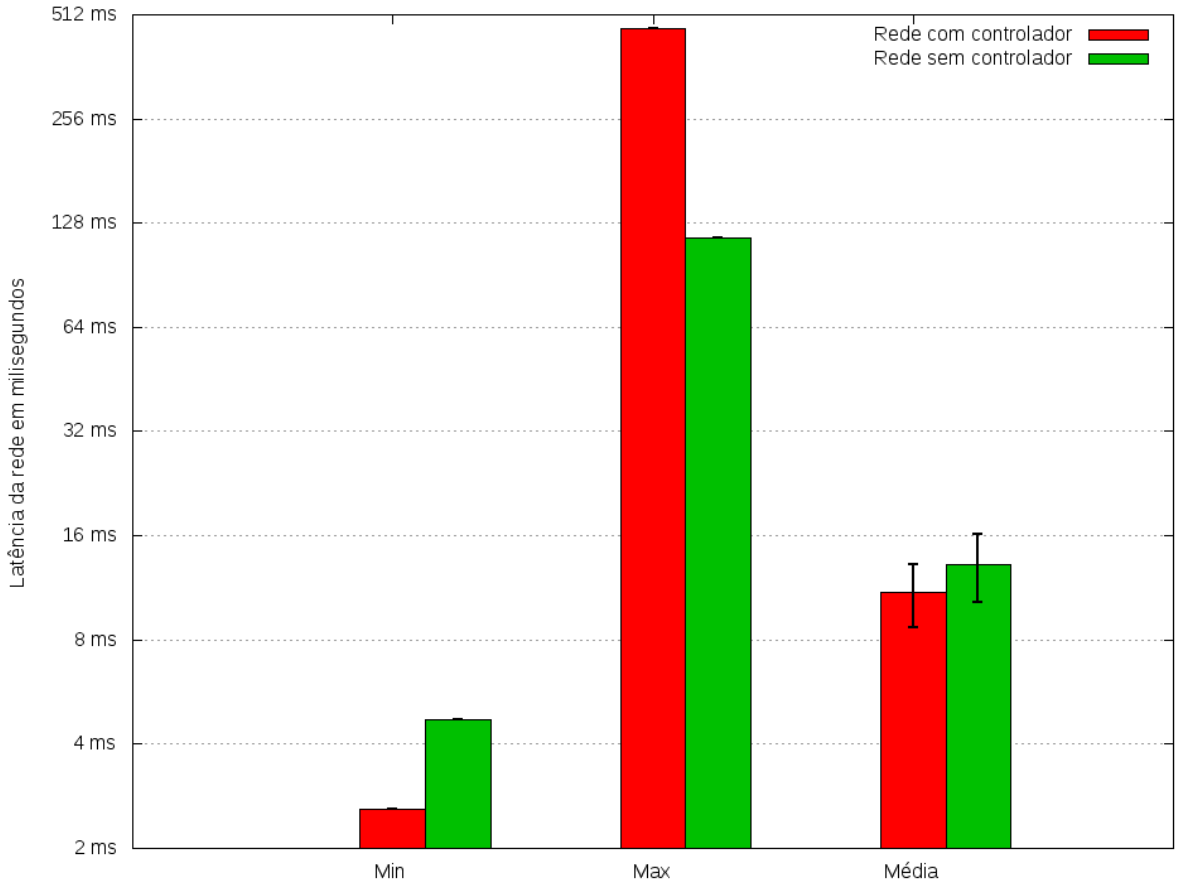

Figura 6.31: Avaliação da latência da rede Ipê considerando os valores mínimos, máximos e médios analisados

6.8.2 Avaliação de largura de banda

A avaliação da largura de banda foi feita utilizando 4 pares de computadores em sub-redes e servidores da rede física diferentes. Para cada par de computadores, foi utilizada a ferramenta de linha comando *iperf*. O experimento consistiu em medir, com intervalos de 5 segundos, durante 5 minutos, a largura de banda através de conexões TCP. A largura de banda medida é representada pela média das quatro medições a cada iteração de coleta durante o experimento.

A largura de banda da rede sem controlador medida durante o experimento foi, em média, 18 *Megabytes*. Na figura 6.32 são apresentados os resultados do experimento. Conforme pode ser visto nesta figura, os pontos representam os valores médios amostrados em todos os pares de computadores que estavam coletando a largura de banda.

Uma interpolação suavizada e uma passando por todos os pontos médios mostram a largura de banda média computada nesse experimento.

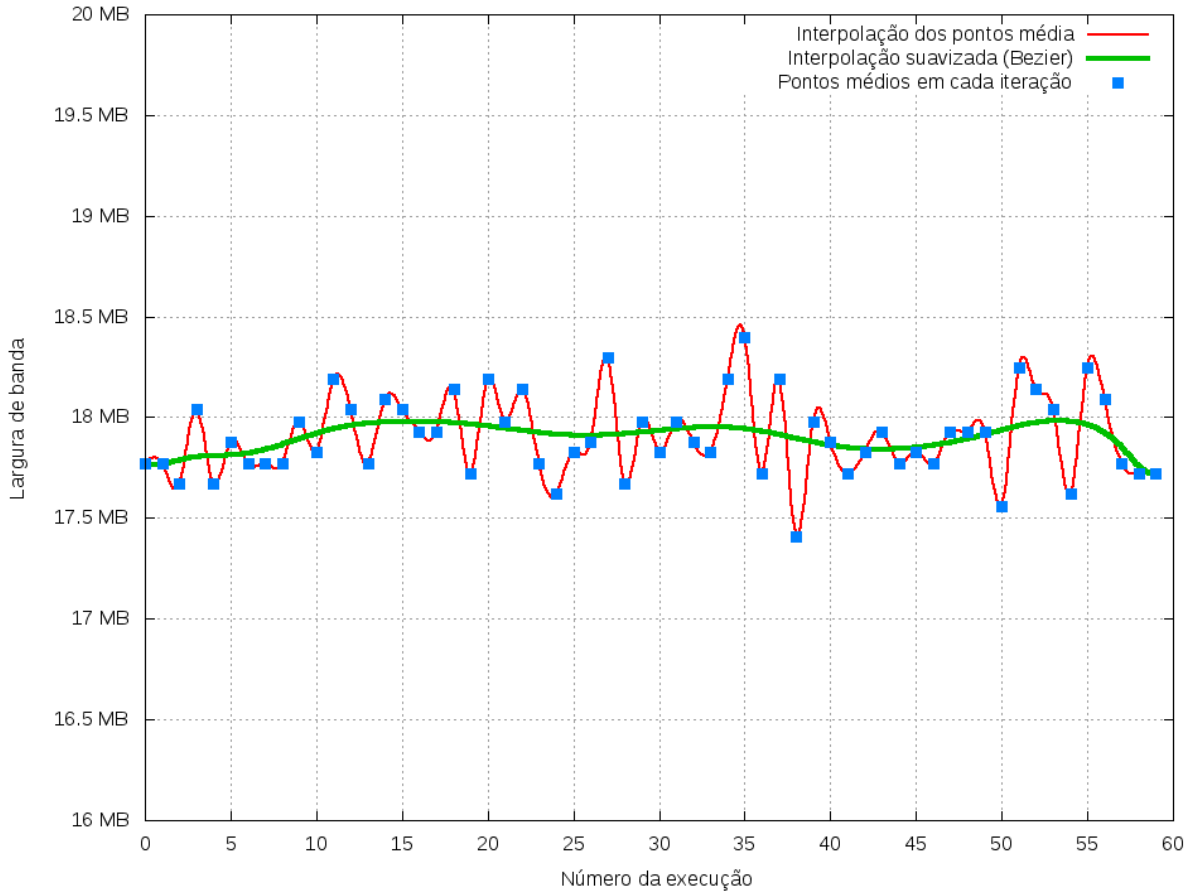

Figura 6.32: Largura de banda média em *Megabytes* da rede sem controlador ao longo de 60 coletas

O mesmo experimento foi executado para a rede com o controlador utilizando a solução do módulo em grafos. Para esse cenário, notou-se uma redução na largura de banda em função da utilização do *OpenFlow* e o módulo em grafos.

A figura 6.33 apresenta os resultados da largura de banda média para o experimento com a solução em grafos. A cada 5 segundos, uma nova conexão foi estabelecida entre cada par de computadores.

O primeiro pacote de cada conexão gerou um pacote de entrada *PacketIn*. Ao ser tratado pelo controlador, esses pacotes reduzem a fluidez da rede, pois ocupam a CPU do controlador. Pacotes na rede sem controlador não precisam ser processados por um computador externo, logo não consomem da mesma forma a largura de banda.

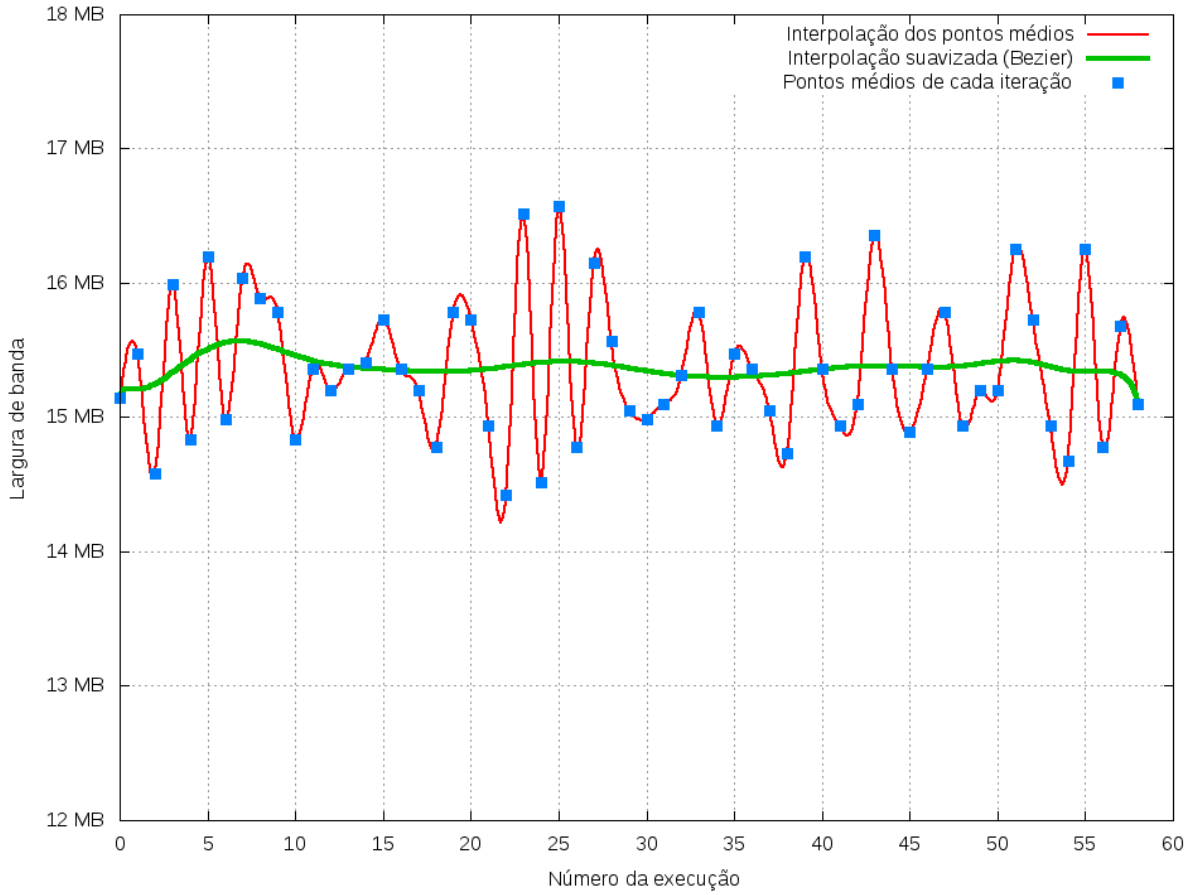

Figura 6.33: Largura de banda média em *Megabytes* da rede com o controlador com a solução em grafos ao longo de 60 coletas

Como o tempo para processar os pacotes de entrada é sempre o mesmo, pode-se inferir que a redução da largura de banda se manteria a mesma em qualquer cenário. Ou seja, em uma rede *Gigabit* a diferença entre a largura de banda média com e sem controlador seria a mesma.

A figura 6.34 apresenta a diferença entre as larguras de banda média nos experimentos sem controlador e com o módulo em grafos utilizando *OpenFlow*. A curva em vermelho representa a largura de banda média da rede sem controlador. A curva em verde, a largura de banda da rede com controlador utilizando a solução em grafos. As curvas representam a interpolação suavizada dos pontos médios computados durante as 60 coletas realizadas durante o experimento.

Uma redução média de $2,5$ *Megabytes* foi registrada durante o experimento da a rede sem controlador para a rede com controlador.

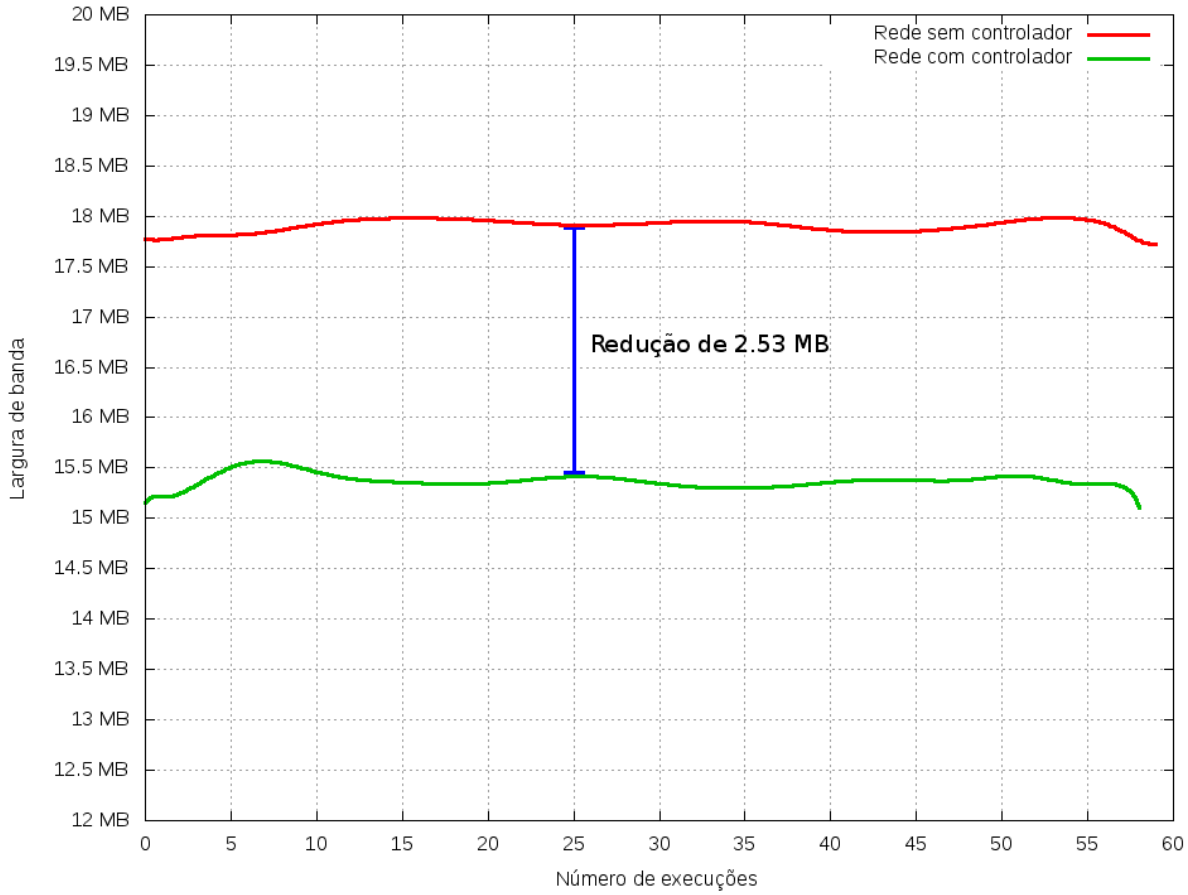

Figura 6.34: Diferença da largura de banda dos experimento com e sem controlador

6.8.3 Avaliação de *jitter*

O experimento para medir o *Jitter* foi feito utilizando a ferramenta de linha de comandos *iperf*. 6 pares de computadores executaram a ferramenta transmitindo pacotes UDP durante 5 minutos. A cada 5 segundos foram coletados os valores do *jitter* da rede.

O experimento descrito acima foi executado na rede sem controlador e na rede com controlador utilizando a solução do módulo em grafos. A rede Ipê simulada foi executada por completo. Ou seja, 27 comutadores e 550 computadores sendo simulados durante o experimento.

A figura 6.35 apresenta o resultado da coleta de amostras com o valor médio do *jitter* em cada instante de execução. É possível notar que a variação média na transmissão de pacotes está entre 1 e 2 milissegundos.

O mesmo experimento foi executado para a rede com controlador utilizando a solução do módulo em grafos. Para esse cenário é possível notar o valor médio do *jitter* bem alto no início de cada experimento. Esse valor representa os primeiros pacotes que necessitam ser avaliados pelo controlador. Logo, a variação da latência (*jitter*) é mais alta

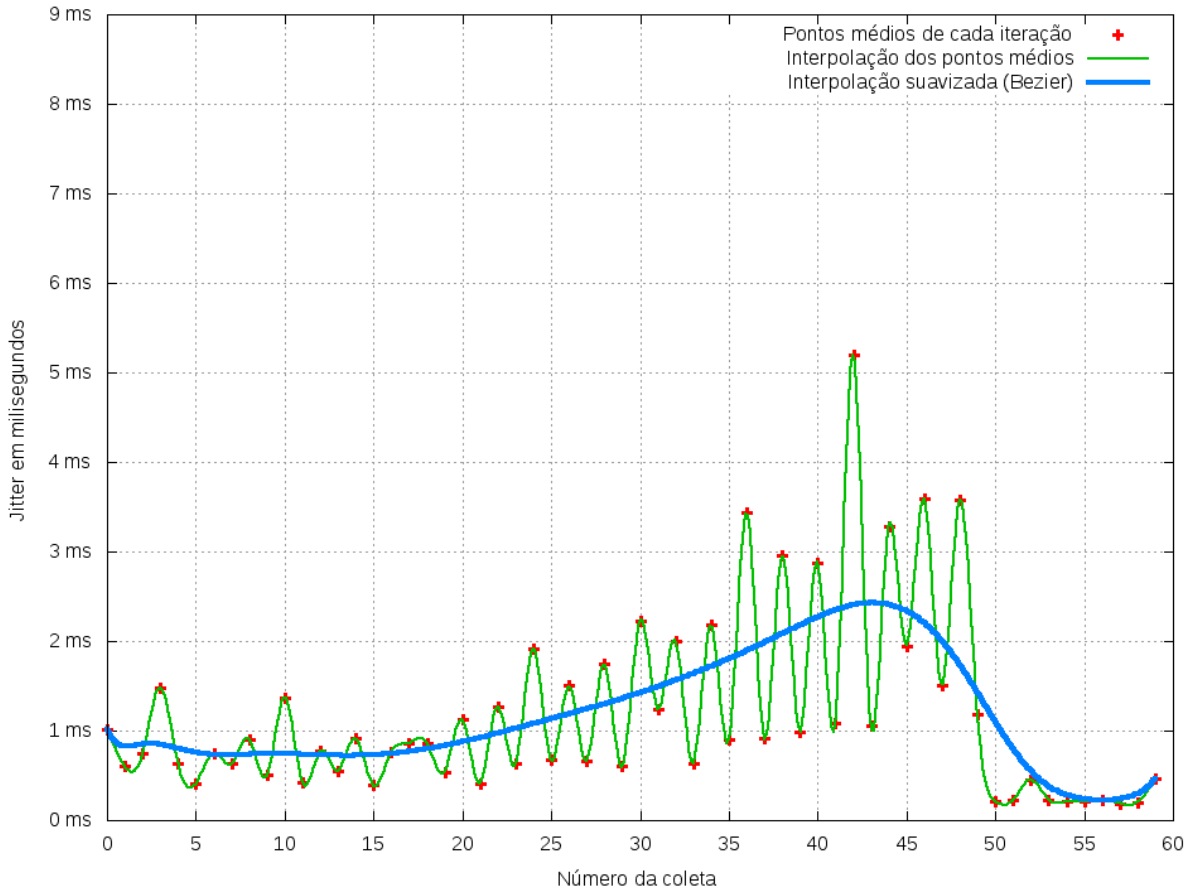

Figura 6.35: Valor médio do *jitter* da rede sem controlador coletada com intervalo de 5 segundos durante 5 minutos em 6 pares de computadores

para esses casos.

É possível notar que, em média, o *jitter* está entre 1 e 2 milissegundos. Observando a sequência das amostras e o valor do coletado em cada período é possível notar que a variação média é baixa, o que demonstra estabilidade na rede do experimento.

A figura 6.36 mostra os resultados da coleta das amostras de *jitter* médio para a rede com controlador utilizando a solução com módulo em grafos. Os pontos em vermelho representam o valor médio coletado dos 6 pares de computadores para o instante do experimento. A curva em cor verde representa a interpolação dos pontos médios. A curva em azul é a interpolação suavizada que apresenta os resultados do experimento.

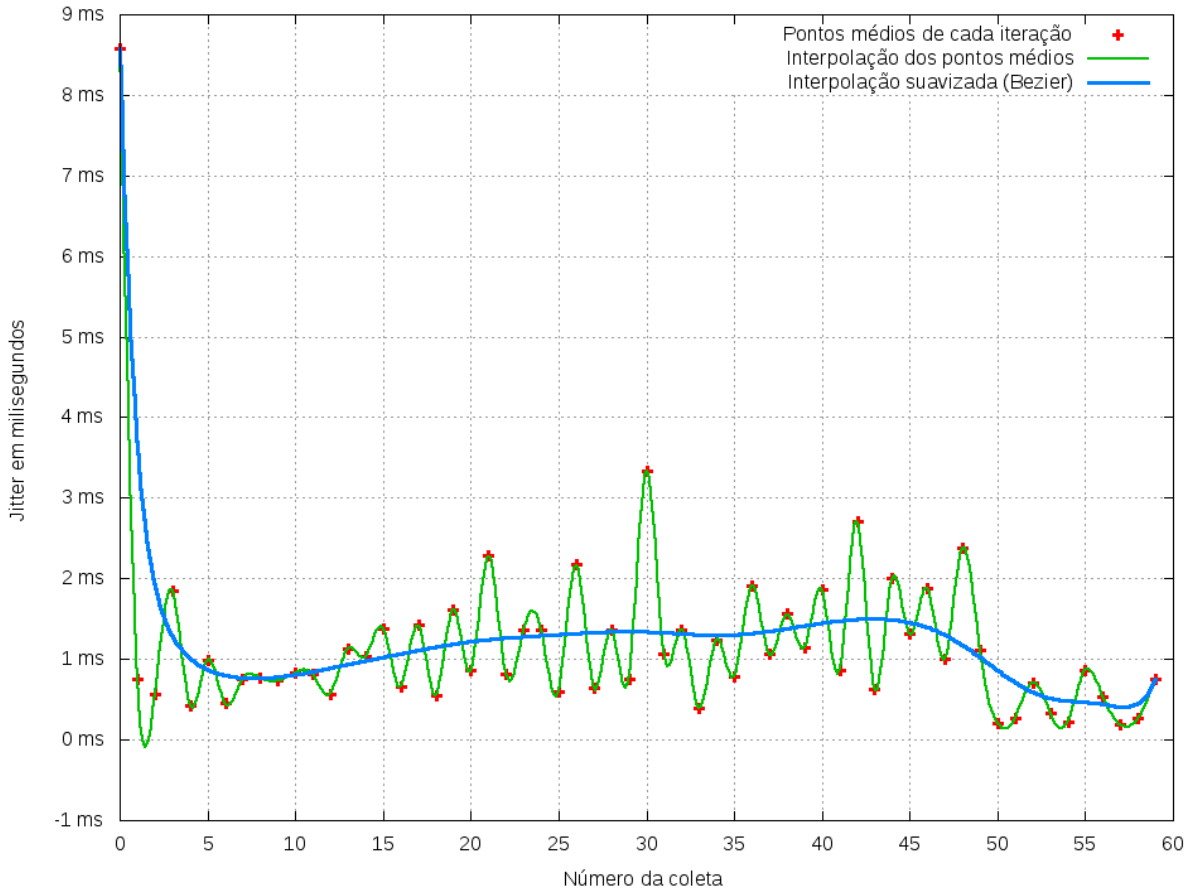

Figura 6.36: Valor médio do *jitter* da rede com controlador utilizando a solução com módulo em grafos coletada com intervalo de 5 segundos durante 5 minutos em 6 pares de computadores

Comparando-se a variação da latência da rede sem controlador e com controlador, nota-se que elas não se diferem muito. A rede com controlador possui um *jitter* alto no início do experimento em função dos primeiros pacotes, que geram *PacketIn*. A rede sem controlador teve uma variação maior, observando-se as iterações do experimento. Apesar das diferenças, o *jitter* médio, nos dois cenários de teste, foi aproximadamente o mesmo; entre 1 e 2 milissegundos.

A figura 6.37 apresenta um gráfico comparando a variação da latência *jitter* nos dois cenários de experimento. A curva em vermelho representa os valores coletados para a rede com controlador utilizando a solução em grafos. A curva em verde representa os valores coletados para a rede sem controlador.

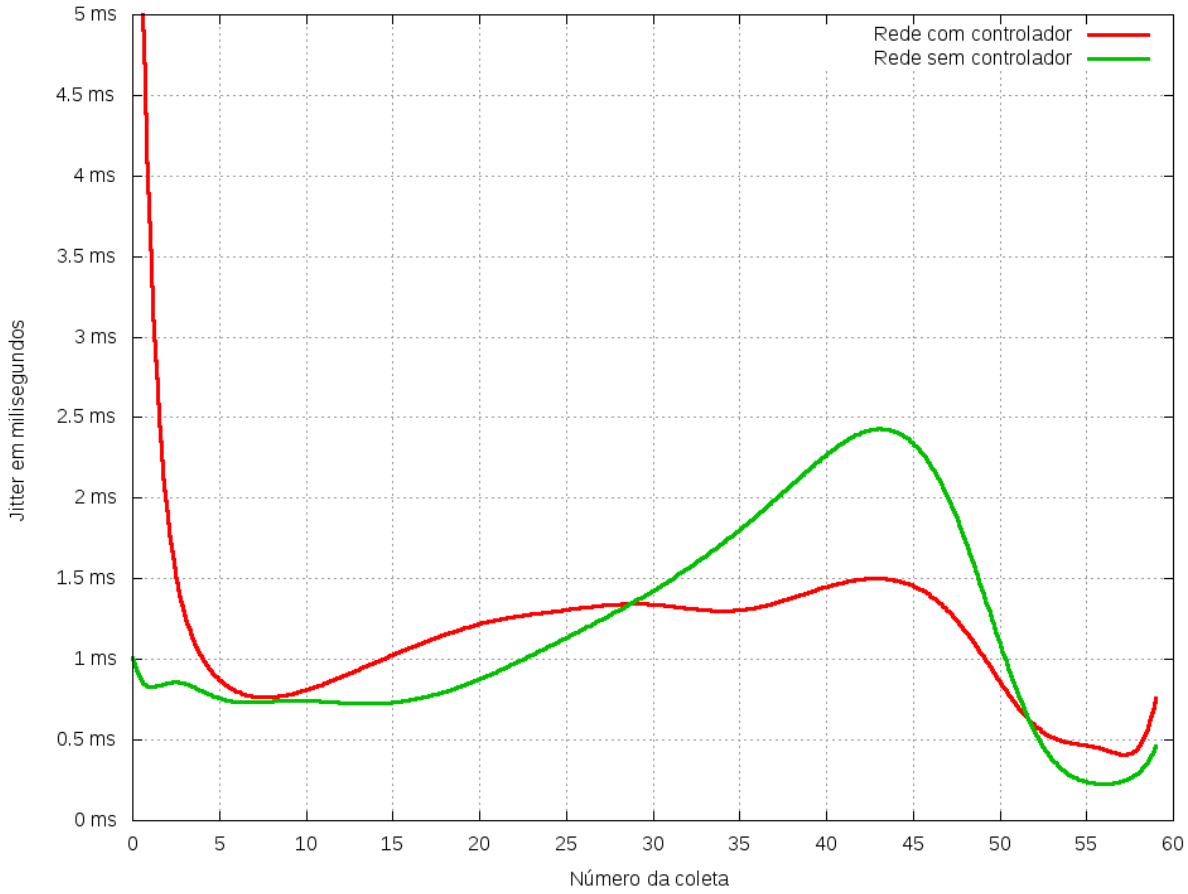

Figura 6.37: Comparação do *jitter* das redes com e sem controlador

6.8.4 Avaliação de percentual de erros

O experimento foi feito no ambiente simulado da rede Ipê. 6 pares de computadores, utilizando o utilitário de linha de comandos *iperf* com o protocolo UDP, coletaram o número de perdas de pacotes durante cada iteração do experimento. Cada iteração representa um intervalo de tempo de 5 segundos. Os experimentos foram executados durante 300 segundos (5 minutos) em dois cenários: um com a rede sem controlador e a outro com a rede com controlador utilizando a solução em grafos.

A figura 6.38 Apresenta os resultados da computação dos valores coletados nos experimentos. As colunas em vermelho representam a rede com controlador. As em verde a rede sem controlador. Cada iteração do experimento transmitiu, aproximadamente, 1000 pacotes. O eixo y do lado esquerdo representa o número de pacotes em escala exponencial. O eixo y do lado direito representa o percentual de perda de pacotes em

relação ao total.

Em média, o número de pacotes perdidos para a rede sem controlador foi de 1 pacote de 1000. Para a rede com controlador foram 2 pacotes. Em ambos os casos, considera-se que perda de pacotes tem baixo impacto. Em percentuais, a rede com controlador obteve uma perda de 0.5%. A rede sem controlador, 0.1%. Nos dois casos, menos de 1% de perda do total de pacotes transmitidos.

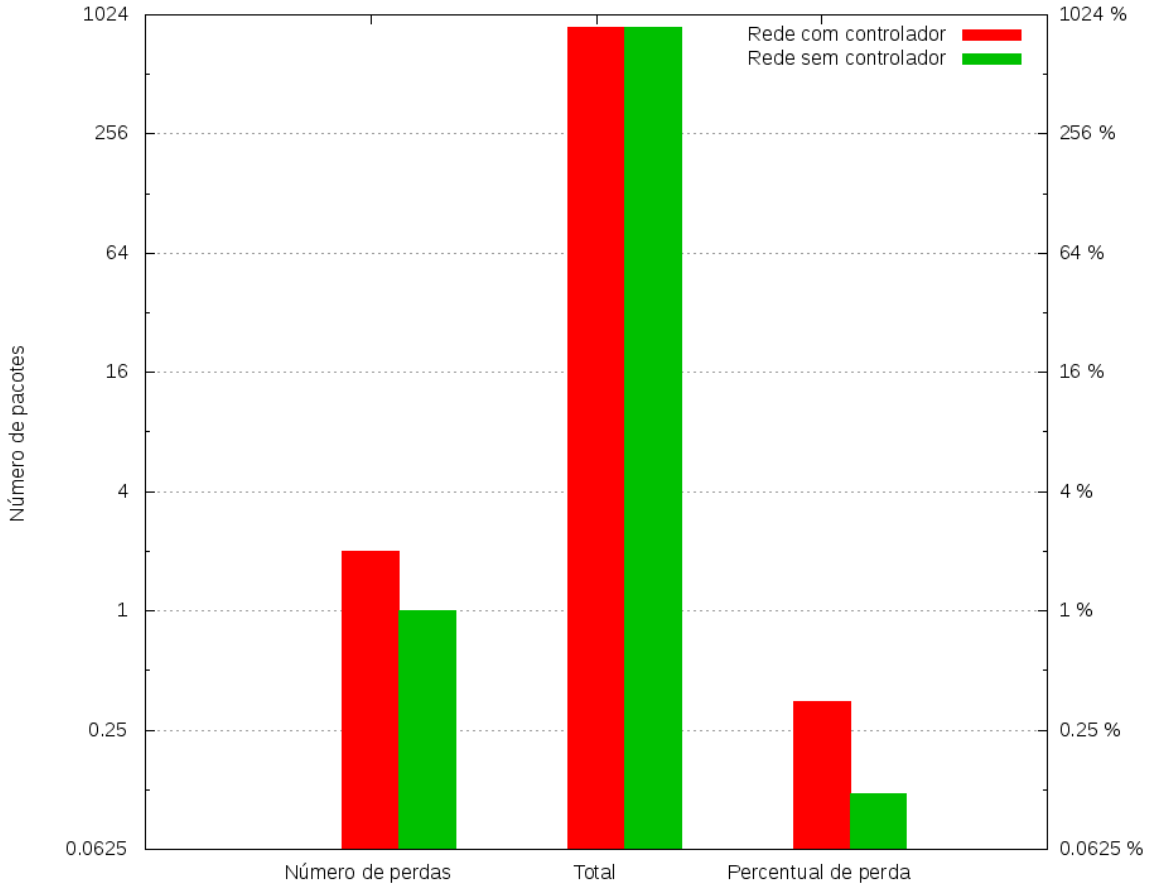

Figura 6.38: Número de perdas de pacotes, total de pacotes transmitidos por iteração e percentual de perda coletados em dois experimentos. Um com a rede sem controlador e outro com a rede com controlador utilizando a solução em grafos

Capítulo 7

Conclusão e trabalhos futuros

O presente trabalho apresentou o uso de grafos no contexto de SDN. Um grafo da rede em tempo real vai de encontro com uma das principais vantagens em se desacoplar o plano de controle do plano de dados que é obter uma visão global da rede. O sistema proposto mantém um estado fiel, consistente e dinâmico da rede real, facilitando a tarefa de gerenciamento de um rede definida por software e reduzindo o volume de computação nos nós da rede. Grafos são uma modelagem direta e precisa da topologia de uma rede. Eles deveriam ser um recurso básico, uma premissa em controladores SDN para representar a rede.

A abstração em grafos foi identificada em outros cenários de rede como *cloud computing*. *OpenStack*, um dos mais populares sistemas para gerenciamento de ambientes virtualizados, possui uma visualização da topologia da rede em um de seus módulos Neutron [1]. Essa abstração foi combinada em uma solução SDN para implementar ambientes isolados com redes *multi-tenant* [44]. Seria interessante avaliar uma possível combinação unificada com uma visão compartilhada da rede.

Como uma proposta futura, pode-se criar um visualizador em tempo real do grafo que interaja com o administrador da rede e mostre, de uma maneira simples, toda a operação da rede.

Assim como o controlador POX, o módulo proposto, ao ter seu processo terminado, não persiste as informações de estado da rede. Todo o grafo computado é perdido. Em função disso, um banco de dados em grafos, distribuído, poderia ser utilizado para persistir o grafo, e as informações do estado da rede, de maneira confiável e tolerante a falhas.

Algoritmos genéricos em grafos poderiam ser implementados como uma biblioteca para o módulo. Assim, estabelecida uma periodicidade, esses algoritmos poderiam ser computados no grafo e seus resultados publicados como extensão da API.

Uma avaliação comparando algoritmos de roteamento como OSPF e BGP com uma solução que faz utilização do grafo seria interessante a fim de mostrar a redução na computação dos nós da rede.

Referências

[1] Openstack:open source software for building private and public clouds. http://openstack.org/, March 2012.

[2] Gember Aaron, Prabhu Prathmesh, Ghadiyali Zainab, and Akella Aditya. Toward software-defined middlebox networking. 2012.

[3] Tavakoli Arsalan, Casado Martin, Koponen Teemu, and Shenker Scott. Applying nox to the datacenter. 2009.

[4] Raghavan Barath, Koponen Teemu, Ghodsi Ali, Casado Martín, Ratnasamy Sylvia, and Shenker Scott. Software-defined internet architecture. 2012.

[5] controller Beacon. Beacon controller, February 2015.

[6] Pankaj Berde, Matteo Gerola, Jonathan Hart, Yuta Higuchi, Masayoshi Kobayashi, Toshio Koide, Bob Lantz, Brian O'Connor, Pavlin Radoslavov, William Snow, and Guru Parulkar. Onos: Towards an open, distributed sdn os. In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, HotSDN '14, pages 1–6, New York, NY, USA, 2014. ACM.

[7] Mark Berman, Jeffrey S. Chase, Lawrence Landweber, Akihiro Nakao, Max Ott, Dipankar Raychaudhuri, Robert Ricci, and Ivan Seskar. Geni: A federated testbed for innovative network experiments. *Computer Networks*, 61(0):5 – 23, 2014. Special issue on Future Internet Testbeds – Part I.

[8] Jun Ko Bong, Pappas Vasileios, Raghavendra Ramya, Song Yang, B. Dilmaghani Raheleh, Lee Kang-Won, and Verma Dinesh. An information-centric architecture for data center networks. 2012.

[9] Martin Casado, Teemu Koponen, Rajiv Ramanathan, and Scott Shenker. Virtualizing the network forwarding plane. In *Proceedings of the Workshop on Programmable Routers for Extensible Services of Tomorrow*, PRESTO '10, pages 8:1–8:6, New York, NY, USA, 2010. ACM.

[10] Dimitris Papadopoulos Charalampos Mavroforakis. Modelling of openflow controllers with alloy: A load-balancing protocol. 2013.

[11] Wenbo Chen, Hui Li, Qiang Ma, and Zhihao Shang. Design and implementation of server cluster dynamic load balancing in virtualization environment based on openflow. In *Proceedings of The Ninth International Conference on Future Internet Technologies*, CFI '14, pages 9:1–9:6, New York, NY, USA, 2014. ACM.

[12] Dave CLARK, Bill LEHR, Steve BAUER, Peyman FARATIN, and Rahul SAMI. *Overlay Networks and the Future of the Internet*. Massachusetts Institute of Technology, 2006.

[13] Andersen David, Balakrishnan Hari, Kaashoek Frans, and Morris Robert. Resilient overlay networks. 2001.

[14] Bruce S.. Davie and Adrian Farrel. *MPLS: Next Steps*. Elsevier, 2008.

[15] WiNet DCC-UFMG. Laboratório winet, March 2015.

[16] Erik de Britto e Silva, Gustavo Pantuza, Frederico Sampaio, Bruno Pereira dos Santos, Luiz Filipe M. Vieira, Marcos Augusto M. Vieira, and Daniel Fernandes Macedo. Enforcing link utilization with traffic engineering on sdn. 2015.

[17] Rede Nacional de Pesquisa. Conectividade rede ipê, December 2015.

[18] Keller† Eric, Ghorbani Soudeh, Caesar Matt, and Jennifer Rexford. Live migration of an entire network (and its hosts). 2012.

[19] Projeto Floodlight. Floodlight - open source software for building sdn, April 2014.

[20] Nate Foster, Rob Harrison, Michael J. Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. Frenetic: a network programming language. *SIGPLAN Not.*, 46(9):279–291, September 2011.

[21] Michael J. Freedman. Experiences with coralcdn: A five-year operational view. 2010.

[22] FriendFeed. Tornado, December 2013.

[23] Geant Geant. Rede geant, March 2015.

[24] Natasha Gude, Teemu Koponen, Justin Pettit, Ben Pfaff, Martín Casado, Nick McKeown, and Scott Shenker. NOX: towards an operating system for networks. *SIGCOMM Computer Communication Review*, 38:105–110, July 2008.

[25] Dorgival Guedes, Luiz Vieira, Marcos Vieira, and Henrique Rodrigues. Redes definidas por software: uma abordagem sistêmica para o desenvolvimento de pesquisas em redes de computadores, 2012.

[26] Anees Shaikh Guohui Wang, T. S. Eugene Ng. Programming your network at runtime for big data applications. 2012.

[27] Nikhil Handigol, Mario Flajslik, Srini Seetharaman, Ramesh Johari, and Nick McKeown. Aster*x: Load-balancing as a network primitive. 2010.

[28] Nikhil Handigol, Mario Flajslik, Srini Seetharaman, and Nick McKeown Ramesh Johari. Plug-n-serve: Load-balancing web trafic using openflow. 2009.

[29] Brandon Heller et al. Ripcord: a modular platform for data center networking. *SIGCOMM Comput. Commun. Rev.*, 40:457–458, 2010.

[30] Timothy L. Hinrichs, Natasha S. Gude, Martin Casado, John C. Mitchell, and Scott Shenker. Practical declarative network management. pages 1–10, 2009.

[31] Internet2 Internet2. Rede internet2, March 2015.

[32] Stoica Ion, Morris Robert, Liben-Nowell David, Karger David R., Kaashoek M. Frans, Dabek Frank, and Balakrishnan Hari. Chord: A scalable peer-to-peer lookup protocol for internet applications. 2001.

[33] Raj Jain, Dah-Ming Chiu, and William R Hawe. *A quantitative measure of fairness and discrimination for resource allocation in shared computer system.* Eastern Research Laboratory, Digital Equipment Corporation Hudson, MA, 1984.

[34] Constantine Dovrolis Jennifer Rexford. Future internet architecture. 2010.

[35] Dilley John, Maggs Bruce, Parikh Jay, Prokop Harald, Sitaraman Ramesh, and Weihl Bill. Globally distributed content delivery. 2002.

[36] Teemu Koponen, Martin Casado, Natasha Gude, Jeremy Stribling, Leon Poutievski, Min Zhu, Rajiv Ramanathan, Yuichiro Iwata, Hiroaki Inoue, Takayuki Hama, and Scott Shenker. Onix: a distributed control platform for large-scale production networks. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–6, Berkeley, CA, USA, 2010. USENIX Association.

[37] Bob Lantz, Brandon Heller, and Nick McKeown. A network in a laptop: Rapid prototyping for software-defined networks. pages 19:1–19:6, 2010.

[38] CNPq Libfluid. Libfluid, February 2015.

[39] Casado Martín, Koponen Teemu, Shenker Scott, and Tootoonchian Amin. Fabric: A retrospective on evolving sdn. 2012.

[40] W. D. Maurer and T. G. Lewis. Hash table methods. *ACM Comput. Surv.*, 7(1):5–19, March 1975.

[41] Nick McKeown, Hari Balakrishnan Tom Anderson, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. *OpenFlow: Enabling Innovation in Campus Networks*. ACM, 2008.

[42] Christopher Monsanto, Joshua Reich, Nate Foster, Jennifer Rexford, and David Walker. Composing software-defined networks. In *Proceedings of the 10th USENIX conference on Networked Systems Design and Implementation*, nsdi'13, pages 1–14, Berkeley, CA, USA, 2013. USENIX Association.

[43] controller NOX. Nox controller, February 2015.

[44] Rogério V Nunes, Raphael L. Pontes, and Dorgival Guedes. Virtualized network isolation using software defined networks. In *Proceedings of the 38th IEEE Conference on Local Computer Networks (LCN)*, pages 700–703. IEEE, 2013.

[45] ONF Open Networking Foundation. Openflow 1.5 specification, December 2014.

[46] OpenVSwitch OpenVSwitch. Switch virtual openvswitch, March 2015.

[47] John Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, Stephen Rumble, Ryan Stutsman, and Stephen Yang. The ramcloud storage system. *ACM Trans. Comput. Syst.*, 33(3):7:1–7:55, August 2015.

[48] Gustavo Pantuza, Frederico Sampaio, Luis Vieira, Dorgival Guedes, and Marcos Vieira. Análise e gerenciamento de rede através de grafos em redes definidas por software. 01 2014.

[49] Gustavo Pantuza, Frederico Sampaio, Luiz F. M. Vieira, Dorgival Guedes, and Marcos A. M. Vieira. Network management through graphs in software defined networks. In *10th International Conference on Network and Service Management (CNSM) and Workshop*, pages 400–405, 2014.

[50] Larry Peterson, Andy Bavier, Marc E. Fiuczynski, and Steve Muir. Experiences building planetlab. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 351–366, Berkeley, CA, USA, 2006. USENIX Association.

[51] controller POX. Pox controller, February 2015.

[52] Ramya Raghavendra, Jorge Lobo, and Kang-Won Lee. Dynamic graph query primitives for SDN-based cloud network management. In *Proceedings of the Workshop on Hot Topics on Software Defined Networking, HotSDN'12*, 2012.

[53] RedCLARA RedCLARA. Rede redclara, March 2015.

[54] RNP Rede IPÊ. Rede ipÊ, March 2015.

[55] FOSTER Ian RIPEANU Matei, IMAMNITCHI Adriana. Mapping the gnutella network. 2002.

[56] Christian Esteve Rothenberg, Marcelo Ribeiro Nascimento, Marcos Rogério Salvador, and Maurício Ferreira Magalhães. Openflow e redes definidas por software: um novo paradigma de controle e inovação em redes de pacotes. *Cad. CPqD Tecnologia, Campinas*, 7(1):65–76, 2010.

[57] controller Ryu. Ryu controller, February 2015.

[58] godard sebastien. Sysstat, March 2015.

[59] Hardeep Uppal and Dane Brandon. Openflow based load balancing. 2010.

[60] van Jacobson. Networking named content. 2009.

[61] Richard Wang, Dana Butnariu, and Jennifer Rexford. Openflow-based server load balancing gone wild. 2011.

Apêndice A

Publicações científicas

Os seguintes artigos foram publicados como consequência direta da pesquisa desenvolvida nesta dissertação.

O primeiro trabalho apresenta uma abstração baseada em grafos para o gerenciamento de redes em ambientes de Software Defined Networking, integrando a representação topológica ao plano de controle do controlador SDN [49]. O sistema modela dispositivos e enlaces como um grafo dinâmico atualizado em tempo real, permitindo a execução de algoritmos clássicos diretamente no controlador. A implementação sobre o POX, avaliada em ambiente Mininet, demonstra que a abordagem fornece uma visão global consistente da rede e simplifica tarefas de monitoramento e gerenciamento.

O segundo artigo descreve a consolidação dessa arquitetura de grafos como uma estrutura central de gerenciamento no controlador SDN [48]. O projeto integra módulos de descoberta de topologia, rastreamento de hosts e coleta de métricas de tráfego para manter um grafo fiel ao estado da rede. Os experimentos evidenciam atualização dinâmica da topologia e reutilização eficiente de algoritmos de grafos, reforçando a viabilidade da abstração como base para aplicações de controle.

O terceiro trabalho explora engenharia de tráfego sobre SDN com o objetivo de forçar a utilização eficiente de múltiplos enlaces por meio da divisão dinâmica de fluxos [16]. A solução, implementada sobre um controlador OpenFlow, distribui conexões segundo políticas simples de balanceamento e foi validada em ambiente físico com switches comerciais. Os resultados mostram ganhos significativos de throughput e maior aproveitamento da infraestrutura, demonstrando a capacidade do sistema de adaptar-se dinamicamente às variações de carga.